PAR2QO: Parametric Penalty-Aware Robust Query Optimization

Haibo Xiu
Duke University
Durham, NC, USA
haibo.xiu@duke.edu

Yang Li
Duke University
Durham, NC, USA
yang.li@duke.edu

Qianyu Yang
Duke University
Durham, NC, USA
qianyu.yang@duke.edu

Pankaj K. Agarwal
Duke University
Durham, NC, USA
pankaj@cs.duke.edu

Jun Yang
Duke University
Durham, NC, USA
junyang@cs.duke.edu

ABSTRACT

Parametric Query Optimization (PQO) is an important problem in database systems, yet existing approaches suffer from high training costs, sensitivity to estimation errors, and vulnerability to severe performance regressions. This paper introduces **PAR^{2}QO** (**PAR**ametric **P**enalty-**A**ware **R**obust **Q**uery **O**ptimization), a system that integrates robust query optimization into PQO. PAR^{2}QO strategically obtains plans from a well-balanced set of probe locations informed by the workload, and caches them as plan-penalty profiles. At runtime, PAR^{2}QO selects the plan with the lowest expected penalty, explicitly accounting for selectivity uncertainties. Extensive experiments show that PAR^{2}QO delivers significant speedups over existing methods while ensuring robustness against performance degradation. Additionally, we introduce **CARVER**, a workload generator aimed at covering possible cardinalities of subqueries. Not only does CARVER provide a more comprehensive way to evaluate PQO methods, but when used for training learned methods, it can also enhance their generalizability and stability.

PVLDB Reference Format:
Haibo Xiu, Yang Li, Qianyu Yang, Pankaj K. Agarwal, and Jun Yang.
PAR2QO: Parametric Penalty-Aware Robust Query Optimization. PVLDB, 18(11): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Hap-Hugh/PAR2QO.

1 INTRODUCTION

A long-standing challenge in databases is *Robust Query Optimization* (*RQO*) [12]. Query optimizers rely on selectivity and cost estimates to pick execution plans. When these estimates are inaccurate, the chosen plan can perform far worse than the optimal. There are various approaches towards robust query optimization; we focus on finding a *robust query plan* in this work: Instead of minimizing the plan cost given the estimated selectivities, which can be wrong, we look for a plan that "performs well" despite uncertainties in the estimates. In other words, when executed at the true selectivities, the plan is expected to incur a low penalty with respect to the cost of the true optimal plan. The resulting optimization problem is challenging because we must consider possible estimation errors and compare with the real optimal plans at the corrected selectivities, which amounts to exploring a neighborhood — rather than a single point — in the selectivity space. The optimization overhead can thus be higher than that for traditional query optimization.

Parametric Query Optimization (*PQO*) is a technique originally motivated by the desire to reduce optimization overhead for queries that follow the same template but differ in their parameter settings, which frequently arise in applications. Once PQO finds the optimal plan for a given query instance, it caches the plan and reuses it for future query instances whose estimated selectivities are close to the cached plan. Pushing the idea further, Kepler [15], the latest state of the art in PQO, employs deep learning to train a model that selects the right plan among a set of cached candidates.

Combining PQO and RQO is natural, because PQO helps to amortize query optimization overhead across multiple query instances. The potential of this combination was demonstrated in the recent RQO work of PARQO [50], which considered caching one robust plan per query template. However, opportunities for such reuses are limited because the choice of the robust plan varies across the selectivity space. There are better ways to reuse the computation in RQO: intuitively, RQO explores the neighborhood of each query instance; even if two query instances are not close enough to justify the same robust plan, the work of exploring their neighborhoods can still be shared.

Existing PQO approaches, on the other hand, are not designed to find robust plans. Some approaches have heuristics that favor caching plans that tend to be robust, e.g., plans that are close to being optimal at multiple points in the selectivity space. However, these approaches do not consider robustness when selecting plans at runtime. Furthermore, their ability to cache a robust plan depends heavily on what candidates were considered for caching in the first place. As a result, their candidate plans are either too focused on training queries (e.g., [44]) without accounting for errors in selectivity estimates, too spread across the entire selectivity space (e.g., [13]), and/or very expensive to acquire (e.g., Kepler [15], or [13] in high dimensions).

To overcome these limitations, we propose a new PQO system, *PAR^{2}QO* (***PAR****ametric* ***P****enalty-****A****ware* ***R****obust* ***Q****uery* ***O****ptimization*).

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:XX.XX/XXX.XX

This work differs from classical PQO approaches, which primarily aim to amortize optimization overhead by caching point-wise optimal plans, without properly accounting for robustness. It also distinguishes itself from traditional RQO approaches, which focus on finding a single robust plan for a given query instance, without considering reuse opportunities across the broader selectivity space. PAR^{2}QO combines both: it aims to produce robust plans as RQO, but leveraging PQO to amortize optimization overhead. Importantly, PAR^{2}QO offers a systematic treatment of robustness, guided by a formal definition and development of new PQO and RQO techniques to enable their efficient integration. It is practical to deploy, requiring only non-intrusive access to existing query optimizers and reasonable training and setup time. It also makes its decisions in a transparent manner, with no "black-boxing," which is easy to understand and debug. Specifically, we make the following technical contributions in this paper:

- PAR^{2}QO employs a novel method to sample the selectivity space to acquire candidate plans. Leveraging the training workload and a learned model of selectivity estimation errors, PAR^{2}QO is able to focus on the most relevant regions of the selectivity space — not only points corresponding to the estimated selectivities of training queries, but also in the neighborhoods induced by uncertainty in these estimates.
- PAR^{2}QO uses a novel caching strategy in the form of *plan-penalty profiles*, for a carefully selected subset of candidate plans. Instead of caching only the plans themselves, this data structure also caches their cost profiles at sampled selectivities, which allows the best robust plan to be computed at runtime, using a rigorously defined robustness objective. Instead of relying on a black box to predict the most robust plan, the optimization decision is explainable using the error model, plan profiles, and robustness objective.
- By focusing its learning efforts on selectivity errors observed from training queries, PAR^{2}QO uses only a tiny fraction of the training cost of Kepler. PAR^{2}QO's learned error model informs the generation of candidate plans at preparation time and plan selection at runtime, and is able to achieve better query performance than Kepler despite shorter training time.
- In the process of evaluating our approach and others, we observed that standard benchmarks such as JOB [31] and DSB [11] are lacking in the PQO setting, because they either offer no guidance on how to generate parameter settings for a query template, or generate them from simple distributions that fail to stress-test PQO. Therefore, we propose a new query workload generator, *CARVER* (***CAR****dinality co****VER****age*), aimed at generating queries to provide good coverage of the range of possible subquery cardinalities that the optimizer might encounter. When used as training workloads, CARVER also improves the robustness of PQO and its ability to generalize to new workloads.

Overall, by defining the robust optimization objective clearly and designing methods accordingly, PAR^{2}QO is able to guard against query performance degradations due to selectivity errors and deliver better overall query performance than previous approaches. Experiments show that PAR^{2}QO consistently outperforms existing methods across various benchmarks and workloads, achieving up to 1.96× speedup over PostgreSQL and 1.83× over Kepler on JOB. It avoids severe regressions, incurs acceptable training cost, and supports fast inference times. Moreover, PAR^{2}QO maintains robust performance under shifting data and query distributions.

2 PRELIMINARIES AND BACKGROUND

2.1 Problem Statements

Prerequisites. As discussed in Section 6, there are many different approaches to robust query optimization, e.g., from worst-case optimal join algorithms to runtime adaptive query processing, but this paper targets query optimizers that generate traditional (non-adaptive) query plans based on cardinality estimates. Our solution is designed to work on existing database systems without changing their plan space, execution engine, or optimization procedure. We only assume that the system supports (or can be easily extended to support) the following calls. First, given a query Q, $\widehat{\mathsf{Sel}}(Q)$ returns a vector of selectivity estimates for subqueries of Q. Second, $\mathsf{Opt}(Q, \mathbf{s})$ returns the optimal execution plan for query Q given the selectivity vector $\mathbf{s}$. Third, $\mathsf{Cost}(\pi, \mathbf{s})$ returns the estimated cost of execution plan π given the vector $\mathbf{s}$ of relevant selectivities. Typically, the cost of optimization is dominated by Opt; individual $\widehat{\mathsf{Sel}}$ and Cost calls are far cheaper in comparison [17]. For our PostgreSQL-based implementation, we use `EXPLAIN` to obtain estimates of costs and relevant selectivities, the same mechanism in earlier work [21, 50] for selectivity injection, and `pg_hint_plan` [38] for specifying the plan.

Parametric Query Optimization (PQO). A query template Γ has a set $\mathsf{params}(\Gamma)$ of query parameters drawn from various domains. A query (instance) Q following Γ, denoted by $Q \sim \Gamma$, assigns a specific set $\mathsf{params}(Q)$ of values to the set of parameters in $\mathsf{params}(\Gamma)$. To avoid repeating expensive Opt calls for workloads with many instances of the same query template, PQO precomputes and/or maintains information for Γ that enables it to efficiently find, for a given query $Q \sim \Gamma$, an execution plan whose cost is not much higher than that of $\mathsf{Opt}(Q, \widehat{\mathsf{Sel}}(Q))$, without making an Opt call.

Typically, PQO uses a per-template *plan cache*, which contains a set of *candidate plans* for the template, and selects the best plan from them for each incoming query at runtime.

Robust Query Optimization (RQO). In reality, when optimizing a query Q, the system does not have the true selectivities $\mathbf{s}$, but instead their estimates $\hat{\mathbf{s}}$. As a result, it may select a plan whose execution cost is much higher than that of the true optimal plan.To quantify the penalty of making a wrong selection and to define a "robust" plan formally, we adopt the following general framework of PARQO [50] based on stochastic optimization, which is flexible enough to encode many notions of robustness found in previous work and applications [1, 4, 48].

We assume a user-defined *penalty function* $\mathsf{Penalty}(\pi, \mathbf{s})$ specifying the penalty incurred by executing a potentially non-optimal plan π under the true selectivities $\mathbf{s}$. We use the same default as PARQO, but note that our approach works for any choice of $\mathsf{Penalty}$:

$$\mathsf{Penalty}(\pi, \mathbf{s}) = \begin{cases} 0 & \text{if } \mathsf{Cost}(\pi, \mathbf{s}) \le (1+\tau) \cdot \mathsf{Cost}^\star(\mathbf{s}), \\ \mathsf{Cost}(\pi, \mathbf{s}) - \mathsf{Cost}^\star(\mathbf{s}) & \text{otherwise.} \end{cases} \quad (1)$$

Here, $\mathsf{Cost}^\star(\mathbf{s}) = \mathsf{Cost}(\mathsf{Opt}(Q, \mathbf{s}), \mathbf{s})$ is the (surrogate) optimal cost, approximated by the estimated cost of the best plan obtained by the optimizer assuming the knowledge of the true selectivities $\mathbf{s}$; $\mathsf{Cost}(\pi, \mathbf{s})$ is the estimated cost of π when executing under $\mathbf{s}$. Equation (1) defines penalty to be proportional to the amount of cost exceeding the optimal, but only if it is beyond the prescribed tolerance ($\tau = 0.2$ following the convention [1, 10, 13, 48, 50]).

Since selectivities are uncertain, we model them as a random vector S. Assuming that $\mathrm{S} \sim f(\mathbf{s}|\hat{\mathbf{s}})$, the probability density function for the distribution of true selectivities $\mathbf{s}$ conditioned on the estimates $\hat{\mathbf{s}}$, we can then define the *most robust* plan for Q as the one that minimizes the expected penalty $\mathsf{E}[\mathsf{Penalty}(\pi, \mathrm{S}) \mid \widehat{\mathsf{Sel}}(Q)]$.

Parametric Robust Query Optimization. The problem is defined as follows. Given a query template Γ, a (training) database instance $\mathcal{D}$, and a (training) query workload W of queries following Γ:

- (Cache preparation) Compute a data structure from $\mathcal{D}$ and W.
- (Runtime plan selection) Given a query $Q \sim \Gamma$, use the cached data structure to find a robust plan for Q.

We first present two baseline solutions in Section 2.2 and then discuss our main solution in Section 3.

Parametric Query Workload Generation. In a "cold-start" variant of the above setup, the training query workload W is not given. This variant requires addressing the following problem: given a query template Γ and a database instance $\mathcal{D}$, generate query workload W following Γ. Informally, we would like the generated workload to serve as a good workload for training and/or testing a parametric robust query optimizer. To this end, the generated queries should provide good coverage of the various selectivity scenarios that the optimizer can potentially encounter, including situations when the data distribution may change and the query distribution may be different. We present our solution in Section 4.

2.2 Baselines: PARQO and PARQO′

PARQO [50], a state-of-the-art robust query optimization framework, proposes caching one robust plan per template and reusing it opportunistically at runtime. This approach serves as a rather weak baseline for PQO; hence, we additionally propose a stronger baseline called PARQO′, which elevates it into a full-fledged solution for parametric robust query optimization. We first briefly review the components of PARQO below and then describe PARQO′.

Error Profiling. PARQO uses the database instance $\mathcal{D}$ and the query workload W to learn an error model $\mathcal{E}$ for query template Γ that allows us to construct the conditional distribution of true selectivities $f(\mathbf{s}|\hat{\mathbf{s}})$ given the estimated selectivities $\hat{\mathbf{s}} = \widehat{\mathsf{Sel}}(Q)$ for a query $Q \sim \Gamma$. In general, optimizing a complex query may involve many selectivities, from single-table selections to multi-table selection-join subqueries; it would be infeasible to acquire all possible selectivities and represent the full joint distribution of all their errors. Therefore, PARQO chooses to model $\mathcal{E}$ using a collection of *error profiles*, each capturing a *querylet*, i.e., a small subquery template (up to a three-table selection-join) in Γ. We collect these profiles by comparing the optimizer estimates with the real subqueries cardinalities observed by executing W over $\mathcal{D}$. Each error profile contributes a factor to the joint distribution $f(\mathbf{s}|\hat{\mathbf{s}})$.

Finding a Robust Plan. Given a plan $Q \sim \Gamma$ and $\hat{\mathbf{s}} = \widehat{\mathsf{Sel}}(Q)$, PARQO computes a robust plan for Q per Equation (1) as follows. First, to tame the complexity of subsequent optimization, PARQO identifies a small number of *sensitive dimensions* among those of $\hat{\mathbf{s}}$ whose uncertainty impacts the penalty the most. This dimensionality reduction is guided by a careful sensitivity analysis that assesses each dimension's contribution to the overall variance in penalty, using Monte-Carlo sampling from $f(\mathbf{s}|\hat{\mathbf{s}})$. Second, PARQO draws a number of selectivity samples from $f(\mathbf{s}|\hat{\mathbf{s}})$ restricted to the sensitive dimensions, and invokes Opt with sampled selectivities to obtain a set of candidate plans. Finally, PARQO returns the candidate plan with the least expected penalty, estimated by recosting the plan at sampled selectivities and comparing its costs with the respective optimal plans at these locations.

Parametric Robust Query Optimization with PARQO. The method above is designed to find a robust plan for a single query instance. Since the overhead of finding a robust plan is high, PARQO caches a robust plan per query template. Let Q_{c} denote the query that the cached robust plan was optimized for, and let $\hat{\mathbf{s}}_{\mathsf{c}}$ denote the estimated selectivities for Q_{c}. At runtime, given a new query Q with the same template and estimated selectivities $\hat{\mathbf{s}}$, PARQO tests if the Kullback–Leibler (KL) divergence of $f(\mathbf{s}|\hat{\mathbf{s}})$ from $f(\mathbf{s}|\hat{\mathbf{s}}_{\mathsf{c}})$ is below a prescribed threshold θ_{KL}. If yes, Q and Q_{c} will be almost indistinguishable from the perspective of a cardinality-based optimizer, so PARQO simply returns the cached plan for Q. Otherwise, PARQO falls back to calling $\mathsf{Opt}(Q, \hat{\mathbf{s}})$ to obtain a plan for Q, with no guarantee of robustness.

PARQO′: A Stronger Baseline. We extend PARQO to obtain a strong baseline for PQO as follows. To prepare the cache:

- Learn an error model $\mathcal{E}$ from $\mathcal{D}$ and W.
- Construct $\mathcal{C} = \{\langle \pi_i, \mathbf{s}_i \rangle\}$, which consists of a set of candidate robust plans (π_i) along with the respective selectivities ($\mathbf{s}_i$) they were obtained under. Starting with an empty $\mathcal{C}$, we iterate through each query $Q \in W$ with estimated selectivities $\hat{\mathbf{s}}$. We calculate the KL divergence of $f(\mathbf{s}|\hat{\mathbf{s}})$ from $f(\mathbf{s}|\mathbf{s}_i)$ for each candidate $\langle \pi_i, \mathbf{s}_i \rangle \in \mathcal{C}$. If the divergence is below θ_{KL} for at least one existing candidate, meaning that Q is already "covered" by $\mathcal{C}$, we move on. Otherwise, we find a robust plan π for Q and add $\langle \pi, \hat{\mathbf{s}} \rangle$ to $\mathcal{C}$.

At runtime, given a query Q, we find the candidate $\langle \pi_i, \mathbf{s}_i \rangle \in \mathcal{C}$ that minimizes the KL divergence of $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ from $f(\mathbf{s}|\hat{\mathbf{s}}_i)$, and return π_i as a robust plan for Q. No Opt or Cost calls are needed.

A main issue with PARQO′ is that in order for a query Q to reuse a cached plan π_i reliably, $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ must be very similar to $f(\mathbf{s}|\mathbf{s}_i)$, so many $\langle \pi_i, \mathbf{s}_i \rangle$ may be needed to cover the entire selectivity space. However, finding each robust plan to add to the cache is expensive, which makes the cache preparation cost high.

3 PARAMETRIC PENALTY-AWARE ROBUST QUERY OPTIMIZATION

A key insight into beating the baseline in Section 2.2 is to reuse the computation involved in finding a robust plan, rather than reusing the result plans directly. We illustrate the opportunities for reusing computation with a simple conceptual example. For brevity, we will refer to $\widehat{\mathsf{Sel}}(Q)$ as the "location" of query Q in the selectivity space. Intuitively, $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ defines a "neighborhood" around (but not necessarily centered at) Q, and finding a robust plan for Q involves exploring this neighborhood. Suppose PARQO′

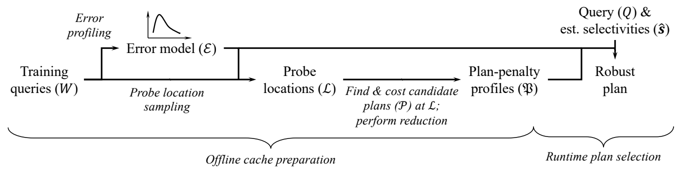

Error profiling
Error model ($\mathcal{E}$)
Query (Q) & est. selectivities ($\hat{\mathbf{s}}$)
Training queries (W)
Probe location sampling
Probe locations ($\mathcal{L}$)
Find & cost candidate plans ($\mathcal{P}$) at $\mathcal{L}$; perform reduction
Plan-penalty profiles ($\mathfrak{P}$)
Robust plan
Offline cache preparation
Runtime plan selection

Figure 1: PAR^{2}QO workflow.

has cached two plans π_1 and π_2 whose neighborhoods are both near Q's, but neither is close enough for Q to justify reuse. Nonetheless, the two neighborhoods together provide sufficient coverage for Q's neighborhood. Recall that finding π_1 and π_2 in the first place required sampling plans in their neighborhoods. The same samples should be useful for finding a robust plan for Q.

This insight leads us to our new solution PAR^{2}QO. It changes the strategy of caching robust plans for previously seen queries of a template to caching a data structure $\mathfrak{P}$ called the *plan-penalty profiles*. On a high level, $\mathfrak{P}$ compactly represents how a set of robust plan candidates performs relative to the optimal plans (per Equation (1)) over a set of *probe locations* in the selectivity space. PAR^{2}QO intelligently samples probe locations and uses the optimal plans to seed the set of candidate plans. This set may be too big, so for efficiency, PAR^{2}QO further reduces this set to construct $\mathfrak{P}$. At runtime, given a query Q, PAR^{2}QO uses $\mathfrak{P}$ to estimate the expected penalties of candidate plans over $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$, by reweighing per-location penalties to remove sampling bias, and selects the best.

While the high-level idea is simple, interesting challenges lie in how to design an effective sampling scheme for probe locations and how to reduce the candidate plan set in order to facilitate the identification of robust plans at runtime. In the remainder of this section, we describe the details of PAR^{2}QO, whose workflow is depicted in Figure 1. Cache preparation proceeds in three steps:

- Given $\mathcal{D}$ and query workload W, learn an error model $\mathcal{E}$. This step is the same as the error profiling step discussed in Section 2.2.
- Sample a set $\mathcal{L}$ of probe locations (Section 3.1).
- Find and cost candidate plans at $\mathcal{L}$, and reduce/summarize the information into the plan-penalty profiles $\mathfrak{P}$ (Section 3.2).

Selection of a robust plan at runtime is covered in Section 3.3. Finally, Section 3.4 presents an in-depth discussion of PAR^{2}QO by contrasting it with other state-of-the-art approaches, highlighting the novelty of our solution and how it represents a more principled approach to achieving plan robustness, whose advantages will also be empirically demonstrated in Section 5.

3.1 Sampling Probe Locations

There are two natural ideas. First, sampling should be informed by the query workload W because, intuitively, we do not want to devote much effort to regions that do not matter to queries. Second, for each query Q, we want to probe not only the location $\widehat{\mathsf{Sel}}(Q)$, but also locations in the neighborhood defined by $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$, because they might yield the true optimal plan for Q.

A naive approach is to randomly sample some locations in the neighborhood of each Q, but there are two pitfalls. First, if the workload W is skewed, some regions of the selectivity space will be over-sampled. Such high concentrations of samples are wasteful because they have diminishing returns on improving our understanding of these regions. Second, overemphasis on popular queries diverts resources from handling less common queries that may turn out to be costly and stand to benefit more from robust plans.

Our idea is to employ a check akin to the reuse check in PARQO′ (Section 2.2) to prevent over-sampling in a region: if a query "hits" a region previously sampled, we do not sample more from this region. We also track how often a region is hit by queries in W, which empirically captures the query distribution and allows us to estimate the sampling probabilities of probe locations, useful for bias correction later (Section 3.3).

The algorithm proceeds as follows. We organize the set of sampled probe locations into clusters $\{\langle \mathcal{L}_i, \mathbf{s}_i, h_i \rangle\}$, where $\mathcal{L}_i$ denotes the set of probe locations in cluster i, $\mathbf{s}_i$ serves to remember the distribution $f(\mathbf{s}|\mathbf{s}_i)$ from which $\mathcal{L}_i$ was sampled, and h_i tracks how many queries in W hits this cluster. For each workload query $Q \in W$, we find the cluster $\langle \mathcal{L}_i, \mathbf{s}_i, h_i \rangle$ that minimizes the KL divergence of $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ from $f(\mathbf{s}|\mathbf{s}_i)$. If this minimum divergence is below the prescribed threshold θ_{KL}, we increment h_i by one and move on. Otherwise, we sample n locations from $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$, and create a new cluster $\{\langle \mathcal{L}_{\mathsf{new}}, \widehat{\mathsf{Sel}}(Q), 1 \rangle\}$, where $\mathcal{L}_{\mathsf{new}}$ contains the newly sampled locations. At the end of the process, let $m \leq |W|$ denote the final number of clusters. Overall, $\mathcal{L}$ is the union of all $\mathcal{L}_i$'s, with mn locations total. The lower-right quarter of Figure 2 graphically illustrates our probe location sampling approach, in comparison with previous work, which is further discussed in Section 3.4.

3.2 Computing Plan-Penalty Profiles

With the set $\mathcal{L}$ of probe locations, the next step is to invoke the query optimizer to find and cost candidate plans at $\mathcal{L}$, and then post-process the resulting information into a plan-penalty profile $\mathfrak{P}$. Conceptually, $\mathfrak{P}$ is a matrix whose rows correspond to candidate plans and columns correspond to the probe locations. The entry $\mathfrak{P}[\pi, \mathbf{s}]$ records the value of $\mathsf{Penalty}(\pi, \mathbf{s})$ defined by Equation (1), and the row vector $\mathfrak{P}[\pi, \cdot]$ represents the penalty profile for π.

As the first step in building $\mathfrak{P}$, we obtain an initial set $\mathcal{P}$ of candidate plans and construct a *plan-cost matrix* $\mathbf{C}$, where each row vector $\mathbf{C}[\pi, \cdot]$ represents the *cost profile* of candidate plan $\pi \in \mathcal{P}$, i.e., its costs at all probe locations. To collect the set $\mathcal{P}$, we call $\mathsf{Opt}(Q, \mathbf{s})$ to obtain the optimal plan for each probe location $\mathbf{s} \in \mathcal{L}$. Note that $|\mathcal{P}| \leq |\mathcal{L}|$ because it is possible for the optimizer to choose the same plan for multiple locations. Then, for each distinct candidate $\pi \in \mathcal{P}$, we call $\mathsf{Cost}(\pi, \mathbf{s})$ for each $\mathbf{s} \in \mathcal{L}$, and store the result in entry $\mathbf{C}[\pi, \mathbf{s}]$. The overall running time of this step is dominated by $|\mathcal{L}|$ calls to Opt and $|\mathcal{P}| \cdot |\mathcal{L}|$ calls to Cost.

At this point, we can simply compute $\mathfrak{P}$ from $\mathbf{C}$ by calculating each entry $\mathfrak{P}[\pi, \cdot]$ using Equation (1), with $\mathsf{Cost}(\pi, \mathbf{s}) = \mathbf{C}[\pi, \mathbf{s}]$ and $\mathsf{Cost}^{\star}(\mathbf{s}) = \min_{\pi' \in \mathcal{P}} \mathbf{C}[\pi', \mathbf{s}]$. However, the size of $\mathfrak{P}$ can be large because of the number of candidate plans. For example, in our experiments in Section 5, templates 22 and 24 from the JOB benchmark [31] produced more than 300 candidate plans. Hence, we next focus on reducing the set $\mathcal{P}$ of candidate plans to a small number k. After reduction, the size of $\mathfrak{P}$ becomes $O(k|\mathcal{L}|)$. Besides saving space and improving inference speed, a good plan reduction method can also help improve robustness (and hence query performance), which we will validate empirically in Section 5.

We consider two plan reduction methods below. The first method prefers choosing plans that are near-optimal at many probe locations, which heuristically reflects robustness. The second method is more conservative: it chooses a set of representative plans that can substitute for others, without second-guessing robustness at this point (later, the runtime plan selection procedure will pick, among these representatives, the best plan for the given query according to the more precise robustness criterion). We give an example comparing the results of these methods in Section 5.3 .

Reduction by τ-Approximate Cover. The first method follows a natural heuristic: we prefer plans that are near-optimal at a large number of probe locations. The same heuristic and its variants have been employed by various PQO methods including QueryLog [44] and Kepler [15]. More precisely, we say that a plan π *covers* probe location $\mathbf{s}$ if $\mathsf{Cost}(\pi, \mathbf{s}) \leq (1+\tau) \cdot \mathsf{Cost}^{\star}(\mathbf{s})$, where τ is the same as in Equation (1). Given k, the target number of candidate plans after reduction, we want to choose a subset of k plans in $\mathcal{P}$ that together cover as many probe locations as possible.

This problem is an instance of the set cover problem [47], where a simple greedy algorithm offers an approximation ratio of $H(|\mathcal{L}|) \leq \ln|\mathcal{L}| + 1$ (in terms of the number of candidates needed to cover all probe locations). In each step, we greedily pick the candidate plan that covers the most number of probe locations that have not been covered yet. The running time is $\mathrm{O}(|\mathcal{P}| \cdot |\mathcal{L}|)$, with no optimizer calls, and is negligible compared with the time to acquire $\mathbf{C}$.

We expect this method to work well in practice, because its heuristic generally reflects robustness. However, it can miss robust plans. The reason is that this heuristic, by considering the overall number of probe locations covered, reflects a "global" measure of robustness, but at runtime, robustness in fact depends on the "local" neighborhood around the query of interest.

Reduction by Cost Profile Similarity. Our second, more conservative, method aims to choose representative plans that can generally substitute for others. To capture the extent to which two plans π and π' can substitute for each other, we want a distance metric reflecting the similarity between their cost profiles $\mathbf{C}[\pi, \cdot]$ and $\mathbf{C}[\pi', \cdot]$. To this end, we adopt the Jensen–Shannon distance between the two discrete distributions P and P' represented by the respective profiles. More precisely, this distance is calculated as the square root of the average between the KL divergence from P to M and the KL divergence from P' to M, where $M = (P + P')/2$ is a mixture distribution of P and P'. An alternative would be the Wasserstein distance, which is better at capturing shape and location differences between distributions, but we choose Jensen–Shannon because of its low computational complexity.

Then, given the target number k, we want to choose a subset of plans in $\mathcal{P}$ that minimizes the maximum distance between any plan in $\mathcal{P}$ to its closest among the k chosen. This problem is an instance of the metric k-center problem [47], and a range of clustering algorithms are applicable. Our implementation uses a simple greedy algorithm based on the farthest-first traversal, which gives an approximation ratio of 2. We choose the first candidate plan randomly from $\mathcal{P}$ and place it in our solution set S. While $|\mathcal{S}| < k$, we add to S the plan π in $\mathcal{P} \setminus S$ that maximizes the distance between π and plans already in S. We stop and return S once $|\mathcal{S}| = k$. The running time is $\mathrm{O}(k \cdot |\mathcal{P}| \cdot |\mathcal{L}|)$, where the $|\mathcal{L}|$ factor reflects the running time of one distance computation. Again, there are no optimizer calls, so the cost is negligible compared with constructing $\mathbf{C}$.

3.3 Runtime Plan Selection

At runtime, PAR^{2}QO has access to the error model $\mathcal{E}$, the plan-penalty profiles $\mathfrak{P}$ with k candidate plans (Section 3.2), as well as additional information on the probe locations $\mathcal{L}$ (Section 3.1). Given a query Q, PAR^{2}QO uses $\mathcal{E}$ to derive $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$, the distribution of true selectivities conditioned on the estimates. Then, PAR^{2}QO estimates the expected penalty (Section 2.1) for each of the k candidate plans, and returns the plan with the lowest estimate. To estimate $\mathsf{E}[\mathsf{Penalty}(\pi, \mathbf{S}) \mid \widehat{\mathsf{Sel}}(Q)]$ for candidate plan π, we use its penalty profile $\mathfrak{P}[\pi, \cdot]$. If the probe locations were sampled from $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$, the average penalty would yield an unbiased estimate of the expected penalty. However, since the probe locations were sampled differently, we must reweigh the penalties to correct the bias using the method of importance sampling.

The core idea is to reweigh the penalty at each probe location based on how likely that location is sampled from the *target distribution* $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ and how likely that location is sampled from our *sampling distribution.* Recall from Section 3.1 that probe locations are organized into m clusters with n locations in each. For the i-th cluster $\langle \mathcal{L}_i, \mathbf{s}_i, h_i \rangle$, its n probe locations $\mathcal{L}_i = \{\mathbf{s}_{i,1}, \mathbf{s}_{i,2}, \ldots, \mathbf{s}_{i,n}\}$ are sampled from $f(\mathbf{s}|\mathbf{s}_i)$. Since the i-th cluster was hit h_i times by training queries, we estimate the sampling probability of this cluster as $h_i / \sum_{i=1}^{m} h_i = h_i/|W|$, which measures how frequently a new query falls into cluster i based on how many times it was encountered during training. Combining both components above, we estimate the sampling probability of a specific location to be $s_{i,j}$ as $(h_i/|W|) \cdot f(\mathbf{s}_{i,j}|\mathbf{s}_i)$. Meanwhile, the likelihood of this location under the target distribution is $f(\mathbf{s}_{i,j}|\widehat{\mathsf{Sel}}(Q))$. Hence, overall, the expected penalty of π is estimated as:

$$\frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{f(\mathbf{s}_{i,j}|\widehat{\mathsf{Sel}}(Q))}{(h_i/|W|) \cdot f(\mathbf{s}_{i,j}|\mathbf{s}_i)} \cdot \mathfrak{P}[\pi, s_{i,j}].$$

Since we use expected penalties to rank plans, we can simplify the above calculation into the following, by dropping terms that do not affect the ranking:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \frac{f(\mathbf{s}_{i,j}|\widehat{\mathsf{Sel}}(Q))}{h_i f(\mathbf{s}_{i,j}|\mathbf{s}_i)} \cdot \mathfrak{P}[\pi, s_{i,j}].$$

As an additional optimization, we can also cache the value of $f(\mathbf{s}_{i,j}|\mathbf{s}_i)$ when generating $\mathbf{s}_{i,j}$. Overall, the selection of the robust plan can be done by a single pass over $\mathfrak{P}$, so the running time is $\mathrm{O}(k|\mathcal{L}|)$. No optimizer calls are needed except the single $\widehat{\mathsf{Sel}}(Q)$ call to obtain the estimated selectivities.

3.4 Discussion

We contrast PAR^{2}QO with other PQO approaches, including two state-of-the-art systems, Kepler [15] and QueryLog [44]. Before delving deeper, we note that previous PQO approaches do not specifically optimize for finding *robust* plans. In fact, traditional PQO would be happy with reproducing the plans selected by the underlying optimizers, with no consideration for plan robustness. However, it would be incorrect to conclude that traditional PQO cannot beat the underlying optimizer by design. Once we closely

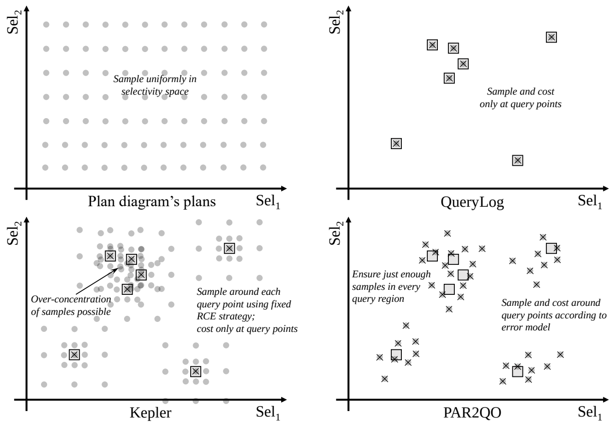

Sel$_2$
Sample uniformly in selectivity space
Plan diagram's plans
Sel$_1$
Sel$_2$
Sample and cost only at query points
QueryLog
Sel$_1$
Sel$_2$
Over-concentration of samples possible
Sample around each query point using fixed RCE strategy; cost only at query points
Kepler
Sel$_1$
Sel$_2$
Ensure just enough samples in every query region
Sample and cost around query points according to error model
PAR2QO
Sel$_1$

Figure 2: Illustration of locations in a 2-d selectivity space where various PQO approaches obtain candidate plans (circles) and cost them (crosses), using training queries (squares) when applicable.

examine these methods, we see that many have components whose heuristics favor keeping robust plans as candidates, thereby allowing them to beat the underlying optimizer when it makes incorrect selectivity estimates, even if this feature is not by design.

We first highlight differences between PAR^{2}QO and Kepler [15], illustrated by the lower-right and lower-left quarters of Figure 2.

(1) Kepler executes plans for all training queries as well as alternative plans obtained via *row-count evolution* (RCE). Measuring actual execution times makes Kepler robust against errors not only in selectivity estimation but also in plan cost models. In contrast, PAR^{2}QO only executes (counting versions of) training queries (and no alternative plans) to learn an error model for selectivities, and it "trusts" $\mathsf{Cost}(\pi, \mathbf{s})$ given correct $\mathbf{s}$. Hence, PAR^{2}QO spends significantly less time preparing its cache. In Section 5, we will see that focusing on selectivity errors delivers comparable or better query performance than Kepler despite a much lower initial cost.
(2) Kepler explores the "neighborhood" of a training query Q using RCE, which perturbs subquery selectivities in order to discover more candidate plans. This method hence improves robustness, analogous to PAR^{2}QO sampling from $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ according to the error model (Section 3.1). However, RCE uses a fixed strategy for generating perturbations (exponentially spaced in selectivity ranges), which we believe is not as effective as PAR^{2}QO's perturbations informed by the error model.
(3) The same τ-approximate cover method in Section 3.2 is employed by Kepler to reduce the set of candidate plans. As discussed earlier, this method helps to improve "global" robustness (over the training workload). However, unlike PAR^{2}QO, which has a built-in mechanism (Section 3.1) to avoid over-representing popular queries, the effectiveness of Kepler's plan reduction is susceptible to an imbalanced query workload. Moreover, in determining coverage, Kepler does not consider the costs of candidate plans at locations obtained by RCE; hence, its notion of coverage is much narrower than ours.
(4) Kepler makes its runtime plan selection using a neural network, combined with a method to assess prediction uncertainty. However, it is difficult to justify plan choices and interpret uncertainty. In contrast, PAR^{2}QO's plan selection is transparent; the availability of the error model and the plan penalty profiles makes the selection process easy to understand.

The last point above, on PAR^{2}QO's advantage of interpretability over Kepler, also applies to other black-box methods. While effective in many scenarios, these methods obscure the rationale behind plan choices, making it difficult to interpret model behavior, diagnose the root causes of regressions, and implement corrections. In contrast, PAR^{2}QO's decisions can be explained by the chosen plan's cost penalty profile and the error model of selectivity estimates, as our robustness objective is clearly defined. A recent system demonstration based on PARQO [51] shows how, given any two plans (e.g., robust vs. optimizer default), it can compare and inspect their robustness by visualizing their costs over the region of uncertainty surrounding the initial selectivity estimates. Users can see potential selectivities for which the robust plan avoids severe performance regressions. If, for any reason, robust plans frequently underperform because actual selectivity errors do not follow the error model, a clear corrective step would be to recalibrate the error model. Such ease of interpretability and diagnosis would be difficult if not impossible to achieve for black-box methods.

Next, we compare PAR^{2}QO with PQO approaches that are not based on deep learning. The most recent representative is QueryLog [44], illustrated by the upper-right quarter of Figure 2. We also touch on some earlier approaches, including plan diagrams [13], illustrated by the upper-left quarter of Figure 2.

(1) To find candidate plans, previous work has generally used one of the following two methods. The first method, used by earlier approaches including the pioneering work of *anorexic plan diagrams* [13], attempts to cover the entire selectivity space, but quickly becomes intractable beyond 5-6 dimensions. The second method, used by QueryLog, is to sample past queries for candidate plans. However, QueryLog has no mechanism for further perturbing selectivities to obtain potentially better plans, like PAR^{2}QO's sampling (Section 3.1) or Kepler's RCE.
(2) To reduce the set of candidate plans, most PQO approaches, like Kepler, use the τ-approximate cover method in Section 3.2 or some variants. For example, anorexic plan diagrams use the same method; QueryLog seeks a cover that minimizes an "aggregate sub-optimality" measure, which is close in spirit to this method. While this reduction naturally favors robust plans, its benefit is limited here because of how the candidates were generated in the first place: neither uniformly sampling the entire selectivity space nor focusing on the locations of the training queries alone comes close to PAR^{2}QO in the ability to capture the effect of selectivity estimation errors on robustness.
(3) To select among candidate plans at runtime, QueryLog employs a model using contextual multi-armed bandit learning, which can adapt dynamically over time, while earlier approaches generally perform no better than recosting all candidate plans for the query and picking the cheapest. None aims at picking a robust plan like PAR^{2}QO does.

In summary, PAR^{2}QO offers a systematic treatment of robustness — driven by a formal definition of robustness against errors in selectivity estimates — from initial learning of the error model, sampling of probe locations and candidate plans, construction of the plan-penalty profiles, to plan selection at runtime.

4 CARVER: WORKLOAD GENERATION GUIDED BY CARDINALITY

Training and testing query optimizers require a query workload, whose choice is not always clear. Synthetic benchmarks (e.g., DSB [11]) typically use standard distributions (e.g., Gaussian) to generate parameter settings for given templates. Alternatively, if the database has been operating for some time, past queries can be used for the workload. In either case, there is no guarantee that such workloads reflect future queries. If our goal is *robust* query optimization, it is natural to make training and testing queries cover the possible scenarios that a query optimizer may potentially encounter, as comprehensively as possible. Indeed, recent work on Kepler [15] generates parameter settings for a given query template by uniformly sampling rows from the full join result (with no selections applied), ensuring that every join result row has a chance to be covered by the generated query workload.

As argued in Section 1, however, we need a still better notion of coverage. Recall that a cost-based optimizer makes its decisions by estimating the selectivities of query predicates. Hence, it would be useful for the workload to include queries that exhibit a diverse range of selectivities. By randomly sampling final join result rows, Kepler will tend to overrepresent popular column values in its query parameters. Second, optimizer decisions are heavily influenced by the cardinalities of subqueries. Therefore, we should also generate scenarios that cover the range of possible subquery cardinalities, instead of focusing only on the final join result as Kepler does.

CARVER meets the above requirements by providing a flexible and powerful way for users to ensure that the generated queries provide good coverage of the optimization scenarios. Formally, given a database instance $\mathcal{D}$ and a query template Γ with a set $\mathsf{params}(\Gamma)$ of parameters, our goal is to generate a set of queries following the template Γ, where each query assigns a specific value to each parameter in $\mathsf{params}(\Gamma)$. Overall, CARVER works as follows. It randomly generates queries following template Γ from a mixture distribution $\{(P_k, w_k)\}$, whose components (P_k's) and associated weights (w_k's) are specified by the user. Each component P_k is specified by a set of subquery templates $\{\gamma_{k,j}\}$ of Γ whose parameters form a disjoint partitioning of $\mathsf{params}(\Gamma)$. P_k is defined as the product distribution $\prod_j P_{k,j}$, where each $P_{k,j}$ is a distribution governing the generation of values for $\mathsf{params}(\gamma_{k,j})$ in the subquery template $\{\gamma_{k,j}\}$.

We now describe the procedure for generating from P_{kj} for each $\mathsf{params}(\gamma_{k,j})$. We assume that $\gamma_{k,j}$ is a selection-join query template, and that it can be rewritten as $\sigma_\theta(q)$, where q is the (parameterless) *base query* obtained from $\gamma_{k,j}$ by "removing" all atomic query predicates involving parameters,[1] and θ is the *remainder predicate* involving $\mathsf{params}(\gamma_{k,j})$. The *selectivity* of a setting in $\mathsf{params}(\gamma_{k,j})$ is defined as $|\sigma_\theta(q(\mathcal{D}))|/|q(\mathcal{D})|$ where θ is instantiated with the given setting. Users can specify a bucketization B of the selectivity range (not necessarily equally spaced), e.g., $B = \{[0, 0.5\%), [0.5\%, 1\%), [1\%, 2\%), \dots, [90\%, 1]\}$; CARVER will generate an equal number of $\mathsf{params}(\gamma_{k,j})$ settings for each bucket of B, up to the number of possible settings for this bucket.

[1]The general procedure needs to handle subtleties that arise because of `OR` and `NOT`, which we omit here.

Intuitively, including a subquery in the definition of mixture component guides CARVER to ensure that its generated parameters provide good coverage of the possible subquery cardinalities. Additionally, including subqueries joining multiple tables and involving multiple query parameters ensures that CARVER accounts for dependencies among join and selection predicates.

Depending on the type and complexity of the remainder predicate θ, the generation procedure varies. Section 4.1 presents several common cases; Appendix E provides a more detailed example and discusses how to handle additional challenges.

4.1 Generation of Subquery Parameters

Equality Selections. Here, θ has the form $A_1 = v_1 \wedge \cdots \wedge A_n = v_n$, where each A_i is a column of some input table in q. First, we group the result rows of q by $A_1, \dots, A_n$ and compute the row count for each group. The selectivity of θ for each group (using the group-by values for parameter setting) is simply the group row count divided by $|q|$, the total of all group row counts. Second, we sort the groups by their counts and bucketize them according to B. Finally, we draw an equal number of samples from each bucket (unless limited by the number of groups available in the bucket). Note that the first and second steps can be computed conveniently in SQL using a `SELECT-FROM-WHERE-GROUPBY-ORDERBY` statement.

Note that the above procedure also handles the case of a single "not-equal-to" selection, i.e., θ has the form $A \neq v$, by computing the selectivity of v as one minus its group row count divided by $|q|$.

Inequality Selections. Suppose that θ has the form $A_1 \leq v_1 \wedge \cdots \wedge A_n \leq v_n$. As with the case of equality selections, we first group the result rows of q by $A_1, \dots, A_n$ and compute the row count for each group. We then sort the groups by $A_1, \dots, A_n$; the result is conceptually a n-dimensional array $\mathbf{A}$ representing a discrete probability density function. Let $\mathbf{A}[v_1, \dots, v_n]$ denotes the array entry for $(v_1, \dots, v_n)$. Next, we compute the cumulative density function (CDF) over $\mathbf{A}$ in-place (details to be presented shortly). We then sort $\mathbf{A}$ by the CDF values and bucketize them according to B. Finally, we draw an equal number of samples from each bucket (again, subject to availability). All processing before the CDF computation can be done easily in SQL.

In more detail, the CDF can be computed as follows, by accumulating $\mathbf{A}$ over one additional dimension at a time. In the first step, for each setting of $A_1, \dots, A_{n-1}$ to $v_1, \dots, v_{n-1}$, we update the subarray $\mathbf{A}[v_1, \dots, v_{n-1}, *]$ to be the conditional CDF over A_n by computing the prefix sum over its elements. In general, the step accumulating over A_i updates each subarray $\mathbf{A}[v_1, \dots, v_{i-1}, *, \dots]$ to be the conditional CDF over $A_i, \dots, A_n$ while setting $A_1, \dots, A_{i-1}$ to $v_1, \dots, v_{i-1}$; each such subarray is updated in-place as the prefix sum over its constituent subarrays updated in the previous step. The last step updates $\mathbf{A}$ to be the overall CDF.

The above procedure can be easily extended to support other inequality predicates: $A_i < v_i$ can be supported by handling the bucket boundaries slightly differently; $A_i > v_i$ and $A_i \geq v_i$ can be handled by reversing A_i (from ascending to descending) in the sort order of the groups. It is also possible to support range predicates in the form of A_i BETWEEN l_i AND u_i by conceptually duplicating the column A_i in input data as columns A_i^l and A_i^u, and handling the predicate as $A_i^l \geq l_i \wedge A_i^u \leq u_i$.

LIKE *Selection.* Suppose that θ has the form A LIKE v. The design space for the generation procedure is large. We describe below the token-based approach currently employed by CARVER. Exploration of alternative approaches, such as using language models to further classify different types of LIKE-operators [3], is left for future work. First, we make one pass over the result rows of q to tokenize all A values. If A in general contains multi-word strings, we tokenize each A value into words. Otherwise, we tokenize each A value into characters. During the pass, we maintain the set of tokens along with the number of q result rows whose A value contains this token. Second, we sort the tokens by their counts and bucketize them according to B. Finally, the same sampling procedure is carried out over the buckets. Each token *token* sampled corresponds to the predicate A LIKE '%*token*%'.

The above procedure can be extended to a conjunction of LIKE selections. For example, to handle A_1 LIKE $v_1 \wedge A_2$ LIKE v_2, for each result row q, we will tokenize both A_1 and A_2 values and consider all pairs of tokens, one from A_1 and the other from A_2. We maintain the set of token pairs along with the number of q result rows whose A_1 and A_2 contain the tokens in the pair, respectively.

5 EXPERIMENTS

Databases and benchmarks. Here, we focus on experiments using **JOB** (Join Order Benchmark) [33], based on real-world IMDB data, consisting of 33 query templates. It is designed to challenge optimizers with skewed/correlated data distributions and intricate join relationships. We have also performed evaluation using **DSB** [11], a recently introduced industrial benchmark built on synthetic data, extending TPC-DS [39] with more sophisticated data distributions and correlations. Due to space constraints, however, its results are discussed only in Appendix G.2 .

Workload synthesis. For JOB, we generate three different workloads: CARVER, Uniform, and Historical. For **CARVER**, we consider the one-component scenario by default, regarding all 2-table querylets as the set of subquery templates $\{\gamma_{k,j}\}$ in our generator, and the selectivity bucketing for each $\text{params}(\gamma_{k,j})$ is performed by partitioning the GROUP BY results into ten equally spaced selectivity ranges, i.e., $B = \{[0, 10\%), [10\%, 20\%), \ldots, [90\%, 1]\}$. For **Uniform**, we first perform a full join across all tables based on the join conditions specified in the query templates. We then sample $\text{params}(\Gamma)$ uniformly from the join result, same as the synthetic method in Kepler [15]. For **Historical**, we follow the methodology in PARQO [50]; additional workload details are available in Appendix G.1 . For all workload generation methods, after initial synthesis, we further ensure only queries that return nonempty results are included. We synthesize 250 queries for each $\text{params}(\Gamma)$: 50 queries are used to initialize and train PQO models; the remaining 200 queries serve as the testing workload for evaluation.

PAR^{2}QO setup. We implement PAR^{2}QO on top of PostgreSQL V16.2, leveraging plan and selectivity injection tools from prior work [21, 38, 50]. For each Γ, we learn the error model $f(\mathbf{s}|\hat{\mathbf{s}})$ using relevant querylets containing up to 3-table subqueries, following the same procedure in PARQO [50]. Unless otherwise specified, we set $n = 50$ as the number of probe locations per cluster and $\theta_{\text{KL}} = \ln(200)$ as the prescribed KL divergence threshold. The sample size is user-configurable, and we demonstrate that even with this relatively small number of probe locations, PAR^{2}QO achieves strong performance. The target upper bound on the number of candidate plans after reduction is $\max(0.2 \cdot |\mathcal{P}|, 10)$, where 10 ensures sufficient diversity when the initial $|\mathcal{P}|$ is small. For plan reduction, we use τ-approximate cover (τ-Cover in short) by default. Section 5.3 further examines the impact of these choices.

The remainder of this section is structured as follows: First, we compare PAR^{2}QO's query performance against others in Section 5.1. Next, Section 5.2 compares their overhead (training and inference). Section 5.3 evaluates the contributions of components and parameter settings in PAR^{2}QO through a series of ablation studies. Then, Section 5.4 assesses plan robustness under data distribution shifts. Finally, Section 5.5 evaluates how CARVER outperforms other synthetic workloads in training optimizers.

5.1 Comparison of Query Performance

We compare PAR^{2}QO with the following methods:

- **PostgreSQL**: Here, we simply rely on PostgreSQL's default behavior, invoking its optimizer separately for each query instance. While this method serves as a useful comparison point, our goal goes beyond matching the optimizer's performance — we aim to enhance plan robustness and reduce optimization time in a parametric setting. To provide a more comprehensive evaluation, Section 5.3 further introduces stronger baselines.
- **Kepler**: Kepler [15] is a state-of-the-art PQO method that learns from real query executions to predict the fastest plan. We use RCE to obtain candidate plans, prune candidates using its default greedy algorithm with a near-latency-optimal threshold of 1.2×, and train a model for each template.
- **PARQO and PARQO′**: PARQO refers to the one-robust-plan-per-template approach previously proposed for PARQO [50], while PARQO′ is the extension introduced in Section 2.2 that considers caching multiple robust plans obtained from training queries. Both methods use the same error profiles and expected-penalty-based metric for robust plan selection as PAR^{2}QO.
- **QueryLog′:** Following the plan population method [44], this baseline obtains its initial candidate set from optimal plans observed in the training workload, and then applies τ-Cover plan reduction. At runtime, it selects the plan with the lowest recost, $\text{Cost}(\pi, \widehat{\text{Sel}}(Q))$.
- **Bao:** Bao [37] is a learned query optimizer that leverages reinforcement learning to generate hints for input queries. It serves as a baseline representing general-purpose learned optimizers that are not specifically tailored for the PQO setting.

For a fair comparison, all methods use the same set of training queries. Configuration of Bao is more involved; we followed the guidelines in its tutorial [36] to obtain a reasonable setting. All configuration details are available in Appendix G.1 . To reduce execution noise and simulate a warm-cache environment, we execute each selected plan 5 times and report the median latency as the final result. All experiments were conducted on a Linux server with 16 Intel(R) Core(TM) i9-11900 @ 2.50GHz processors.

Table 1: Average execution latency (ms) and counts of significant speedup (↑) / regression (↓) cases, by different PQO methods and their variants, across three workloads of JOB. Latencies within 5% of the best in respective columns are highlighted.

	CARVER				Uniform				Historical			
	Latency (ms)	↑ > 1.2×	↓ > 1.2×	↓ > 2×	Latency (ms)	↑ > 1.2×	↓ > 1.2×	↓ > 2×	Latency (ms)	↑ > 1.2×	↓ > 1.2×	↓ > 2×
PostgreSQL	536 ms	-	-	-	410 ms	-	-	-	637 ms	-	-	-
PAR^{2}QO	**273 (1.96×)**	15	3	0	**254 (1.61×)**	14	2	0	**326 (1.95×)**	15	7	0
Kepler	499 (1.07×)	19	5	4	306 (1.34×)	15	5	2	**328 (1.94×)**	19	2	1
PARQO′	305 (1.76×)	14	7	0	305 (1.34×)	13	5	0	348 (1.83×)	16	4	0
PARQO	350 (1.53×)	9	2	0	383 (1.07×)	3	2	0	433 (1.47×)	9	5	0
QueryLog′	587 (-1.10×)	4	6	0	428 (-1.04×)	5	4	1	668 (-1.05×)	2	7	0
Bao	443 (1.21×)	5	1	0	325 (1.26×)	6	1	0	613 (1.04×)	8	2	1
Other Variants:												
Uni-BestCost	658 (-1.23×)	3	10	3	426 (-1.04×)	4	9	5	725 (-1.14×)	3	9	3
BestCost	344 (1.56×)	12	3	0	355 (1.15×)	12	3	0	400 (1.59×)	12	1	0
BestCost-bound	513 (1.04×)	10	2	0	362 (1.13×)	11	3	0	591 (1.08×)	9	1	0
Nearest	401 (1.34×)	13	9	2	422 (-1.03×)	9	6	1	377 (1.69×)	15	6	1
PAR^{2}QO-sim	**287 (1.87×)**	14	4	0	270 (1.52×)	14	1	0	**330 (1.93×)**	17	5	0
PAR^{2}QO-no-reduce	**285 (1.88×)**	14	4	0	**255(1.61×)**	14	2	0	339 (1.88×)	13	8	1
BestCost-sim	420 (1.28×)	11	9	1	402 (1.02×)	9	2	1	446 (1.43×)	8	3	0
BestCost-no-reduce	366 (1.46×)	8	3	0	390 (1.05×)	9	3	0	444 (1.43×)	7	4	0

Table 1 (ignore the *Other Variants* section for now) summarizes the results on JOB. PAR^{2}QO outperforms other methods in terms of average execution latency and delivers the highest improvement on CARVER, achieving a 1.96× speedup over PostgreSQL. Recall that CARVER is a "harder" workload than Uniform and Historical, because it seeks to cover the full range of possible selectivities (Section 4). Kepler shows strong speedups on Uniform and Historical, but it achieves only a 1.07× speedup on CARVER. On Historical, both Kepler and PAR^{2}QO reach approximately a speedup of 1.95× over PostgreSQL, higher than other baselines. PARQO′, PARQO, and Bao all achieve speedups, with PARQO′ outperforming the other two, but it still underperforms PAR^{2}QO. Meanwhile, QueryLog′ shows slight regressions below PostgreSQL across all workloads, with the largest observed on CARVER at −1.10×.

Beyond average latency, our results show that PAR^{2}QO achieves a superior balance between efficiency and robustness. To quantify, Table 1 shows the number of templates exhibiting either *significant* speedup/regression or *severe* regression, defined as follows:

- Significant speedup/regression: A change in query latency exceeding 1.2×/−1.2× compared with the PostgreSQL baseline.
- Severe regression: A performance degradation worse than −2×.

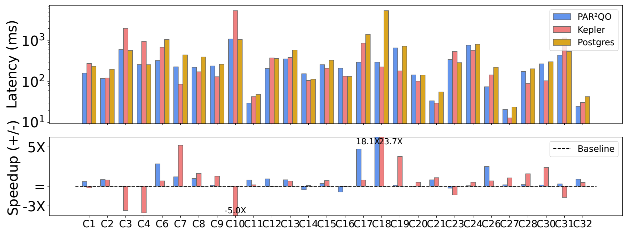
Significant speedup highlights a method's ability to find more efficient plans; however, equally critical is the ability to avoid significant/severe regressions. While Kepler achieves the largest number of significant speedups (even outperforming PAR^{2}QO in this regard), it is fragile — it frequently encounters severe regressions that significantly degrade its overall average latency. In contrast, PAR^{2}QO avoids such extreme cases and maintains stable performance. On the other hand, while other baselines also rarely exhibit severe regressions, they are too conservative — most achieve much fewer significant speedups compared with PAR^{2}QO (with the exception of PARQO′, which gets closer to PAR^{2}QO but still underperforms). Bao fares very well in avoiding both significant and severe regressions, partly because it relies on hinting rather than direct plan injection, allowing PostgreSQL to moderate its behavior; however, it also achieves far fewer significant speedups.

To further analyze the performance at template-level, we focus on PAR^{2}QO and Kepler, presenting detailed execution latency and speedup results in Figure 3. For clarity and simplicity, each template is labeled with the workload's initial letter plus its ID (e.g., $C2$ represents template 2 in CARVER, and $H1$ is template 1 in Historical). We exclude templates in Figure 3 where neither PAR^{2}QO nor Kepler exhibits significant speedup or regression. Results show that, overall, Kepler can achieve markedly higher speedups in certain scenarios. For instance, on $C7$ and $C19$, it attains 5.17× and 4.03× speedups, respectively, whereas PAR^{2}QO only reaches 1.96× and 1.10×. Moreover, there are templates where Kepler provides a speedup but PAR^{2}QO does not, such as $H4$, $H8$, $H19$, $U22$, and $U26$. However, Figure 3 also highlights that PAR^{2}QO effectively mitigates severe regressions: the worst performance degradation for PAR^{2}QO occurs in $H4$ (−1.81×), while Kepler's largest regression, observed in $C10$, exceeds −5×. These findings indicate that Kepler, by learning from real executions, can sometimes identify *faster* plans. However, these plans lack robustness and may fail to deliver consistent performance in practice. In contrast, PAR^{2}QO effectively mitigates extreme performance degradations, leading to better overall execution latency.

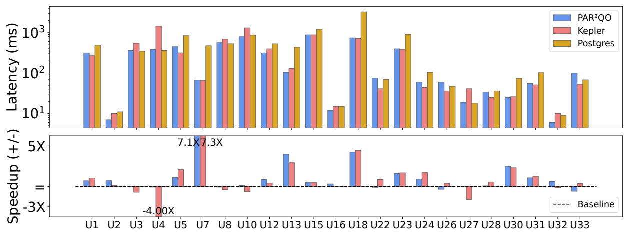
5.2 Comparison of Overhead

Besides the execution latencies of query plans, other types of overhead — notably the one-time (or periodic) per-workload model preparation cost and the runtime per-query optimization cost — are also important practical considerations. After all, PQO was initially motivated by the desire to reduce the per-query optimization cost, which increases the query latencies observed by users. This subsection examines the overhead of various methods in comparison with PostgreSQL. Additional details are available in Appendix G.1 .

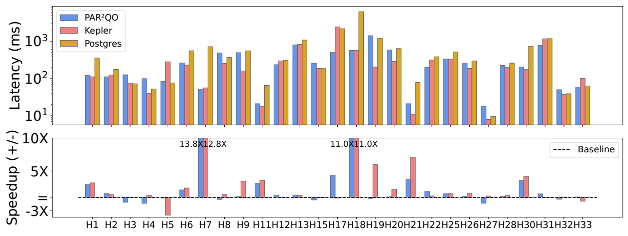
Model preparation time. By model preparation time, we mean the period from receiving the training workload to completing the model setup and being ready to generate plans for new queries. For PAR^{2}QO, the first step is profiling errors using the training workload, which takes less than 13 minutes on average per query template. Then, preparing the plan-cost matrix takes about 8.5 minutes per template on average (with the default setting of $n = 50$,



(a) CARVER Workload



(b) Uniform Workload



(c) Historical Workload

Figure 3: Template-level execution latency and speedup comparison on three workloads for JOB. The top panel of each figure compares the average latency (ms) on a log scale, and the bottom shows the average speedup or regression over PostgreSQL (baseline).

which results in $|\mathcal{L}| = 300$ and $|\mathcal{P}| = 88$). Subsequent plan reduction is very fast, completing under 500ms per template. Thus, the total preparation time for PAR^{2}QO averages 21.5 minutes per JOB template. In contrast, Kepler requires a large number of query executions for training, where every query incurs many executions. Overall, Kepler's preparation time is over 6.5 hours per JOB template — more than 18× slower than PAR^{2}QO. As a comparison point, PARQO′, without PAR^{2}QO's smart method of sharing computation, suffers from the high cost of finding the robust plan for each anchor independently; it takes an average of 2.4 hours per JOB template, 6.7× slower than PAR^{2}QO. Both Bao and QueryLog incur lower model preparation overhead, but it comes at the cost of reduced performance.

Inference time and model size. As a baseline, PostgreSQL's optimizers adds on average 67ms optimization overhead per JOB query. In comparison, Kepler is the fastest, under 1ms per query, thanks to its model directly outputting the query plan; PAR^{2}QO's runtime overhead is 26ms, still considerably faster than PostgreSQL; Bao is the slowest, with 376ms per query, because it does not have the benefit of PQO. While PAR^{2}QO's inference time is longer than Kepler because it still needs to carry out robustness-based plan selection at runtime, this overhead is a good trade-off in order to secure more robust and generally more efficient plans than Kepler, as shown in Section 5.1. As another comparison point, QueryLog′, despite its much simpler objective requiring no robustness evaluation, has a higher overhead (35.4ms) than PAR^{2}QO, because merely recosting plan candidates adds considerable cost. This observation further demonstrates the effectiveness of computation caching in PAR^{2}QO. Finally, all methods have practically acceptable model sizes: PAR^{2}QO has a 0.2MB error model plus a 1.8MB cache (2.6MB before plan reduction), totaling no more than 2.1MB at runtime; Kepler and Bao's models use very compact neural networks, which take less than 1MB space at runtime.

5.3 Ablation Study of PAR^{2}QO

To gain deeper insights, we design a series of experiments to evaluate the impact of various components and parameter settings in PAR^{2}QO. Because of space constraints, we discuss only a subset of the results below and show an even smaller subset in Table 1, under the *Other Variants* section. Other results can be found in Appendix G.3 .

Impact of sample locations. First, we examine how PQO performance is affected by the locations from which we obtain candidate plans. To this end, we create a new variant called **Uni-BestCost**, which probes for its candidate plans using the same number of locations as PAR^{2}QO, and similarly uses τ-Cover to reduce this set. However, instead of PAR^{2}QO's workload- and error-aware sampling-by-cluster strategy, Uni-BestCost samples locations uniformly at random within a hyperrectangle bounding all selectivities observed during training. Finally, when selecting plans at runtime, instead of optimizing for penalty-based robustness like PAR^{2}QO, Uni-BestCost picks the candidate plan whose cost is the lowest given the estimated selectivities, just like QueryLog′. To isolate the effect of candidate plans for comparison, we create a second variant called **BestCost**, which is identical to Uni-BestCost except that it uses the same reduced set of candidate plans as PAR^{2}QO.

Results for Uni-BestCost in Table 1 show that using more uniform samples from the observed selectivity range is not effective — not only does it perform far worse than PAR^{2}QO, but it also underperforms QueryLog′ despite examining more locations. On the other hand, BestCost performs significantly better and reverses Uni-BestCost's slowdown: for CARVER, the −1.23× slowdown improves to a 1.56× speedup, surpassing Kepler's 1.07×; for Uniform and Historical, while BestCost remains behind Kepler, it now achieves speedups over the PostgreSQL baseline. The dramatic improvement from Uni-BestCost to BestCost highlights the effectiveness of PAR^{2}QO's sampling strategy in Section 3.1.

Impact of # of probe locations per cluster. We further examine the impact of n, the number of probe locations per sampling cluster, which influences $|\mathcal{L}|$, the total number of sample locations. A larger n can improve plan quality, but also increases training and inference costs. When increasing n from 50 to 200, we observe improvements in some specific templates — for instance, $H4$, which showed the largest regression in Figure 3, improves from a −1.90× regression to a 1.41× speedup. However, the overall performance across workloads remains largely unchanged, indicating that a larger n provides diminishing returns in most cases. Since n influences $|\mathcal{L}|$ directly and $|\mathcal{P}|$ indirectly, it can, in the worst case, blow up overhead quadratically. In practice, increases in overhead are milder but still significant: 3.3× training time and 2.1× inference time compared with $n = 50$. On the other hand, when decreasing n from 50 to 10, we observe a clear degradation in plan quality, but even in this case, PAR^{2}QO ($n = 10$) is still better than Kepler for CARVER and comparable for Uniform and Historical, and consistently beats PostgreSQL, Bao, and PARQO. Further details are available in Appendix G.3 . Overall, we find the setting of $n = 50$ to be a good default, although smarter sampling strategies that increases n adaptively on a per-template basis might be a fruitful future direction.

Impact of runtime plan selection criteria. Next, we study the impact of plan selection criteria on PQO performance. Note that the BestCost variant considered above differs from PAR^{2}QO only in how it selects a plan at runtime. Since BestCost significantly underperforms PAR^{2}QO as shown in Table 1, we can conclude that PAR^{2}QO's penalty-based robustness objective is far superior to simply picking the plan with the lowest cost at estimated selectivities $\hat{\mathbf{s}}$. Here, for further comparison, we create two additional variants that consider alternative plan selection criteria used in previous work. First, **BestCost-bound** extends BestCost by incorporating a safety check, similar to the "cost check" in SRC [17]: it checks whether the cost of a candidate plan π at $\hat{\mathbf{s}}$ remains within a bound $(1 + \lambda)$ of the cost of π at the location where π was optimized (we set $\lambda = 2$, as suggested by SRC [17]); if not, the system falls back to invoking the optimizer. Second, **Nearest** picks the candidate plan whose location in the selectivity space is closest to $\hat{\mathbf{s}}$ in ℓ_2 distance, following the idea of "selectivity check" in SRC [17].

From Table 1, we observe that applying the $(1 + \lambda)$ safety check degrades BestCost's performance, indicating that the fallback mechanism does not guarantee a plan with truly optimal execution performance. The root cause is that the estimated selectivities $\hat{\mathbf{s}}$ for the given query may be wrong to begin with; using $\hat{\mathbf{s}}$ for the safety check and falling back to the optimal plan at $\hat{\mathbf{s}}$ may not be safe in reality. From Table 1, we also see that Nearest fares no better than BestCost and remains far behind PAR^{2}QO. To conclude, with proper accounting for uncertainty in $\hat{\mathbf{s}}$ in its plan selection strategy, PAR^{2}QO has a clear performance advantage over approaches that ignore uncertainty and/or are heuristic in nature.

Impact of plan reduction. By default, PAR^{2}QO employs τ-Cover for plan reduction. Recall that PAR^{2}QO sets the upper bound on the number of plans (after reduction) to be $\max(0.2 \cdot |\mathcal{P}|, 10)$. For JOB, the average number of candidate plans obtained per template via sampling, $|\mathcal{P}|$, is 88. We found that, on average, 9 candidate plans per template were sufficient to τ-cover all candidates, justifying our choice of the default upper bound. In the end, PAR^{2}QO kept on average 8 candidate plans per template. A smarter, template-specific selection of this upper bound and a theoretical justification of what is needed for complete coverage are interesting future directions.

To evaluate other options, we create two variants: **PAR^{2}QO-sim**, which reduces the candidate plan set by cost profile similarity (Section 3.3), and **PAR^{2}QO-no-reduce**, which does not reduce the set at all. Results, shown in Table 1, indicate that both PAR^{2}QO-sim and PAR^{2}QO-no-reduce offer comparable performance as PAR^{2}QO overall. In other words, effective plan reduction, which reduces runtime overhead, does not sacrifice plan performance. In fact, despite plan reduction, PAR^{2}QO is able to slightly but consistently outperform PAR^{2}QO-no-reduce, indicating that τ-Cover may also help improve robustness — a point that we examine further below. A closer look at the resulting plans under both reduction methods is provided in Appendix F .

We also extend this ablation study to BestCost, which employs τ-Cover for plan reduction but does not consider robustness explicitly when picking plans at runtime. We call the corresponding variants **BestCost-sim** and **BestCost-no-reduce**. Interestingly, from Table 1, we see that BestCost, with τ-Cover plan reduction, noticeably outperforms BestCost-no-reduce, with no reduction. This improvement, more significant than that of PAR^{2}QO over PAR^{2}QO-no-reduce, confirms the benefit of τ-Cover in improving robustness, especially in the setting of BestCost where runtime plan selection is not robustness-based. In contrast, BestCost-sim does not offer such a benefit because its reduction method favors preserving plan diversity over robustness.

5.4 Robustness Against Data Shifts

Previous experiments were conducted on a static database state. However, true robustness requires plans to maintain good performance even when data and their distributions evolve. In this experiment, we simulate real-world scenarios with shifting data distributions, which can introduce unforeseen variations in selectivity errors and plan performance.

We create multiple database instances from the original IMDB dataset using three "slicing" methods to get varying degrees of data distribution shifts. First, ***category-slicing*** [50] splits data by item categories (*kind_type.kind*), which produces the most dramatic distribution shifts.[2] Second, ***time-slicing*** [50] applies a 20% sliding window on *production_year*, generating 9 instances. Adjacent instances share overlapping data, while distant ones exhibit larger differences, producing smooth transitions and gradual shifts. Third, ***random-slicing*** randomly samples 20% of *title* along with associated data from other tables; it largely maintains the overall data distribution, representing the least challenging scenario.

For each dataset-slicing method, we synthesize three workloads (CARVER, Uniform, and Historical) by the same procedures in Section 5.1. PAR^{2}QO and Kepler select plans based on one instance alone (called the "base" instance), which are then evaluated on the remaining instances. Note that PAR^{2}QO's error model $\mathcal{E}$ is also derived from the base instance alone. As the baseline, we consider PostgreSQL's plan, which is optimized using statistics refreshed for each instance (i.e., not subject to the same handicap as PAR^{2}QO and Kepler). We select four representative templates to evaluate — 4, 14, 17, and 19 — such that they cover various query complexities and different performance characteristics observed in earlier experiments. The detailed selection criteria are described in Appendix G.5 , along with additional experimental results. In summary, we find PAR^{2}QO plans to be more robust than Kepler's across different data drift scenarios. For example, even under the most challenging category-slicing, the largest regression experienced by PAR^{2}QO on non-base instances is no more than −2×. Interestingly, while we do

[2]Table sizes can vary up to 51× (e.g. *title* in "Movie" vs. "Video game"), and template execution times may fluctuate up to 176× (e.g. C17 on "Movie" vs. "Video game").

Table 2: Performance of models trained under different workloads for JOB, tested under the mixed workload.

		CARVER	Uniform	Historical
PostgreSQL	Avg latency	552 ms	552 ms	552 ms
PAR^{2}QO	Avg latency	315 (1.75×)	407 (1.35×)	400 (1.38×)
	↑ > 1.2×	15	12	15
	↓ > 1.2×	2	8	7
	↓ > 2×	1	1	3
Kepler	Avg latency	555 (-1.01×)	553 (-1.00×)	909 (-1.65×)
	↑ > 1.2×	15	8	9
	↓ > 1.2×	8	14	10
	↓ > 2×	3	5	5
Bao	Avg latency	562 (-1.02×)	626 (-1.14×)	626 (-1.14×)
	↑ > 1.2×	1	3	4
	↓ > 1.2×	14	17	17
	↓ > 2×	0	1	4

not expect to beat PostgreSQL's plans (which have access to correct statistics on these instances), PAR^{2}QO in fact performs better than PostgreSQL in quite a few cases, demonstrating that robust plans can offer protection against selectivity errors regardless of their causes. In contrast, Kepler suffers severe regression in some cases, with slowdowns of up to $-6\times$ compared to the baseline.

5.5 Unforeseen Query Workloads and Effectiveness of CARVER for Training

We design the following experiment to see how different learned methods perform when presented with unforeseen query workloads, and to evaluate how effective CARVER is as a training workload (in comparison with Uniform and Historical). First, we train each method exclusively on each one of these workloads; then, we test the method on a *mixed* workload, which samples queries with equal probability from all three workloads. This mixed testing workload simulates a setting where the testing queries are drawn from a distribution from the training one, which commonly arise in practice. We want to evaluate how PAR^{2}QO, Kepler, and Bao perform this setting, and determine which training workload generally leads to models with best performance.

The results are shown in Table 2. First, we see that PAR^{2}QO adapts to the unforeseen testing workload much better than Kepler and Bao, regardless of the training workload used. PAR^{2}QO still outperforms the PostgreSQL baseline significantly: 1.75× when trained with CARVER and $\geq 1.35\times$ otherwise. On the other hand, Kepler and Bao experience overall slowdown, with far more templates experiencing significant regression. When trained using Historical, Kepler's $-1.65\times$ slowdown — a stark reversal of its 1.94× speedup in Table 1 — can be explained by the fact that Historical queries have less diverse parameter values, which makes it harder for Kepler's model to generalize. We also provide detailed template-level performance in Figure 4.[3] Similar to Figure 3, while Kepler achieves strong performance on some templates (e.g., $C19$ and $C30$), it also suffers severe degradation in others (e.g., $H3$-4, $H23$, and $C31$). In contrast, PAR^{2}QO again avoids such extreme regressions effectively, even over this unforeseen query workload.

Second, training using CARVER generally gives better performance than training using Uniform or Historical, regardless of the learned method tested. For PAR^{2}QO and Bao, the advantages are clear (>25% and >11%, respectively). For Kepler, while CARVER's advantage over Historical is clear, its advantage over Uniform is more subtle but still evident from the fewer number of significant/severe regressions. Overall, these experiments confirm the superiority of CARVER over others for training learned optimizers.

Finally, we also experimented with a variant of CARVER that samples only from individual tables independently (unlike our default CARVER, which samples from join subquery results). Experimental results, presented in Appendix G.5 , show that this variant is less effective than our default CARVER.

[3]Templates 14, 20, and 29 are excluded in Figure 4, as both PAR^{2}QO and Kepler's results show no significant speedup/regression beyond 1.2×.

6 RELATED WORK

Parametric Query Optimization. PQO has been extensively studied over the years [24, 25]. The primary focus of traditional PQO methods is minimizing the optimizer's invocation through plan caching and reuse strategies, ensuring the plan's cost remains acceptable while avoiding unnecessary re-optimization [2, 7, 17]. Different approaches have been explored to address this challenge, including caching only *one* plan with the lowest expected cost for the entire selectivity space [1, 7, 8], caching a *set* of plans [17, 20, 26], or progressively caching multiple plans over time [2, 6, 29]. Some works propose to build density maps through query clustering to better capture selectivity variations [2, 20], or building a "plan diagram" as a color-code visualization [13, 41] to provide a structured view of the selectivity space and how different plans dominate various regions. For a more comprehensive review of traditional PQO techniques, we direct readers to [12]. Recent work, SCR [17] applies an online selectivity and cost check to evaluate if a cached plan can be reused guaranteeing the bound on cost sub-optimality, and if a new plan needs to be added into the cache. Most recent efforts, including QueryLog [44] and Kepler [15], extend PQO by incorporating execution performance optimization and learning from query executions. QueryLog formalizes PQO problem as a two-step decision process, generating candidate plans and selecting the most suitable one. Kepler and PAR^{2}QO adhere to this structure. Since Kepler and QueryLog have been thoroughly discussed and compared with PAR^{2}QO in Section 3.4, we omit further details here.

Robust query optimization. Traditional query optimizers often select plans that perform poorly in practice, with cardinality estimation errors being a primary cause of such degradations [31, 32]. RQO aims to select robust plans that remain stable despite inaccuracies in cardinality estimates [35, 52]. To quantify these uncertainties and guide robust plan selection, prior research assumes cardinality as a probability density function [4, 9], a uniform distribution over the entire space [1], or within a bounding box [5, 19]. More recent work has developed advanced plan selection techniques: Some approaches consider worst-case sub-optimality [16, 40, 45], define robustness metrics based on cost slope and integral [48], or compute optimality or near-optimality ranges as robustness indicators [14, 49]. More recent work selects plans based on learned execution time distribution [34], and prioritizes plans using expected penalty [50]. Another line of research explores *adaptive processing*, where runtime observations guide dynamic plan switching to avoid poor plan executions [16, 18, 43, 46, 48]. For a comprehensive overview of techniques in robust query processing, we refer readers

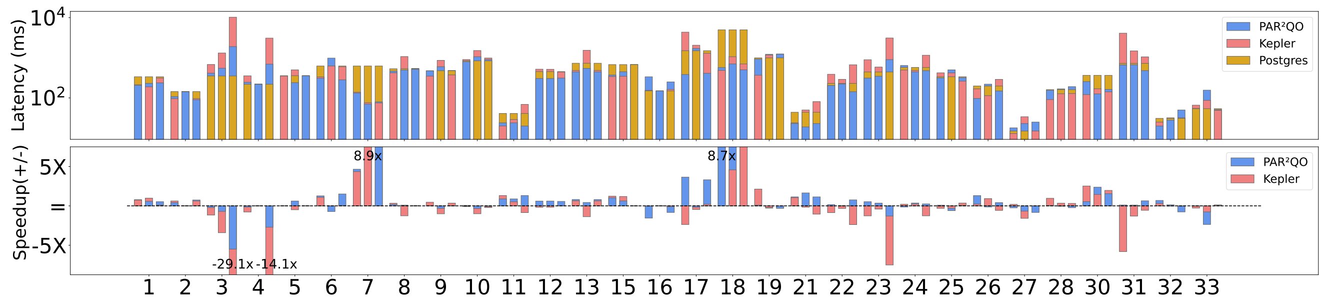


Figure 4: Workload effectiveness evaluation on JOB. Each query template shows three bars (left to right: CARVER, Uniform, Historical), representing different workloads used for training. Each bar shows three overlapping segments, corresponding to PAR^{2}QO, Kepler and PostgreSQL. The top panel compares execution latency; the lower shows the speedup/regression of PAR^{2}QO and Kepler relative to PostgreSQL.

to this survey [22]. Previous work [10, 13–15, 41, 42, 44, 49] implicitly considers robustness when pruning plans, while PAR^{2}QO takes a step further by directly integrating plan robustness into PQO.

7 CONCLUSION AND FUTURE WORK

Selecting a PQO method in practice involves a trade-off. Kepler can sometimes find the fastest plans with low inference latency but requires expensive model training and is prone to severe regressions. QueryLog is easy to set up but may fail to improve execution performance beyond the optimizer. By integrating penalty-aware robustness into PQO, PAR^{2}QO achieves a better balance between efficiency, stability, and practicality. Future work includes further expanding the search space of robust plans; sharing computation across templates; and adapting the error model/cache dynamically for evolving database states and workloads.

ACKNOWLEDGMENTS

This paper is supported by NSF grant IIS-2402823. Work by Agarwal is also supported by NSF grant CCF-2223870 and by US-Israel Binational Science Foundation Grant 2022131. Work by Yang is also supported by NSF grants IIS-2008107 and II-2211526.

REFERENCES

[1] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. 2010. On the Stability of Plan Costs and the Costs of Plan Stability. *Proc. VLDB Endow.* 3, 1 (2010), 1137–1148. https://doi.org/10.14778/1920841.1920983

[2] 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*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 402–413. https://doi.org/10.1109/ICDE.2012.57

[3] 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, Article 54 (March 2024), 25 pages. https://doi.org/10.1145/3639309

[4] Brian Babcock and Surajit Chaudhuri. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 119–130. https://doi.org/10.1145/1066157.1066172

[5] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. 2005. Proactive Re-optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 107–118. https://doi.org/10.1145/1066157.1066171

[6] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. Knowl. Data Eng.* 21, 4 (2009), 582–594. https://doi.org/10.1109/TKDE.2008.160

[7] 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*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 531–542. https://doi.org/10.1145/1807167.1807226

[8] Francis C. Chu, Joseph Y. Halpern, and Johannes Gehrke. 2002. Least Expected Cost Query Optimization: What Can We Expect?. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis (Eds.). ACM, 293–302. https://doi.org/10.1145/543613.543651

[9] Francis C. Chu, Joseph Y. Halpern, and Praveen Seshadri. 1999. Least Expected Cost Query Optimization: An Exercise in Utility. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, Victor Vianu and Christos H. Papadimitriou (Eds.). ACM Press, 138–147. https://doi.org/10.1145/303976.303990

[10] Atreyee Dey, Sourjya Bhaumik, Harish Doraiswamy, and Jayant R. Haritsa. 2008. Efficiently approximating query optimizer plan diagrams. *Proc. VLDB Endow.* 1, 2 (2008), 1325–1336. https://doi.org/10.14778/1454159.1454173

[11] 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. https://doi.org/10.14778/3484224.3484234

[12] Bailu Ding, Vivek Narasayya, and Surajit Chaudhuri. 2024. Extensible Query Optimizers in Practice. *Foundations and Trends® in Databases* 14, 3-4 (2024), 186–402. https://doi.org/10.1561/1900000077

[13] 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*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, 1081–1092. http://www.vldb.org/conf/2007/papers/research/p1081-d.pdf

[14] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *Proc. VLDB Endow.* 1, 1 (2008), 1124–1140. https://doi.org/10.14778/1453856.1453976

[15] 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. https://doi.org/10.1145/3588963

[16] Anshuman Dutt and Jayant R Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1039–1050.

[17] 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*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1539–1554. https://doi.org/10.1145/3035918.3064040

[18] Anshuman Dutt, Sumit Neelam, and Jayant R Haritsa. 2014. QUEST: An exploratory approach to robust query processing. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1585–1588.

[19] Belgin Ergenc, Franck Morvan, and Abdelkader Hameurlain. 2007. Robust Placement of Mobile Relational Operators for Large Scale Distributed Query

Optimization. In *Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2007), 3-6 December 2007, Adelaide, Australia*, David S. Munro, Hong Shen, Quan Z. Sheng, Henry Detmold, Katrina Falkner, Cruz Izu, Paul D. Coddington, Bradley Alexander, and Si-Qing Zheng (Eds.). IEEE Computer Society, 227–235. https://doi.org/10.1109/PDCAT.2007.53

[20] Antara Ghosh, Jignashu Parikh, Vibhuti S. Sengar, and Jayant R. Haritsa. 2002. Plan Selection Based on Query Clustering. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 179–190. https://doi.org/10.1016/B978-155860869-6/50024-X

[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. https://doi.org/10.14778/3503585.3503586

[22] Jayant R. Haritsa. 2024. Robust Query Processing: A Survey. *Found. Trends Databases* 15, 1 (2024), 1–114. https://doi.org/10.1561/1900000089

[23] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. 2019. Join on Samples: A Theoretical Guide for Practitioners. *Proc. VLDB Endow.* 13, 4 (2019), 547–560. https://doi.org/10.14778/3372716.3372726

[24] 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. https://doi.org/10.1016/B978-155860869-6/50023-8

[25] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1992. Parametric Query Optimization. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, Li-Yan Yuan (Ed.). Morgan Kaufmann, 103–114. http://www.vldb.org/conf/1992/P103.PDF

[26] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1997. Parametric Query Optimization. *VLDB J.* 6, 2 (1997), 132–151. https://doi.org/10.1007/S007780050037

[27] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 631–646. https://doi.org/10.1145/2882903.2882940

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[29] Allison W. Lee and Mohamed Zaït. 2008. Closing the query processing loop in Oracle 11g. *Proc. VLDB Endow.* 1, 2 (2008), 1368–1378. https://doi.org/10.14778/1454159.1454178

[30] Kukjin Lee, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (2023), 2871–2883. https://doi.org/10.14778/3611479.3611494

[31] 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. https://doi.org/10.14778/2850583.2850594

[32] 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. https://doi.org/10.14778/2850583.2850594

[33] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/S00778-017-0480-7

[34] Yifan Li, Xiaohui Yu, Nick Koudas, Shu Lin, Calvin Sun, and Chong Chen. 2023. dbET: Execution Time Distribution-based Plan Selection. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[35] Guy M. Lohman. 2017. Query Optimization - Are We There Yet?. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (LNI)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.), Vol. P-265. GI, 25–26. https://dl.gi.de/handle/20.500.12116/646

[36] Ryan Marcus. 2020. https://github.com/learnedsystems/BaoForPostgreSQL.

[37] 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*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1275–1288. https://doi.org/10.1145/3448016.3452838

[38] Satoshi Nagayasu. 2023. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.

[39] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058. http://dl.acm.org/citation.cfm?id=1164217

[40] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, 37–48. https://doi.org/10.1145/2213556.2213565

[41] 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*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 1228–1240. http://www.vldb.org/archives/website/2005/program/paper/fri/p1228-reddy.pdf

[42] 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*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 1228–1240. http://www.vldb.org/archives/website/2005/program/paper/fri/p1228-reddy.pdf

[43] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Transactions on Database Systems (TODS)* 46, 3 (2021), 1–45.

[44] 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. https://doi.org/10.14778/3494124.3494126

[45] Srinivas Karthik Venkatesh, Jayant R. Haritsa, Sreyash Kenkre, and Vinayaka Pandit. 2018. A Concave Path to Low-overhead Robust Query Processing. *Proc. VLDB Endow.* 11, 13 (2018), 2183–2195. https://doi.org/10.14778/3275366.3275368

[46] Ziyun Wei and Immanuel Trummer. 2024. ROME: Robust Query Optimization via Parallel Multi-Plan Execution. *Proc. ACM Manag. Data* 2, 3 (2024), 170. https://doi.org/10.1145/3654973

[47] David P Williamson and David B Shmoys. 2011. *The design of approximation algorithms*. Cambridge university press.

[48] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. 2018. Robustness Metrics for Relational Query Execution Plans. *Proc. VLDB Endow.* 11, 11 (2018), 1360–1372. https://doi.org/10.14778/3236187.3236191

[49] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. 2018. On the Calculation of Optimality Ranges for Relational Query Execution Plans. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 663–675. https://doi.org/10.1145/3183713.3183742

[50] Haibo Xiu, Pankaj K. Agarwal, and Jun Yang. 2024. PARQO: Penalty-Aware Robust Plan Selection in Query Optimization. *Proc. VLDB Endow.* 17, 13 (2024), 4627–4640. https://doi.org/10.14778/3704965.3704971

[51] Haibo Xiu, Yang Li, Qianyu Yang, Weihang Guo, Yuxi Liu, Pankaj K. Agarwal, Sudeepa Roy, and Jun Yang. 2025. Hint-QPT: Hints for Robust Query Performance Tuning. *Proc. VLDB Endow.* 18, 12 (2025).

[52] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust query optimization methods with respect to estimation errors: A survey. *ACM Sigmod Record* 44, 3 (2015), 25–36.

Table 3: Summary of notations.

Symbol	Description
Γ	A parameterized query template
Q	A query instance
W	A workload of query Q from Γ
$\mathcal{D}$	A database instance
π	A physical query execution plan
$\widehat{\mathsf{Sel}}(Q)$	A vector of estimated selectivities for Q
$f(\mathbf{s}\mid\hat{\mathbf{s}})$	Distribution of true selectivity $\mathbf{s}$ conditioned on $\hat{\mathbf{s}}$
$\mathsf{Opt}(Q, \mathbf{s})$	Optimizer-chosen plan for Q at selectivity $\mathbf{s}$
$\mathsf{Cost}(\pi, \mathbf{s})$	Optimizer-estimated cost of plan π at selectivity $\mathbf{s}$
τ	Tolerance threshold of estimated cost differences
$\mathsf{Penalty}(\pi, \mathbf{s})$	Penalty incurred by plan π under selectivity $\mathbf{s}$
$\mathsf{E}[\cdot]$	Expectation over selectivity distribution $f(\mathbf{s}\mid\hat{\mathbf{s}})$
$\mathcal{E}$	Learned selectivity error model from $\mathcal{D}$ and W
$\mathcal{C}$	PARQO′'s cache
$\mathcal{P}$	Set of initial candidate plans for a query template
$\mathcal{L}$	Set of sampled probe locations
m	# of clusters
$\mathfrak{P}$	Plan-penalty profiles; $\mathfrak{P}[\pi, \cdot]$ is the penalty profile of π
$\mathbf{C}$	Plan-cost matrix; $\mathbf{C}[\pi, \cdot]$ is the cost profile of π
n	# of probe locations per cluster
k	Target # of plans for reducing $\mathcal{P}$
θ_{KL}	Threshold for KL divergence
h_i	# of queries in W that hits cluster i

A NOTATION SUMMARY

Please see summarized notations used in this paper in Table 3.

B PSEUDOCODE OF ALGORITHMS

Please see detailed pseudocode for PAR^{2}QO in Algorithms 1 to 4. Besides, we present the pseudocode for PARQO′ in Algorithm 5.

C JUSTIFICATION FOR BIAS CORRECTION AT RUNTIME PLAN SELECTION

Given $\widehat{\mathsf{Sel}}(Q)$, we aim to estimated the expected penalty $\mathsf{E}[\mathsf{Penalty}(\pi, \mathbf{S}) \mid \widehat{\mathsf{Sel}}(Q)]$ for each π using $\mathfrak{P}$ and probe locations $\mathcal{L}$. This expectation is derived over the target distribution conditioned on the new estimates $f(\mathbf{s}\mid\widehat{\mathsf{Sel}}(Q))$, which is obtained from the error model $\mathcal{E}$.

Suppose we have an *overall* probability density distribution of selectivity $\mathbf{F}(\mathbf{y})$, and R samples $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_R\}$ drawn from $\mathbf{F}(\mathbf{y})$. Using importance sampling, we estimate the target expectation as:

$$\mathsf{E}[\mathsf{Penalty}(\pi, \mathbf{S}) \mid \widehat{\mathsf{Sel}}(Q)] = \sum_{i=1}^{R} \frac{f(\mathbf{y}\mid\widehat{\mathsf{Sel}}(Q))}{\mathbf{F}(\mathbf{y})} \cdot \mathfrak{P}[\pi, \mathbf{y}].$$

These R samples can be divided into m independent clusters $\{M_1, M_2, .., M_m\}$, where the i-th cluster M_i contains $R_i = h_i \cdot N$ samples, with $h_i \geq 1$ representing the density of cluster M_i and N is a constant. Thus, $R = \sum_{i=1}^{m} h_i \cdot N$.

While $\mathbf{F}(\mathbf{y})$ is unknown, we know the probability of $\mathbf{y}$ being sampled within the i-th cluster, $\mathbf{Q}_i(\mathbf{y})$. Since clusters are independent, the probability of selecting the i-th cluster is $\frac{h_i}{\sum_{i=1}^{m} h_i}$. Since a sample $\mathbf{y}$ can be drawn from *any* cluster in m clusters, the probability of $\mathbf{y}$ being sampled from the $\mathbf{F}(\mathbf{y})$ can be calculated as the cumulated probability of being sampled from each cluster: $\sum_{i=1}^{m} \frac{h_i}{\sum_{i=1}^{m} h_i} \cdot \mathbf{Q}_i(\mathbf{y}) = \frac{1}{R} \sum_{i=1}^{m} h_i \mathbf{Q}_i(\mathbf{y})$. And finally, after dropping terms that do not affect plan ranking, the expected penalty is

Algorithm 1 PAR^{2}QO: Cache Preparation (Offline)

```
Input: Workload $W$ with queries from template $\Gamma$; database $\mathcal{D}$
Output: Plan-penalty profiles $\mathfrak{P}$, error model $\mathcal{E}$, probe clusters
    $\{\langle \mathcal{L}_i, \mathbf{s}_i, h_i \rangle\}$
    — Step 1: Error Model Construction —
 1: Learn error model $\mathcal{E}$ from $W$ and $\mathcal{D}$
    — Step 2: Sampling Probe Locations —
 2: Initialize cluster cache as empty
 3: for each query $Q_i \in W$ do
 4:     Derive $f(\mathbf{s}\mid\widehat{\mathsf{Sel}}(Q_i))$ from $\mathcal{E}$
 5:     if minimum KL divergence to existing cluster $< \theta_{\mathsf{KL}}$ then
 6:         Reuse cluster and increment hit counter $h_i$
 7:     else
 8:         Sample $n$ probe locations from $f(\mathbf{s}\mid\widehat{\mathsf{Sel}}(Q_i))$
 9:         Cache new cluster $\langle \mathcal{L}_i, \widehat{\mathsf{Sel}}(Q_i), h_i{=}1 \rangle$
10:     end if
11: end for
12: $\mathcal{L} \leftarrow \bigcup_i \mathcal{L}_i$                    ▷ Total $mn$ probe locations
    — Step 3: Computing Plan-Penalty Profiles —
13: Call $\mathsf{Opt}(Q, \mathbf{s})$ for each $\mathbf{s} \in \mathcal{L}$ to collect $\mathcal{P}$
14: for each $\pi \in \mathcal{P}$ and $\mathbf{s} \in \mathcal{L}$ do
15:     Compute $\mathsf{Cost}(\pi, \mathbf{s})$ and store in $\mathbf{C}[\pi, \cdot]$
16:     Compute $\mathsf{Penalty}(\pi, \mathbf{s})$ and store in $\mathfrak{P}[\pi, \cdot]$
17: end for
```

Algorithm 2 PlanReduction-τCover: Reducing Candidate Plans via τ-Coverage

```
Input: Candidate plans $\mathcal{P}$, plan-cost matrix $\mathbf{C}$, target size $k$
Output: Reduced plan set $\mathcal{S}$ of size at most $k$
 1: Initialize reduced plan set $\mathcal{S} \leftarrow \emptyset$
 2: while $|\mathcal{S}| < k$ and $\mathcal{P} \setminus \mathcal{S} \neq \emptyset$ do
 3:     Identify plan $\pi \in \mathcal{P} \setminus \mathcal{S}$ that covers the most uncovered
        probe locations where:
            $\mathsf{Cost}(\pi, \mathbf{s}) \leq (1 + \tau) \cdot \mathsf{Cost}^{\star}(\mathbf{s})$
 4:     Add $\pi$ to $\mathcal{S}$
 5: end while
 6: return $\mathcal{S}$
```

Algorithm 3 PlanReduction-Similarity: Reducing Candidate Plans via Cost Profile Dissimilarity

```
Input: Candidate plans $\mathcal{P}$, plan-cost matrix $\mathbf{C}$, target size $k$
Output: Reduced plan set $\mathcal{S}$ of size at most $k$
 1: Initialize reduced plan set $\mathcal{S} \leftarrow \emptyset$
 2: Randomly select an initial plan $\pi$ and add to $\mathcal{S}$
 3: while $|\mathcal{S}| < k$ and $\mathcal{P} \setminus \mathcal{S} \neq \emptyset$ do
 4:     For each remaining plan, compute the minimum cost profile
        distance to plans in $\mathcal{S}$
 5:     Select the plan with the maximum such distance
 6:     Add selected plan to $\mathcal{S}$
 7: end while
 8: return $\mathcal{S}$
```

Algorithm 4 PAR^{2}QO: Runtime Plan Selection (Online)

```
Input: Query instance $Q$, plan-penalty profiles $\mathfrak{P}$, error model $\mathcal{E}$, clusters $\{\langle \mathcal{L}_i, \mathbf{s}_i, h_i \rangle\}$
Output: Robust plan $\pi$ for query $Q$
1: Derive $f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q))$ from $\mathcal{E}$
2: for each plan $\pi \in \mathfrak{P}$ do
3:     Estimate expected penalty via importance sampling:
           $\sum_{i=1}^{m} \sum_{j=1}^{n} \frac{f(\mathbf{s}_{i,j}|\widehat{\mathsf{Sel}}(Q))}{h_i f(\mathbf{s}_{i,j}|\mathbf{s}_i)} \cdot \mathfrak{P}[\pi, \mathbf{s}_{i,j}]$
4: end for
5: return plan $\pi$ with the lowest estimated expected penalty
```

Algorithm 5 PARQO′: An extended PARQO baseline

```
Input: Workload $W$ with queries from template $\Gamma$; database $\mathcal{D}$
Output: Plan cache $\mathcal{C} = \{\langle \pi_i, \mathbf{s}_i \rangle\}$ for query template $\Gamma$
    Offline Cache Preparation
1: Learn error model $\mathcal{E}$ from $W$ and $\mathcal{D}$
2: Initialize plan cache $\mathcal{C} \leftarrow \emptyset$
3: for each query $Q \in W$ do
4:     covered ← False
5:     for each cached $\langle \pi_i, \mathbf{s}_i \rangle \in \mathcal{C}$ do
6:         if $\mathrm{KL}(f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q)) \parallel f(\mathbf{s}|\mathbf{s}_i)) < \theta_{\mathrm{KL}}$ then
7:             covered ← True
8:             break                              ▷ $Q$ is already covered
9:         end if
10:    end for
11:    if not covered then
12:        $\pi \leftarrow$ FindRobustPlan($Q, \mathcal{E}$)    ▷ Same as PARQO [50]
13:        Add $\langle \pi, \widehat{\mathsf{Sel}}(Q) \rangle$ to $\mathcal{C}$
14:    end if
15: end for
    Online Plan Selection
16: function SelectPlan($Q$)
17:     $\langle \pi^*, \mathbf{s}^* \rangle \leftarrow \arg\min_{\langle \pi_i, \mathbf{s}_i \rangle \in \mathcal{C}} \mathrm{KL}(f(\mathbf{s}|\widehat{\mathsf{Sel}}(Q)) \parallel f(\mathbf{s}|\mathbf{s}_i))$
18:     return $\pi^*$
19: end function
```

estimated as:

$$\sum_{i=1}^{R} \frac{f(\mathbf{y}|\widehat{\mathsf{Sel}}(Q))}{\sum_{i=1}^{m} h_i \mathbf{Q}_i(\mathbf{y})} \cdot \mathfrak{P}[\pi, \mathbf{y}].$$

If $R = mn$, this becomes,

$$\mathsf{E}[\mathsf{Penalty}(\pi, \mathbf{S}) \mid \widehat{\mathsf{Sel}}(Q)] = \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{f(\mathbf{y}|\widehat{\mathsf{Sel}}(Q))}{h_i \mathbf{Q}_i(\mathbf{y})} \cdot \mathfrak{P}[\pi, \mathbf{y}].$$

This aligns with our bias correction in Section 3.3 by simply replacing $\mathbf{y}$ with $s_{i,j}$ and $\mathbf{Q}_i(\mathbf{y})$ with $f(\mathbf{s}_{i,j}|\mathbf{s}_i)$.

D A DETAILED EXAMPLE USING PAR^{2}QO

In this section, we present a detailed walkthrough of how to apply PAR^{2}QO (Section 3) to populate cache and select robust plans for parametric queries. Consider template 17 below from the JOB benchmark [32].

```sql
SELECT MIN(n.name) AS member_in_charnamed_american_movie,
       MIN(n.name) AS a1
FROM cast_info AS ci, company_name AS cn, keyword AS k,
     movie_companies AS mc, movie_keyword AS mk,
     name AS n, title AS t
WHERE cn.country_code ='XXX' AND k.keyword ='XXX' AND n.name LIKE
       'XXX'
  AND n.id = ci.person_id AND ci.movie_id = t.id
  AND t.id = mk.movie_id AND mk.keyword_id = k.id
  AND t.id = mc.movie_id AND mc.company_id = cn.id
  AND ci.movie_id = mc.movie_id AND ci.movie_id = mk.movie_id
  AND mc.movie_id = mk.movie_id;
```

The relevant dimensions we defined for this template are cn^σ, k^σ, n^σ, $mc \bowtie cn^\sigma$, $k^\sigma \bowtie mk$, and $n^\sigma \bowtie ci$, where the superscript σ denotes the presence of at least one local selection condition on a table. The error model built from the training workload is a joint distribution profiling the selectivity errors observed from these relevant dimensions.

We begin by identifying representative clusters and sampling probe locations from the selectivity distribution $f(\mathbf{S}|\widehat{\mathsf{Sel}}(Q))$ of each cluster, using the balanced sampling strategy described in Section 3.1 and Algorithm 1. Specifically, each query is treated as a selectivity distribution derived from the error model, conditioned on the selectivity estimates. If a query is not sufficiently covered by any existing cluster—determined via a KL divergence threshold—we create a new cluster centered at that query and sample from it. During this process, we also track how many training queries fall into each cluster. For template 17, this procedure produces 6 clusters, from which we draw 50 probe locations each, resulting in a total of 300 probe points; and the most frequently hit cluster is matched by 22 (of 50) training queries. By invoking the optimizer at each probe location, we collect execution plans, leading to 53 unique candidate plans across all probes.

Next, we construct a penalty profile for each plan. Specifically, we evaluate the optimizer-estimated cost of each plan at every probe location, forming a plan-cost matrix $\mathbf{C}$ of shape $[53 \times 300]$, where each row captures the cost profile of a candidate plan across all probes. Using Equation (1), we compute the corresponding plan-penalty matrix $\mathfrak{P}$.

To finalize the cache, we reduce the number of candidate plans by applying the τ-Cover reduction strategy with a target size $k = \max(0.2 \cdot |\mathcal{P}|, 10)$. This pruning step results in 10 selected plans, reducing $\mathfrak{P}$ to a $[10\times300]$ matrix. Appendix F visualizes the reduced plan space, and Appendix G.3 explores the impact of different values of k on performance. At runtime, we estimate the robustness of each cached plan using importance sampling with bias-corrected weights, as detailed in Section 3.3 and Appendix C. Interestingly, despite caching 10 plans, we find that two plans dominate the space: one is assigned as the robust plan for 81 (of 200) test queries, and the other for 75.

E ADDITIONAL DETAILS ON CARVER

A Detailed Example using CARVER. With an example, we illustrate how to use CARVER to generate synthetic workloads for training and evaluation. As described in Section 4, CARVER provides the flexibility of using a mixture distribution consisting of multiple components. For the query template 17 from JOB, the first possible component, which we denote by P_1, involves three two-table querylets, which are $mc \bowtie cn^{\sigma}$, $k^{\sigma} \bowtie mk$, and $n^{\sigma} \bowtie ci$; P_1 is then defined as the product distribution of the parameter distribution of each querylet. By default, the CARVER setup used by our experiments in Section 5 would include just P_1.

The second possible component, P_2, represents the full query (joining all tables and including all local selections). If we restrict CARVER to include P_2 only, it would be similar to the Uniform workload used by our experiments in Section 5 and by Kepler [15] in that it does not try to cover the ranges of possible subquery cardinalities. However, it still provides a more balanced coverage of selectivities for full parameter combinations than Uniform.

Finally, the third possible component, P_3, involve all single-table subqueries, i.e., with possible selections but no joins. If we restrict CARVER to include P_3 only, it amounts to sampling independently from individual tables. Our default CARVER setup in experiments does not choose this option for the following reasons. First, sampling independently from single tables can be highly inefficient if we want to produce parameter settings that yield non-empty query results after joining. Second, selectivity uncertainty is often amplified by correlations and dependencies among joining tables, which single-table sampling fails to capture. Therefore, using P_3 alone may not reflect real query behaviors and may degrade the training effectiveness of the workload. In Appendix G.5, we confirm this observation by comparing this alternative with the default CARVER setup using P_2.

Additional Discussion on CARVER. There are several points worth noting about CARVER. First, we emphasize that CARVER by design goes beyond merely covering ranges of selectivities for individual, single-table selections. The generation procedures in Section 4 generally support multi-table join subqueries and, when applicable, joint sampling of multiple query parameters. This feature ensures that the workload reflects the dependencies that exist among join and selection predicates, as handling such dependencies has been widely recognized [28, 30, 31] as more challenging for query optimizers than single-table selections selectivities.

Second, we note that in some cases, there are methods for combining the procedures in Section 4 to jointly generate parameters for more complex predicates. For example, it is not difficult to handle a conjunction of equality selections and inequality selections, or a conjunction of equality selections and `LIKE` selections. We will not exhaustively list all such possibilities here. Moreover, there always exists a default method for handling a combination of multiple parameterized predicates, simply by generating parameters independently using subquery templates that share the same base query but differ in their parameterized remainder predicates. This method can be applied when the complex predicate does not fall into a case discussed earlier, or when we prefer a cheaper method than the earlier ones.

Third, the base query q may be too expensive to compute if q joins many large tables and has few selection predicates (recall that all parameterized predicates go into the remainder predicate θ instead). In this case, we instead compute a sample of q's result rows. CARVER currently implements the simple approach of taking a uniform random sample of each table involved in the join. While this approach is widely known to suffer from low join rate, it has worked sufficiently well for experiments in Section 5. If needed, more sophisticated methods for sampling from join can be used instead [23, 27].

Finally, it is sometimes useful to require that every generated query returns a non-empty result set, particularly for production workloads where queries typically yield results, unlike ad hoc exploration workloads. There are also other technical reasons (see Section 5 for details). Therefore, CARVER offers an option to enforce non-empty results. When query parameters are sampled independently from multiple distributions, CARVER executes the query to verify validity and rejects settings that yield empty results. If the rejection rate is too high, CARVER suggests switching to a mixture component that samples directly from the full join result, which automatically ensures non-empty results by design.

F PLAN VISUALIZATION AFTER PLAN REDUCTION

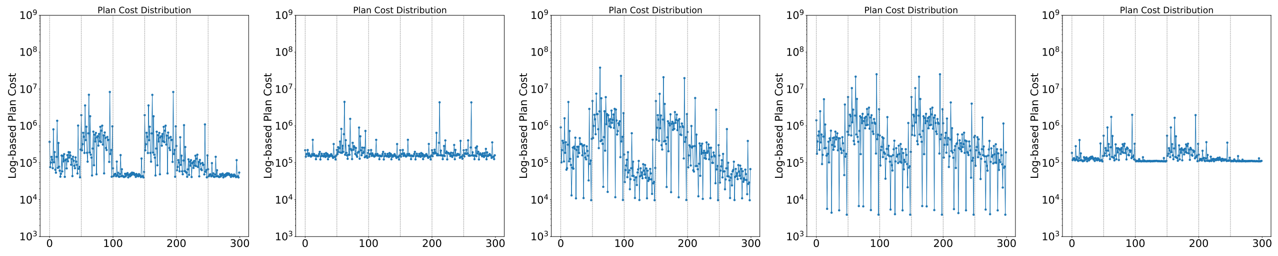
Figure 5 visualizes the cost distribution of plans after applying different reduction methods across selectivity probes for JOB template 17, with CARVER as the underlined workload. For the λ-Cover reduction method, we present the top five queries with the largest coverage range. For similarity-based reduction, we set the target plan size $k = 5$. The remaining plans from both methods show significant overlaps. Using the set by either reduction method as the plan cache achieves over a 4× speedup compared to PostgreSQL, demonstrating their effectiveness in selecting robust plans. Additionally, for plans reduced by cost profile similarity, we observe that those plans within the same group exhibit highly similar cost distributions, while plans from different groups display distinct behavior. These observations suggest the JS divergence is a suitable metric for measuring plan similarity, and the k-center greedy method can effectively preserve plan diversity.

G ADDITIONAL DETAILS ON EXPERIMENTS

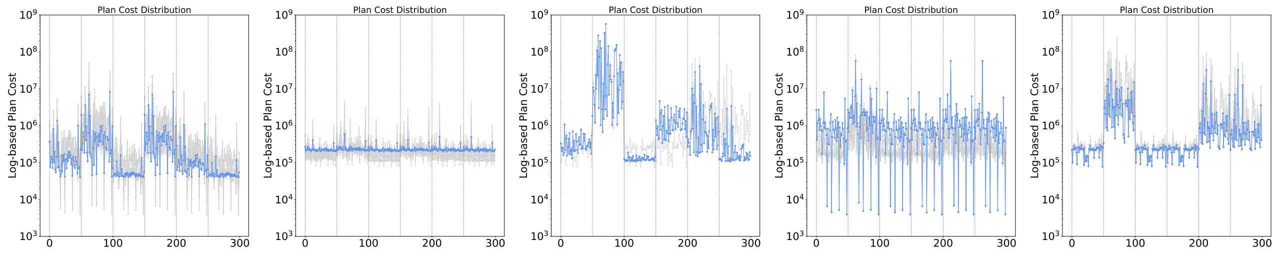
G.1 Details on Experimental Configurations

CARVER, Uniform, Historical setup. For **CARVER**, as mentioned in Section 4, if $\gamma_{k,j}$ is a querylet with a combination of multiple parameterized predicates, we generate parameters for each binding independently. For **Uniform** workload generation, certain JOB templates, including 25, 28-31, may encounter memory overflow when performing full-table joins. To mitigate this, we apply a random sampling approach. We initially sample 10% data from each table and incrementally increase the sampling rate if the number of unique parameter settings in $\text{params}(\Gamma)$ is insufficient. For **Historical** workload generation, parameter settings for each predicate are extracted from the original benchmark queries in JOB and then permuted across all predicates within each template Γ to generate $\text{params}(\Gamma)$. The synthesized queries are then split into two distinct sets for training and testing.

For each workload generation method, we apply a posteriors filter after the initial synthesis to ensure only *valid* queries are included. Similar guarantees are also discussed in Kepler [15]. This filter removes queries that return zero rows (before final aggregation), ensuring all queries in the workload produce at least one row. Beyond practical considerations, there is a technical necessity for this filtering due to PostgreSQL's plan hinting mechanism



(a) Reduction by τ-Approximate Cover



(b) Reduction by Cost Profile Similarity

Figure 5: Plan cost distribution of remaining plans after different reduction methods for template 17 in JOB. Initially, 53 candidate plans are obtained by Opt from 50 selectivity probe locations across 6 clusters, separated by vertical dashed lines. We manually set the target remaining plan size k to 5 (only for this presentation, not the same as $\max(0.2 \cdot |\mathcal{P}|, 10)$, which we used as default in our experiments), with each plot presenting one of these plans. For similarity-based reduction, plans within the same group of according remaining plan are also shown (gray lines).

[38]. Without hinting, if a query contains a subquery that produces zero rows, the optimizer may detect this early and terminate execution prematurely. Such premature termination can lead to misleading performance evaluations, as the observed speed may stem from early termination rather than actual plan efficiency. However, when a complete physical plan is hinted, PostgreSQL follows the prescribed join order and join strategies step-by-step, executing the full plan regardless of intermediate zero-row results. To prevent premature termination and ensure meaningful query execution, we manually enforce that all queries in the workload return at least one row. We observe that in certain templates, the one-component scenario is inefficient when this filter is applied. Even after synthesizing over 10,000 initial queries, the number of valid queries remains insufficient—for example, in templates 24 and 29. For these cases, CARVER switches to the two-component mixture model (a mixture of P_1 and P_2 as described in Appendix E), using equal weights. And this mixture ensures non-empty results.

Error model setup. In our experiments, PAR^{2}QO, PARQO$'$, and PARQO use the same error model constructed by executing relevant subqueries from the training workload to collect selectivity estimation errors. Then we use kernel density estimation to build the probability distribution. This process takes less than 13 minutes on average per JOB template and less than 9 minutes per DSB template.

DSB setup. For DSB [11], we use a scale factor of 2 for data generation, and we concentrate on the 15 SPJ query templates. The only reason we are limited to SPJ queries is `pg_hint_plan`, the plan injection mechanism we currenuse for PostgreSQL. Our general approach, however, is not limited to SPJ queries in principle.

For query workloads based on the above templates, in addition to CARVER, we generate a **Gaussian** workload by defining a Gaussian distribution centered at 0 with a standard deviation of 2. We then use `dsqgen` [11] to independently sample params for each binding parameter from this distribution.

Kepler setup. In our experiments, we use three generations of RCE (Row Count Evolution)[15] to perturb all sub-queries within the given template. For the exponent base and range, we also refer to the default value: $b = 10$ and $m = 2$. On average, Kepler populates 199 candidate plans for JOB and 130 for DSB. To prune these candidate plans into a small plan set (referred to as "plan covers" [15]) that ensures near-optimal execution latency for all training queries, we execute each candidate plan three times per training query. Then collect the *fast* plans using the default greedy "Set Cover" method. Since the initial training workload is small, we set δ=0.01, ensuring that all queries will be covered by the pruned plans. When training with 400 queries, as in Section 5.3, the resulting set of plans will cover 396 queries ($400 \cdot (1 - \delta)$). We also follow the timeout policy to avoid executing poor plans. After this pruning, an average of 3.4 plans per template remain for JOB, and 1.23 for DSB. Given the small training size, we observe that Kepler's confidence threshold is consistently below 0.6. Setting the threshold to ≥ 0.6 forces all predictions to fall back to the optimizer. Therefore, following the recommendations in [15], we set the confidence threshold to 0.4.

PARQO$'$ and PARQO setup. PARQO$'$ and PARQO utilize the same selectivity error model as PAR^{2}QO. For each respective selectivities $\mathbf{s}_i$ (referred to as "anchors" [50]), we use Sobols method with 128 seeds to identify sensitive dimensions and by default the top 3 sensitive dimensions are selected to find robust plans. A same

θ_{KL} threshold ($\ln(200)$ as recommended [50]) is used when calculating the KL divergence of $f(\mathbf{s}|\hat{\mathbf{s}})$ from $f(\mathbf{s}|\mathbf{s}_i)$. For PARQO, we only identify one anchor and cache only one robust plan based on this anchor. If the testing query is not within the θ_{KL} threshold, we fall back to default PostgreSQL's optimal plan since the cached robust plan can not be reused in this case. Therefore, the inference time for PARQO includes the time spend on KL-divergence testing plus the planning time incurred by falling back to the optimizer when the test fails. For PARQO′, the inference time includes only the KL-divergence testing time to find the nearest anchor. PARQO′ and PARQO also leverage the same robustness metric (i.e. Equation (1)) as in PAR^{2}QO.

Since PARQO creates only a single anchor and caches one robust plan, our experiments show that its average reuse fraction across all three workloads over 33 JOB templates is just 49%. Furthermore, as shown in Table 6, although PARQO′ incurs approximately 4× higher model preparation time compared to PARQO, it achieves faster runtime inference. T his is because falling back to the default optimizer in PARQO introduces additional overhead from both KL-divergence testing and the optimizer's planning process, making it inefficient in practice.

Bao setup. We configure and train Bao following the procedures outlined in its official tutorial [36]. During training, we execute the full set of training queries across all templates to collect initial data. To improve training stability and coverage, the entire training workload is shuffled three times using different random seeds, with Bao retrained after every 100 queries. For each query, Bao evaluates all 48 valid hint combinations (referred to as arms), which represent different configurations of join and scan methods. While the training cost is amortized across templates—resulting in relatively low average training time per template—the inference time is notably high. As shown in Table 6, Bao's inference time significantly exceeds that of all other methods, including PostgreSQL. This drawback is also acknowledged in Bao's original paper [37], making it less suitable for applications where fast query response is critical.

G.2 Comparison of Query Performance for DSB

For DSB, we compare PAR^{2}QO, Kepler, and PARQO′ with results summarized in Section 5.1. As DSB's data is synthetic, optimizing parametric queries is generally less challenging than JOB. For CARVER, PAR^{2}QO achieves an average latency of 155ms, a 2.19× speedup over PostgreSQL, and 1.05× over Kepler. For Gaussian, both PAR^{2}QO and Kepler achieve a 2.3× speedup over PostgreSQL. Neither PAR^{2}QO nor Kepler exhibits significant regression (i.e., more than 1.2×) across templates. In terms of efficiency, PAR^{2}QO demonstrates a significantly lower model preparation time, requiring only 9.5 minutes per template, compared to over 4 hours for Kepler. PARQO′ also shows strong robustness across both workloads with a moderate training cost of 33.2 minutes per template; however, its speedups fall short of those achieved by PAR^{2}QO and Kepler.

Table 4: Comparison of average execution latency (ms) and number of speedup (↑) / regression (↓) counts across two workloads (CARVER and Gaussian) on DSB.

		CARVER	Gaussian
PostgreSQL	Avg latency	339 ms	351 ms
PAR^{2}QO	Avg latency	155 (2.19×)	152 (2.31×)
	↑ > 1.2×	5	3
	↓ > 1.2×	0	0
	↓ > 2×	0	0
Kepler	Avg latency	163 (2.08×)	152 (2.30×)
	↑ > 1.2×	4	3
	↓ > 1.2×	0	0
	↓ > 2×	0	0
PARQO′	Avg latency	183 (1.83×)	170 (2.06×)
	↑ > 1.2×	5	2
	↓ > 1.2×	0	0
	↓ > 2×	0	0

G.3 Additional Results on Ablation Study for PAR^{2}QO

To understand the impact of different hyperparameters in PAR^{2}QO and verify our design choice, we conduct a systematic ablation study by varying one parameter at a time while keeping the others fixed. Results are reported across three workloads in Table 5, with corresponding system-level statistics in Table 6.

Impact of # of probe locations per cluster (n). By default, PAR^{2}QO uses $n = 50$ probes per cluster to cover the local selectivity space. Increasing n to 200 does not leads to huge performance improvements in latency, however, both training and inference costs rise significantly due to the increased number of candidate plans and heavier calculation of plan penalty and robustness. Conversely, reducing n to 10 leads to noticeable drops in speedup and more significant regressions, indicating insufficient coverage of selectivity uncertainty. Despite the smaller n, PAR^{2}QO still outperforms PostgreSQL, highlighting the inherent robustness protection provided by our method. This also confirms that a moderate n (e.g., 50) strikes a good balance between performance and overhead.

Impact of Plan Reduction Size (k). To validate this choice, we evaluate two extreme cases.

- First, setting $k = 1$ (i.e., selecting only a single plan) leads to a substantial drop in performance and increases the number of severe regressions—particularly under CARVER—highlighting the limited generality of single-plan reuse, as also observed in PARQO. That said, caching one plan still delivers non-trivial speedups over PostgreSQL on the Uniform and Historical workloads, suggesting that these workloads are relatively simple and less sensitive to selectivity variation. These findings also emphasize that CARVER generates more diverse and challenging queries, making it a more rigorous and effective benchmark for evaluating the robustness and generalization of PQO methods.
  Moreover, the $k = 1$ configuration incurs minimal inference

Table 5: Ablation study of PAR^{2}QO across three workloads on JOB. By default, PAR^{2}QO uses $n = 50$, $k = \max(0.2 \cdot |\mathcal{P}|, 10)$, and $\theta_{KL} = \ln(200)$. Each variant modifies a single parameter while keeping all others fixed to isolate its effect.

	CARVER				Uniform				Historical			
	Latency (ms)	$\uparrow > 1.2\times$	$\downarrow > 1.2\times$	$\downarrow > 2\times$	Latency (ms)	$\uparrow > 1.2\times$	$\downarrow > 1.2\times$	$\downarrow > 2\times$	Latency (ms)	$\uparrow > 1.2\times$	$\downarrow > 1.2\times$	$\downarrow > 2\times$
PostgreSQL	536 ms	-	-	-	410 ms	-	-	-	637 ms	-	-	-
PAR^{2}QO (default)	273 (1.96×)	15	3	0	254 (1.61×)	14	2	0	326 (1.95×)	15	7	0
PAR^{2}QO (n=200)	270 (1.98×)	14	4	0	257 (1.59×)	12	3	0	319 (2.00×)	17	7	0
PAR^{2}QO (n=10)	319 (1.68×)	13	9	0	301 (1.36×)	12	5	0	330 (1.93×)	13	6	0
PAR^{2}QO (k=1)	506 (1.06×)	15	8	5	318 (1.29×)	9	3	0	366 (1.74×)	13	3	1
PAR^{2}QO (k=5)	250 (2.14×)	17	2	0	266 (1.54×)	13	3	0	318 (2.00×)	16	6	0
PAR^{2}QO (k=max$(0.5 \cdot |\mathcal{P}|, 10)$)	276 (1.94×)	16	3	0	265 (1.54×)	13	4	0	328 (1.94×)	16	7	0
PAR^{2}QO (k=$|\mathcal{P}|$)	285 (1.88×)	14	4	0	255(1.61×)	14	2	0	339 (1.88×)	13	8	1
PAR^{2}QO ($\theta_{KL} = \ln(2000)$)	271 (1.98×)	16	2	0	263 (1.56×)	13	3	0	327 (1.95×)	15	7	0
PAR^{2}QO ($\theta_{KL} = \ln(20000)$)	281 (1.91×)	15	4	0	287 (1.43×)	10	4	0	344 (1.85×)	12	8	1

Table 6: Detailed comparison of system-level statistics across methods. For variants based on PAR^{2}QO and PARQO, the model preparation time includes the upfront error model construction time, which is less than 13 minutes on average per template. For the number of plans, we show both the initial plan size and the size of robust plans after plan reduction.

Method	Avg Model Preparation Time	Avg Inference Time	Avg # of Clusters	Avg # of Plans
PostgreSQL	0.0 min	67.3 ms	–	–
PAR^{2}QO	21.5 min	26.2 ms	6	88 → 8
Kepler	390.1 min	0.8 ms	–	197 → 4
Bao	14.5 min	376.1 ms	–	–
PARQO′	144.5 min	36.2 ms	6 (anchors)	59 → 6
PARQO	36.5 min	42.3 ms	1 (anchor)	15 → 1
QueryLog′	0.1 min	35.4 ms	–	5
PAR^{2}QO (n=200)	72.1 min	54.0 ms	6	152 → 10
PAR^{2}QO (n=10)	17.2 min	5.8 ms	6	29 → 5
PAR^{2}QO (k=1)	21.5 min	0.1 ms	6	88 → 1
PAR^{2}QO (k=5)	21.5 min	26.6 ms	6	88 → 4
PAR^{2}QO (k=max$(0.5 \cdot |\mathcal{P}|, 10)$)	21.5 min	29.9 ms	6	88 → 8
PAR^{2}QO (k=$|\mathcal{P}|$)	21.5 min	31.4 ms	6	88 → 8
PAR^{2}QO ($\theta_{KL} = \ln(2000)$)	19.7 min	23.4 ms	5	85 → 8
PAR^{2}QO ($\theta_{KL} = \ln(20000)$)	16.2 min	18.1 ms	4	74 → 7

overhead, suggesting potential practicality in simpler or latency-sensitive scenarios.

- Second, retaining all candidate plans without any reduction (same as PAR^{2}QO-no-reduce in Section 5.3) slightly worsens latency compared to the default setting. This indicates that plan reduction not only reduces cache size and inference time but also helps eliminate less robust plans, thereby improving overall performance.

We further experiment with intermediate values such as $k = 5$ and $k = \max(0.5 \cdot |\mathcal{P}|, 10)$, both of which yield performance close to the default configuration. These results suggest that moderate values of k (e.g., $5 — 0.5 \cdot |\mathcal{P}|$) are sufficient to balance robustness and efficiency. Developing an adaptive method to automatically select k remains an interesting direction for future work.

Impact of KL Threshold (θ_{KL}). The threshold θ_{KL} controls how aggressively queries in the training workload are clustered. Smaller thresholds lead to more clusters and finer-grained coverage of the selectivity space. As shown in Table 5, increasing θ_{KL} results in slightly worse performance and more regressions. The system-level statistics in Table 6 show that higher thresholds reduce the number of clusters and overall plan diversity, which limits the ability to generalize across unseen queries. Our default choice of $\ln(200)$ offers a good trade-off between coverage granularity and training efficiency.

Overall, these results highlight the importance of tuning the hyperparameters in PAR^{2}QO to balance performance, robustness, and overhead. The default settings used in the main experiments provide consistently strong performance across diverse workloads.

Impact of training size ($|W|$). For those templates where Kepler exhibits severe regressions, e.g., $C3$-4, $U4$, $H5$, and $C10$, as shown in Figure 3, we expand the original training set of each template from 50 to 400. The larger training size increases Kepler's model preparation time substantially — up to 120 hours per template — but the impact on query performance varies across templates. Some regressions are mitigated by additional training: $C4$ improves from $-3.71\times$ to $-1.50\times$, $U4$ from $-4.0\times$ to a $1.31\times$ speedup, and $H5$ from $-3.70\times$ to $-1.30\times$. However, $C10$ and $C3$ remain unchanged at $-5.04\times$ and $-3.46\times$ respectively. These results suggest that while a larger training size may improve performance, the degree of improvement depends on the query workload and is not guaranteed. Notably, for all templates except $U4$, Kepler with increased training size still underperforms PAR^{2}QO using only the original 50 training queries. We also expanded the training size for PAR^{2}QO; the resulting performance change is minimal, indicating that a small training size is sufficient for PAR^{2}QO.

G.4 Additional Details on Robustness Against Data Shifts

The four representative queries are selected based on the following criteria. Template 4 shows the largest degradation for PAR^{2}QO ($H4$ in Figure 3). Template 14 represents a scenario where neither PAR^{2}QO nor Kepler shows significant speedup or regression. Template 17 is a case where PAR^{2}QO outperforms Kepler, while Template 19 is where Kepler significantly outperforms PAR^{2}QO. These templates also vary in complexity, ranging from 5 to 10-table joins and 3 to 12 parameter bindings, ensuring a diverse evaluation. The detailed template-level performance across different slicing methods and workloads is presented in Figures 6 to 8. For clarity, we plot a horizontal dashed line (red) in some figures as a marker for significant speedups or regressions exceeding 1.2×. First, in category-slicing, where data distribution varies the most, we use "category" as the base instance. Query performance fluctuates across instances, yet Figure 6 shows that plans selected by PAR^{2}QO maintain robust performance across templates and workloads. Although PAR^{2}QO exhibits some degradation over PostgreSQL in certain cases (e.g., $H4$ and $C17$) when tested on other instances, the largest regression does not exceed −2×. Interestingly, some templates with modest initial speedups on the base instance perform significantly better on other instances, even surpassing PostgreSQL. Examples include $C14$, $U14$, $H14$, and $H19$. We do not intend to beat PostgreSQL's refreshed plan on these instances, since our current error model observed from the base instance is a static profile and does not explicitly account for possible data shifts. But the robustness consideration in PAR^{2}QO inherently promotes stability and prevents severe regressions. In contrast, Kepler suffers severe regressions in $C4$ and $C19$, with performance dropping reaching up to −6× compared to the baseline.

Second, in time-slicing, we use DB5 as the base instance. The speedups generally follow a unimodal distribution, peaking in middle instances and decreasing toward both extremes (e.g., $C19$). However, some templates deviate from this trend, such as $H17$ and $C17$. For example, the plan obtained for $H17$ from DB5 achieves a 3× speedup, but on DB1, which is farther from the base, the speedup increases to 7.15×. The only severe regression observed for PAR^{2}QO is $H19$ on DB1 (−2.68×), though it remains better than Kepler's −3.05×.

Finally, in random-slicing, we regard DB5 as the base instance. Most templates exhibit consistent execution performance across instances. PAR^{2}QO maintains both its speedups and low regression levels across different instances, demonstrating its robustness.

G.5 Additional Results on Unforeseen Query Workloads and Effectiveness of CARVER for Training

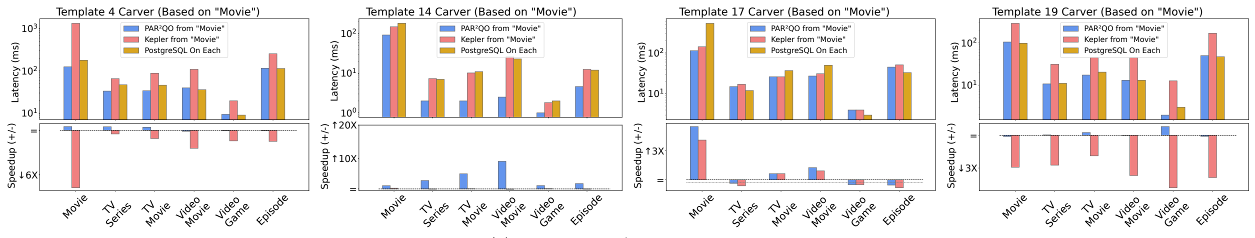
Detailed analysis separating outliers. As discussed in Section 5.1, Kepler lacks the robustness protection of PAR^{2}QO, potentially leading to poor plan selection. To help ensure that our conclusion is not biased by outliers, we reexamine the results by excluding the single template causing the largest performance decline when trained on each workload. Note that this exclusion is based on performance decline, not execution time. After this adjustment, Kepler trained on CARVER performs better, reducing its latency to 432ms and achieving a speedup of 1.28×, surpassing Uniform (524ms) and Historical (508ms). Still, PAR^{2}QO trained with CARVER maintains its superior performance, achieving a latency of 359ms and a 1.54× speedup over PostgreSQL, compared to Uniform (1.16×) and Historical (1.24×). These results underscore CARVER's effectiveness in generating workloads with comprehensive cardinality coverages: using CARVER as the training workload can enhance plan efficiency as well as robustness, and improve the ability to generalize to new workloads. These advantages position CARVER as a valuable tool not only for evaluating PQO model performance but also for guiding its improvement.

Table 7: Latency (ms) on selected JOB templates. Each row shows PAR^{2}QO's performance when trained with a different workload and tested on the same mixed workload (CARVER+ Uniform + CARVER′ + Historical).

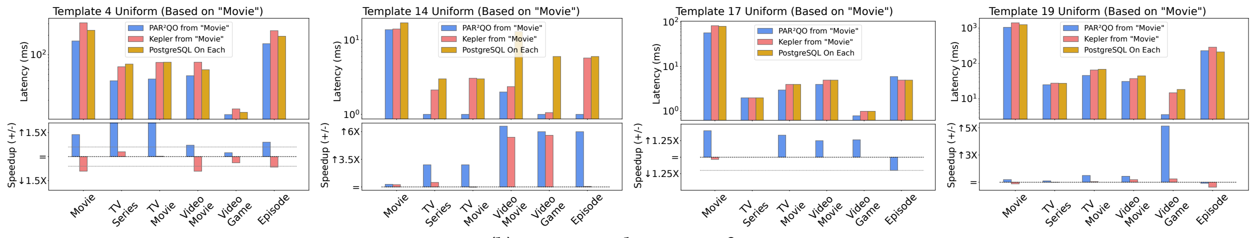
Training Workload	Q3	Q4	Q7	Q14	Q17	Q19	Q23
PostgreSQL	300	205	522	133	1382	762	442
CARVER	354	218	109	130	389	834	276
Uniform	475	195	60	129	1437	1012	329
CARVER′	335	265	269	138	1326	898	659
Historical	2114	634	72	127	365	1006	907

Comparison with CARVER restricted to independent sampling from individual tables. Recall that CARVER samples from join subquery results. To evaluate this design choice, we conduct an experiment similar to Section 5.5 to compare CARVER, Uniform, Historical, and CARVER′ — which restricts CARVER to sampling independently from individual tables, corresponding to the option of using P_3 only as discussed in Appendix E. Again, we train PAR^{2}QO with each of the four workloads, but test each of the result models on queries drawn from a mixture of all four workload distributions. Table 7 shows results for several representative JOB templates. Although CARVER′ is comparable to CARVER in some cases, such as Q3 and Q14, it performs significantly worse on others, including Q7, Q17, and Q23. This result supports the choice of our default setting for CARVER, which also significantly outperforms Uniform and Historical, and confirms the importance of capturing join subquery selectivities for robust PQO.

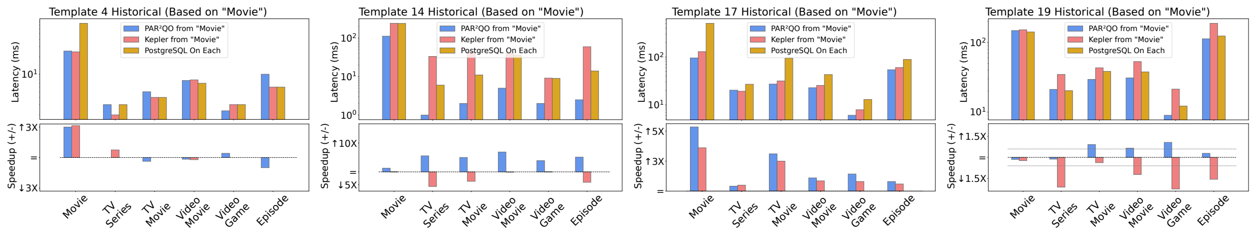
Results for DSB. On DSB, we also compare CARVER and Gaussian for training, following the same experimental design in Section 5.5. Again, we train using each of the workloads, but test using a mixed workload including queries sampled from both. The results are summarized in Table 8. PAR^{2}QO achieves a 2.28× speedup over PostgreSQL when trained using CARVER, surpassing the 1.67× speedup when trained using Gaussian. In comparison, Kepler achieves a 2.25× speedup when trained using CARVER, compared



(a) Category-slicing: CARVER

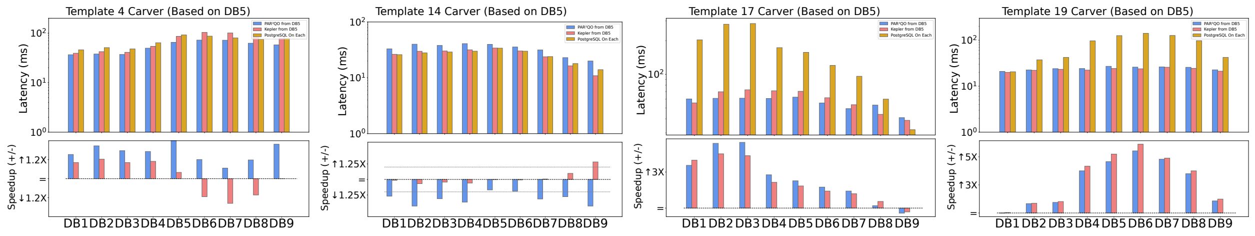


(b) Category-slicing: Uniform

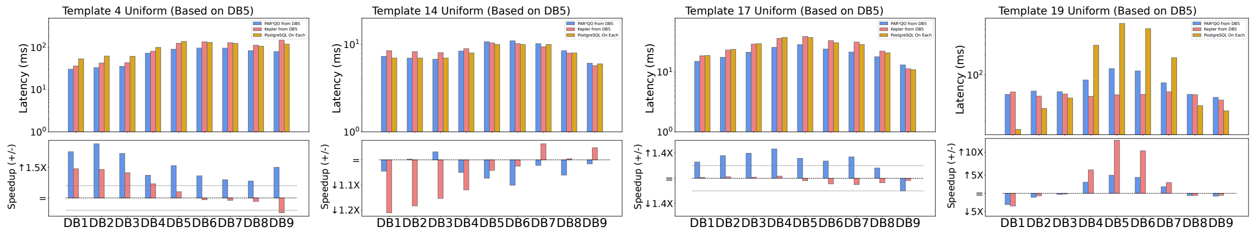


(c) Category-slicing: Historical

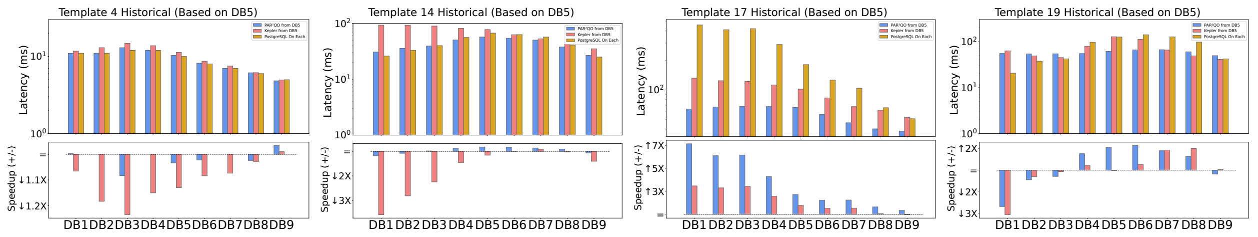
Figure 6: Robustness verification by category-slicing. "Movie" is the base database instance.



(a) Time-slicing: CARVER

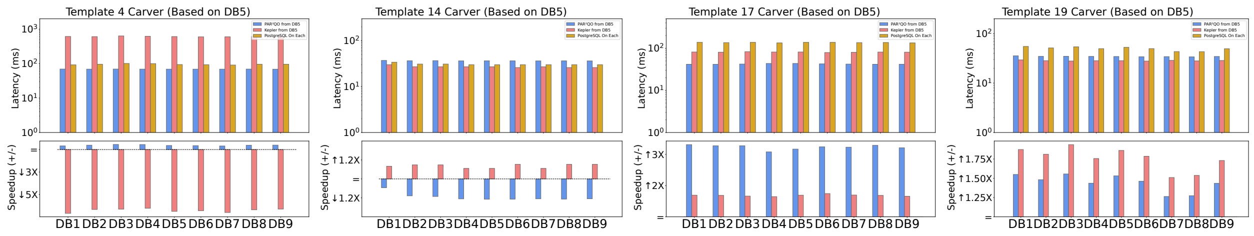


(b) Time-slicing: Uniform

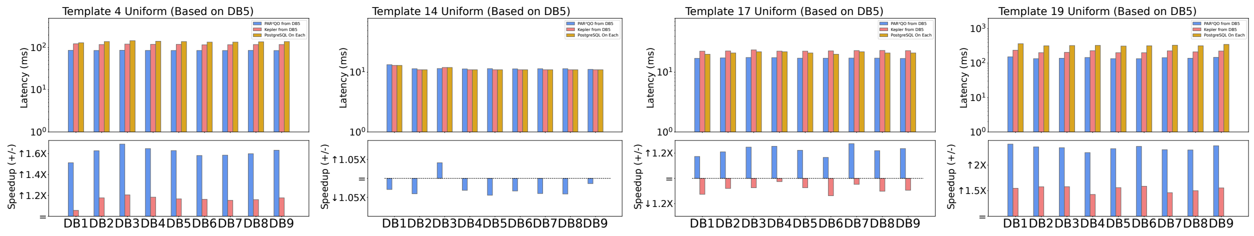


(c) Time-slicing: Historical

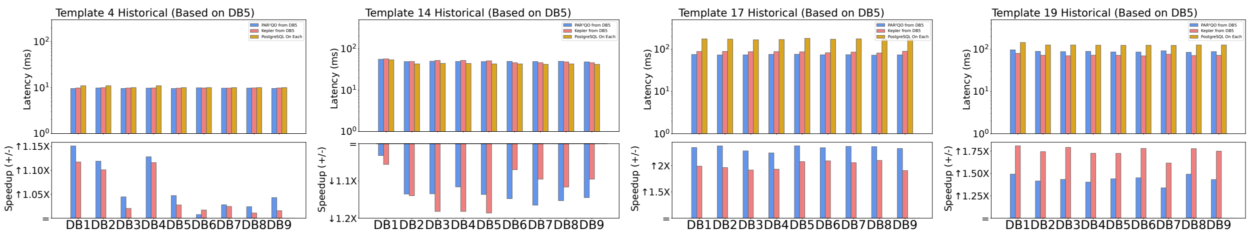
Figure 7: Robustness verification by time-slicing. DB5 is the base database instance.



(a) Random slicing: CARVER



(b) Random slicing: Uniform



(c) Random slicing: Historical

Figure 8: Robustness verification by random-slicing. DB5 is the base database instance.

Table 8: Performance of models trained under each workload (CARVER or Gaussian) for DSB, tested under the same mixed workload (CARVER + Gaussian).

		CARVER	**Gaussian**
PostgreSQL	Avg latency	347 ms	347 ms
PAR^{2}QO	Avg latency	152 (2.28×)	207 (1.67×)
	↑ > 1.2×	5	4
	↓ > 1.2×	0	0
	↓ > 2×	0	0
Kepler	Avg latency	154 (2.25×)	206 (1.68×)
	↑ > 1.2×	5	3
	↓ > 1.2×	0	0
	↓ > 2×	0	0

with 1.68× using Gaussian. Therefore, we arrive at the same conclusion as Section 5.5 that CARVER works better for training.