RankPQO: Learning-to-Rank for Parametric Query Optimization [Technical Report]

Songsong Mo[1], Yue Zhao[1], Zhifeng Bao[2], Quanqing Xu[3], Chuanhui Yang[3], Gao Cong[1]
[1]Nanyang Technological University, [2]RMIT University, [3]OceanBase
{songsong.mo@,zhao0342@e.,gaocong@}ntu.edu.sg,zhifeng.bao@rmit.edu.au,{xuquanqing.xqq,rizhao.ych}@oceanbase.com

ABSTRACT

Parametric Query Optimization (PQO) is crucial for efficiently handling parametrized queries (PQ) in many database applications. This paper addresses two key challenges in existing PQO techniques, focusing on plan set generation and best plan selection. Regarding plan set generation, existing methods rely on modifying sub-plan cardinalities, often resulting in inefficiency and sub-optimal performance due to unclear extents of modifications needed. To overcome this issue, we propose a hybrid plan enumeration algorithm that adeptly adjusts both cardinality and join order. Regarding best plan selection, recent methods rely on machine learning models to choose plans with minimum predicted latency, but they struggle with accurate predictions when parameter bindings vary. Even minor variations in parameters can significantly impact cardinality, affecting plan optimality. To overcome this issue, we propose to utilize a learning-to-rank model, which uses relative rankings as a more reliable performance indicator. Our approach, integrated into PostgreSQL, undergoes extensive experiments on real datasets, showcasing significant improvements in both efficiency and accuracy, as compared to baselines. Specifically, it accelerates the PostgreSQL optimizer by up to 2.57× and surpasses the best existing baseline by up to 1.36×.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/songsong945/RankPQO.

1 INTRODUCTION

Parameterized queries (PQ) are extensively employed in database applications, where the same SQL statements are executed multiple times, each with different parameter bindings. R3W3 A parameterized query uses placeholders for parameters, and the parameter values are supplied at execution time, enhancing efficiency and security by reducing the need for repeated compilations and mitigating SQL injection attacks. In dealing with complex SQL queries, a common approach is the Opt-Always strategy. This method involves optimizing each query instance individually, which can lead to considerable CPU and memory usage. In contrast, many commercial database systems [1, 2] prefer the Opt-Once strategy. This approach optimizes only the initial query instance and reuses this plan for subsequent queries, which can result in sub-optimal performance in later executions. Parametric query optimization (PQO) [5, 7, 8, 11–13, 16–18, 32] offers a balanced solution. It aims to significantly reduce optimization overheads compared to the Opt-Always method while also minimizing the risk of execution sub-optimality associated with the Opt-Once strategy.

In this work, we focus on the query optimization problem within the parameterized query setting for select-project-join-aggregate queries. Adopting a decoupled architecture for candidate plan generation and best plan prediction [11, 32], the PQO problem contains two main tasks: identifying a relatively small number of plans to cache for a parameterized query offline, and efficiently selecting the best cached plan at runtime to execute an instance of the parameterized query. R2W1 R2D1 R3W2 R3D2

To address these tasks, our approach operates in three steps. First, we enumerate a large subset (about a few hundred) of potential plans for a parameterized query. The primary problem here is the expansive search space, making an exhaustive evaluation of each potential plan infeasible. Therefore, we aim to efficiently generate a subset of all possible plans that effectively covers high-quality (i.e., low-latency) plans. Second, from this large subset, we select a relatively small number (about a few dozen) of plans to cache. This step presents two key problems: selecting k candidate plans to cache that maximize coverage of the most optimal plans tailored to the range of user input parameter vectors (a problem known to be NP-complete [10]), and evaluating the quality of each plan without actual execution. Third, at runtime, we select the best cached plan to execute any instance of the parameterized query. The main problem here is evaluating the quality of each plan without actual execution, which is essential for selecting the optimal plan for a parameter vector input from a user. We summarize these problems into two primary challenges.

Challenge 1: Efficiently and efficiently generating a candidate plan set, that both explores the expansive search space in the first step and addresses the k candidate plans coverage problem in the second step. To address Challenge 1, early approaches [12, 32] typically involve invoking traditional query optimizers for different parameter combinations, followed by heuristic methods to select the candidate plan set. More recently, Kepler [11] has introduced a method of varying sub-plan cardinalities to enumerate new plans for generating candidate plan sets. Indeed, altering parameters and adjusting sub-plan cardinalities may change cardinalities, as different parameters also result in different query paths. However, the challenge is that it is difficult to know how much we need to modify cardinality so that we can generate a new plan. This makes the plan generation very inefficient. For instance, in a scenario where the cardinality of a sub-plan is increased tenfold, PostgreSQL might still generate the same plan as before. To address this, we propose a hybrid plan enumeration algorithm that modifies both the cardinality and the join order. Specifically, we first enumerate parameters to change the selectivities of tables and select those that result in distinct query plans, as parameters yielding identical plans are likely to correspond to similar predicate selectivities. For each query instance (i.e., the template bound with each parameter vector), we then enumerate different join orders at the logical level. Finally, R1W2 R1D3 R3W2 we specify the join order and invoke the PostgreSQL optimizer to generate new query plans for this query instance.

Additionally, we improve the quality of the plans generated (where high quality means low latency) by constructing a join graph and employing cardinality to aid in the sampling process for generating join orders. This hybrid approach of modifying both cardinality and join orders not only speeds up plan generation but also helps to generate high-quality plans. For selecting a set of k candidate plans to cache, existing solutions [11, 32] typically employ a greedy approach based on cost or latency. However, these methods require additional time to gather cost or latency data, as shown in Table 5. To address this, we employ a learning-to-rank model to efficiently select a subset of the enumerated plans through a greedy algorithm. R3W2

R3D2 Challenge 2: Evaluating different plans for various parameters without actual execution is a non-trivial task, which is critical to the second problem in the second step and the entire third step in our solution. To address this challenge, early work [5, 7, 12] focuses on designing cost functions for estimation. However, these methods are often constrained by the assumptions of linearity or monotonicity of the cost. With the increasing application of machine learning (ML) in query cost estimation [21], recent studies [11, 32] have shifted towards using ML models for this challenge (see Section 3.2 for more details). Nonetheless, existing models take the parameter vector as the sole input. These approaches assume local continuities, meaning they expect that similar parameters lead to similar plans. However, as Example 1 and Figure 1 demonstrate, even subtle differences in parameters can significantly affect the latency of queries for the same plan, indicating local discontinuities. This suggests that relying solely on parameter similarity may lead to selecting suboptimal plans. To address the problem of local discontinuities, we propose a learning-to-rank model. The rationale is that while absolute cost prediction is challenging due to local discontinuities, relative rankings can still offer a reliable indicator of performance. Furthermore, for PQO, neither candidate plan generation nor best plan prediction requires precise plan cost estimation. Understanding the relative costs of different plans, based on varying query parameters, is sufficient to identify the optimal execution plan. R3W2

In summary, we make the following contributions:

- We propose a novel hybrid candidate plan enumeration algorithm for PQO. This algorithm outperforms existing solutions in terms of both the efficiency of plan enumeration and the quality of the plans generated.
- To our best knowledge, this work is the first of its kind designing a learning-to-rank model specifically for PQO. Utilizing this model, we develop novel approaches for candidate plan selection and optimal plan prediction, addressing two major challenges in PQO.
- The extensive experimentation conducted on real-world datasets attests to the superiority of our approach, RankPQO. Our integrated solution within PostgreSQL not only surpasses the performance of PostgreSQL optimizer by up to 2.57× but also outshines the leading baseline by up to 1.36×.

2 RELATED WORK

Parametric Query Optimization. PQO has been extensively studied [5, 7, 8, 11–13, 16–18, 32]. These studies share common objectives: (1) considering a set of alternative plans for a query template across different ranges of parameter values (i.e., candidate plan generation) and (2) selecting the most appropriate plan based on the actual parameter values at runtime(i.e., best plan selection).

For the candidate plan generation task, previous work [13, 16, 18] has focused on finding an optimal set of plans covering the entire range of selectivities for parameterized predicates, which is an NP-hard task [10]. It is demonstrated in [17] that a small set of plans can be sufficient for achieving near-optimality throughout the selectivity space. Consequently, most proposals [8, 32] have focused on identifying a plan set across the selectivity space through heuristic search, to match the performance of existing optimizers by using them to generate query plans for parameter sets. For the best plan selection task, early approaches, such as Ellipse [7], density-based clustering [5] and SCR [12], are limited by assumptions of linear or monotonic cost functions, leading to poor performance for complex SQL queries [28].

Recently, end-to-end learned solutions [11, 32] have been proposed for parametric query optimization, by utilizing traditional algorithms for generating candidate plans and Machine Learning(ML)-based models for selecting the best plan. For the candidate plan generation task, the query-log-driven solution [32] select minimal cost plans from the query log, which aims to match the performance of the built-in optimizer, but is limited by the performance of the built-in optimizer. Kepler [11] attempts to expand this by heuristically enumerating query plans through cardinality adjustments. However, this strategy faces efficiency challenges due to the uncertainty in the extent of cardinality modification needed. To enhance this, we propose a hybrid method that modifies both cardinality and join orders for more effective plan enumeration. Please refer to the Challenge 1 part of Section 1 for more details.

For the best plan selection task, existing methods [11, 32] typically approach this as a classification or regression problem, using the parameter vector as the sole input, which assumes that similar parameters yield similar plans. However, minor parameter variations can cause significant cardinality changes and need different plans. To address this, we propose a learning-to-rank-based method in Section 3.2 for best plan selection.

ML for Query Optimization. The topic of applying machine learning to query optimization has received a lot of interest [21], including techniques for cardinality estimation [15, 20, 31], end-to-end query optimizer [24, 26, 33, 34, 38], etc. The most relevant to our work is end-to-end query optimizers. Neo [26] and Balsa [33] employ cost estimation models for model-based query plan generation. In contrast, Bao [24], HybridQO [34], and Lero [38] utilize traditional optimizers for generating query plan candidates, supplemented by cost estimation models for plan selection.

Our work differs by focusing on parametric query optimization, aiming at caching good plans for a query template and selecting the optimal plan for specific query parameters during execution. These models are designed to estimate the execution time of a specific query plan. In contrast, our requirement is to predict the execution time of a plan for various user input parameter vectors. Taking Lero as an example, it enumerates a plan set for a specific query and uses one ranking model to compare any two plans within this set to select the best plan. Our challenge, however, involves generating a much larger plan set for a template, selecting a subset of plans to cache, and then selecting a plan based on different user-provided parameter vectors at query time. Consequently, a model that can predict which plan has the minimum cost for varying parameter vectors is necessary, making these existing models for query optimization inapplicable to our problem. Please refer to Section 4.1 for more details.

R3D3 **Learning-to-rank Paradigm.** RankPQO adopts a learning-to-rank paradigm [23], a method widely used for tasks such as document retrieval [19] and query optimization in Lero [38]. The existing learn-to-rank techniques can be categorized into pointwise [37], pairwise [19], and listwise [14] approaches. Notably, our work is the first that applies the learning-to-rank paradigm to the problem of parametric query optimization, and is based on the pairwise approach, which focuses on comparing item pairs to determine which is better.

3 OVERVIEW

We present the problem definition (Section 3.1), discuss the limitation of existing work (Section 3.2), and give an overview of our approach (Section 3.3).

3.1 Problem Definition

Following previous studies [11, 32], our focus lies on parameterized queries. Let Q denote a parametric query (a.k.a. query template) with d parameterized predicates. Correspondingly, a query instance q denotes a query template Q bound to the parameter vector $V = [v_1, ...v_d]$. Let p represent a query plan for Q. Moreover, the actual execution time when employing p to execute q is represented as $Latency(p, q)$. In the following, we define two sub-problems (candidate plan generation and best plan selection) of Parametric Query Optimization (PQO). R3D5

Definition 3.1 (Candidate Plan Generation Problem). Given a workload $W = \{q_1, q_2, \ldots, q_n\}$ for template Q with parameter vectors $Vs = \{V_1, V_2, \ldots, V_n\}$, our goal is to select a k-size subset $P = \{p_1, p_2, \ldots, p_k\}$ from all the possible plans of Q to cache, such that the overall latency of all query instances, $\mathcal{G}(W, P)$, is minimized:

$$\mathcal{G}(W, P) = \sum_{i=1}^{n} \min_{j=1}^{k} Latency(p_i, q_j). \quad (1)$$

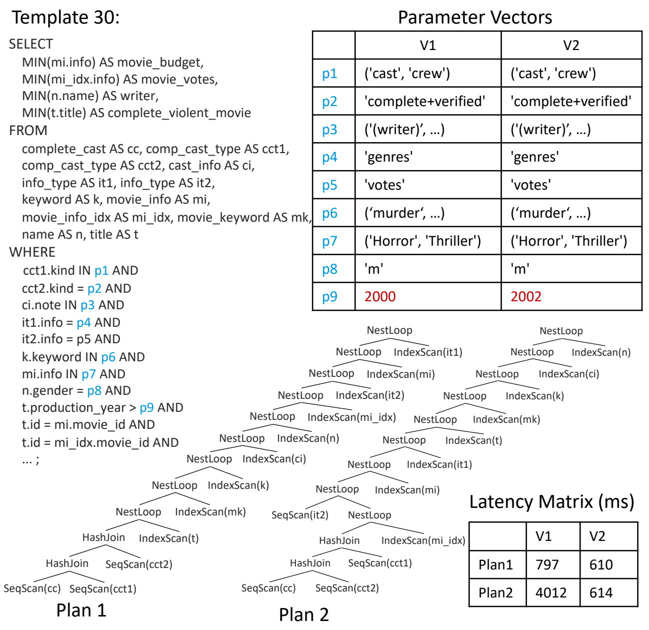
Definition 3.2 (Best Plan Selection Problem). Given a k-size cached plan set P, for a query instance q, we aim to find the optimal plan from P, which can minimize the query latency. This can be formally expressed as:

$$p_i = \arg\min_{p_j \in P} Latency(p_j, q). \quad (2)$$

3.2 Motivation of Our Approach

We proceed to briefly describe query-log-driven solutions [32] (LogPQO) and Kepler [11] for PQO and discuss the limitations of these methods. Both methodologies adopt a pipeline of candidate plan generation followed by best plan prediction. R1D8 R3D6

In addressing Challenge 1, the objective of candidate plan generation is to identify a plan set for caching for a query template. The query-log-driven solution aims to match the performance of the built-in optimizer by selecting plans with minimal cost, which are generated by the built-in optimizer. However, this approach is inherently capped by the performance of the built-in optimizer and


Template 30:

```
SELECT
   MIN(mi.info) AS movie_budget,
   MIN(mi_idx.info) AS movie_votes,
   MIN(n.name) AS writer,
   MIN(t.title) AS complete_violent_movie
FROM
   complete_cast AS cc, comp_cast_type AS cct1,
   comp_cast_type AS cct2, cast_info AS ci,
   info_type AS it1, info_type AS it2,
   keyword AS k, movie_info AS mi,
   movie_info_idx AS mi_idx, movie_keyword AS mk,
   name AS n, title AS t
WHERE
   cct1.kind IN p1 AND
   cct2.kind = p2 AND
   ci.note IN p3 AND
   it1.info = p4 AND
   it2.info = p5 AND
   k.keyword IN p6 AND
   mi.info IN p7 AND
   n.gender = p8 AND
   t.production_year > p9 AND
   t.id = mi.movie_id AND
   t.id = mi_idx.movie_id AND
   ... ;
```

Parameter Vectors

	V1	V2
p1	('cast', 'crew')	('cast', 'crew')
p2	'complete+verified'	'complete+verified'
p3	('(writer)', ...)	('(writer)', ...)
p4	'genres'	'genres'
p5	'votes'	'votes'
p6	('murder', ...)	('murder', ...)
p7	('Horror', 'Thriller')	('Horror', 'Thriller')
p8	'm'	'm'
p9	2000	2002

Latency Matrix (ms)

	V1	V2
Plan1	797	610
Plan2	4012	614

Figure 1: A counter example demonstrating the impact of slight parameter variations on query plan latencies

cannot yield plans that are superior to it (Limitation 1, L1). Additionally, Kepler heuristically enumerates query plans by changing cardinality. This technique encounters two primary issues: Firstly, plan generation lacks efficiency as there is no clear indication of the extent of cardinality modification required to derive a new plan (Limitation 2, L2). Secondly, in an effort to generate new plans, cardinalities might be altered to the extent that they significantly deviate from their true values, resulting in the creation of low-quality (i.e., high-latency) plans (Limitation 3, L3).

To overcome Limitation L1, we employ a novel plan generation method. Specifically, we enhance efficiency by directly generating new query plans through specified join orders for addressing L2. Additionally, we improve the quality of the plans generated by constructing a join graph and employing cardinality to aid in the sampling process for generating join orders for addressing L3. As to be shown in Table 8 of experiments, both strategies contribute to the quality of plan generation.

For the best plan prediction, a key challenge (Challenge 2 in Section 1) lies in evaluating different plans without actual execution. Existing methods typically treat this problem as either a classification or a regression problem. Specifically, it can be formulated as a multi-class classification problem [32], where the given k plans correspond to k classes. The model is trained to partition the feature space based on the best plan id labels. Alternatively, it can be modeled as a regression problem [11, 32] to approximate the estimated cost of the plan for the input features of the query instance. However, these models [11, 32] only utilize the parameter vector as input, adhering to the local consistency assumption that in the parameter space, similar parameter values will yield similar or identical plans (Limitation 4, L4). Yet, in reality, minor differences in parameters can lead to significant changes of cardinality, subsequently resulting in different query plans, as demonstrated in Example 1. In query-log-driven solutions [32], there have been

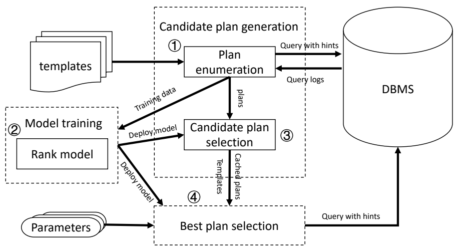


Figure 2: System architecture

considerations to take parameter-related selectivity rates as inputs. However, as demonstrated in Kepler [11], this strategy performs far worse due to the poor cardinality estimates. To overcome L4, we propose to take both query plan and parameter information as input, and employ learning-to-rank techniques for plan selection. This is because, while determining the absolute best plan is challenging due to local discontinuities, relative rankings could still provide a reliable performance indicator.

Example 1. Figure 1 illustrates an instance of query plan generation for template 30 with two similar parameter vectors, V1 and V2, where the minor difference is highlighted in red color. Plan 1 and Plan 2 are the query plans generated by PostgreSQL for V1 and V2, respectively. The latencies for V1 and V2 using Plan 1 and Plan 2 are shown in the latency matrix. If existing PQO cost models are used to estimate the execution times of Plan 1 and Plan 2 for V2, the predicted times are nearly identical for the two plans, which would lead to an arbitrary selection between the two; PostgreSQL chooses Plan 2, which is marginally worse. For a query with parameter V1, the existing cost models predict similar execution times for Plan 1 and Plan 2 for V1. However, their actual execution times differ by a factor of 5. Our learning-to-rank-based model can discern that Plan 1 outperforms Plan 2 for both V1 and V2, demonstrating the advantage of our model over existing cost prediction models in cases where minor parameter variations lead to significant performance differences.

3.3 System Overview

Figure 2 presents an overview of our system (RankPQO), which includes four components: *plan enumeration*, *model building*, *candidate plan selection*, and *best plan selection*. Specifically, the process begins with plan enumeration (1), followed by model training (2), then candidate plan selection (3), and finally best plan selection (4). Next, we describe the functionality of each component. R3D4

Plan enumeration. For a given template, an ideal solution is to enumerate all possible plans and then select k for caching. However, enumerating all plans can be very time-consuming due to the vast search space involved. Therefore, our plan enumeration module introduces a hybrid enumeration algorithm that generates n plans for the subsequent selection stage. Specifically, we efficiently generate plans by simultaneously enumerating both cardinalities and join orders. As to be shown in Table 6, our hybrid enumeration algorithm allows us to generate a greater number of distinct plans per minute. This capability increases the diversity of the plans and provides more comprehensive coverage of the optimal plan set for a template with various parameter bindings. As illustrated in Figure 8, our generated plan set achieves the highest speedup ratio when employing the same method for best plan prediction, evidencing the quality of plans generated by our method.

Model building. We generate training data using a traditional query optimizer. More specifically, we sample a subset of query

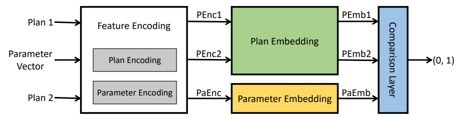


Figure 3: Rank model architecture

plans from the enumerated plans and execute them to answer various query instances derived from a template Q with different parameter vectors. We then collect latency data to serve as our training data. Based on this training data, we construct a pairwise rank model, PRank. This model takes in two plans (denoted as p_1 and p_2) and one parameter vector V_1 as input. It then predicts the relative latency of plans for executing the corresponding query instance q_1 derived from Q and V_1. Specifically, if $Latency(p_1, V_1) < Latency(p_2, V_1)$, the rank model outputs 0; otherwise, it outputs 1.

Candidate plan selection. Using the rank model, we are able to predict the relative latencies for all plans across all query instances in W. With these relative latencies as a basis, and given a budget of k, we devise a greedy-based algorithm for candidate plan selection. This algorithm outputs a set P of k plans. The goal is to minimize $\mathcal{G}(W, P)$, which represents the overall latency of all query instances, by selecting the most efficient plans from P for execution.

Best plan selection. Upon deploying the rank model and caching the k cached plans for each template, we introduce a best plan prediction algorithm for real-time execution. Specifically, whenever a user submits a query instance, our algorithm utilizes the rank model to rank the k cached plans. The algorithm then selects and outputs the optimal plan for executing that particular query instance. The selected plan is converted into a hint plan, specifying the join order and operator type through pg_hint_plan [4], which we refer to as the query with hints. R3D4

4 RANK MODEL

As our ranking model PRank is used in both candidate plan generation and best plan selection, in this section we present the design of PRank, and the training procedure of PRank.

4.1 Model Design

The architecture of our rank model, PRank, is shown in Figure 3. Specifically, it includes the feature extraction and encoding layer (highlighted in grey), the plan embedding layer (highlighted in green), the parameter embedding layer (highlighted in yellow), and the comparison layer (highlighted in blue).

In comparison to existing models proposed for PQO [11, 32], our model diverges in two aspects. Firstly, PRank incorporates both plans and the parameter vector as inputs. This contrasts with existing models, which solely rely on the parameter vector or its corresponding selectivity vector. Secondly, our objective focuses on predicting the relative latency between plans for a given query instance. This comparative approach is notably simpler than either predicting the cost of plans (regression) or identifying which plan offers the fastest execution time (classification), which is used as objectives in existing models. These distinguished designs effectively address Limitation 4. Overall, the learning-to-rank framework we adopt operates by taking a parameter vector and two plans as inputs, and determining which plan has a lower execution latency for this input parameter vector. This method leverages the relative performance of plans rather than relying on absolute cost or execution time predictions. Below, we describe the details of each layer shown in the figure. R3W3

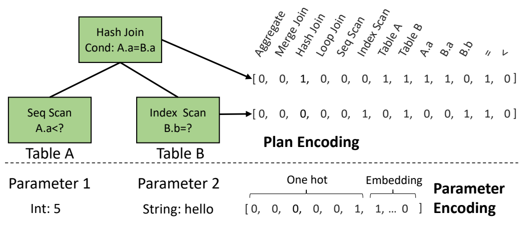


Figure 4: Feature extraction and encoding

Feature extraction and encoding layer. PRank uses two encodings: a plan encoding, which represents the partial execution plan, and a parameter vector encoding, which encodes information regarding the parameters.

As illustrated in the upper half of Figure 4, we represent a plan using a tree structure of vectors. Each sub-plan p (represented by a tree node) features a vector that combines a one-hot encoding reflecting the last operation p, a binary encoding indicating the tables touched by p, and an embedding for the predicates linked with p. For operation information, we aggregate them into six types, i.e., hash join, merge join, loop join, index scan, seq scan, and aggregate. Each type of operation is encoded as a six-dimensional one-hot vector. We use one-hot encoding for table information, where each bit indicates whether the node touches the corresponding table.
R1D2 Next, we encode each predicate by representing the involved operators (e.g., <, >, =, BETWEEN, LIKE, etc.) and columns as one-hot vectors, followed by embedding them into a fixed length using a fully connected neural network. For nodes with multiple predicates, we use average pooling to achieve a consistent embedding size. Notably, for clarity in our visual representations, as seen in Figure 4, we demonstrate one-hot encoding for both the column and comparator. Unlike encoding methods found in other learned query optimizers [24, 38] or in cardinality estimation problem [6, 29], our vector does not incorporate the estimated cost, estimated cardinality, row width or sampling-based embedding. These factors are highly correlated with parameters we cannot know in advance due to user input variability.

As shown in the lower half of Figure 4, for a parameter vector of dimension m, we exclusively utilize the m parameter values as input features. The supported types encompass integer, float, and string. To encode each type, we employ standard techniques: one-hot encoding for integers, normalization to the range (0, 1) for floats, and embeddings for strings.

Plan embedding layer. In the plan embedding layer (PlanEmb) of PRank, plans are mapped from their original feature space into a 32-dimensional embedding space. Based on an early experiment comparing plan representation models (specifically Transformer-based [36], TCNN-based [24, 25], and TreeLSTM-based [30, 35] models), when selecting the same features, the performances among these models are comparable. However, Tree Convolutional Neural Networks [27] (TCNN) offers higher efficiency. To efficiently capture the information from the tree-structured plan, we employ TCNN for this plan embedding. As shown in Figure 5a, the vectorized query plan tree passes three layers of tree convolution. After the final convolution layer, dynamic pooling flattens the tree structure into a singular vector. Subsequently, a fully connected layer translates this pooled vector into a 32-dimensional embedding for the subsequent comparison.

Parameter embedding layer. We employ a lightweight feed-forward neural network (ParamEmb) for parameter embedding for efficiency. As shown in Figure 5b, this network ingests the encoded parameter vector and converts it into a 32-dimensional embedding.

Comparison layer. For a given user input parameter vector, our goal is to determine the better plan between each pair of plans. As shown in Figure 5c, the comparison layer takes the embeddings of the parameter vector (PaEmb) and two plans (PEmb1 and PEmb2) as input. It outputs a binary label to indicate the better plan between the pair. Specifically, we first compute the distances between the embeddings of the plan and the parameter vector: $d_1 = \|\text{PaEmb} - \text{PEmb1}\|, d_2 = \|\text{PaEmb} - \text{PEmb2}\|$. We then compute the difference of the two distances: $d = d_1 - d_2$. Finally, we feed d into a logistic activation function: $\phi(d) = \frac{1}{1+\exp(-d)}$ to generate the final output of PRank. We can interpret which plan is preferable for the parameter vector from this output. Specifically, if $\text{PRank}(V, p_1, p_2) < 0.5$, it implies $Latency(p_1, V) < Latency(p_2, V)$; otherwise, $Latency(p_1, V) \geq Latency(p_2, V)$.

4.2 Model Training

Loss function. To enable our model to accurately predict the superior plan between each pair, we train it using the Binary Cross Entropy (BCE) loss. The loss function is defined as:

$$\text{Loss} = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{3}$$

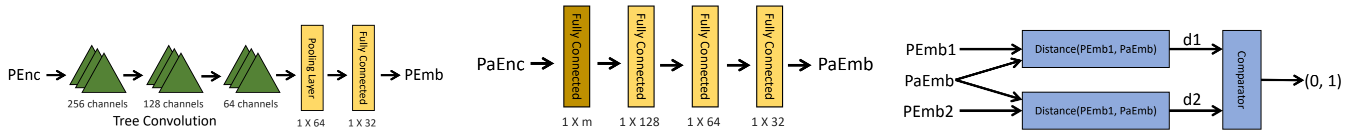
where

- y_i is the ground-truth label indicating the superior plan for the i-th pair in the training data. A value of 1 implies that p_1 is superior, while 0 implies that p_2 is superior.
- $\hat{y}_i$ is the predicted probability by our model that p_1 is superior to p_2 for the i-th pair, computed as $\hat{y}_i = \frac{1}{1+\exp(-d)}$, where d is the difference in distances between the embeddings of the plan and the parameter vector.
- m is the total number of plan pairs in the training data.

By minimizing this loss, we aim to enhance the model's capability to correctly identify the superior plan for a given parameter vector.

Pairwise training. We train PRank using a pairwise comparison framework. For each parameter vector V and its cached candidate plans $\{p_1, \ldots, p_k\}$ discussed in Section 5.2, we generate $k(k-1)$ training data examples based on the loss function presented in Eq. 3. Specifically, for every pair (i, j) where $1 \leq i \neq j \leq n$, we create a data example characterized by the features (V, p_i, p_j). The label is set to 1 if $Latency(p_i, V) \geq Latency(p_j, V)$, and 0 otherwise.

Training pipeline. In model training, our initial approach involves training a separate model for each query template, a strategy we refer to as RankPQO-NS (No Model Share). As indicated in Table 12 and Table 16, this strategy encounters several issues: susceptibility to overfitting, high inference time, and the generation of considerable model size, approximately n MB, where n denotes the number of query templates. To mitigate these issues, we adopt a strategy where one model is shared across all templates, with each template having a distinct input layer for parameter embedding. This approach is imperative because of the distinct parameter vector associated with each template. The training process of this shared model R2W2 R2D3



(a) Plan embedding layer (b) Parameter embedding layer (c) Comparison layer

Figure 5: Embedding and comparison layers

involves sequential training with the training data of each template. We refer to this as a naive shared model strategy. R2W2 R2D3 Nevertheless, as Table 14 illustrates, this method (Steps = 10 in experiments) leads to overfitting for templates trained later and underfitting for those trained earlier. To resolve this issue, we introduce a modification where training data from each template is alternated during the training of the shared model. We refer to this method as RankPQO-S (Shared Model with Alternative Training). As demonstrated in Table 14, RankPQO-S (Steps = 1) achieves the best performance by balancing the training across different templates, effectively mitigating the issues of overfitting and underfitting.

Model re-training. Our current implementation is designed with the assumption that the database setup, the behavior of the optimizer, and the data patterns remain relatively stable. However, in the event of occasional changes, even if infrequent, we need to update our model accordingly. This involves gathering updated data that reflects the altered database configuration and then prioritizing the retraining of our model with this new information.

5 PROPOSED SOLUTION

Building upon the rank model, PRank, designed for plan ranking, this section presents the algorithms used for candidate plan generation and best plan prediction. For candidate plan generation, our goal is to efficiently produce diverse query plans that ideally cover the optimal plans, as outlined in Challenge 1. Subsequently, we evaluate the performance of these plans to identify the best-performing one for a query.

To achieve this, we propose a framework that combines plan enumeration with plan selection for candidate plan generation. More specifically, we present a hybrid plan enumeration algorithm to enumerate multiple distinct plans efficiently by modifying the cardinality estimates and the join orders via certain rules (Section 5.1). R2W2 R2D3 Then we introduce how we collect the training data for model training based on the enumerated plans (Section 5.2). Subsequently, we devise a model-based candidate plan selection algorithm to select plans for caching (Section 5.3). Finally, we present the best plan prediction algorithm that ranks all candidate plans for a query (Section 5.4).

5.1 Plan Enumeration

Given the vast search space of plans, evaluating each potential plan becomes impractical. Thus, we turn to generate a sub-set of plans by manual rules to prune poor plans. Specifically, we introduce a hybrid plan enumeration algorithm that utilizes manual rules to enumerate both the cardinality estimates and the join orders, resulting in the generation of multiple distinct plans.

Cardinality enumeration. Instead of directly modifying the cardinality, as employed in RCE [11], we adopt a strategy of enumerating parameters for an indirect alteration of the cardinality. R1W2 R1D3 R3W3 RCE first generates a query plan using PostgreSQL and then enlarges or reduces the cardinality of sub-plans. Subsequently, it uses hints to

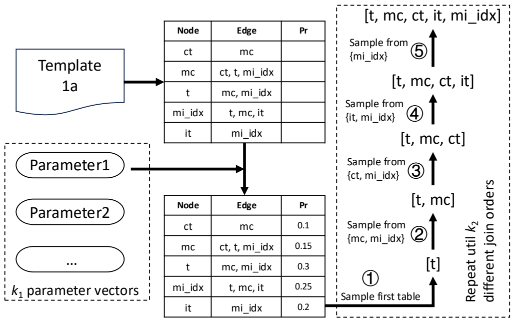


Figure 6: A running example demonstrating cardinality and join order enumeration

specify the adjusted cardinalities and invokes PostgreSQL to generate new plans. In contrast, our approach modifies the parameters of predicates to influence the cardinality of each table, which in turn leads to the generation of new plans by PostgreSQL. Our decision stems from concerns that directly adjusting cardinality for different plans might lead to significant overestimations or underestimations of its true value. Furthermore, solely relying on modifying cardinality to obtain distinct plans is inefficient. This is because the extent of cardinality change needed to yield a distinct plan is uncertain. To address this, we devise a strategy for enumerating join orders, allowing for a more efficient generation of diverse query plans.

Join order enumeration. Unlike the enumeration of cardinalities, specifying different join orders is inherently more efficient in generating distinct plans. This is because varying the join order almost always results in distinct plans, providing a more diversified set of options for optimization. However, simply setting join orders can lead to Cartesian joins, resulting in poor plans (long-running plans). To address this issue, we first construct a join graph based on the query's join conditions. Then, by sampling from the join graph, we can generate join orders that effectively avoid Cartesian joins. Furthermore, building upon our earlier enumeration of cardinalities, we determine the cardinality for each table. During sampling, we prioritize joining tables with smaller cardinalities. These combined strategies effectively alleviate the generation of poor plans. R1D4

The details of our hybrid plan enumeration algorithm are presented in Algorithm 1, and a running example is shown in Figure 6. R1W2 R1D3 R2W1 R2D2 The core idea of our approach is leveraging PostgreSQL to generate plans based on specified selectivities and join orders. Specifically, the first step involves modifying the parameters of predicates to influence the selectivities of each table. Then, for each set of parameters that results in distinct plans, we proceed to enumerate different join orders. This step is based on the understanding that parameters yielding identical plans are likely to correspond to similar predicate selectivities.

Initially, the algorithm uniformly samples k_1 parameter vectors (shown in the left of Figure 6) to change the selectivities of each table, indirectly modifying the cardinality (lines 3-5). For each distinct parameter vector, it generates plan p_0 by the PostgreSQL optimizer (lines 8-9). Subsequently, for every distinct plan p_0, the algorithm enumerates k_2 join orders (shown in the right of Figure 6). For the generation of each join order, the algorithm adopts a probabilistic approach: it initiates by randomly selecting a table (weighed by its sampling probability) as the starting point of the join list (lines 26-28), as shown in step 1 of the running example. In subsequent iterations, the next table to be joined is sampled probabilistically from neighboring nodes that have not been visited, to preclude Cartesian joins (line 35), as illustrated in steps 2, 3, and 5 of the running example. In scenarios where neighboring nodes are either exhausted or already included in the join list, a yet-to-be-included table is randomly picked based on its sampling probability as the next join candidate (line 33), also demonstrated in steps 4 of the running example.

Finally, for each specified join order, the algorithm produces plan p by the conventional query optimizer. All distinct plans are added to the final plan set for output.

Remark. We compare our plan enumeration algorithm with those employed in LogPQO [32] and Kepler [11]. In terms of efficiency, our method achieves higher efficiency because specifying a join order directly leads to a new query plan, whereas modifying cardinalities, the strategy used in both LogPQO and Kepler, may not yield new plans.

Regarding effectiveness, LogPQO adopts the plan outputted by Postgres for each query instance in the training workload. Kepler centers around the plan outputted by Postgres for each query instance in the training workload and attempts to generate similar query plans by perturbing sub-plan cardinalities. However, the enumerated query plans are optimized for the user-submitted query instances. Thus, focusing solely on finding the best query plan for the training workload as a candidate plan might lead to 'overfitting'. In contrast, our approach samples across the entire join order space. This allows us to potentially cover those query plans that perform well across a variety of user-supplied parameter vectors, thereby enhancing overall performance, rather than just covering the optimal query plan for a specific parameter vector.

5.2 Training Data Collection

R1W1
R1D1

After enumerating k_1 parameter vectors Vs and a candidate plan set P (where m=$|P|$) as outlined in Algorithm 1, we proceed to execute each plan over Q bound with each parameter vector, generating a dataset of execution latencies for model training. This produces a dataset of execution latencies for model training. Yet, the generation of these latencies is time-consuming, requiring approximately 48 CPU days to collect the full dataset for each workload.

Taking into account the unique objective of our model, which focuses on predicting the relative magnitudes of latencies and primarily identifying the top-k good query plans, our model inherently exhibits heightened robustness and generalization capability, enabling it to learn effectively from a limited dataset. To further reduce data collection time, we implemented three strategies: 1. Random sampling: We randomly sample parameter vector and plan pairs from the full dataset. For example, in the JOB dataset, enumerating all parameter vectors and plans results in 12,300 training data pairs. However, we only sample 3,000 pairs to collect latency data for training. 2. Multi-core parallelization: We leverage multiple CPU cores to speed up data collection. 3. Timeout mechanism: Since we are focused on finding the top-k good query plans, we use the execution time of PostgreSQL's default plan for a parameter set as a time threshold t. If other plans exceed $3t$, we terminate them and record the execution time as a large value ($10t$). For the JOB dataset, approximately 10.8% of the long-running queries are terminated via this mechanism.

Algorithm 1: Hybrid Plan Enumeration(Q, P_{info}, k_1, k_2)

```
Input: $Q$: an query template, $P_{info}$: the information of parameter
       vector, $k_1$: the number of sampled parameter vectors, $k_2$: the
       number of sampled join orders
Output: $P$: a plan set
1  Initialize parameter vectors $Vs$ as $\emptyset$;
2  Initialize unseen plans $P$ as $\emptyset$;
3  while $|Vs| < k_1$ do
4      $V \leftarrow$ uniformly sampled value for each value in $P_{info}$;
5      $Vs \leftarrow Vs \cup V$;
6  Generate join graph $G$ from $Q$;
7  for each $V \in Vs$ do
8      Query instance $q \leftarrow Q$ bind with $V$;
9      $p_0 \leftarrow$ getOptimizerPlan($q$);
10     if $p_0 \notin P$ then
11         $P \leftarrow P \cup p_0$;
12         Get cardinalities $Cards$ for each table from $p_0$;
13         $Os \leftarrow$ SampleJoinOrders($G$, $Cards$, $k_2$);
14         for each join order $O \in Os$ do
15             $p \leftarrow$ getOptimizerPlan($q$, $O$);
16             $P \leftarrow P \cup p$;
17 Return $P$;
18 ------------------------------------------------------------
   Function SampleJoinOrders($G$, $Cards$, $k_2$):
19     --------------------------------------------------------
20     Initialize unseen join orders $Os$ as $\emptyset$;
21     Tables $T \leftarrow$ all nodes of $G$;
22     $Pr \leftarrow$ getSelectionProb($T$, $Cards$);
23     while $|Os| \leq k_2$ do
24         Initialize $visited$ as $\emptyset$;
25         Initialize join order $O$ as empty list;
26         $nextNode \leftarrow$ getSampleNode($T \backslash visited$, $Pr$);
27         $O$.append($nextNode$);
28         $visited \leftarrow visited \cup nextNode$;
29         while $|O| < |T|$ do
30             $t \leftarrow O$.getback();
31             $neighbors \leftarrow$ getNeighbors($t$) $\backslash visited$;
32             if $neighbors$ is empty then
33                 $nextNode \leftarrow$ getSampleNode($T \backslash visited$, $Pr$);
34             else
35                 $nextNode \leftarrow$ getSampleNode( $neighbors$, $Pr$);
36             $O$.append($nextNode$);
37             $visited \leftarrow visited \cup nextNode$;
38         $Os \leftarrow Os \cup O$;
39     return $Os$;
```

Algorithm 2: Candidate Plan Selection(PRank, Vs, P, k)

```
Input: $Vs$: the parameter vectors, $P$: a plan set
Output: $P'$: a plan set for caching
1  ParamEmbs ← PRank.ParamEmb($Vs$);
2  PlanEmbs ← PRank.PlanEmb($P$);
3  for i, ParamEmb ∈ ParamEmbs do
4      for j, PlanEmb ∈ PlanEmbs do
5          DistanceMatrix[$i$][$j$] ← ||PlanEmb − ParamEmb||
6  Initialize selected plans $P'$ as $\emptyset$;
7  $PlanIds$ ← set(range(len($P$)));
8  while $|P'| < k$ do
9      for $p \in PlanIds$ do
10         Dis[p] ← getDistance($P' \cup p$,Vs, $DistanceMatrix$);
11     $p \leftarrow \arg\min_{p \in PlanIds} (Dis[p])$;
12     $P' \leftarrow P' \cup p$;
13     $PlanIds \leftarrow PlanIds \backslash p$;
14 Return $P'$;
```

Algorithm 3: Best Plan Predication(PRank, V, P)

```
Input: $V$: the parameter vector, $P$: a cached plan set
Output: $p$: the best plan for $V$
1  ParamEmb ← PRank.ParamEmb($V$);
2  PlanEmbs ← PRank.PlanEmb($P$);
3  for i, PlanEmb ∈ PlanEmbs do
4      Distances[$i$] ← ||PlanEmb − ParamEmb||
5  $p \leftarrow \arg\min(Distances[p])$;
6  Return $p$;
```

5.3 Candidate Plan Selection

As emphasized in Challenge 1, the vast search space and the identification of the k optimal plans are primary challenges for candidate plan generation. Initially, our approach employs the hybrid plan enumeration algorithm to efficiently identify m viable plans within this expansive search space. Subsequently, our goal is to select a subset of plans constrained by the plan number budget k, aiming to minimize $\mathcal{G}(W, P)$. However, this k-set identification problem is NP-complete [10]. Considering this computational challenge, we adopt a simple greedy approach. Notably, the key distinction between our method and populateCache [32] is our use of a rank model to guide the greedy search iterations.

Algorithm 2 presents the pseudo-code for candidate plan selection. Initially, we utilize the PRank model to compute the embeddings for both parameters and plans. Then, we compute their similarity, denoted as a DistanceMatrix of dimensions |parameters| x |plans| (lines 1-5). Subsequently, based on the DistanceMatrix, we greedily select a plan (lines 8-13). Specifically, in each iteration, for each the candidate plan $p \in PlanIds$, we use the **getDistance** function to calculate the sum of the minimum distances from all parameters to the current selected plan set $P' \cup p$. Notably, we use the sum of these minimum distances as an approximation for $\mathcal{G}(W, P)$. This is because, for a given set of parameters, the model tends to embed plans with shorter execution times at closer distances. The plan with the smallest summed distance is then added to P'. Ultimately, P' serves as the final result.

5.4 Best Plan Prediction

We proceed to present the best plan prediction algorithm, which aims to identify the optimal plan from the set of cached plans for a query with a parameter vector during runtime. Utilizing our proposed rank model, we embed both the input parameter vector and the cached plans. The plan with the closest embedding distance to the input parameter vector is then identified as the optimal plan. Details of this approach are presented in Algorithm 3.

6 EXPERIMENTS

We evaluate the performance of RankPQO in comparison to baselines regarding execution latency speedups on parameterized query workloads. Our main findings are summarized as follows:

- An end-to-end implementation of RankPQO on PostgreSQL significantly outperforms the built-in optimizer and the state-of-the-art solutions for PQ. (Section 6.2)
- Our candidate plan generation algorithm efficiently and effectively identifies markedly superior plans compared to existing candidate plan generation baselines. (Section 6.3)
- The Learning to Rank model demonstrates better performance over regression models for the PQO problem. (Section 6.4)

6.1 Experimental Setup

Datasets. Four datasets are used in our experiments:

- **Join Order Benchmark (JOB)**. This dataset, designed by Leis et al. [22], comprises 33 parameterized queries using the Internet Movie Database (IMDB). For parameter value generation, we synthetically generate 200 parameter vectors for each parameterized query by uniformly sampling rows from the result set of a derived query, which selects column values for each parameterized predicate.
- **TPCH**. The TPCH benchmark is designed to test the response time of database systems to complex queries. It includes eight tables with the scale factor set to 10 in our experiments and encompasses 22 parameterized queries. 200 parameter vectors for each parameterized query are generated following the official TPCH specification [3].
- **Stack** [11]. This database consists of real-world StackExchange data. It includes 42 queries from the original benchmark and 45 manually-written query templates. We source parameter values from Kepler [11].
- **DSB** [9]. DSB enhances the TPC-DS benchmark with complex data distribution and challenging query templates. We choose 15 SPJ queries [1] from the original benchmark. The scale factor is set to 100, and 5 parameter vectors are generated for each parameterized query.

For each parameterized query, we randomly choose 80% of the parameter vectors as the training set and use the rest 20% as the test set. We use JOB as the default dataset unless specified otherwise.

Baselines. We compare RankPQO against three key baselines: PostgreSQL, and two state-of-the-art solutions, LogPQO [32] and Kepler [11]. For our solution, we use RankPQO-S and RankPQO-NS

[1]They include q13, q18, q19, q25, q27, q40, q50, q72, q84, q85, q91, q99, q100, q101, q102.

to represent the shared rank model and non-shared rank model, respectively. When referring to both variants collectively, we simply use RankPQO. In some tables, we use -S and -NS to denote RankPQO-S and RankPQO-NS for short. Additionally, we compare RankPQO with the learned query optimizer, Lero [38]. Lero is used in an Opt-Always fashion, i.e., it optimizes every query instance independently. Since Lero has been demonstrated to surpass other learned query optimizers [38], we exclude such baselines from comparison in our experiments. R1D7

Metrics and default parameter setting. To evaluate the performance of different methods, we employ two key metrics: running time and speedup ratio. The speedup ratio, or simply speedup, is calculated as the runtime of the PostgreSQL divided by the runtimes of compared solutions. Additionally, we evaluate the accuracy of models, which is defined as the correctness rate of successfully ranking the pairwise plans for each query. For model training, we train all the tasks using the Adam optimizer with a learning rate of 0.001. By default, we have configured our system with the following settings: the number of cached candidate plans is set to 30; the training epoch count is 10; the pairwise training data size is fixed at 3000; for Algorithm 1, we sample k_1=200 parameter vectors and k_2=50 join orders; the embedding dimension for columns, comparators, and string parameters is set to 32; and the alternation step for RankPQO-S is set to 1. For the parameters in baselines, we opted for the default values as specified in their publications. Each result presented is the median of 5 runs.

Setup. All experiments are conducted on a machine equipped with an Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz (20 physical cores), 128GB RAM, and an NVIDIA GeForce RTX 3080 GPU with 10GB of memory. PostgreSQL version is 12.5.

6.2 Overall Performance

We present the evaluation of the overall performance for RankPQO-S, RankPQO-NS, LogPQO, Kepler, Lero, and PostgreSQL.

R1D6 **Varying dataset.** Table 1 depicts the speedup ratios of two variants of RankPQO compared to PostgreSQL, LogPQO, Kepler, Lero, True Cardinality, and Fastest Found Plan across various datasets. True Cardinality refers to PostgreSQL generating a query plan with true cardinalities, while Fastest Found Plan refers to the fastest plan generated by Hybrid Plan Enumeration for each query. We observe the following: (1) Fastest Found Plan performs the best, which demonstrates that Hybrid Plan Enumeration can identify good plans. RankPQO does not match the performance of Fastest Found Plan because we select top-k query plans to serve different parameter combinations. (2) RankPQO-S and RankPQO-NS consistently deliver improved speedup ratios in comparison to the baseline methods, except for Fastest Found Plan. Specifically, RankPQO-NS reduces the plan execution time of PostgreSQL by 61.1% and Kepler by 26.4% on the JOB dataset. This enhanced performance can be attributed to two key factors: firstly, our hybrid enumeration algorithm, which generates a larger pool of high-quality plans (Please refer to Table 5 and Table 7), and secondly, the employment of a learning-to-rank model that serves as a more reliable indicator for plan selection (Please refer to Figure 11). (3) The efficacy of RankPQO is influenced by the unique complexities of each dataset. Specifically, JOB has a higher average number of joins per query compared to TPCH, making it more complex and resulting in a larger optimization space.

Table 1: Overall performance on all datasets.

	JOB	Stack	TPCH	DSB
LogPQO	1.67	1.47	1.11	1.69
Kepler	1.89	2.03	1.18	1.81
Lero	2.01	1.92	1.18	2.04
RankPQO-S	2.52	**2.41**	**1.19**	**2.3**
RankPQO-NS	**2.57**	2.38	1.18	2.11
True Cardinality	2.12	1.35	1.16	1.26
Fastest Found Plan	**3.45**	**3.11**	**1.29**	**2.92**

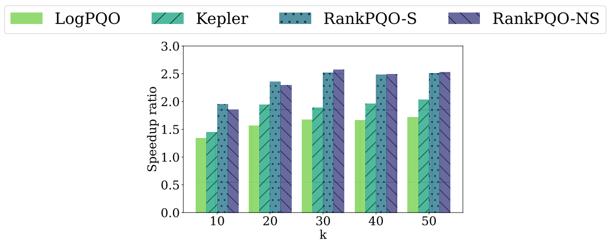


Figure 7: Overall performance by varying k

Varying the number of cached plans k. Figure 7 shows the variation of performance with the number of cached plans k for RankPQO-S and RankPQO-NS alongside other methods. As k increases, the performances for all methods improve, which can be attributed to the broader selection of cached plans for enhancing optimization flexibility. However, the trend also intimates the onset of diminishing returns when k is increased further. This can be ascribed to two factors: (1) the increased selection complexity that elevates the time required to choose the optimal plan; (2) the saturation of optimal plans within the cache, meaning that additional plans contribute marginally to performance improvements. Consequently, we have selected $k = 30$ as the default value for subsequent experiments, a trade-off between the benefits of caching and the costs to optimize performance.

Per template performance. To provide additional details, Table 2 and Table 3 show the average execution time for individual query templates with different parameters in JOB and DSB, which are the smallest and largest datasets, respectively, among the four datasets used in our experiments. Table 2 shows half of the query templates in JOB due to the limited space. Several observations can be made based on these results. First, for queries with short execution time, PostgreSQL is more efficient due to the overhead from additional optimizations of other methods. In templates such as q2, q4, and q16 in JOB, and q100 in DSB, the queries are simple. The optimization processes, which involve selecting plans from a cache, tend to add unnecessary overhead for these simple queries. Second, in contrast, for more expensive queries, RankPQO demonstrates superior performance over all baseline methods, particularly for templates q24 and q30 in JOB, and q25 and q101 in DSB. The learning-to-rank mechanism of RankPQO is able to identify the most efficient execution plans effectively for these complex queries. On average, R1D7 R2W3 R2D5

RankPQO-S achieves the best results, demonstrating the learning-to-rank model acting as a reliable indicator for plan selection.

Table 2: Per template performance over JOB (seconds)

Template	PG	LogPQO	Kelper	-S	-NS	Lero
q2	**0.016**	0.023	0.073	0.028	0.029	0.066
q4	**0.014**	0.023	0.4	0.018	0.017	0.36
q6	0.2	0.21	0.13	0.19	0.19	**0.12**
q8	0.19	0.13	0.16	0.064	**0.062**	0.14
q10	3.73	2.61	2.28	2.11	**2.07**	**2.07**
q12	0.16	0.16	**0.098**	0.18	0.17	0.88
q14	1	1.02	0.77	1.03	**0.31**	0.701
q16	**0.016**	0.023	0.024	0.025	0.028	0.022
q18	0.42	**0.41**	0.45	0.44	0.54	**0.41**
q20	3.56	1.82	1.63	0.82	**0.51**	1.48
q22	0.27	0.27	0.26	0.34	0.3	**0.23**
q24	1.5	0.74	0.82	**0.58**	**0.58**	0.74
q26	0.8	**0.48**	0.55	0.5	0.5	0.5
q28	0.41	**0.21**	0.27	0.23	0.22	0.25
q30	4.33	1.54	0.92	**0.71**	0.77	0.83
q32	**0.0093**	0.014	0.015	0.013	0.012	0.013
average	1.039	0.605	0.553	0.454	**0.394**	0.502

Table 3: Per template performance over DSB (seconds)

Template	PG	LogPQO	Kelper	-S	-NS	Lero
q13	120.6	235	225	**118**	161	185
q18	31.6	74.1	**24.5**	53.7	48.2	62.3
q19	46.6	46.3	47.1	**45.6**	46	46.2
q25	4368	3012	2934	**2367**	2725	2867
q27	0.016	0.012	0.013	0.25	**0.0085**	0.56
q40	1.66	1.82	**1.54**	8.47	1.77	2.31
q50	208	127	108	**91.3**	95.7	105
q72	302	298	342	608	**28.8**	213
q84	**4.72**	5.67	6.34	5.27	4.81	5.34
q85	522	325	286	**63.9**	340	290
q91	5.64	5.63	4.28	**3.74**	5.49	5.67
q99	524	274	178	**136**	**136**	145
q100	**54.2**	325	104	274	305	189
q101	2724	491	609	**57.4**	302	232
q102	102.8	94.2	106	86.4	**70.8**	83
average	601	354	331	**261**	284	295

Comparison with Lero. This experiment is to compare RankPQO with Lero, a learning-to-rank-based optimizer, over JOB and DSB, which are the smallest and largest datasets, respectively, among the four datasets used in our experiments. Table 4 shows the planning and execution times of PG, Lero, RankPQO-S, and RankPQO-NS. We have the following two findings: First, PG has the shortest planning time, while Lero has the longest. This is because, unlike PG, both Lero and RankPQO spend additional time optimizing the execution plan. RankPQO has a shorter planning time than Lero because it uses cached plans instead of enumerating plans during execution, which reduces the overall planning time. Second, RankPQO has the shortest execution time. The main advantage over Lero is that RankPQO can explore a much larger candidate pool, thereby increasing the probability of finding better query plans. Specifically, RankPQO first enumerates hundreds of query plans and then selects dozens to cache as candidates offline. In contrast, Lero only enumerates a few plans for each query online, which limits the potential for finding the optimal plan. Overall, RankPQO-S outperforms Lero in terms of total time, with an improvement of 17.3% for JOB and 11.4% for DSB. R2W3 R2D5

Table 4: Comparison with Lero (milliseconds)

		Planning	Execution	Total
JOB	PG	12	745	757
	Lero	59	318	377
	RankPQO-S	21	291	312
	RankPQO-NS	20	285	305
DSB	PG	4.2	601,055	601,059.2
	Lero	54	294,705	294,759
	RankPQO-S	28	261,268	261,296
	RankPQO-NS	30	284,675	284,705

6.3 Analyzing candidate plan generation

Efficiency. The efficiency of generating candidate plans is assessed in three stages: plan enumeration, plan costing, and plan selection. LogPQO employs PostgreSQL for plan enumeration to generate plans across different parameter sets for each query template. Specifically, it uses the explain command to generate a query plan for each set of parameter vectors, followed by a deduplication step to remove duplicate plans. R3D8 Cost estimation is achieved through "what-if" analysis, and plan selection utilizes a greedy algorithm for caching decisions. Kepler employs the Row Count Evolution method for plan enumeration and collects plan latency by 20 sampling parameter sets for each query template. Notably, while plan costing in Kepler is typically an online process, we have adapted it for offline analysis in this experiment. Plan selection in Kepler is executed using the plan cover pruning method. RankPQO engages Algorithm 1 for plan enumeration. The system avoids a distinct costing phase by integrating the rank model directly into the plan selection process, which is shown in Algorithm 2.

Table 5 shows the time used for plan enumeration and plan costing of the three methods. We observe that the time taken for plan enumeration increases from LogPQO to Kepler and further to RankPQO. This trend can be attributed to the progressively expanding enumeration space, resulting in a higher diversity of plans. Specifically, LogPQO depends solely on the optimizer of PostgreSQL, Kepler modifies sub-plan cardinalities, and RankPQO ajusts both cardinalities and join orders. Although this expansion leads to increased enumeration time, it also contributes to enhanced end-to-end performance (as discussed in Section 6.2). As mentioned earlier, the time for plan costing is collecting the cost or latency for selecting plans to cache. RankPQO avoids a distinct costing phase by integrating the rank model directly.

Table 6 provides a comparison of plan generation across three solutions. It presents the number and speed of distinct plans generated by different methods. Here, speed means the number of distinct plans generated per second by each method. R3D7 The results show that

Table 5: Efficency of plan generation and costing (seconds)

	LogPQO	Kepler	RankPQO
Plan enumeration	87	127	382
Plan costing	3331	7429	NA

Table 6: Efficency of distinct plan generation

	LogPQO	Kepler	RankPQO
Number of distinct plans	367	421	2547
Speed (number/second)	4.22	3.31	6.67

Table 7: Efficency of candidate plan selection (seconds)

k	LogPQO	Kepler	RankPQO-S	RankPQO-NS
10	6.36	2.38	149	152
20	19.76	7.41	240	245
30	39.98	14.99	367	362
40	66.29	24.85	527	526
50	97.64	36.61	712	716

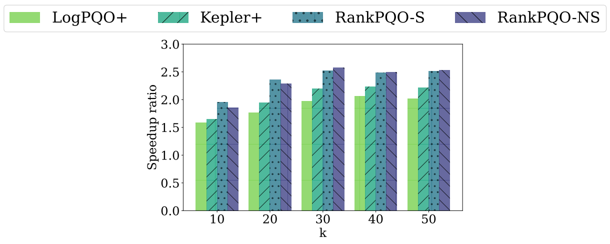


Figure 8: Effectiveness of candidate plan generation

RankPQO outperforms all baselines in both the total number of distinct plans generated and the speed of generation, demonstrating its superior efficiency in generating distinct plans. This result also corroborates our earlier discussion regarding the inefficiency of generating distinct plans by merely modifying cardinalities.

Table 7 shows the time for plan selection under different k values. We observe that, during the plan selection phase, RankPQO consumes the most time since it does not need the distinct plan costing phase. Instead, it invokes the rank model directly during plan selection to serve as the cost indicator. When $k = 30$, the total time of generating candidate plans for LogPQO, Kelper, RankPQO is about 3458s, 7571s, and 749s, respectively. Overall, the approach of RankPQO in generating candidate plans is the most efficient because it circumvents the time-intensive plan costing phase. Note that the process of generating candidate plans is done offline.

Effectiveness. To compare the effectiveness of candidate plan generation between our solutions and the baselines, we incorporate our learning-to-rank model into the best plan selection components of Kepler and LogPQO, resulting in the variants Kepler+ and LogPQO+. Figure 8 depicts the speedup ratio for each method as the number of candidate plans k changes. We observe that both RankPQO-S and RankPQO-NS consistently surpass the baselines, showing the effectiveness of our candidate plan generation approach.

Ablation study of hybrid plan enumeration. Our hybrid plan enumeration method involves enumerating both cardinalities and join orders to generate query plans. Removing the enumeration of join orders aligns our method with LogPQO+, and the comparison results are already in Figure 8. Here, we focus on evaluating two optimization strategies for enumerating join orders: constructing a join graph and sampling from this graph based on the cardinalities of each table. Specifically, we denote the method of random sampling on the join graph without cardinality-based sampling as Method -C; and the method that eliminates the join graph entirely, resulting in random sampling across all tables, as Method -G.

Table 8 presents a comparison of execution time for a subset of queries. Here, RankPQO-S has the best performance, demonstrating the usefulness of both strategies to the quality of plan generation. When comparing the two strategies, removing cardinality-based sampling increases the average execution time by 0.128 seconds (from 0.312 to 0.44 seconds), and eliminating the join graph altogether increases it by 0.179 seconds (from 0.44 to 0.619 seconds). This suggests that constructing a join graph plays a more significant role in enhancing the quality of generated plans.

Table 8: Ablation study of hybrid plan enumeration (seconds)

Template	RankPQO-S	-C	-G
q2	0.028	0.038	0.07
q4	0.018	0.1	0.045
q16	0.025	0.019	0.12
q24	0.58	0.71	0.93
q29	0.53	0.87	1.19
q30	0.71	0.9	1.36
Average	0.312	0.44	0.619

Table 9: Time (hours) of collecting training data with varying data sizes (All = entire dataset)

Data size	1k	2k	3k	4k	5k	All
No time out	7.14	14.56	19.72	24.24	28.31	89.42
Time out	1.66	2.88	3.99	5.07	5.97	22.59

6.4 Analyzing ML models

R1W1
R1D1

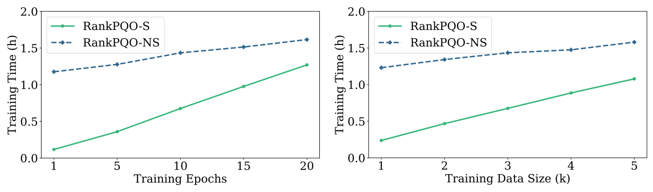
Time of collecting training data.. Table 9 shows the time required to collect training data with varying data sizes. Both the sampling strategy and the timeout mechanism significantly reduce data collection time, especially when collecting the entire dataset. The sampling strategy limits the number of parameter vectors and plan pairs executed, further speeding up the process. Table 10 demonstrates the impact of using different numbers of processes on data collection time. We find that increasing the number of processes effectively reduces the data collection time, though the improvement diminishes as the number of processes increases. This is because we assign all queries for a given template and its parameters to the same process, so the total execution time depends on the slowest process. Table 11 compares the effectiveness of different data collection strategies across methods. The sampling strategy, multi-core parallelization, and timeout mechanism improve data collection efficiency without affecting the end-to-end performance.

Table 10: Time (hours) of collecting training data with varying number of processes

Number of processes	6	12	18
Time	7.47	3.99	2.87

Table 11: Effectiveness for different training data collecting strategies

	LogPQO	Kepler	-S	-NS
No time out + 12	1.67	1.89	2.52	2.57
Time out + 6	1.67	1.91	2.54	2.51
Time out + 12	1.67	1.93	2.53	2.55
Time out + 18	1.67	1.91	2.53	2.53



(a) Varying training epochs (b) Varying training data size

Figure 9: Learning efficiency of rank models

Table 12: Effectiveness for varying training epochs

Training Epochs	RankPQO-S		RankPQO-NS	
	Accuracy	Speedup	Accuracy	Speedup
1	0.59	1.21	0.77	1.34
5	0.68	2.36	0.83	2.34
10	0.79	2.52	0.86	2.57
15	0.81	2.42	0.88	2.41
20	0.83	2.59	0.89	2.26

Table 13: Effectiveness for varying training data size

Training Data Size (k)	RankPQO-S		RankPQO-NS	
	Accuracy	Speedup	Accuracy	Speedup
1	0.79	2.00	0.89	2.22
2	0.78	2.42	0.86	2.44
3	0.79	2.52	0.86	2.57
4	0.78	2.54	0.85	2.48
5	0.77	2.55	0.85	2.47

Varying training epochs. As shown in Figure 9a, we vary the number of training epochs to demonstrate the learning time efficiency of our models. RankPQO-NS exhibits a higher training time demand, which is attributed to the necessity of creating and initializing multiple models. As the number of epochs increases, the training time for RankPQO-S rises significantly. This is because RankPQO-S undergoes more frequent switches between training on different query template data (for further details, please see the experiment about varying alternation steps).

The effectiveness of our models with varying epochs is shown in Table 12. At one training epoch, RankPQO-NS has better accuracy and speedup ratio compared to RankPQO-S. This is because RankPQO-S, being a single model for all query templates, requires a larger amount of training to generalize across different patterns. As training epochs increase, the performance of RankPQO-S initially improves and then stabilizes. However, for RankPQO-NS, although the accuracy continues to improve as training epochs increase, the speedup saturates and eventually drops. The higher accuracy of RankPQO-NS can be attributed to this specific training, and the decrease in overall performance can be attributed to the overfitting problem from repeated training on their respective query template This also illustrates that accuracy should not be the sole performance indicator for the task, and the shared model strategy employed by RankPQO-S can effectively prevent overfitting, a critical advantage in maintaining model robustness over multiple training iterations.

Varying training data size. We vary the size of the training data pairs, where k = 1000, and each pair consists of two plans and R3D9 one parameter vector to evaluate the learning data efficiency of our models. Figure 9b demonstrates how the training time of our methods changes with varying data sizes. Similar observations as those made when varying the training epochs can be made here: as the training data size increases, the training time for RankPQO-S rises more sharply. This trend is also related to the process of training a single model with data from different query templates. Increasing the number of training epochs augments the number of switches between different query templates, while increasing the training data size elevates the cost of loading and transferring data for each switch. This reflects the complexity of managing larger datasets within a unified model and underscores the trade-offs between data volume and training efficiency.

Table 13 shows the effectiveness of RankPQO-S and RankPQO-NS as the size of the training data increases. The consistent accuracy with varying data sizes indicates that both models have learned the essential patterns for distinguishing between two plans, and adding more data does not significantly enhance this capability. With the increase in data size, both models initially show an improvement in speedup ratio, suggesting they become better at generating embeddings that accurately capture execution times. This leads to a more precise selection of plans with minimal latency, thus resulting in a higher speedup ratio. However, beyond a certain point, the models have sufficiently refined their embeddings for efficient plan selection, and additional data does not contribute to further improvement in this process.

Varying alternation steps. Alternation steps refer to the number of epochs during which RankPQO-S trains on the training data of the same template before switching to the training data of the next template. It is intuitive that switching training data incurs additional overhead time, particularly in terms of reading and writing data. Thus, as we decrease the number of steps, leading to more frequent switches, the training time increases. This relationship is confirmed by the experimental results shown in Figure 10, where a lower number of steps corresponds to a longer training time.

Table 15: Accuracy of per template

Sequence	Template	step = 10	step = 5	step = 2	step = 1
1	q31	0.58	0.70	0.72	0.80
5	q14	0.48	0.62	0.74	0.79
10	q6	0.63	0.72	0.77	0.83
15	q12	0.59	0.68	0.70	0.79
20	q20	0.68	0.77	0.76	0.83
25	q10	0.61	0.68	0.71	0.77
30	q16	0.72	0.76	0.73	0.80
33	q5	0.77	0.78	0.75	0.80

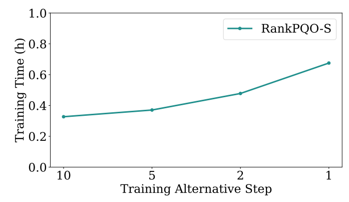


Figure 10: Efficiency for varying alternation steps

Table 14: Effectiveness for varying alternation steps

Steps	RankPQO-S	
	Accuracy	Speedup
10	0.61	1.00
5	0.69	1.85
2	0.72	2.33
1	0.79	2.52

However, as demonstrated by Table 14, both accuracy and speedup ratio improve as the number of alternation steps decreases. This improvement can be attributed to the ability of the model to retain information from one template to the next more effectively when the switch occurs more frequently.

Table 15 displays the accuracy for templates at different positions within the training sequence, offering a detailed view of how varying the frequency of alternation between templates impacts learning. In the first column, where there is no alternation (step size equal to the total number of epochs), we see a trend: the earlier a template is in the training sequence, the lower its accuracy tends to be. This trend supports the hypothesis that the model may "forget" previously learned patterns when not revisited frequently, as the training focuses on subsequent templates. In contrast, the last column, which represents the highest alternation frequency (step size of 1, meaning constant alternation), shows higher and more consistent accuracy across all templates. This consistency suggests that frequent alternation mitigates the "forgetting" of patterns, enabling the model to maintain a more uniform understanding across the different templates, thereby enhancing overall performance.

Comparison with baselines. Table 16 shows the training time. RankPQO takes the most time, which can be attributed to two factors: (1) unlike the baselines that only take parameter vectors as input, RankPQO takes both parameter vectors and the plan as input; (2) RankPQO employs a more complex model to encode and predict relative costs, whereas baselines utilize simpler methods such as XGBoost or 3-layer feedforward neural networks.

Table 16 shows the model sizes of all solutions. RankPQO-S has the minimal model size. This is because RankPQO-S optimizes the total number of models by utilizing a shared model across all query templates, thus minimizing the model size. For instance, with m query templates and k cached plans per template, LogPQO would train mk separate models; Kepler would have m models plus mk output heads; RankPQO-NS would have m models, one for each

Table 16: Efficency of model training and model size

	LogPQO	Kepler	-S	-NS
Training time (s)	22.8	484	2431	5164
Model size (MB)	38	6.8	5.9	34

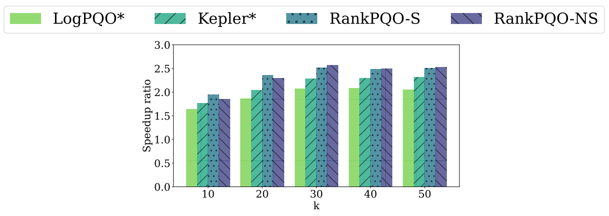


Figure 11: Effectiveness of different models

query template. In contrast, RankPQO-S employs a single shared model with distinct input layers for each query template to handle varying parameter dimensions, resulting in one model and m input layers. As indicated by Table 16, the size of the RankPQO-S model is on par with that of Kepler. With an increasing number of query templates, the size efficiency of our model becomes more evident. The shared model approach in RankPQO-S not only optimizes for model size but also scales effectively with the growing number of query templates, a crucial feature for dynamic and varied query workloads.

To benchmark the best plan selection capabilities across all methods, we standardize the candidate plan set outputted by the hybrid plan enumeration for each method and utilize the best plan selection components from each, denoted as Kepler* and LogPQO*. Figure 11 illustrates the speedup ratio trends of each method as the number of candidate plans k varies. It clearly shows that RankPQO-S and RankPQO-NS consistently outperform the enhanced baseline methods, demonstrating the strength of our approach in selecting the most efficient execution plans. Compared to Kepler* and LogPQO* with Kepler+ and LogPQO+ shown in Figure 8, Kepler* and LogPQO* outperform them, indicating that the hybrid plan enumeration significantly contributes to their effectiveness.

Ablation study of rank model. We conduct two ablation studies to investigate the impact of different components of our rank model, namely No Rank and Plan Distance. Here, we also standardize the candidate plan set output by the hybrid plan enumeration. The No Rank variant removes the relative ranking mechanism, instead of directly using the plan and parameter embeddings to estimate the cost without comparing them. The Plan Distance variant concatenates the parameter encodings to each node of the plan encodings and then obtains the plan embedding through the embedding module. Finally, it computes the distance between the plan embeddings, rather than calculating the distance between the parameter and plan embeddings as in the original model. R2W3 R2D4 R3W1

As shown in Table 17, RankPQO-S achieves the best speedup ratio, confirming that our rank model is the most effective. However, the No Rank variant also performs well, surpassing both Kepler and Kepler* variants, demonstrating that using the plan as an input improves performance. On the other hand, Plan Distance shows more modest results, performing similarly to Lero, despite using a larger candidate plan set. This is because Plan Distance concatenates the

Table 17: Ablation study of rank model

	RankPQO-S	No Rank	Plan Distance
Speedup ratio	2.52	2.41	2.14

parameter encoding to the plan encoding at every node; however, in reality, the parameter primarily influences the selectivity of the leaf nodes, and its impact on internal nodes depends on factors such as the plan structure.

7 CONCLUSION

In this study, we have introduced RankPQO, a novel approach to Parametric Query Optimization (PQO) that specifically addresses the inefficiencies in plan set generation and best plan selection for parametrized queries. The cornerstone of RankPQO is a hybrid plan enumeration algorithm that adeptly adjusts sub-plan cardinalities and join orders to navigate the challenges of traditional PQO techniques. Complementing this, we have proposed a learning-to-rank model to provide a more reliable best plan selection, especially in scenarios where minor parameter variations can drastically alter query execution performance. The extensive experimentation conducted on real-world datasets attests to the superiority of RankPQO. Our integrated solution within PostgreSQL not only surpasses the performance of PostgreSQL optimizer by up to 2.57× but also outshines the leading baseline by up to 1.36×. These results confirm the practical efficacy of RankPQO and its potential to improve PQO in database applications significantly.

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] 2022. TPC-H Benchmark. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp.

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

[5] 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.

[6] Mehmet Aytimur, Silvan Reiner, Leonard Wörteler, Theodoros Chondrogiannis, and Michael Grossniklaus. 2024. LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 54:1–54:25.

[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] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2007. On the Production of Anorexic Plan Diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007.* ACM, 1081–1092.

[11] 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.

[12] 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.

[13] 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.

[14] Masoud Reyhani Hamedani, Jin-Su Ryu, and Sang-Wook Kim. 2023. GELTOR: A Graph Embedding Method based on Listwise Learning to Rank. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023.* ACM, 6–16.

[15] 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.

[16] 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.

[17] 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.

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

[19] Yiling Jia, Huazheng Wang, Stephen Guo, and Hongning Wang. 2021. PairRank: Online Pairwise Learning to Rank by Divide-and-Conquer. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021.* ACM / IW3C2, 146–157.

[20] 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.

[21] 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.

[22] 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.

[23] Hang Li. 2014. *Learning to Rank for Information Retrieval and Natural Language Processing, Second Edition.* Morgan & Claypool Publishers.

[24] 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.

[25] Ryan 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.

[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] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.* AAAI Press, 1287–1293.

[28] Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005.* ACM, 1228–1240.

[29] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (2023), 740–752.

[30] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *PVLDB* 13, 3 (2019), 307–319.

[31] 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.

[32] 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.

[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] 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.

[35] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* 1297–1308.

[36] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670.

[37] Nengjun Zhu, Jian Cao, Xinjiang Lu, and Qi Gu. 2021. Leveraging pointwise prediction with learning to rank for top-N recommendation. *World Wide Web* 24, 1 (2021), 375–396.

[38] 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.