Practical Parameterized Query Optimization via Efficient Plan Reuse and List-wise Ranking

HAI LAN, School of Electrical Engineering and Computer Science, The University of Queensland, Australia
YANG YU, School of Computer Science, Wuhan University, China
ZHIFENG BAO, School of Electrical Engineering and Computer Science, The University of Queensland, Australia
ZI HUANG, School of Electrical Engineering and Computer Science, The University of Queensland, Australia
YUWEI PENG, School of Computer Science, Wuhan University, China

Modern analytical workloads, particularly those powering business intelligence dashboards, are dominated by parameterized queries, where a small number of query templates are executed repeatedly with varying predicate values. Classical cost-based optimizers frequently fail to produce efficient execution plans in this setting due to inaccurate cardinality estimation and unstable plan choices. Although numerous learning-based optimizers have been proposed, most are designed for general, ad-hoc query optimization and overlook the repetitive structure of parameterized workloads. As a result, they often incur high training or serving overheads or require substantial modifications to existing database systems, limiting their practical adoption. We propose PLARQ, a practical learned optimizer tailored for Parameterized Query Optimization (PQO). We begin by analyzing existing approaches through a unified two-stage framework: (1) Plan Candidate Generation (PCG), which forms a set of plausible execution plans, and (2) Plan Ranking (PR), which selects the most promising one. This abstraction captures the core design principles of prior work and provides a foundation for systematic comparison. Building on this framework, PLARQ employs a similarity-based PCG approach to retrieve a compact, high-quality set of plan candidates from a precomputed plan pool, and a list-wise, attention-based ranking model to effectively identify the optimal plan among them. PLARQ integrates seamlessly with PostgreSQL without modifying the optimizer internals. Extensive experiments across five benchmarks demonstrate that PLARQ improves end-to-end query performance, achieving speedups of up to 420.31x over PostgreSQL and up to 2x over existing learned methods.

CCS Concepts: • **Information systems → Database management system engines**; **Query optimization**.

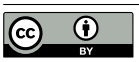
Additional Key Words and Phrases: Learned Query Optimization, Parameterized Query Optimization

ACM Reference Format:
Hai Lan, Yang Yu, Zhifeng Bao, Zi Huang, and Yuwei Peng. 2026. Practical Parameterized Query Optimization via Efficient Plan Reuse and List-wise Ranking. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 93 (February 2026), 26 pages. https://doi.org/10.1145/3788254

1 Introduction

Data warehousing and business intelligence applications rely heavily on parameterized queries, where fixed query templates are repeatedly executed with varying predicate values [52, 53]. In these

Authors' Contact Information: Hai Lan, h.lan@uq.edu.au, School of Electrical Engineering and Computer Science, The University of Queensland, Brisbane, Queensland, Australia; Yang Yu, yu_yang@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China; Zhifeng Bao, zhifeng.bao@uq.edu.au, School of Electrical Engineering and Computer Science, The University of Queensland, Brisbane, Queensland, Australia; Zi Huang, huang@itee.uq.edu.au, School of Electrical Engineering and Computer Science, The University of Queensland, Brisbane, Queensland, Australia; Yuwei Peng, ywpeng@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, Hubei, China.



ACM 2836-6573/2026/2-ART93
https://doi.org/10.1145/3788254

scenarios, often referred to as the Parameterized Query Optimization (PQO) setting [15, 16, 20, 51], stability and low latency are paramount. Existing database systems typically rely on cost-based optimizers to generate execution plans for these queries. However, despite decades of development, cost-based optimizers still struggle with complex templates due to persistent errors in cardinality estimation, cost modeling, and join order selection [29, 31].

With the rise of learning techniques, many methods have been proposed to tackle the limitations in the query optimizer, falling into two main categories: ***C1: Replacing Query Optimizer or Its Components*** – These methods replace specific internal modules, such as cardinality estimation [21, 23, 25, 26, 36, 46, 54, 55, 59, 60, 62, 63, 66, 71], cost estimation [11, 38, 40, 47, 48], or join enumerators [22, 27, 39, 50, 64]. While effective in isolation, integrating them into production systems requires significant engineering effort [67] and holistic system modification. However, integrating only one component is often insufficient [24], as other optimizer limitations remain. ***C2: Enhancing An Existing Query Optimizer with A Better Decision*** – Methods in this category still use an existing query optimizer in the plan generation process while they introduce learning-based methods to explore the plan space through hint settings [37, 58] or determine the final plan by ranking plans [14, 51, 70].

Table 1. Related Studies under Two-Stage Framework

Work	Scope	PCG	PR
Bao [37]	General QO	Hints	*Point-wise*
FASTgres [58]	PQO	Hints	-
Lero [70]	General QO	Cardinality Editing	*Pair-wise*
LogPQO [51]	PQO	Greedy Search	*Point-wise*
LTR [14]	General QO	Optimizer	*List-wise*
Cost-based Optimizer	General QO	Optimizer	*Point-wise*

Note: FASTgres directly predicts the hint set ID, rather than relying on a ranking-based approach.

1.1 Two-stage Framework for Plan Generation

Methods in Category ***C2*** share a common processing framework that can be abstracted into a *two-stage framework*: (1) Plan Candidate Generation (PCG), which constructs a set of plausible (sub-)plan candidates, and (2) Plan Ranking (PR), which selects the most promising execution plan from that set. This framework closely reflects how practical PQO solutions operate in real systems –leveraging existing optimizers to explore multiple alternative plans, followed by a component to guide final decision-making [16, 51]. Table 1 summarizes representative approaches under this unified view. In the table, the column "Scope" denotes whether each method targets general ad-hoc query optimization (General QO) or explicitly focuses on PQO[1].

In terms of Plan Candidates Generation (PCG), there are four different strategies in plan candidates generation. For a given query, Bao [37] and FASTgres [58] generate the different plans by setting different hints. These hints control the implementations of each operator to be considered, e.g., whether hash join can be used. Lero [70] first re-scales sub-plans' cardinality estimated by the default query optimizer, and then, it calls the optimizer to generate new plans based on the new cardinality. LogPQO [51] selects k plans from m plans by greedily adding a plan in each iteration that minimizes a predefined metric. The m plans are generated from the historic workloads. Differently, LTR [14] integrates the ranking procedure into the plan enumeration process, and the (sub-)plan candidates are generated by the optimizer's enumeration algorithm. We argue that a good PCG

[1]We categorize FASTgres as PQO because it utilizes template-specific models (contexts) to select plans based on predicate parameters. This design mirrors the PQO setting, where recurring query templates are optimized by mapping specific parameter bindings to efficient execution configurations.

strategy for PQO workload should address two key aspects: (1) the time in generating the plan candidates for an online query should be low; (2) the best plan found should yield large performance gains, and many candidates should outperform the default plan. *However, our empirical study shows that none of them is able to fully satisfy both aspects (cf. Section 5.2).*

As for Plan Ranking (PR), there are three ranking approaches: *point-wise*, *pair-wise*, and *list-wise* approaches [35]. The *point-wise* based methods, Bao and LogPQO, select the plan based on the latency prediction, which is trained in a supervised learning manner to predict an absolute value. It is less effective compared to *pair-wise* and *list-wise* approaches [35] and may select a inferior plan. The *pair-wise* based method, Lero, only compares two plans at a time and does not consider more contextual information by looking at more plans at a time. Thus, it could select a sub-optimal plan. Moreover, when selecting the final plan, Lero will trigger the model several times, which is inefficient. LTR calls the learned model to rank the candidates in each sub-plan decision, which is time-consuming. Moreover, for each candidate, LTR calls the built model once to calculate the ranking score for each candidate. *In summary, existing methods are either inefficient or ineffective in the PR* stage.

To summarize, existing approaches remain inadequate for PQO: PCG strategies are either slow or fail to include high-quality plans, and PR strategies struggle with accuracy or runtime overhead. This creates a clear need for a PQO-focused solution that is both efficient and deployable in real systems.

1.2 Our Solution

In this paper, we propose PLARQ, an efficient **PLA**n **R**euse and list-wise plan **R**anking for **Q**uery optimization, within the two-stage framework. We introduce a novel PCG method that leverages query similarity to retrieve a compact, high-quality set of candidate plans from a pre-computed pool. This method is motivated by a key observation: *Two queries that are close in the query space (i.e., all possible instances of the same template) are highly likely to share the same optimal execution plan.* Given a query template, we first construct a pool of global candidates with a tailored sampling strategy to span the query space. Each item in the pool comprises a query and the corresponding (near−)optimal plan, obtained through offline exploration. For each coming query, we select the k nearest candidates from the pool by computing the query-level similarity and their associated plans are then used as the candidate plans.

To build an efficient and effective PR stage, we apply a *list-wise* attention model to jointly rank these candidates and select the most efficient plan with three key features: (1) we only consider the ranking process at plan level instead of sub-plan level, which reduces the number of calls in ranking and does not modify the code of the database; (2) thanks to the effectiveness of our PCG stage, we only consider a small number of plan candidates, which reduces the inference time of the proposed model; (3) we propose a new model to rank the candidates in a *list-wise* manner to capture inter-plan relationships to identify the optimal plan from the candidates.

In summary, the main contributions of this paper are:

- Based on our observations (cf. Section 2.1) of the (near−)optimal plans of different queries from the same query template, we propose a new PCG strategy by selecting the plan candidates for an incoming query from a plan candidates pool. We propose novel strategies for the plan candidates pool construction and plan candidates selection (cf. Section 3).
- We propose a new *list-wise* plan ranking method and adopt the attention-based model to capture the relationships among different plans at the same time (cf. Section 4).

- We implement our method on PostgreSQL [5] and conduct a comprehensive evaluation to show the efficiency and effectiveness of our proposed methods (cf. Section 5). For exmaple, the end-to-end query performance achieves speedups of up to 420.31x over PostgreSQL and up to 2x over existing learned methods.

2 Core Idea & System Overview

We first introduce the notations and concepts used under the Parameterized Query Optimization setting. A *query template*, Q, is a parameterized SQL query that can be used to run a variety of specific queries by substituting placeholders, i.e., the values in the predicates. A *query*, q, for one query template is generated by substituting placeholders with actual values. A *query space*, S, refers to the entire set of possible queries that can be generated from a given query template. A *similarity function* between two queries, q_i and q_j, from the same query template, $f(q_i, q_j)$, measures how close q_i and q_j are within the corresponding query space. We now introduce two core components of the two-stage framework:

Definition 1 (Plan Candidates Generation (PCG)). *Given a query q, PCG aims to generate a set of plan candidates, P_q, for q such that every plan $p_i \in P_q$ is a valid plan for q and P_q includes some plans that are more efficient than the plan generated from the native query optimizer with a high probability.*

Definition 2 (Plan Ranking (PR)). *Given a set of plan candidates, $P_q = \{p_1, p_2, ..., p_n\}$, of a query q, PR takes P_q as input and outputs the rank of each plan in P_q, such that a plan with a lower rank is more efficient. Let $r(p_i)$ denote the rank of p_i and $t(p_i)$ denote the running time of p_i. If $r(p_i) < r(p_j)$, then $t(p_i) < t(p_j)$.*

To this end, the quality of the generated execution plan under this two-stage framework for an incoming query depends on whether the plans better than the default one are included in the candidates (PCG) and whether we can select the best plan from the candidates correctly (PR). The efficiency of the plan generation process depends on the efficiency of these two stages.

2.1 Core Ideas

We next introduce two core ideas for designing an efficient and effective learned optimizer under the two-stage framework.

Core Idea 1: PCG– Leverage Similar queries to Generate High-quality Plan Candidates. Given a query q of one query template Q, we carefully select the plans of other queries from the same query template as the candidates for q. This idea is attractive if we know which queries could have the same (near-)optimal plan as q. This idea is feasible based on the following two observations:

Observation 1. *Two close queries in the query space of the same query template have the same (near-)optimal execution plan with a high probability.*

Observation 2. *There exist some sensitive regions where a small change in the predicates' values leads to different plans.*

These observations are well supported by prior studies [19, 45]. For example, Reddy and Haritsa [45] visualize the mapping from queries to execution plans and show that large contiguous regions are often covered by the same plan. However, they also identify optimizer instability near plan boundaries, where even minor changes in selectivity can trigger abrupt plan changes.

Motivated by these insights, we propose *selecting k nearest neighbors of query q and using their (near-)optimal plans as the plan candidates*. Observation 1 supports using nearest neighbors to

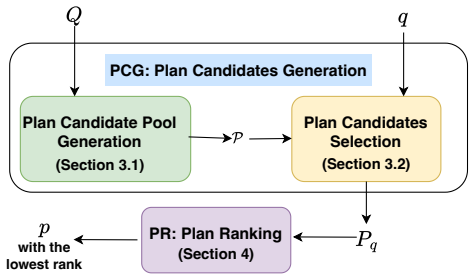


Fig. 1. PLARQ overview.

include high-quality plans, while Observation 2 motivates choosing multiple neighbors (via k) to reduce the risk of including only suboptimal plans from unstable regions.

Table 2. Profiling PR stage of Bao and Lero on CEB and Stack.

Query	#Plans		Opt. Time		Model Time		Exe. Time	
	Bao	Lero	Bao	Lero	Bao	Lero	Bao	Lero
CEB-1	5	40	23.2	195.8	5.7	60.4	557.1	309.3
CEB-2	5	45	29.4	371.2	5.8	75.9	36.2	61.5
CEB-3	5	25	11.9	67.2	4.7	31.7	682.2	253.2
CEB-4	5	65	339.5	6088.9	6.8	140.8	52593.0	4345.7
CEB-5	5	75	1802.9	34696.4	7.9	218.2	39252.4	6868.1
Stack-1	5	15	12.1	40.0	4.2	20.1	16.382	13.2
Stack-2	5	30	27.0	240.9	5.0	40.7	1388.632	329.0
Stack-3	5	40	392.2	8591.5	5.8	72.3	28.55	30.6
Stack-4	5	35	63.4	858.0	5.9	60.5	53.111	37.8
Stack-5	5	15	11.3	43.1	4.2	21.6	33.667	23.4

Core Idea 2: PR– Rank A Small Set of Good Candidates Simultaneously. We conduct mini-experiments on Bao and Lero by adopting CEB and Stack benchmarks. We sample five queries from each and the results are shown in Table 2. We report the number of plan candidates (#Plans), time used in calling optimizer (Opt. Time) to annotate the plans, model call (Model Time), and plan execution time (Exe. Time). We report the time in *milliseconds*.

From Table 2, we find that: (1) Lero generates more plan candidates than Bao and usually finds better plan candidates resulting in shorter execution times; (2) however, its larger candidate set makes plan selection significantly slower – sometimes even exceeding execution time (e.g., CEB-5, Stack-3). Thus, selecting a *small set of good plan candidates* is crucial in PR stage.

Moreover, existing methods either adopt a *point-wise* or *pair-wise* ranking idea, which is ineffective in capturing more contextual information to select the final execution plan. To address this, we adopt a *list-wise* ranking approach, which jointly considers all plan candidates and enables more accurate final plan selection.

2.2 System Overview

In this paper, we propose a learned query optimizer, PLARQ, under the two-stage framework for generating execution plan for a given query, grounded in the two core ideas introduced above.

Each stage is carefully designed to balance efficiency (i.e., the time required for plan generation) and effectiveness (i.e., the quality of the selected execution plan). Figure 1 provides an overview of the proposed system, which we refer to as PLARQ.

Before processing an incoming query, the PCG module constructs a candidate plan pool $\mathcal{P}$ for a query template Q. Each item in the pool is a tuple (q_i, p_i), where q_i is a historical query and p_i is its associated execution plan. We assume p_i is (near-)optimal for q_i. When a new query q of template Q arrives, a similarity-based search algorithm retrieves the top-k most similar items from $\mathcal{P}$, forming the candidate set P_q.

The PR module then ranks the plans in P_q and selects the one with the lowest estimated rank. Lower ranks correspond to more efficient execution plans.

3 Efficient Plan Candidate Generation via Reusable queries

The goal of the Plan Candidate Generation (PCG) module is to efficiently produce a small set of high-quality plan candidates for each incoming query. PLARQ achieves this by reusing plans previously generated for similar queries within the same query template, hence avoiding expensive online plan enumeration. Before serving a new query, PLARQ constructs a candidate plan pool (cf. Section 3.1.2). At runtime, it retrieves the top-k nearest candidates from the pool based on a predefined similarity measure and reuses their associated (near-)optimal plans as candidates (cf. Section 3.2).

3.1 Offline Candidate Plan Pool Construction

PLARQ focuses on the Parameterized Query Optimization (PQO) setting, where query templates are known in advance and repeatedly executed with different parameter bindings. Consequently, plan pools are constructed entirely *offline*, ensuring that no additional cost is incurred during query time. This offline setup allows the system to explore promising plans once per template and reuse them across thousands of future query instances. When constructing the pool, we consider three key properties: plan quality, pool size, and candidate coverage. Plan quality reflects whether the included plans are (near-)optimal, as suboptimal candidates are unlikely to yield optimal results for incoming queries based on Observation 1 (cf. Section 2.1).

A larger pool size allows for a more diverse set of candidates, increasing the likelihood of finding a better plan for an incoming query. Meanwhile, it incurs higher computational cost, as more candidates must be evaluated during plan generation. To balance effectiveness and efficiency, we treat the pool size as a tunable parameter. Given a certain pool size, our goal is to select candidates that can serve a wide range of incoming queries – i.e., candidates that provide broad coverage of the query space.

3.1.1 (Near-)optimal Plan Generation. Unlike the online process for an incoming query, where we have a strict constraint on the optimization time (e.g., tens of milliseconds), this module operates offline and so we can spend more time on plan generation.

Naive Methods. A naive way to obtain a (near-)optimal plan is to enumerate all valid plans by varying join orders and physical operators, execute them, and select the fastest. However, this is infeasible due to the large plan space. A more practical alternative is to guide the optimizer by injecting true cardinalities. In practice, "injecting true cardinalities" serves only as a heuristic to guide the optimizer toward high-quality plans: perfect cardinalities do not guarantee globally optimal plans but effectively mitigate major estimation errors [31]. addressing a key source of suboptimality: cardinality estimation errors. Yet, this approach is also costly, as it requires computing exact cardinalities for all subplans.

Algorithm 1: (Near-)optimal Plan Generation

Input: Query instance q, error bound eb, iteration threshold s
Output: Execution plan p_{best} of q

1 $p_{best} \leftarrow NIL$, $t_{best} \leftarrow +\infty$; current best plan and its runtime;
2 $i \leftarrow 0$, $errs \leftarrow \{\}$; $errs$ stores nodes' Q-errors;
3 $hints \leftarrow \{\}$; execution hints for q;
4 **while** $i = 0 \vee (i \leq s \wedge \exists\ err\ in\ errs > eb)$ **do**
5 $p, t \leftarrow$ RunQuery(q, $hints$);
6 **if** $t < t_{best}$ **then**
7 $t_{best} \leftarrow t$, $p_{best} \leftarrow p$;
8 $errs \leftarrow$ ComputeQError(p); get the Q-error of each node in p;
9 $hints \leftarrow$ UpdateHints(p, $hints$); add hint for node in p with the inaccurate cardinality estimation;
10 $i = i + 1$;
11 **return** p_{best};

Proposed Method. Instead of injecting the true cardinality for all possible nodes, we find that *injecting true cardinality for some nodes is already effective in finding a good plan (much better than the default plan) for a query.* But a question is raised – which nodes do we need to inject the true cardinality? We adopt a simple yet efficient way – injecting the true cardinality obtained by running the execution plan generated the query optimizer. We iteratively correct the most significant estimation errors by injecting true cardinalities only at critical nodes, efficiently identifying high-quality plans without exhaustive search. Algorithm 1 shows the process. To control the overhead of the whole process, we introduce two parameters, error bound eb and iteration threshold s. The former determines the quality of cardinality estimation for the node in a plan. The latter specifies how many rounds we can inject true cardinality at most. Usually, a smaller eb or a larger s will incur a longer running time.

Given a query q, we execute it under a set of injected true cardinality, we refer to as $hints$[2], and obtain the execution plan p and running time t. If t is smaller than t_{best}, the current best plan's running time, we store p and t (Lines 5 - 7). Then, we compute the Q-error of each node in p and update the $hints$ by adding the new hints for the node with the inaccurate cardinality estimation in p (Lines 8 - 9). We repeat the above process until all nodes' Q-errors are no larger than eb or we already run s rounds (Line 4), and return p_{best} (Line 11).

Enhancing Cost Model Support for Injected Cardinalities. To build the candidate pool, we rely on PostgreSQL's `pg_hint_plan` extension, which enables fine-grained control over join orders and operator choices, requiring no internal code modification and ensuring practicality. Similar hinting mechanisms exist in other commercial systems (e.g., Oracle [3], SQL Server [6]), making our approach practical and portable. To generate high-quality plans through cardinality injection during plan exploration, we identified a critical limitation in the PostgreSQL extension pg_hint_plan: it does not incorporate the injected output cardinalities of intermediate join nodes when computing their individual costs. As a result, even when we fix cardinalities using hints in pg_hint_plan, the optimizer's cost model may still rely on its internal (and potentially inaccurate) cardinality estimates, leading to suboptimal cost assessments. This limitation becomes particularly problematic in complex queries involving multi-way joins, where accurate intermediate cardinality estimation is crucial for meaningful plan evaluation. To overcome it, we extend pg_hint_plan to allow explicit

[2]We conduct our work on PostgreSQL [5] and use pg_hint_plan [4] here, to inject the true cardinality. Thus, we call the injected cardinality as hints.

injection of true cardinality at each join node. This enhancement enables PostgreSQL's cost model to reflect the true cost implications of the provided cardinalities, supporting more accurate evaluation of plans involving nested loop, merge join, and hash join operators. Although this modification is engineering in nature,it is essential for enabling the optimizer to make meaningful use of our hints to output a better plan.

3.1.2 Plan Candidates Pool Construction. To enable effective plan reuse, we construct a candidate plan pool $\mathcal{P}$ for each query template Q. The goal is to populate $\mathcal{P}$ with a diverse set of queries that collectively span a broad range of predicate selectivities and execution characteristics. Each item in $\mathcal{P}$ is a pair (q_i, p_i), where q_i is a query derived from Q, and p_i is the (near-)optimal execution plan for q_i. The construction process consists of three steps and the construction time of candidates pool are analyzed in Section 5.5.2, confirming that the offline cost is practical for real workloads.
Step 1: Predicate Value Sampling. We begin by identifying all parameterized predicates in the template query Q, which we classify into four types: `IN`, `EQ`, `LIKE`, and `RANGE`. For each type, we adopt a data-aware sampling strategy to ensure representative coverage of the predicate selectivity space.

- **`IN` and `EQ` Predicates.** For categorical attributes that encode semantic types (e.g., `region`, `category`), we collect all distinct values from the column. For non-type categorical attributes with large domains, we sort values by their frequency in the base table and partition them into three strata: high-frequency (top 33%), medium-frequency (middle 34%), and low-frequency (bottom 33%). We then uniformly sample n values from each stratum to form a set of $3n$ candidate values, where we set $n = 6$ in our experiments.
- **`LIKE` Predicates.** We manually or synthetically generate a pool of wildcard patterns, then evaluate their selectivities using the base data. To ensure diverse behavior, we retain patterns with significantly different selectivities, choosing multiple values from each order-of-magnitude bucket.
- **`RANGE` Predicates.** For numeric attributes, we first determine whether value distributions are skewed. If not, we use the column's minimum and maximum values as bounds. If skew is present, we extract the tightest range covering 90% of the data and partition it into a fixed set of intervals based on δ, the range span. We then select representative intervals such as $[min + \delta/2, max]$, $[min, min + \delta/4]$, etc., covering both narrow and wide ranges.

Step 2: Query Generation. Using the sampled predicate values, we enumerate all valid combinations of predicates to generate the full set of queries for the template Q. Each instance corresponds to a unique binding of values across all predicates. This exhaustive enumeration ensures that the candidate space captures diverse logical variations of the template.
Step 3: Random Sampling and Plan Annotation. To limit construction cost, we randomly sample a certain number l of queries from the enumerated set to populate the pool. Each sampled query q_i is executed under the target DBMS, and its corresponding execution plan p_i is obtained using the procedure described in Section 3.1.1 and selecting the fastest plan among candidates. We treat the observed plan as (near-)optimal and include the pair (q_i, p_i) in the candidate plan pool $\mathcal{P}$. This ensures that the pool contains both structural diversity in query predicates and plan diversity.

3.2 Candidate Selection via Similarity Search

After constructing the candidate pool $\mathcal{P}$ for a given query template, our goal is to select the k nearest neighbors of a new incoming query q from $\mathcal{P}$ and use their associated plans as the candidate plans for q (cf. Section 2.1). To enable this, we must define a similarity function that measures how close two queries are within the same template.

Since different query templates may have different sets of placeholders (i.e., predicates), we design template-specific similarity functions that compare queries based on the semantics of their predicates. In the following, we introduce the similarity computation for common predicate types, including range, equality, IN, and LIKE predicates.

3.2.1 Similarity of Each Predicate. Since most query templates in existing benchmarks [7, 9, 10], place placeholders only within the predicates, we adopt the same assumption in this work. We focus on seven common predicate operator types: $>$, $<$, $\geq$, $\leq$, $=$, IN, and LIKE. Below, we define the similarity function for each operator individually.

Case 1: Similarity on RANGE Predicates. We use a range predicate, $v_l \leq A \leq v_u$ (i.e., the interval $[v_l, v_u]$), as an example to introduce our similarity function, where A is the attribute associated with the predicate placeholder, and v_l and v_u are the lower and upper bounds, respectively.

Given two queries, q_1 and q_2, we assume their range predicates on attribute A are $R_1 = [v_l^1, v_u^1]$ and $R_2 = [v_l^2, v_u^2]$, respectively. We also assume that a histogram $\mathcal{H}_A$ has been constructed for attribute A. Let $\mathcal{H}_A(R)$ denote the estimated number of tuples falling within range R according to the histogram.

We define the similarity between R_1 and R_2 as:

$$\text{sim}(R_1, R_2) = \frac{\mathcal{H}_A(R_1 \cap R_2)}{\mathcal{H}_A(R_1 \cup R_2)},$$

which reflects the degree of overlap between the two ranges in terms of their estimated selectivity. A higher ratio indicates greater similarity in filtering behavior.

This definition can be naturally extended to other forms of range predicates by appropriately computing their intersection and union on the histogram.

Case 2: Similarity on IN Predicates. The value of an IN predicate is a set of constants. Suppose q_1 and q_2 contain IN predicates on attribute A with value sets S_1 and S_2, respectively. A straightforward approach is to follow the strategy from **Case 1** and compute similarity as:

$$\text{sim}(q_1, q_2) = \frac{\mathcal{H}_A(S_1 \cap S_2)}{\mathcal{H}_A(S_1 \cup S_2)},$$

where $\mathcal{H}_A$ is the histogram of attribute A.

However, this approach is often ineffective for IN predicates. Typically, the attribute used in an IN predicate is categorical and resides in a dimension or reference table, where each value appears only once. In such cases, $\mathcal{H}_A$ yields uniform frequencies, making it difficult to differentiate similar from dissimilar queries.

To address this, we instead build the histogram over the attribute A after joining its base table T with a related fact table T'. This reflects the *actual impact* of the IN predicate after joining. Let $\mathcal{H}_A^{T \bowtie T'}$ denote the histogram of A after the join. We then define the similarity as:

$$\text{sim}(q_1, q_2) = \frac{\mathcal{H}_A^{T \bowtie T'}(S_1 \cap S_2)}{\mathcal{H}_A^{T \bowtie T'}(S_1 \cup S_2)}.$$

In most cases, the dimension table T involved in the IN predicate participates in a single join relationship with one fact table T', making this join-based histogram both feasible and effective. Histogram construction is performed *offline*, since query templates and their predicates are known in advance. At query time, PLARQ only reads these precomputed summaries to locate the most similar historical query, incurring negligible overhead.

Case 3: Similarity on EQ Predicates. Consider an equality predicate of the form $A = v$. There are two ways to interpret such predicates: (1) as a special case of an IN predicate where the value set

contains only one element; (2) as a special case of a range predicate where the lower and upper bounds are equal, i.e., $[v, v]$.

Based on the data type of attribute A, we choose the appropriate strategy. If A is a categorical attribute, we compute similarity using the same approach as in **Case 2** for `IN` predicates. If A is a numerical attribute, we compute similarity based on the estimated cardinalities of the specific values. Let v_1 and v_2 be the equality values in queries q_1 and q_2, respectively. The similarity is defined as:

$$\text{sim}(q_1, q_2) = \frac{\min(\mathcal{H}(v_1), \mathcal{H}(v_2))}{\max(\mathcal{H}(v_1), \mathcal{H}(v_2))},$$

where $\mathcal{H}(v)$ denotes the estimated number of tuples with value v in the histogram of attribute A.

Case 4: Similarity on `LIKE` Predicates. Let q_1 and q_2 be two queries containing `LIKE` predicates with values v_1 and v_2, respectively. String patterns in `LIKE` clauses can vary significantly in structure and semantics (e.g., wildcards, prefixes, substrings), making it difficult to define a meaningful syntactic or semantic distance between them. Instead of comparing the patterns directly, we measure the similarity between q_1 and q_2 based on the effect of their predicates – specifically, how many tuples each predicate is expected to match. Let c_1 and c_2 be the cardinalities of the predicates `LIKE` v_1 and `LIKE` v_2, respectively. We define their similarity as: $\frac{min(c_1,c_2)}{max(c_1,c_2)}$, which reflects how similar the two predicates are in terms of filtering strength – i.e., the closer their selectivities, the higher the similarity.

Computing the exact cardinality for arbitrary `LIKE` predicates is often infeasible, as it would require full scans or specialized index lookups, incurring high overhead. To make the similarity computation efficient, we instead use the query optimizer's cardinality estimates for c_1 and c_2. While these estimates may not always be accurate, they provide a low-cost and consistent approximation that enables practical and scalable similarity computation.

3.2.2 Aggregated Similarity from Predicates. A query template typically contains multiple placeholders, each corresponding to a specific predicate on an attribute. Given two queries derived from the same template, we first compute the similarity for each predicate individually using the methods described in the previous cases.

To obtain the aggregated similarity between the two queries, we aggregate the predicate-level similarities. By default, we compute the equal-weight sum of all predicate similarities. This aggregated similarity score captures the overall closeness between the two queries in terms of their filtering conditions.

We then use this aggregated similarity to perform nearest neighbor search over queries, enabling context-aware plan retrieval based on predicate similarity.

4 List-wise Plan Ranking via Contextualized Embeddings

After retrieving the candidate plans for a query, we now describe how to select the final plan to be executed. PLARQ ranks the candidate plans using a *list-wise* ranking approach and selects the plan with the lowest predicted rank. Ideally, a lower rank corresponds to a lower actual execution time. In Section 4.1, we describe how each plan candidate is embedded into a vector representation by capturing its structural features, query-specific information, and its relationship to other plan candidates. Section 4.2 then introduces the ranking model that operates on these embeddings, and Section 4.3 details the training procedure.

Figure 2 illustrates the overall architecture of our ranking model, which consists of two components: the Plan Embedding Layer, which generates embedding vectors e_p for each candidate plan p, and the Rank Prediction Layer, which predicts the execution rank based on these embeddings.

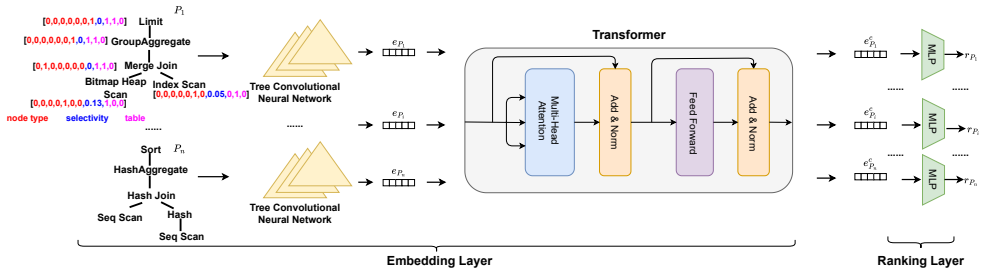


Fig. 2. Overview of the plan ranking model architecture.

4.1 Plan Embedding Layer

An effective plan embedding layer should capture two complementary types of information: (1) the detailed characteristics of each individual plan, and (2) the contextual relationships among all candidate plans. Both are critical for accurate ranking.

First, the execution performance of a plan depends on its structure – such as join order, physical operator choices, and the shape of the plan tree. Second, a plan's relative quality is influenced by the presence of alternative candidates. The same plan may be ranked differently depending on which other plans it is compared against.

To incorporate both aspects, our embedding layer is designed in two stages. The first stage uses a neural encoder to generate an embedding for each plan based on its structure and query-specific features. The second stage refines these embeddings using a self-attention mechanism, allowing each plan to incorporate information from the other candidates in the set.

4.1.1 Single-Plan Feature Extraction. For each query plan, PLARQ first generates an initial encoding aligned with the plan's tree structure, as illustrated in Figure 2. Each node in the tree corresponds to a physical operator, and we extract two primary types of features for each node: the operator type and the accessed tables, which aligns existing studies [37, 70].

Operator Type. We categorize physical operator types into seven main groups: Hash Join, Merge Join, Nested Loop, Sequential Scan, Bitmap Heap/Index Scan, Index (Only) Scan, and a general category for all other operators (e.g., Hash, Materialize, Aggregate). These categories are selected based on their prevalence in query plans across various workloads. Each operator type is represented using one-hot encoding. For example, the vector for Hash Join is $[1, 0, 0, 0, 0, 0, 0]$.

Table Access. We also encode table access information using one-hot encoding, where the length of the vector corresponds to the total number of tables in the database. For each plan node, we examine which tables are accessed. If a node accesses a specific table, the corresponding position in the vector is set to 1; otherwise, it remains 0. This representation enables the model to capture which parts of the schema are involved in different parts of the plan, which is important for understanding the data access pattern and for reasoning about plan performance.

Selectivity Information. To incorporate query-level information, we include selectivity only for scan nodes. Selectivity (or normalized cardinality) is omitted for non-scan nodes due to two reasons: (1) obtaining accurate values for intermediate nodes incurs high overhead, and (2) using estimated values can introduce noise and degrade embedding quality. Therefore, we assign a selectivity value of 0 to all non-scan nodes.

For scan nodes, we estimate selectivity based on available statistics. Specifically, we use the same statistics employed by PCG to estimate selectivity for tables with predicates. For LIKE predicates,

we directly use the optimizer's estimate. For tables without predicates (or with only join conditions), and for all other non-scan node types, selectivity is set to 0.

Node Encoding. Finally, we concatenate the operator type encoding, table access vector, and selectivity value to form the complete feature vector for each node in the plan tree.

4.1.2 Structural Embedding via TreeCNN. A number of methods have been proposed to embed individual query plans in order to capture their structural and semantic details. Among them, Tree Convolutional Neural Networks (TreeCNNs) [42] have emerged as a highly effective and efficient approach. As demonstrated in a recent experimental study [68], TreeCNN consistently achieves strong performance on cost estimation tasks and outperforms alternative models on in-distribution (ID) workloads, where relative plan rankings are critical. While models such as LSTM may offer slightly better accuracy on more complex, out-of-distribution (OOD) queries, TreeCNN provides a significantly better trade-off between accuracy and computational overhead. Its lightweight architecture enables faster training and inference, making it a practical and scalable solution for real-world query optimization tasks. Based on this observation, PLARQ adopts TreeCNN as the backbone for encoding individual plans.

TreeCNN is a variant of convolutional neural networks designed specifically for tree-structured data such as query plans. It applies a set of triangular-shaped convolutional filters over all local subtrees of the form (parent, child, child), effectively capturing local structural patterns. These local features are then aggregated using dynamic pooling to produce a fixed-size vector representation for the entire plan.

As shown in Figure 2, we employ a three-layer TreeCNN to generate the initial embedding of each plan. We refer to this as the "initial" embedding because it captures only the internal structure of the plan, without incorporating information from other candidates in the same query context. That contextual information is handled in the next stage.

4.1.3 Modeling Inter-Plan Relationships. Existing methods typically transform the output of a tree-based plan encoder (e.g., tree convolution) into a scalar through additional layers, such as pooling and fully connected layers. This scalar is then used either to directly estimate the query latency [37] or to derive a ranking score for plan selection [70]. While all plan candidates from the same query share the same model parameters, these approaches treat each plan independently during scoring, without considering the context of other competing plans in the candidate set. As a result, the learned representation for a plan fails to reflect its relative merit among alternatives—an aspect that is crucial for effective ranking.

To address this limitation, we propose to explicitly model the interdependence among candidate plans for the same query. We achieve this by feeding the output embeddings of the tree convolution encoder into a multi-head self-attention module. This transformation enables *contextualization*: the representation of each plan is refined based on its interactions with other plans in the same candidate set. In other words, instead of learning an isolated embedding, the model learns a relational representation that captures how each plan compares to others, which is essential for accurate *list-wise* ranking.

4.2 Rank Prediction Layer

After obtaining the contextualized embedding for each candidate plan, we apply a three-layer fully connected neural network to transform each embedding and produce a ranking score. A final softmax layer converts these scores into class probabilities. This formulation casts the plan ranking task as a multi-class classification problem: each candidate plan is assigned to one of k classes. Specifically, the plan predicted to be the fastest is assigned to class 0, while the slowest is assigned to class $k - 1$.

The fully connected network is shared across all candidate plans, meaning each plan embedding is processed independently but identically using the same network parameters. This design ensures consistent scoring behavior across candidates and enables efficient batched inference during plan selection.

To determine the final output, we examine the predicted classes. If exactly one plan is assigned to class 0, it is selected as the final output. If multiple plans are assigned to class 0, one of them is selected uniformly at random. This strategy reflects the model's confidence that the selected plans are similarly optimal and interchangeable from a performance perspective.

4.3 Model Training

4.3.1 Loss function. We design our loss function by drawing inspiration from pairwise ranking methods, with the key objective of capturing the relative efficiency among a set of execution plans. Rather than evaluating plans in isolation, our approach emphasizes pairwise comparisons to learn the overall ranking within the candidate set.

To model *list-wise* ranking behavior, we define a composite loss function consisting of two components:

$$\begin{aligned} Loss = & -\frac{1}{N}\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i \\ & -\sum_{i=1}^{n}\sum_{j=i+1}^{n} (sign(e^{y_i} - e^{y_j}) \cdot \frac{y_i + y_j}{1 + e^{\hat{y}_j - \hat{y}_i}}) \end{aligned} \tag{1}$$

The first term is the standard cross-entropy loss, where y_i is the true class label (i.e., rank) and $\hat{y}_i$ is the predicted probability for candidate i. This component encourages accurate classification of each candidate plan's relative rank, indirectly capturing ranking quality through label distribution.

The second term enforces consistency in pairwise comparisons across all candidate plans. For each pair (i, j), we compute a margin-based loss weighted by the signed difference in ground-truth efficiencies. The sign function returns -1, 0, or 1 based on the difference $e^{y_i} - e^{y_j}$, indicating whether plan i should be ranked before, equal to, or after plan j. The denominator $1 + e^{\hat{y}_j - \hat{y}_i}$ encourages the predicted ranking $\hat{y}_i < \hat{y}_j$ when $y_i < y_j$, reinforcing correct pairwise orderings.

Together, this loss formulation integrates both pointwise classification and pairwise ordering signals, aligning with our goal of learning robust list-wise rankings for execution plans.

4.3.2 Training Data Generation. For each training query q, we generate a set of plan candidates P_q by retrieving its top-k nearest neighbors from the corresponding candidate pool. Each candidate plan $p_i \in P_q$ is executed to obtain its runtime t_i, which serves as the supervision signal for training. The collected execution records for query q are denoted as $\{(q, p_i, t_i) \mid p_i \in P_q\}$.

Let W denote the set of training queries. We aggregate all execution records from queries in W into a unified training pool:

$$\mathcal{U}_W = \bigcup_{q \in W} \{(q, p_i, t_i) \mid p_i \in P_q\}.$$

To construct a training instance, we randomly sample k records from $\mathcal{U}_W$. The running time t_i of these sampled items are then used to derive their ranking labels.

Each sampled item is assigned a one-hot label vector of length k, indicating its relative rank among the k candidates. Specifically, if an item has the j-th smallest running time (starting from 0), the j-th position in its label vector is set to 1, and all others are set to 0. This encoding provides the

model with a clear supervision signal that reflects the true performance ordering of the candidate plans.

If ranking is restricted to plan candidates of the same query, the number of training instances equals the number of training queries. In contrast, our sampling-based procedure allows ranking across plans from different queries, thereby generating significantly more training instances without increasing the number of training queries. This cross-query supervision enriches the training signal and enhances the model's generalization ability.

5 Experiments

We conduct a comprehensive experimental evaluation of PLARQ as well as existing learning-based query optimization methods, aiming to answer the following questions:
Q1: How effective are the plan candidates generated by PLARQ, compared to those produced by baseline methods?
Q2: How efficient is PLARQ in terms of end-to-end query processing performance?
Q3: How well can PLARQ adapt to dynamic scenarios?
Q4: What is the preprocessing overhead incurred by PLARQ before it is able to serve online queries?
Q5: How do different parameter settings and system design choices impact the performance of PLARQ?

We start from presenting the experimental setup in Section 5.1. Then, we give the evaluation details for **Q1** – **Q5** from Section 5.2 to Section 5.6.

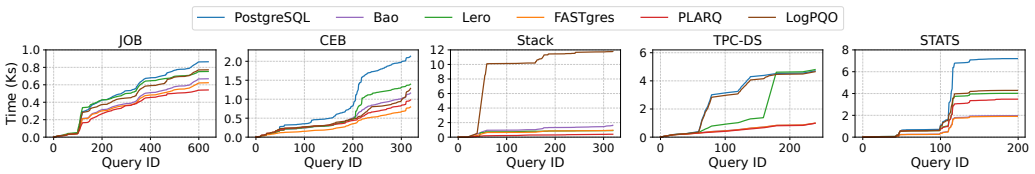


Fig. 3. Cumulative running time of best plan candidate generated from different methods.

5.1 Experimental Setup

Datasets and Workloads. We conduct experiments on five representative benchmarks that are also widely used by state-of-the-art methods [37, 58, 70]:

- **JOB** [31]: This benchmark includes 113 complex SQL queries across 33 templates derived from the IMDB dataset, which captures information about movies, actors, directors, and related entities. We generate training and testing queries with non-empty results, as modern databases typically optimize empty-result queries efficiently. Template #29 is excluded due to insufficient non-empty queries after extensive sampling, leaving 32 templates in our evaluation.
- **CEB** [43]: This benchmark contains 13,646 queries across 16 templates on the IMDB schema. All templates are included.
- **Stack** [7]: This workload comprises 6,191 queries over 10 tables, organized into 11 templates. All templates are included.
- **TPC-DS** [9]: A widely used benchmark modeling a retail data warehouse with 24 tables. We use 12 representative templates[3] out of the 99 available, generating training and testing queries using the default query generator with a scale factor of 10.

[3]Specifically, we use templates q13, q15, q18, q25, q26, q27, q29, q62, q72, q84, q91, and q99 following the setting in Lero [70].

- **STATS** [21]: This benchmark contains 146 query templates over the STATS dataset [8], which includes eight relational tables populated from the Stats Stack Exchange network. We randomly select 10 templates for evaluation. Compared to IMDB, STATS features more diverse and skewed data distributions, making it a suitable testbed for evaluating optimizer robustness under dynamic conditions.

Query Generation. Overall, for each query template, we randomly generate 20 *non-overlapping* training queries and 20 testing queries, ensuring that all selected queries produce *non-empty* results and are *not duplicated* in the candidate plan pool.

For JOB and STATS, which do not provide sufficient instances for each template, we generate training and testing instances following the query generation procedure described in Section 3.1.2. For the remaining benchmarks, we directly sample the training and testing queries from the available query sets provided by each benchmark or its generator.

Methods for Comparison. We compare the following approaches[4]:

- PLARQ: Our proposed method retrieves plan candidates from a global candidate pool constructed per query template, and selects the final plan using a *list-wise* ranking model. The candidate pool is built following the procedure in Section 3.1.2, with a fixed pool size of 500. At runtime, PLARQ selects the top-5 most similar candidates for each incoming query by default. Source code is available at https://github.com/DataAutonomyLab/l2r.
- Bao [37]: Bao generates the plan candidates by setting different hints, adopts a model to predict the running time of each plan, and selects the plan with the lowest predicted running time. We adopt the authors' source code [1] and use 5 arms and do not enable parallelism.
- FASTgres [58]: FASTgres explores a broader space of hint configurations than Bao and trains a separate model for each query template, in contrast to Bao's global model.
- Lero [70]: Lero generates plan candidates by perturbing the cardinality estimates from the optimizer and ranks them using a *pair-wise* ranking model. We use the authors' implementation [2] with default settings.
- LogPQO [51]: LogPQO generates plan candidates from historical workloads and employs two methods for plan selection: a classification-based approach and a regression-based approach.
- PostgreSQL [5]: We include the default cost-based optimizer of PostgreSQL (version 13.1), which generates a single execution plan without learning-based enhancements.

Metrics. To answer **Q1**, we measure the cumulative running time of the best plan among the candidates for each test query. For **Q2**, we report end-to-end running time from query submission to result retrieval and also break down running time into *plan generation*, *execution*. For **Q3**, we evaluate end-to-end running time under data updates. For **Q4**, we report the offline *preprocessing* time.

System Settings. All methods are implemented on PostgreSQL 13.1 with pg_hint_plan. Experiments are run on CentOS 7.9 with Intel i7-7700 CPU, 64GB RAM, 1TB SSD, and NVIDIA RTX 3090. Timeout for execution plan generation is set to 500 seconds. During the pool construction, we set the iteration threshold s to 6 and error bound eb to 1.5 in Algorithm 1.

5.2 Plan Candidate Quality

5.2.1 Quality of Best Candidate. Table 3 reports the average speedup of the best plan candidates obtained by each method relative to the plan generated by PostgreSQL. Due to excessive running time of PostgreSQL on Stack, we omit its result for Stack and instead compute speedup relative to Bao. We observe the following: (1) Except for STATS, PLARQ consistently achieves the highest average speedup. For example, on JOB, PLARQ attains a speedup of 2.47, while the second-best

[4]Since the implementation of LTR [14] is not publicly available, we exclude it from our experimental comparison.

method reaches only 1.65. (2) Other methods show unstable speedup across different benchmarks. For instance, Bao ranks second on STATS but performs worst on Stack. (3) On STATS, Lero and PLARQ exhibit lower speedup than FASTgres and Bao due to a common limitation: both rely on inefficient strategies to guide cardinality adjustments with predefined scales in Lero or fixed number of iteration in PLARQ. Upon our profiling, they fail to identify high-quality plans in the queries with IDs 44, 45, 49, 101, 104, 112, 114, 116, where a Hash Join is preferred; instead, they select suboptimal plans with Nested Loop Join or Merge Join. As shown in Figure 3, on STATS, there is a noticeable increase in running time for queries with IDs around 50 and 100. In contrast, Bao and FASTgres can explicitly enable Hash Join via hints, allowing them to discover better plans in such cases.

Table 3. Average best-case speedup over PostgreSQL.

Dataset	Bao	Lero	FASTgres	LogPQO	PLARQ
JOB	1.45	1.63	1.65	1.51	2.47
CEB	1.62	2.0	2.22	2.33	2.88
Stack⋆	1.0	6.65	2.89	1.45	32.39
TPC-DS	4.06	1.61	4.12	1	5.53
STATS	6.25	2.94	6.89	7.7	2.75

⋆: we compute speedup relative to Bao due to the excessive time of PostgreSQL.

Figure 3 shows the cumulative running time of the best plan candidates across all test queries. Results are consistent with the speedup analysis: PLARQ achieves the best or second-best across all benchmarks except STATS. On STATS, we observe a noticeable increase in running time for queries around IDs 50 and 100, which aligns with suboptimal plan selection previously discussed.

Table 4. Number of unique plan candidates (#Uni.) and number of plan candidates (#All) generated by each method.

Method	JOB		CEB		Stack		TPC-DS		STATS	
	#Uni.	#All	#Uni.	#All	#Uni.	#All	#Uni.	#All	#Uni.	#All
Bao	3	5	4	5	3	5	3	5	4	5
Lero	9	38	16	47	5	32	11	27	6	20
FASTgres	19	64	27	64	23	64	21	64	22	64
LogPQO	2	8	4	29	1	2	1	2	4	17
PLARQ	3	5	4	5	2	5	2	5	3	5

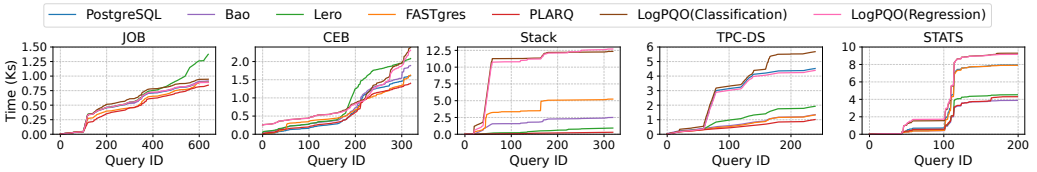


Fig. 4. Cumulative running time of end-to-end performance of different methods.

5.2.2 Statistics of Plan Candidates. Table 4 reports the average number of total and unique plan candidates generated by each method. We observe that PLARQ, Bao, and LogPQO consider significantly fewer candidates compared to Lero and FASTgres. When combined with the results in Table 3, which shows that PLARQ consistently achieves the best or second-best average speedup across most benchmarks, this demonstrates the precision of PLARQ's candidate selection strategy.

Specifically, PLARQ leverages a small, fixed set of high-quality candidates (typically 5 per query) to achieve strong performance. These findings underscore the effectiveness of selecting a compact, diverse, and relevant candidate set, a key design strength of PLARQ, rather than relying on exhaustive plan enumeration.

Discussion on RCE-based Candidate Generation. A recent study, Kepler [18], introduces a candidate generation strategy called *Row Count Evolution* (RCE). RCE is an evolutionary-style algorithm that discovers new plans by iteratively feeding perturbed cardinality estimates of sub-plans back into the optimizer. All plans obtained by different perturbations and queries of the same template are then merged to form a large candidate pool.

To better understand its behavior, we evaluate RCE on the first 25 templates of the JOB benchmark. We observe that RCE typically generates far more distinct plans than other methods studied in this paper. In 9 out of 25 templates, RCE produces over 100 unique plans. While such exhaustive exploration yields a rich pool of candidate plans, the actual speedup achieved by the best plan may remain moderate on our tested workloads. For instance, on template #24, RCE attains a higher speedup than PLARQ (6.14 vs. 4.19), whereas on template #15, it discovers 816 distinct plans but achieves a lower speedup (1.01 vs. 1.13). These results suggest that although RCE effectively enlarges the plan space, many of the generated plans may be redundant or only marginally better than existing ones.

5.3 End-to-end Performance

Figure 4 reports the end-to-end performance of each method, measured by the total time from query submission to result retrieval. As in Section 5.2, we omit PostgreSQL's result on Stack due to its extremely long running time. We can observe:

(1) PLARQ consistently achieves the lowest total running time on four out of five benchmarks. For example, on Stack and TPC-DS, it clearly outperforms all baselines.

(2) On STATS, PLARQ performs competitively with Bao, incurring noticeably higher running time only on queries with IDs near 175. Although the best plan candidates of PLARQ are significantly worse than those of Bao, the competitive overall performance highlights the effectiveness of our ranking module.

(3) On CEB and JOB, although Lero achieves shorter plan execution times than Bao, its overall end-to-end running time is higher. This is because Lero incurs significantly higher optimization overhead compared to Bao.

(4) Although learning-based methods often generate better plans than PostgreSQL (see Figure 3), their end-to-end performance can vary and may even be worse than PostgreSQL. For example, on CEB, Bao, Lero, and LogPQO have a longer running time than PostgreSQL.

The worse performance can be attributed to two main reasons. The first reason is that *the planning overhead is non-negligible, especially for queries with short running time.* Figure 5 breaks down the end-to-end running time into optimization (**Opt**) and plan running time (**Run**) for each method. As Figure 5 shows only the overall runtime breakdown, the optimization overhead of Bao, FASTgres, and PLARQ on short queries is not fully visible. For example, on CEB (Figure 4), the first 180 queries are lightweight, with running time ranging from 100 ms to 2200 ms under PostgreSQL. PostgreSQL's planning time is minimal (2–8 ms), whereas Bao and FASTgres require 50–150 ms, PLARQ takes 150–200 ms, and Lero ranges from 500–800 ms. In such cases, the high inference cost of learned methods can offset the benefits of improved plans. This highlights a key limitation: when query execution is fast, traditional optimizers remain competitive due to their low overhead.

The second reason is *the inaccurate plan selection through learning-based idea.* In return, it leads to suboptimal plan choices and degraded performance. For example, on Stack in Figure 4, FASTgres surprisingly underperforms Bao despite leveraging a larger hint space (64 vs. 5). This degradation

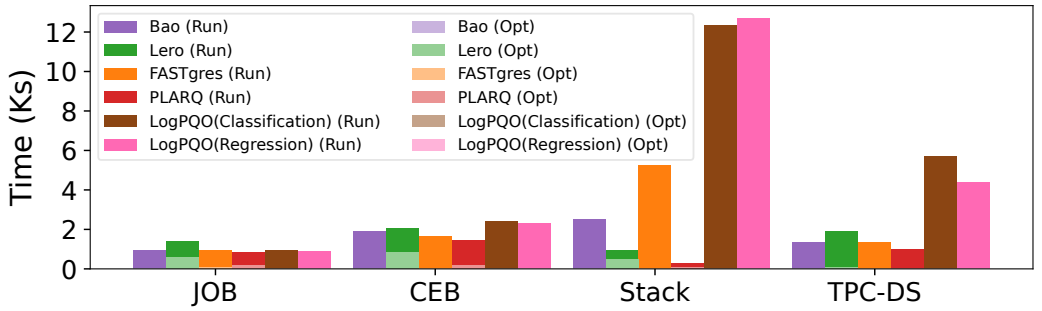


Fig. 5. Breakdown of end-to-end total running time.

is driven by several outlier queries with IDs 20, 45, 47, and 164, where FASTgres selects suboptimal hints, resulting in inefficient execution plans and running time spikes. Similar behavior is observed on STATS, where FASTgres exhibits irregular performance across templates.

These results highlight PLARQ's balance between efficiency and effectiveness. Because PCG produces a small yet high-quality set of plan candidates, the ranking cost remains bounded even for short-running queries. For longer-running queries, these high-quality candidates and accurate ranking lead to substantial end-to-end gains. As a result, PLARQ maintains consistent performance across diverse workloads.

5.4 Dynamic Study

Settings. As we focus on the PQO setting, our dynamic evaluation is designed to test data changes, which is the main challenge under this setting. Other dynamic scenarios remain interesting future work. We initialize the database with 50% of the tuples from each table and incrementally increase the data volume by 12.5% in the following subsequent four rounds, resulting in five total evaluation rounds (the first round evaluates the queries on the initial database). For query workload generation, we randomly select 10 query templates from STATS. In each round, 20 new queries are generated per template, yielding 200 queries in total across all templates. The models are retrained in every 400 queries being evaluated. For all intermediate rounds, the previously trained model (including the plan candidates pool) is reused without retraining, allowing us to test each method's ability to tolerate moderate data shifts.

Results. Figure 6 presents the cumulative end-to-end running time for each optimizer over 1,000 test queries. We observe the following: (1) Overall, PLARQ and Lero almost achieve the best performance across all queries, demonstrating strong and robust results. (2) PLARQ outperforms Bao on the first ~900 queries but is eventually overtaken by Bao in the final round. This is primarily due to the suboptimal quality of plan candidates on the full dataset in PLARQ, as also indicated in Figure 3, which offsets the earlier performance gains. (3) FASTgres shows performance comparable to PostgreSQL, offering limited improvements and exhibiting weaker adaptability to evolving data. Although it considers more hints than Bao and can potentially generate better plans, its performance degradation mainly results from its model's inability to reliably select effective hint sets, a trend also observed in Section 5.3.

5.5 Cost of Data Preparation

5.5.1 Cost of Training Data Construction. Training data is crucial to the effectiveness of learning-based query optimizers, and the overhead of generating this data significantly impacts how quickly

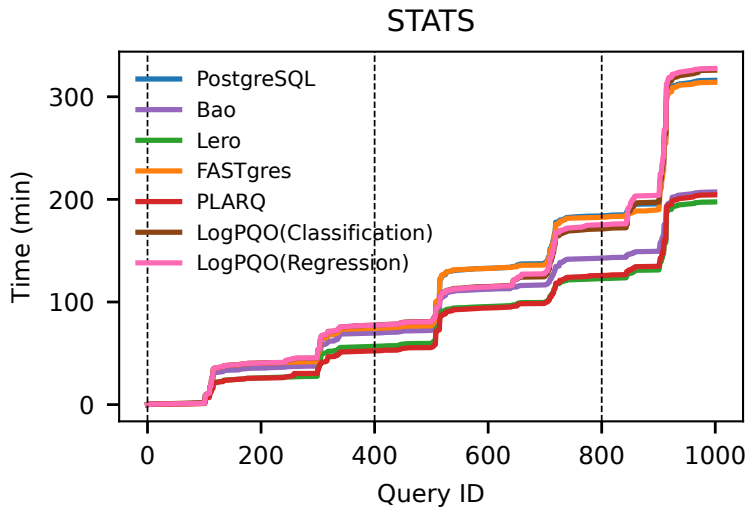


Fig. 6. End-to-End query running time under dynamic data changes.

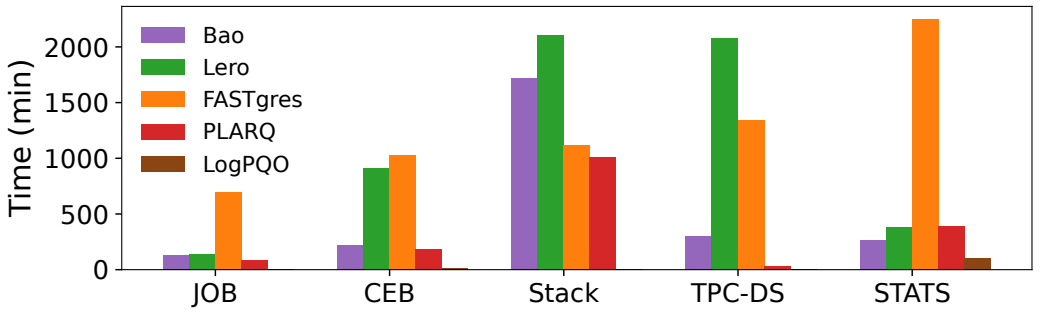


Fig. 7. Construction time of training data.

such systems can be deployed. In our study, the core process involves recording the running time of each query under a plan.

Figure 7 reports the training data construction cost of PLARQ, Lero, Bao, and FASTgres.[5] For Bao, we include the time required to collect the runtime of each training query under all hint configurations. Since LogPQO only relies on the optimizer's estimated cost rather than actual execution time, the collection overhead is extremely low. For instance, on the TPC-DS benchmark, the entire process completes within just 53 seconds. For the remaining methods, we observe that PLARQ consistently incurs the lowest overhead across all benchmarks. FASTgres exhibits significantly higher costs—ranging from approximately 750 to 1250 minutes. Lero performs even worse, with notably high costs on Stack and TPC-DS. Bao shows higher cost only on Stack, but its overall overhead remains lower than that of Lero and FASTgres.

These differences in training data construction cost can be attributed to two key factors: (1) the number of plans evaluated per query, and (2) the quality of these plans. Evaluating more plans per query naturally increases the total time. Additionally, lower-quality plans tend to have longer runtimes, further contributing to the cost. Compared to Bao and PLARQ, both Lero and FASTgres

[5]For all methods, the per-plan timeout is set to 500 seconds, except for FASTgres, which adopts its own adaptive timeout strategy.

verify significantly more plans per query, resulting in higher overhead. For example, on JOB, both Lero and FASTgres require over 750 minutes, while Bao and PLARQ finish within 250 minutes. Although Bao and PLARQ evaluate the same number of plans per query (five), the plan quality variance affects their construction time. On Stack, for instance, Bao incurs approximately 1750 minutes of overhead, whereas PLARQ completes in about 1000 minutes.

5.5.2 Offline Cost of Plan Candidates Pool Construction. Table 5 reports the offline time for constructing candidate pools across benchmarks, ranging from 10.5 hours on JOB to 111 hours on STACK. All pools were built sequentially – processing templates and their queries one by one – so the results represent an upper bound. As this process occurs once per template, the cost is easily amortized over repeated executions and can be further reduced with parallelization, confirming the practicality of PLARQ's offline phase.

Table 5. Time (hours) of plan candidates pool construction.

CEB	JOB	TPC-DS	STACK	STATS
50.51	10.53	41.76	111.0	97.63

5.6 Ablation Study on PLARQ

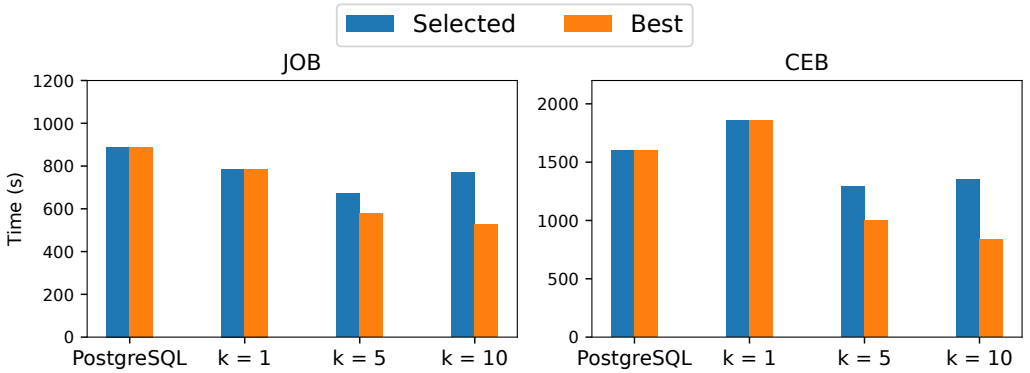


Fig. 8. Evaluating the impact of k.

5.6.1 Impact of k. We evaluate the impact of the candidate size k. Figure 8 reports the total running time of the fastest plan among the k candidates (**Best**) and the plan selected by PLARQ or PostgreSQL (**Selected**) for each query. Note that for PostgreSQL and PLARQ with $k = 1$, there is only one candidate plan, so the Best and Selected times are identical.

From Figure 8, we observe the following: (1) As k increases, the total running time of the fastest plan decreases, since a larger candidate size increases the likelihood of including a better plan. (2) When $k = 1$, PLARQ's selected plan can perform worse than the default PostgreSQL plan, particularly on CEB. This occurs when the test query lies in a sensitive region of the query space, i.e., the nearest neighbor from the pool yields a suboptimal plan. (3) While a larger k improves the upper bound (i.e., reducing the Best time), the total running time of plans selected by PLARQ tends to increase. This is due to (i) the inclusion of more diverse, and potentially poorer, plan candidates, and (ii) the increased difficulty for the model to accurately rank and select the optimal plan from a larger set. Therefore, a moderate value of $k = 5$ is chosen to strike a balance between candidate quality and ranking.

While it is theoretically possible to union the candidate plan spaces of multiple methods (e.g., FASTgres and PLARQ), we find this approach impractical. The union introduces substantial redundancy and increases online inference cost, as confirmed by our ablation in which larger candidate sets do not always yield better end-to-end performance.

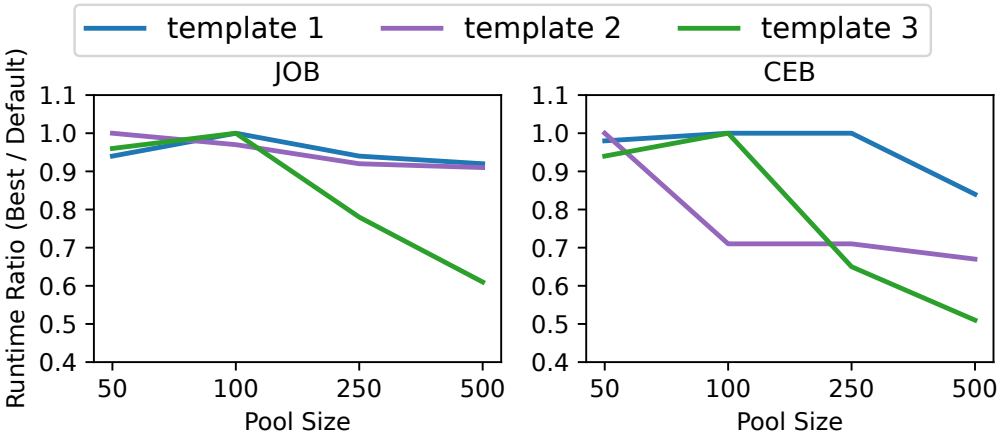


Fig. 9. Evaluating the influence of pool size.

5.6.2 Impact of Pool Size. Figure 9 shows the total running time of the best plan candidates for each query on three sampled templates from JOB and CEB. To enable clearer trend visualization across templates with varying time scales, we normalize the running times to the range $[0, 1]$ We observe that: (1) A larger pool size generally results in lower total running time, as it tends to include more unique plans in the pool, thereby increasing the likelihood of selecting a more efficient plan. (2) Interestingly, when increasing the pool size from 50 to 100, we observe a slight increase in running time on some templates. This is likely due to a few test queries for which the additional plans introduced at this pool size are suboptimal, slightly offsetting the overall gain.

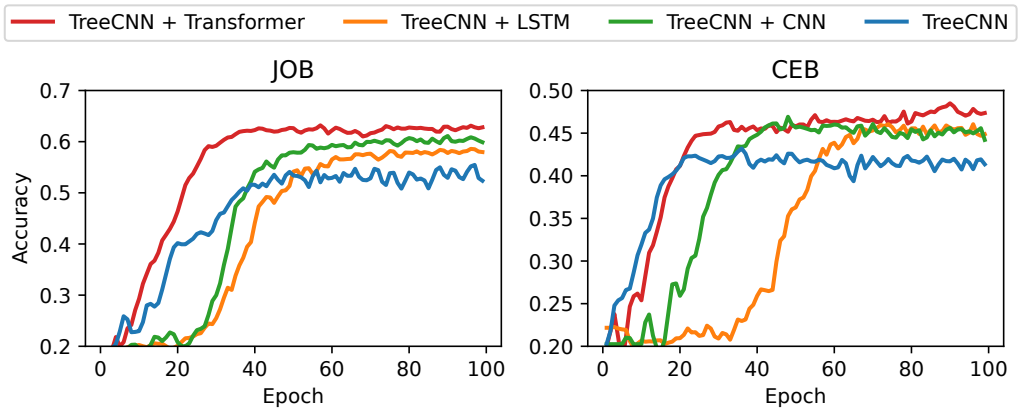


Fig. 10. Evaluating the influence of different models.

5.6.3 Ranking Model Design. We evaluate the effectiveness of different models in capturing inter-plan relationships for plan ranking. Specifically, we compare four variants: TreeCNN alone (which

does not model inter-plan relationships), and TreeCNN combined with Transformer (attention-based), LSTM, or CNN that explicitly captures such relationships. Figure 10 reports the ranking accuracy (i.e., the ability to correctly identify the best plan candidate). We observe that: (1) explicitly modeling inter-plan relationships significantly improves ranking accuracy; (2) among all variants, TreeCNN+Transformer achieves the highest accuracy and most stable convergence, demonstrating its superior capability in leveraging inter-plan dependencies; (3) TreeCNN+LSTM exhibits slower and less stable convergence, particularly on the JOB benchmark, suggesting that LSTM is less effective for capturing complex inter-plan relationships. (4) Moreover, TreeCNN+Transformer achieves fast convergence in coverage, reaching stable performance within approximately 30 epochs.

6 Related Work

Recently, a growing number of studies have explored learning-based methods for query optimization. These approaches can be broadly categorized into two groups:

Learned Optimizer Components. This category encompasses four main tasks: learned cardinality estimation, learned cost modeling, learned join order selection, and learned query rewriting. Since errors in cardinality estimation are a primary cause of suboptimal plans [31], the majority of learning-based query optimization work falls into cardinality estimation. Early studies [26] adopt a workload-driven approach, where models are trained on a specific query workload. However, such models tend to become outdated when the workload shifts. To overcome the limitations, several studies work on how to train a robust model [32, 44]. In contrast, data-driven methods [23, 36, 54, 59, 60, 62, 63, 71] aim to learn the underlying data distribution, making them more robust to workload changes. Beyond traditional relational data, several recent efforts have also tackled cardinality estimation for string data [13, 28, 46, 65] and high-dimensional data[30, 49, 56, 57]. Learned cost estimation leverages machine learning models to estimate the execution cost of a given query plan. Many methods [37, 40, 48] design models that capture the hierarchical (tree) structure of query plans to improve estimation accuracy. The study in [47] heuristically introduces a set of features, such as the number of containers, and trains a model to predict cost based on these handcrafted features. Learned join order selection aims to identify an effective join order for queries involving joins. Existing studies [22, 27, 39, 50, 64] formulate this task as a reinforcement learning problem and adopt various RL-based approaches to solve it. Learned query rewriting applies learning-based techniques to transform an input SQL query into a semantically equivalent form with improved execution performance. LearnedRewrite [69] employs Monte Carlo Tree Search (MCTS), guided by learned cost-improvement models, to efficiently explore the exponential space of possible rewrites. Motivated by the impressive reasoning capabilities of large language models (LLMs) [41], several recent approaches[17, 33, 34] leverage LLMs to rewrite queries using carefully designed prompts, in-context demonstrations (e.g., historical rewrite examples), and selected rewrite rules.

Learned Query Optimizer. As aforementioned in Section 1, methods in this category aim to directly produce execution plans, either by replacing traditional query optimizers or by augmenting them with learned components to improve plan quality. Bao [37] enhances the optimizer by leveraging learned hints to guide plan generation. It employs a Tree Convolutional Neural Network (TreeCNN) to predict the performance of candidate plans under different hint configurations, ultimately selecting the most promising one. FASTgres [58] also adopts a hint-based approach, but extends Bao by considering a broader variety of hint types. It formulates the task as a classification problem, training a model to directly predict the optimal hint set for each query. Kepler [18] generates plan candidates using Row Count Evolution (RCE). RCE is an evolutionary-style algorithm that discovers new plans by iteratively feeding perturbed cardinality estimates of sub-plans back into the optimizer. This process allows it to explore a wider plan space than the default optimizer.

For plan selection, Kepler then adopts a robust classification model to predict the fastest plan for an incoming query based on its parameters. Same as ad Kepler, Lero [70] also generate the plan candidates by injecting learned cardinality estimates into the optimizer. However, differently, it directly revises the cardinality estimation on the default plan generated by the optimizer instead of a multi-generation process. It then performs plan selection through a pair-wise ranking strategy, comparing plans based on execution efficiency. LTR [14] integrates the learning-to-rank paradigm into the plan enumeration phase. It selects both full and partial plans during enumeration, with candidate plans generated by the underlying optimizer's enumeration algorithm, thus offering tighter integration with traditional planning mechanisms. Balsa [61] proposes a fully learned optimizer that avoids dependence on expert cost models or pre-collected execution plans. Instead, it bootstraps the learning process using a simple simulator with a minimal cost model, and gradually refines its value network through safe exploration and execution in a real environment.

Parametric Query Optimization. Traditional studies on PQO [12, 15, 16, 19, 20] aim to generate efficient execution plans for parameterized queries by modeling how plan optimality varies with parameters. Early work partitions the selectivity space to find representative plans covering all regions [20]. Later methods such as Ellipse [15], density-based clustering [12], and SCR [19] introduce heuristic solutions under simplified cost assumptions, while others [16, 51] rely on heuristic search.

7 Conclusion

In this paper, we first unify existing practical learned query optimizers under a common two-stage framework and identify key limitations that hinder their effectiveness in real-world settings. To overcome these challenges, we propose PLARQ, a new learned optimizer designed within this framework. In the Plan Candidate Generation stage, PLARQ maintains a carefully constructed query pool and employs a tailored similarity function to retrieve the top-k most similar queries, enabling the selection of a compact, diverse, and high-quality candidate plan set. In the Plan Ranking stage, PLARQ adopts a list-wise ranking approach that models inter-plan relationships to more accurately identify the optimal execution plan. Extensive experiments demonstrate that PLARQ achieves substantial end-to-end performance gains, delivering significant speedups over PostgreSQL.

Acknowledgments

This project is supported in part by ARC FT240100832 and DP240101211. Yuwei Peng is supported in part by the National Key Research and Development Program of China under Grant No. 2023YFB4503604. Yuwei Peng is the corresponding author.

References

[1] n.d.. Bao's Source Code. https://github.com/learnedsystems/BaoForPostgreSQL.
[2] n.d.. Lero's Source Code. https://github.com/AlibabaIncubator/Lero-on-PostgreSQL.
[3] n.d.. Oracle. https://www.oracle.com/.
[4] n.d.. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.
[5] n.d.. PostgreSQL. http://www.postgresql.org/.
[6] n.d.. SQL Server. https://www.microsoft.com/en-au/sql-server/sql-server-2019.
[7] n.d.. Stack. https://rmarcus.info/stack.html.
[8] n.d.. STATS. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark.
[9] n.d.. TPC-DS. https://www.tpc.org/tpcds/.
[10] n.d.. TPC-H. https://www.tpc.org/tpch/.
[11] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*. IEEE Computer Society, 390–401.
[12] Günes Aluç, David DeHaan, and Ivan T. Bowman. 2012. Parametric Plan Caching Using Density-Based Clustering. In *ICDE*. IEEE Computer Society, 402–413.
[13] 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.
[14] Henriette Behr, Volker Markl, and Zoi Kaoudi. 2023. Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization. In *BTW (LNI, Vol. P-331)*. Gesellschaft für Informatik e.V., 535–554.
[15] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. Knowl. Data Eng.* 21, 4 (2009), 582–594.
[16] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance aware optimization of parameterized queries. In *SIGMOD*. ACM, 531–542.
[17] Sriram Dharwada, Himanshu Devrani, Jayant R. Haritsa, and Harish Doraiswamy. 2025. Query Rewriting via LLMs. *CoRR* abs/2502.12918 (2025).
[18] 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.
[19] Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2017. Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees. In *SIGMOD*. 1539–1554.
[20] Sumit Ganguly. 1998. Design and Analysis of Parametric Query Optimization Algorithms. In *VLDB*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). 228–238.
[21] 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.
[22] Jonas Heitz and Kurt Stockinger. 2019. Join Query Optimization with Deep Reinforcement Learning Algorithms. *CoRR* abs/1911.11689 (2019).
[23] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.
[24] Alekh Jindal and Jyoti Leeka. 2022. Query Optimizer as a Service: An Idea Whose Time Has Come! *SIGMOD Rec.* 51, 3 (2022), 49–55.
[25] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *SIGMOD*. ACM, 1214–1227.
[26] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.
[27] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).
[28] Suyong Kwon, Kyuseok Shim, and Woohwan Jung. 2025. Cardinality Estimation of LIKE Predicate Queries using Deep Learning. *Proceedings of the ACM on Management of Data* 3 (02 2025), 1–26. doi:10.1145/3709670
[29] 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.
[30] Hai Lan, Shixun Huang, Zhifeng Bao, and Renata Borovica-Gajic. 2024. Cardinality Estimation for Similarity Search on High-Dimensional Data Objects: The Impact of Reference Objects. *Proc. VLDB Endow.* 18, 3 (2024), 544–556.
[31] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the Vldb Endowment* 9, 3 (2015), 204–215.
[32] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *SIGMOD, 2022*. ACM, 1920–1933.

[33] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *Proc. VLDB Endow.* 18, 1 (2024), 53–65.
[34] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. *CoRR* abs/2403.09060 (2024).
[35] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Found. Trends Inf. Retr.* 3, 3 (2009), 225–331.
[36] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. *Proc. VLDB Endow.* 15, 3 (2021), 414–426.
[37] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. ACM, 1275–1288.
[38] 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.
[39] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD 2018*. ACM, 3:1–3:4.
[40] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.
[41] Shervin Minaee, Tomás Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. *CoRR* abs/2402.06196 (2024).
[42] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. AAAI Press, 1287–1293. https://doi.org/10.1609/aaai.v30i1.10139
[43] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032.
[44] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533.
[45] 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.
[46] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. *Proc. VLDB Endow.* 14, 4 (2020), 471–484.
[47] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*. ACM, 99–113.
[48] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.
[49] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD*. ACM, 1745–1757.
[50] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. ACM, 1153–1170.
[51] 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.
[52] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706.
[53] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD 2018*. ACM, 1:1–1:6.
[54] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84.
[55] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.
[56] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Xin Cao, Yifang Sun, Wei Wang, and Makoto Onizuka. 2020. Monotonic Cardinality Estimation of Similarity Selection: A Deep Learning Approach. In *SIGMOD*. ACM, 1197–1212.
[57] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Rui Mao, Makoto Onizuka, Wei Wang, Rui Zhang, and Yoshiharu Ishikawa. 2021. Consistent and Flexible Selectivity Estimation for High-Dimensional Data. In *SIGMOD*. ACM, 2319–2327.
[58] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.* 16, 11 (2023), 3310–3322.
[59] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2009–2022.

[60] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *CoRR* abs/2012.14743 (2020).
[61] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD, 2022*. ACM, 931–944.
[62] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.
[63] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.
[64] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308.
[65] Yirui Zhan, Wen Nie, and Jun Gao. 2025. SSCard: Substring Cardinality Estimation using Suffix Tree-Guided Learned FM-Index. *arXiv preprint arXiv:2505.24312* (2025).
[66] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2023. AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2621–2633.
[67] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc T. Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *SIGMOD 2022*. ACM, 2299–2311.
[68] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.* 17, 4 (2023), 823–835.
[69] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.
[70] 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.
[71] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502.

Received July 2025; revised October 2025; accepted November 2025