PARQO: Penalty-Aware Robust Query Optimization

Haibo Xiu, Pankaj K. Agarwal, and Jun Yang
Duke University, Durham, NC, USA
haibo.xiu@duke.edu,{pankaj,junyang}@cs.duke.edu

ABSTRACT

The effectiveness of a cost-based query optimizer relies on the accuracy of selectivity estimates. The execution plan generated by the optimizer can be extremely poor in reality due to uncertainty in these estimates. This paper presents **PARQO** (**P**enalty-**A**ware **R**obust **Q**uery **O**ptimization), a novel system where users can define powerful robustness metrics that assess the expected penalty of a plan with respect to true optimal plans under a model of uncertainty in selectivity estimates. PARQO uses workload-informed profiling to build error models, and employs principled sensitivity analysis techniques to identify selectivity dimensions with the largest impact on penalty. Experimental evaluation on three benchmarks demonstrates how PARQO is able to find robust, performant plans, and how it enables efficient and effective parametric optimization.

PVLDB Reference Format:
Haibo Xiu, Pankaj K. Agarwal, and Jun Yang. PARQO: Penalty-Aware Robust Query Optimization. PVLDB, 14(1): XXX-XXX, 2020.
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/PARQO.

1 INTRODUCTION

Given a query and a set of candidate execution plans, a standard cost-based query optimizer chooses the plan with the lowest *estimated cost*. Errors in cost estimates can lead to suboptimal, and sometimes catastrophic, plan choices. Research has attributed the main source of such errors to inaccurate selectivity estimates [33, 42], which are used by the optimizer to predict the cardinalities of result sets returned by subplans without executing them. Despite decades of research on improving selectivity and cardinality estimates, be it using better data summaries [45], samples [55], or machine learning models [30, 51], the problem remains unresolved. Since higher accuracy comes at some cost, such as runtime monitoring and ongoing maintenance, a database system must strike a balance between the cost and benefit of accurate selectivity estimates. Therefore, coping with uncertainty in selectivity estimates will likely remain a long-term challenge. At a high level, this paper studies how uncertainties in selectivity estimates affect plan optimality, and how to find "robust" plans that work well despite such uncertainties.

The idea of *robust query optimization* has been around for years [8, 10, 28]. Despite extensive research that has introduced various notions of robustness and approaches for finding robust plans (see Section 7 for more discussion), wide adoption of these results has yet to happen in practice. Several challenges must be addressed in a concerted effort to enable impact for this line of research.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

First, there is no "one-size-fit-all" when it comes to the notion of robustness. For some applications, robustness may mean that the chosen plan's cost must be insensitive to errors in cardinality estimates. Some may care instead about how errors affect the cost optimality of the chosen plan: if, over all likely estimation errors, a plan still remains optimal or nearly optimal among all alternatives, whether its cost is sensitive to any estimate is irrelevant. Others may define robustness in more nuanced ways: e.g., to meet a service-level agreement, they may want to "penalize" performance degradation over the optimal proportionally, but only if the degradation exceeds a certain threshold. Because of the myriad of application needs, any system that specializes in one robustness metric or makes design decisions that implicitly encode specific assumptions about robustness will be limited in its applicability. Therefore, we must strive for a general framework and associated techniques that support flexible and powerful robustness definitions.

Related to this point, the notion of robustness is strongly tied to the degree of uncertainty. Analogous to a traditional optimizer prone to making catastrophic choices by ignoring uncertainty, tailoring robust optimization to unlikely scenarios often forces overly conservative choices that perform poorly in practice. In many application settings, it is possible to naturally accumulate or proactively acquire knowledge on how errors are distributed given our data and query workload. The framework should incorporate such knowledge in defining and optimizing robustness in a principled way.

Second, robust query optimization inherits all scalability and efficiency challenges of traditional query optimization and then adds more. Real queries often contain many joining tables and predicates; for example, in the JOB benchmark [33] based on a real dataset (IMDB), query Q29 is a 17-way join, with two tables involved in self-joins, and PostgreSQL makes more than 13,000 cardinality estimates when optimizing this query. Such high dimensionality of the selectivity space makes the problem of robust query optimization very challenging. For many useful robustness metrics, it is impossible to assess a plan's robustness by itself, without examining how other competing plans would perform under uncertainty in high-dimensional space. Therefore, we must tame the overhead of robust query optimization in order to justify its use.

Third, practical adoption may also require integration with existing database systems. Past research has seen many examples where robust query optimization requires modifications to traditional database optimizers and and execution engines. Coupled with the limitation that they only support specific notions of robustness, adoption is a hard sell. Therefore, there is an argument for solutions that interface with traditional systems to support more general notions of robustness in a scalable and efficient way.

To address these challenges, we introduce ***PARQO*** (***P****enalty-* ***A****ware* ***R****obust* ***Q****uery* ***O****ptimization*):

- We develop a framework that takes the general approach of *stochastic optimization* [44] and defines the robustness objective as minimizing *expected penalty*. The key components in this definition include a flexible user-defined penalty function, which assesses the penalty incurred by a plan with respect to the true optimal, and a statistical model of the selectivity estimation errors, which we show how to obtain by profiling the database workload. This powerful combination allows PARQO to tailor to a broader range of application needs. To the best of our knowledge, previous work (Section 7) either does not follow a stochastic optimization approach, or chooses less general robustness measures (and/or employs heuristics reflecting such objectives) that depend only on an individual plan (e.g., whether its own cost is sensitive to error) but not how it compares with other alternatives (such as the true optimal).
- On taming high dimensionality of the optimization problem, existing heuristics for selecting dimensions are not always aligned to the definition of robustness, and methodically, they rely on rather rudimentary techniques that analyze dimensions one at a time. We identify principled techniques from the *sensitivity analysis* literature [41, 49] that account for interactions among multiple dimensions, and apply them to our setting. Our selection of *sensitive dimensions* considers the given expected penalty objective, and therefore is fully aware of and tailored to the error distribution as well as the penalty definition. Finally, selected sensitive dimensions correspond to selection/join condition combinations in the query, which are interpretable and actionable — for example, a user may investigate a sensitive dimension and improve its accuracy (by selectively reanalyzing statistics and/or sampling) in order to obtain a better plan.
- Given a query and its selectivity estimates, we show how to find a robust plan that minimizes expected penalty, focusing on the sensitive dimensions. Except for very expensive queries, the amount of work that goes into finding a robust plan from end to end (including the identification of sensitive dimensions through sensitivity analysis) may not justify doing so to optimize just a single execution. We show how to reuse the work in robust query optimization and amortize its cost in the setting of *parametric query optimization* (*PQO*) [27], where the optimization overhead is shared among multiple queries with the same template but different query parameters, which frequently arise in practice.
- Despite the generality of our framework, PARQO is designed to work with existing optimizers and cardinality estimation methods. We have implemented it on top of PostgreSQL and conducted an end-to-end evaluation using three popular benchmarks: JOB [33], DSB [12], and STATS-CEB [20]. We demonstrate how PARQO is able to suggest hints that are interpretable and actionable for understanding and improving query performance, how resilient PARQO's robust plans are against inaccurate selectivity estimates and how they soundly outperform traditional plans, and finally, how PARQO delivers significant benefits to multiple queries in the PQO setting. We illustrate PARQO's effectiveness in these scenarios using an example below.

Example 1. *Consider Q17 below from the JOB benchmark [33].*

```
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 ='[us]' AND k.keyword
      ='character-name-in-title' AND n.name LIKE 'B%'
  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;
```

Based on the error profiles on selectivity estimation collected from a workload (not specifically for this query), PARQO carries out a sensitivity analysis for PostgreSQL's plan for Q17 and identifies the most sensitive selectivity dimension *to be $mk \bowtie k^{\sigma}$ (here we use the σ superscript on a table to highlight the presence of a local selection condition). This suggestion is interpreted and actionable. Indeed, if we pay extra diligence to learn the true selectivity of $mk \bowtie k^{\sigma}$ and hint it to PostgreSQL, the new plan will achieve a 5.84× speedup (in actual execution time). Additional details and more experiments along this line can be found in Section 6.*

PARQO can also suggest a robust plan *for Q17. To get a sense of how this plan would fare in real-world situations where selectivity errors arise inevitably due to data updates, we simulated a scenario of an evolving database by time-partitioning the IMDB database used by JOB into 9 different instances (DB1 through DB9), each with titles and associated data from a contiguous time period. PARQO only has access to DB5 when choosing the robust plan, and we execute this same plan on all 9 instances and compare its running time with PostgreSQL's plans (each obtained for the specific instance). As can be seen from Figure 3, PARQO's single robust plan consistently beats PostgreSQL on all 9 instances, with a 3.86× speedup on average. This is of course just one data point—more experiments can be found in Section 6.*

Finally, improving the performance of just a single execution would not justify the overhead of robust query optimization, but PARQO shines when combined with PQO, where we share the optimization overhead across many queries with the same template — in this case queries that differ from Q17 only in the choice of literals (e.g., `'[us]'`, `'character-name-in-title'`, `'B%'`*). By caching and reusing the work done on behalf of Q17, PARQO eliminates the need to call the optimizer for 66% of the queries with same template. Furthermore, for these queries, the average speedup over PostgreSQL plans is 7.14×, resulting in an overall improvement of 2.4× for the entire workload. Again, we refer readers to Section 6 for additional details.*

2 PARQO FRAMEWORK

Preliminaries and Problem Statement. A *query template* Q is a query where literal values in its expressions are represented by *parameters*. To optimize a query with template Q and specific parameters values, a traditional query optimizer considers a space of *(execution) plans* $\mathbf{\Pi}(Q)$. For each plan $\pi \in \mathbf{\Pi}(Q)$, the optimizer uses a set of *selectivities* $\mathbf{s} = (s_1, \ldots, s_d) \in [0, 1]^d$ relevant to Q to calculate the cardinalities of results returned by various subplans of π and in turn the overall cost of π, denoted $\mathsf{Cost}(\pi, \mathbf{s})$. The goal of traditional query optimization is to find the *optimal plan* given selectivities $\mathbf{s}$, denoted by $\pi^{\star}(Q, \mathbf{s}) = \arg\min_{\pi \in \mathbf{\Pi}(Q)} \mathsf{Cost}(\pi, \mathbf{s})$; we further denote its cost by $\mathsf{Cost}^{\star}(Q, \mathbf{s})$. When it is clear that we are referring to a given template Q, we omit Q from these notations.

In reality, we do not have the true selectivities $\mathbf{s}$, but only their estimates instead. Acting on this uncertain information, suppose the optimizer picks a plan π. We would like to quantify the *penalty* incurred by executing π relative to the real optimal plan, where their costs are based on the true selectivities $\mathbf{s}$. There are many reasonable options for defining penalty. For example, it can be defined using a tolerance factor τ:

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

In other words, the penalty is proportional to the amount of cost exceeding the optimal, but only if it is beyond the prescribed tolerance. This particular definition would capture the scenario where a provider aims to fulfill a service-level agreement under which any performance degradation above a certain threshold will incur a porportional monetary penalty.

However, since we do not know true selectivities $\mathbf{s}$ in advance, we cannot evaluate the penalty directly at optimization time. Instead, PARQO models selectivities as a random vector S, and evaluates the expected penalty $\mathrm{E}[\mathsf{Penalty}(\pi, \mathrm{S}) | \hat{\mathbf{s}}]$. Let $f(\mathbf{s}|\hat{\mathbf{s}})$ denote the probability density function for the distribution of true selectivities $\mathbf{s}$ conditioned on the current estimate $\hat{\mathbf{s}}$. We formally define the problem of *finding a robust plan* as follows:

(Robust plan) *Given a query with template Q, selectivity estimates $\hat{\mathbf{s}} \in [0,1]^d$, and a conditional distribution of true selectivities $\mathrm{S} \sim f(\mathbf{s}|\hat{\mathbf{s}})$, find a plan $\pi \in \mathbf{\Pi}(Q)$ that minimizes:*

$$E[\mathsf{Penalty}(\pi, \mathrm{S}) | \hat{\mathbf{s}}] = \int \mathsf{Penalty}(\pi, \mathbf{s}) \cdot f(\mathbf{s}|\hat{\mathbf{s}})\, d\mathbf{s}. \quad (2)$$

We also define the problem of *finding sensitive (selectivity) dimensions*, informally at this point, as follows:

(Sensitive selectivity dimensions) *Given Q, $\hat{\mathbf{s}}$, $f(\mathbf{s}|\hat{\mathbf{s}})$, and a plan $\pi \in \mathbf{\Pi}(Q)$, find up to k dimensions among $1, \ldots, d$ having the "largest impact" on $\mathsf{Penalty}(\pi, \mathbf{s})$.*

We defer a detailed discussion on various options of defining "largest impact" to Section 4, but as a preview, PARQO prefers defining the impact of selectivity dimension i as the contribution to the variance in $\mathsf{Penalty}(\pi, \mathrm{S})$ due to uncertainty in s_i.

We note that our framework works with any user-defined penalty definition, not just the example definition in Equation (2). Other possibilities include probability of exceeding the tolerance threshold, standard deviation in cost difference, or simply the cost difference itself, etc.; see the extended version of this paper [57] for details. Our experiments in Section 6 chooses Equation (2) because it is easy to interpret yet still illustrates two important features supported by our framework. First, it is defined relative to the would-be optimal plan, allowing it to model a broader range of notions of robustness than those that are defined only using the plan π itself (such as how sensitive $\mathsf{Cost}(\pi, \mathbf{s})$ is to variation in $\mathbf{s}$). Second, it is not merely linear in $\mathsf{Cost}(\pi, \mathbf{s})$, which would make the problem considerably easier because of linearity of expectation.[1] We want to have a framework and techniques capable of handling more general cases.

Finally, we acknowledge that there are many other issues contributing to poor plan quality besides selectivity estimation errors, including inaccuracy in the cost function $\mathsf{Cost}(\pi, \mathbf{s})$ itself as well as suboptimality of the optimization algorithm; we focus only on selectivity estimation because it has been identified as the primary culprit [33, 42]. In the remainder of this paper, we shall assume that Cost is exact and that we can obtain the optimal plan $\pi^\star(\mathbf{s})$ if given true selectivities.

System Overview and Paper Outline. PARQO is designed to work with any traditional query optimizer that supports (or can be extended to support) two primitives: $\mathsf{Opt}(Q, \mathbf{s})$ returns the optimal plan $\pi^\star(\mathbf{s})$ for Q given selectivities $\mathbf{s}$; $\mathsf{Cost}(\pi, \mathbf{s})$ returns the cost of plan π for selectivities $\mathbf{s}$. We followed the strategy of [20] and the `pg_hint_plan` extension [38] to inject $\mathbf{s}$ and π into PostgreSQL for our implementation. When analyzing the complexity of our algorithms, we count the number of calls to Opt and Cost. Note that Cost is much cheaper than Opt.

A prerequisite of our framework is the distribution $f(\mathbf{s}|\hat{\mathbf{s}})$ of true selectivities conditioned on their estimates. While any distribution could be plugged in, including non-informative ones in case no prior knowledge is available, an informative distribution will make PARQO more effective. We outline a strategy in Section 3 for constructing this distribution by collecting error profiles for query fragments called *querylets* from the database workload. These profiles are able to capture some errors due to dependencies among query predicates. Finally, Section 3 also clarifies what relevant selectivity dimensions are for a given query template.

Next, building on this knowledge of how errors are distributed, we tackle the problem of finding sensitive dimensions in Section 4 for a given query plan π obtained under selectivity estimates $\hat{\mathbf{s}}$. We employ principled sensitivity analysis methods to identify a handful of selectivity dimensions with biggest impact on the user-defined penalty function. In particular, PARQO proposes using *Sobol's method*, which offers an interpretable measure of "impact" based on an analysis of the variance in $\mathsf{Penalty}(\pi, \mathrm{S})$ over $\mathrm{S} \sim f(\mathbf{s}|\hat{\mathbf{s}})$. We also show in Section 4 how automatically identified sensitive dimensions can help with performance debugging of query plans.

Then, we show in Section 5 how to find robust query plans by focusing on the selectivity subspace consisting of only the sensitive dimensions. By sampling from the distribution of true selectivities conditioned on their estimates, we build a pool of candidate robust plans and select the one with the lowest expected penalty. We show how sample caching and reuse can significantly reduce the number of Opt and Cost calls to the optimizer. To further mitigate the overhead of robust query optimization, PARQO combines it with parametric query optimization so that the work devoted to

[1] For example, consider the alternative definition of $\mathsf{Penalty}(\pi, \mathbf{s}) = \mathsf{Cost}(\pi, \mathbf{s}) - \mathsf{Cost}^\star(\mathbf{s})$. Because of the linearity of expectation, the optimization problem boils down to a much simpler version of minimizing expected cost $\mathrm{E}[\mathsf{Cost}(\pi, \mathrm{S})]$, which is independent of the optimal plan costs. While this definition may be appropriate if our overall goal is system throughput, it does not particularly penalize bad cases, which users with low risk tolerance may be more concerned with. As another example that is "pseudo-dependent" on the optimal plan costs, the *P-error* metric recently proposed in [20] defines $\mathsf{Penalty}(\pi, \mathbf{s}) = \mathsf{Cost}(\pi, \mathbf{s}) / \mathsf{Cost}^\star(\mathbf{s})$, but let us consider the logarithm of P-error instead. Because $\log(\mathsf{Cost}(\pi, \mathbf{s}) / \mathsf{Cost}^\star(\mathbf{s})) = \log \mathsf{Cost}(\pi, \mathbf{s}) - \log \mathsf{Cost}^\star(\mathbf{s})$, we see that minimizing expected log-P-error again can be done without regard to the optimal plan costs by the linearity of expectation.

finding a robust plan can be reused for a different query with the same template. We develop a principled test for determining when to allow such reuse.

Finally, Section 6 presents a full experimental evaluation of PARQO using three different benchmarks; Section 7 discusses related work; Section 8 concludes and outlines future directions.

3 ERROR PROFILING

The goal of this step is to build a model that approximates $f(\mathbf{s}|\hat{\mathbf{s}})$ given a query with template Q and selectivity estimates $\hat{\mathbf{s}}$, or equivalently, a model of the error between S and $\hat{\mathbf{s}}$. Some learned selectivity estimators are able to output estimates as well as some measures of uncertainty, which we may readily adopt if we deem them reliable. However, we still need a procedure for obtaining $f(\mathbf{s}|\hat{\mathbf{s}})$ in the general case where such measures are not already available. Despite the notation $f(\mathbf{s}|\hat{\mathbf{s}})$, which involves the true selectivities $\mathbf{s}$, we do not want to supplant the original selectivity model; instead, we simply seek to characterize the errors. Nonetheless, there are some high-level desiderata. First, we would like this model be informed by the database workload.[2] Second, the independence assumption made by many traditional optimizers is often blamed for throwing off cardinality estimates; hence, we need to go further than profiling each selection predicate and join predicate in isolation, so we can account for the effect of their interactions on estimation errors. One the other hand, it is impractical to track estimation error for every possible subquery that shows up during query optimization — recall from Section 1 that PostgreSQL invokes more than 13,000 cardinality estimates for optimizing Q29 alone. Guided by these considerations, PARQO adopts the following design.

Querylets. Given a query or query workload, we build one error profile per "querylet." A *querylet* is subquery pattern involving joins and/or local selection conditions, e.g., $R^\sigma \bowtie_p S$, where superscript σ denotes the presence of at least one local selection condition on a table. Querylets are uniquely identified by the set of tables, join conditions among them, and the subset of the tables with local selection conditions. During query execution, for each subquery matching a querylet, we track the estimated and actual cardinalities of its result. We maintain a sample of all such pairs observed for this querylet in a workload, which constitutes its *error profile*.

We cannot afford to profile all possible querylets, so we choose the following: all single-table querylets, all two-table querylets, plus any additional three-table querylet with the pattern $R^\sigma \bowtie_{p_1} S \bowtie_{p_2} T$, if it appears in some query where none of S and T has any local selection. The cutoff at length two to three is for practicality. The allowance for some three-table querylets is to capture at least some data dependency beyond binary joins.

For example, in Q17 (Example 1), one querylet would be n^σ, which covers all local selection conditions on n. Another example is $mc \bowtie cn^\sigma$. A third example would be $k^\sigma \bowtie mk \bowtie ci$: since mk and ci have no selection conditions, this querylet captures any potential dependency between the local selection on k and the join between mk and ci. As an example of a 3-table querylet that is not profiled, consider $mc^\sigma \bowtie t \bowtie cn^\sigma$ (this case does not arise in Q17), because both $mc^\sigma \bowtie t$ and $t \bowtie cn^\sigma$ would have been profiled already.

Note that one could choose to further differentiate querylets by the columns or query constants involved in the selection conditions, at the expense of collecting more error files. For this paper, we specifically want to keep error profiling simple and practical, so we did not explore more sophisticated strategies. Despite this rather coarse level of error profiling, we obtain good results in practice in Section 6. That said, there are particular cases where we observe limitation of our current approach (also further explained in Section 6). Our framework allows for any error model to be plugged in, so further improvements are certainly possible.

Relevant Dimensions and Error Distributions. For a query template Q, we derive the set of relevant dimensions and corresponding error distributions from the set of querylets contained in the template. Specifically, for each table R with local selection in Q, we use the querylet R^σ (otherwise the estimate should be precise). For each join condition in Q, say between R and S, we select the most specific two-table querylet matching Q. For example, Q15 of JOB joins mc and cn with local selections on both, so the querylet selected is $mc^\sigma \bowtie cn^\sigma$. However, if neither R and S has any local selection, we look for the most specific three-table querylets we have profiled. If there are multiple such error files, we simply merge them. For example, in Q17, neither ci or mc has any local selection, but two three-table querylets matching Q17 contain ci and mc: $n^\sigma \bowtie ci \bowtie mc$ and $ci \bowtie mc \bowtie cn^\sigma$. We merge the collected error data according to these two querylets together and build one error distribution attributed to the join between ci and mc. In the end, the set of relevant selectivities correspond to the set of selection and join conditions in the query template.

As a complete example, for Q17, we arrive at $d = 12$ relevant dimensions as follows. Error profiles for the three local selection selectivities are readily derived from single-table querylets cn^σ, k^σ, and n^σ. Note that 4 tables have no local selections in Q17; we do not consider them relevant dimensions because base table cardinalities are not estimated. Error profiles for three (out of nine) relevant join selectivities are derived from two-table querylets $n^\sigma \bowtie ci$, $mc \bowtie cn^\sigma$, and $mk \bowtie k^\sigma$. Error profiles for the next three relevant join selectivities, for $t \bowtie ci$, $t \bowtie mc$, and $t \bowtie mk$, are derived from three-table querylets $t \bowtie ci \bowtie n^\sigma$, $t \bowtie mc \bowtie cn^\sigma$, $t \bowtie mk \bowtie k^\sigma$, respectively. Finally, for $mc \bowtie ci$, we derive its error profile by merging error profiles for three-table querylets $cn^\sigma \bowtie mc \bowtie ci$ and $mc \bowtie ci \bowtie n^\sigma$; for $mk \bowtie ci$, we merge $k^\sigma \bowtie mk \bowtie ci$ and $mk \bowtie ci \bowtie n^\sigma$; and for $mk \bowtie mc$, we merge $k^\sigma \bowtie mk \bowtie mc$ and $mk \bowtie mc \bowtie cn^\sigma$.

For each selectivity s_i, we create two models, one for low selectivity estimates and one for high selectivity estimates. In this paper, we set the low-high cutoff as the median error observed in s_i's error profile. This simple bucketization is motivated by the observation that errors tend to differ across low and high estimates: e.g., high selectivity estimates naturally have less room for overestimation. Each model simply uses a kernel density estimator to approximate the distribution of *log-relative* errors calculated from the error profiles. Given an estimate $\hat{s}_i$, we pick one of the two models to predict its error depending on how $\hat{s}_i$ compares with the low-high cutoff. We use $g_i(\varepsilon_i|\hat{s}_i)$ to denote this combined density estimator for log-relative errors in dimension i.

[2] If no such workload exists to start with, one can generate a random query workload aimed at coverage, or simply adopt an non-informative error model that conservatively assumes that true selectivities can be arbitrary in $[0, 1]$, and then redo the process after a query workload emerges.

Finally, to put together the error distribution in $\hat{\mathbf{s}}$ in the full d-dimensional selectivity space, we assume independence of errors estimated by the g_i's. Therefore, the conditional pdf in Equation (2) is approximated using the following factorized form:

$$f(\mathbf{s}|\hat{\mathbf{s}}) \approx \prod_{i=1}^{d} g_i(\log(\hat{s}_i/s_i)|\hat{s}_i). \tag{3}$$

Discussion. It is worth noting that while we assume independence among the g_i's above, those g_i's derived from the error profiles of two- and three-table querylets already capture dependencies among the join and selection conditions appearing together in them in a query workload. This approach follows the same intuition as the factor-graph representations for high-dimensional distributions to avoid the high cost of tracking the full distribution. To demonstrate the effectiveness of this approach, we experimentally validate in Section 6 its advantage over a baseline where errors for join and selection selectivities are separately and independently profiled.

Of course, since we cap the size of querylets to profile at 3, dependencies that span longer join chains are not captured. We also note that our g_i's are rather coarse: higher accuracy can certainly be achieved by higher-resolution models and additional profiling effort, e.g., with finer-grained buckets and separate models for different forms of predicates. More sophisticated models can be easily plugged in; PARQO only assumes that we can efficiently draw samples from the error distribution. Here, we only wish to demonstrate a simple approach that does a reasonable job; our overall model size is under 15KB for each of the three benchmarks tested in Section 6.

A Note on Recentering. A good estimator should not exhibit a large bias, meaning that its error distribution should have a mean around 0. After error profiling for PostgreSQL, however, we have observed that this is sometimes not the case. Since PARQO uses error profiles, it is fair to ask how much of its overall advantage simply comes from more careful modeling of errors. To this end, in Section 6, we also experimented with a simple fix called *recentering*, where we calculate the expectation of the true selectivities based on $f(\mathbf{s}|\hat{\mathbf{s}})$ and ask PostgreSQL to use them in optimization. As we shall see in Section 6, while this simple fix shows some improvements, PARQO overall is able to achieve much more.

4 SENSITIVITY ANALYSIS

Given a query template Q and selectivity estimates $\hat{\mathbf{s}} \in [0,1]^d$, consider the plan π chosen by a traditional optimizer $\hat{\mathbf{s}}$: i.e., $\pi = \pi^\star(\hat{\mathbf{s}})$. Given $f(\mathbf{s}|\hat{\mathbf{s}})$, we want to select a subset of up to k out of d dimensions as *sensitive dimensions.* We have two goals. First, we would like these dimensions to serve as interpretable and actionable hints that help user understand and improve the performance of π. Second, for the subsequent task of finding robust plans, we would like sensitive dimensions to help us reduce dimensionality and tame complexity. In the following, we will first review previous approaches and basic sensitivity analysis methods, and then introduce more principled methods. Then, we briefly discuss how sensitive dimensions can be used to help tune query performance.

4.1 From Local to Global Analysis

Before presenting PARQO's approach, we first briefly explain some alternative approaches for contrast. Given a plan π, a number of previous papers [28, 40, 52] define the sensitivity of a dimension i using merely the local properties of the plan's cost function, e.g., the partial derivative $\partial\,\mathsf{Cost}(\pi,\mathbf{s})/\partial s_i$ respect to dimension i evaluated at $\hat{\mathbf{s}}$, the current selectivity estimates. One fundamental limitation of this definition is that it does not address the question "what would we have done differently." It may well be the case that the cost of π is highly sensitive to s_i, but the optimality of π (or its penalty with respect to the optimal plan) is insensitive to s_i for all likely values of s_i. Hence, PARQO focuses instead on penalty-aware analysis.

One obvious improvement is to replace the cost function with the penalty function, which gives us $\partial\,\mathsf{Penalty}(\pi,\hat{\mathbf{s}})/\partial\hat{s}_i$ as a penalty-aware sensitivity measure for dimension i. We can further improve it by incorporating our knowledge of the error distribution and considering the expected penalty incurred by error in each dimension, resulting in the following definition: $\xi_i^{\mathsf{local}}(\pi,\hat{\mathbf{s}}) = \mathsf{E}[\mathsf{Penalty}(\pi,\mathbf{S}))|\hat{\mathbf{s}}]$, where $\mathbf{S} = (\ldots,\hat{s}_{i-1},S_i,\hat{s}_{i+1},\ldots)$ have identical component values as $\hat{\mathbf{s}}$ except dimension i for which $S_i \sim g_i(\log(\hat{s}_i/s_i)|\hat{s}_i)$ (see also Equation (3)). However, such a definition is still limited to *One-At-a-Time* (*OAT*) analysis, which fails to capture interaction among errors across dimensions. In the following, we present principled methods for *global* sensitive analysis to overcome this limitation.

Two popular methods from the sensitivity analysis literature [41, 49] are *Morris* and *Sobol's.* The *Morris Method* [37] is global in the sense that it considers a collection of "seeds" from the whole input space, but it still relies on local, derivative-based measures (called "elementary effects") at each seed that are OAT. We have adapted this method to our setting to incorporate knowledge of the error distribution; see [57] for details. However, as we will see in Section 6, *Sobol's Method* turns out to be more effective; therefore, it will be our focus in the following.

Sobol's Method. *Sobol's Method* [41, 43], based on analysis of variance, performs a fully global analysis and accounts for interactions among all dimensions. Given a function $h : [0,1]^d \to \mathbb{R}$, this method considers its stochastic version $Y = h(\mathbf{X})$, where $\mathbf{X}$ is a random input vector characterized by pdf $f_{\mathbf{X}}$. The variance of Y can be decomposed as follows:

$$\mathsf{Var}[Y] = \sum_{1\le i\le d} V_i + \sum_{1\le i<j\le d} V_{ij} + \sum_{1\le i<j<k\le d} V_{ijk} + \cdots + V_{1\ldots d}.$$

In the above, each $V_{\mathbf{u}}$, where $\mathbf{u}$ is a non-empty subset of the dimensions, is the contribution to the total variance attributed to the interactions among the components of $\mathbf{u}$. For each input dimension i, $V_i = \mathsf{Var}[\mathsf{E}[Y|X_i]]$, where the (inner) expectation, conditioned on a particular value for dimension i, is over all variations in other dimensions, and the (outer) variance is over all variations in dimension i. For each subset of two dimensions i and j, $V_{ij} = \mathsf{Var}[\mathsf{E}[Y|X_iX_j]] - V_i - V_j$, and similarly for larger subsets of dimensions. Normalizing each $V_{\mathbf{u}}$ by $\mathsf{Var}[Y]$ yields the *Sobol's index* $S_{\mathbf{u}} = V_{\mathbf{u}}/\mathsf{Var}[Y]$ for the combination of input dimensions in $\mathbf{u}$. Of particular interests are the so-called *first-order index* $S_i = V_i/\mathsf{Var}[Y]$, which is the portion of the total variance attributed to X_i alone; and the *total-order index* $S_i^T = \sum_{i\in\mathbf{u}\subset[1..n]} S_{\mathbf{u}}$, which is the portion of the total variance that X_i contributes to (alone or together with other dimensions). The latter can be computed as $S_i^T = 1 - V_i^T/\mathrm{Var}[Y]$, where $V_i^T = \mathrm{Var}[\mathrm{E}[Y|X_{\sim i}]]$, without summing an exponential number of Sobol's indices.

Sobol's indices are computed using a quasi-Monte Carlo method, using $2K$ sample points drawn randomly from $f_{\mathbf{X}}$. Given two sample points $\mathbf{a}$ and $\mathbf{b}$, it generates d more points, one for each dimension, by replacing the i-th component of $\mathbf{a}$ with the corresponding one in $\mathbf{b}$, obtaining a new point $\mathbf{a_b}^{[i]}$. Given sample points $\mathbf{a}_1, \ldots, \mathbf{a}_K$ and $\mathbf{b}_1, \ldots, \mathbf{b}_K$, the first-order and total-order indices for dimension i can be estimated through $V_i \approx \frac{1}{K}\sum_{j=1}^{K} h(\mathbf{b}_j)(h(\mathbf{a_b}_j^{[i]}) - h(\mathbf{a}_j))$ and $V_i^T \approx \frac{1}{2K}\sum_{j=1}^{K}(h(\mathbf{a_b}_j^{[i]}) - h(\mathbf{a}_j))^2$.

Sobol's method suits our setting perfectly. Given a plan π obtained under selectivity estimates $\hat{\mathbf{s}}$, we analyze the function $h(\mathbf{s}) = \mathsf{Penalty}(\pi, \mathbf{s})$ by drawing the $2K$ samples from $f(\mathbf{s}|\hat{\mathbf{s}})$. The first-order and total-order indices give principled and interpretable measures of sensitivities that are tailored to the user-defined notion of penalty and are informed by error profiles observed from the database workload. There are good arguments for using either first-order or total-order indices (or even both); our current implementation simply uses the first-order indices.

We denote the *Sobol-sensitivity* for dimension i as $\xi_i^{\text{sobol}}(\pi, \hat{\mathbf{s}})$. Overall, this analysis uses K pairs of sample points, each requiring evaluating $\mathsf{Penalty}$ $d + 2$ times. The total cost of Sobol is $O(Kd)$ Opt and Cost calls. We show practical K values to reach convergence in Section 6; Sobol is generally slower to converge than Morris.

4.2 Sensitive Dimensions as Tuning Hints

PARQO uses Sobol-sensitivity by default to identify sensitive selectivity dimensions for a given plan. Practically, as we have found through experiments in Section 6, the actual Sobol-sensitivity values of the dimensions make it easy to identify a small number of dimensions that clearly stand out. For example, for all queries in JOB, this number varies between 2 to 6. We now describe how these sensitive dimensions are presented by PARQO to users to help them understand and fine-tune plan performance.

Recall from Section 3 that all relevant dimensions are pegged to selection and join conditions in the query, but their error profiles in fact capture more than a single predicate. Hence, PARQO is careful in presenting such dimensions to users. For example, the most sensitive dimension for Q17 is $mk \bowtie k^{\sigma}$ (Example 1). This selectivity needs to be understood as the join selectivity between mk and k *assuming a local selection on* k, which is different from the "plain" join selectivity of $mk \bowtie k$ (which should have no estimation error at all by itself since it is a join between foreign and primary keys). The second, and the only other sensitive dimension for Q17, is associated with the join between mk and ci, and will be presented to users as $(mk \ltimes k^{\sigma}) \bowtie (ci \ltimes n^{\sigma})$. Since neither mk nor ci has any local selection in Q17, the error distribution is derived from the error profiles for querylets $k^{\sigma} \bowtie mk \bowtie ci$ and $mk \bowtie ci \bowtie n^{\sigma}$.

With this information, users may decide to investigate further and take action in several ways, focusing now on these two dimensions instead of all 12 relevant dimensions originally in Q17. For example, they may want to devote more resources to collecting statistics and/or training models relevant to these two dimensions, or simply do some additional probing to get better selectivity estimates for these dimensions and ask the optimizer to re-optimize under these new estimates. Example 1 already mentioned that correcting the error in the most sensitive dimension ($mk \bowtie k^{\sigma}$) leads to a 5.84× speedup in actual execution time of Q17. If we instead correct the error for the second most sensitive dimension $(mk \ltimes k^{\sigma}) \bowtie (ci \ltimes n^{\sigma})$ alone, the speedup will be 1.38×. Finally, if we correct both errors, the speedup will be 6.4×.

Beside presenting the sensitive dimensions appropriately to users and allows them to experiment with different selectivities, PARQO currently does not offer any additional user-friendly interfaces. There are abundant opportunities for developing future work and applying complementary work (e.g., [23, 28, 46, 50]) on visualizations and interfaces, such as tools for interactively exploring the penalty and optimal plan landscapes along sensitive dimensions.

5 FINDING ROBUST PLANS

Given a query template Q and selectivity estimates $\hat{\mathbf{s}}$, our goal is to find a plan π that minimizes the expected penalty $\mathrm{E}[\mathsf{Penalty}(\pi, \mathbf{S})|\hat{\mathbf{s}}]$. This penalty-aware formulation allows for powerful notions of robustness that are based on global properties of the plan space (since penalties are relative to optimal plans with true selectivities), as opposed to simple measures such as those based on the local properties of the cost function for π itself [52]. This stochastic optimization formulation further enables optimization informed by distributions of selectivity estimation errors observed in workloads, which are more focused and less conservative than formulations that consider the entire selectivity space, e.g. [1, 8]. In the following, we first describe the end-to-end procedure for finding a robust plan for a single query, and then discuss how to reuse its effort across multiple queries, in the setting of parametric query optimization.

5.1 Finding One Robust Plan

As the first step, PARQO performs the sensitivity analysis in Section 4 on the optimizer plan $\pi^{\star}(\hat{\mathbf{s}})$ to identify a small subset of sensitive dimensions. Subsequent steps then operate in the subspace consisting of only the sensitive dimensions. In remainder of this subsection, with an abuse of notation, we shall continue to use d for the now reduced number of dimensions and $\mathbf{s}, \hat{\mathbf{s}}$ for their projected versions; $f(\mathbf{s}; \hat{\mathbf{s}})$ would be obtained by Equation (3) in Section 3 using only g_i's for sensitive dimensions.

Next, PARQO computes a set of plans, called the *robust candidate plan pool*, as follows. We draw a sequence of S samples from $f(\mathbf{s}; \hat{\mathbf{s}})$. For each sample $\mathbf{s}$, we call Opt with these selectivities to obtain the optimal plan $\pi^{\star}(\mathbf{s})$ at $\mathbf{s}$ and its cost $\mathsf{Cost}^{\star}(\mathbf{s})$ at $\mathbf{s}$. We cache the triple $\langle \mathbf{s}, \pi^{\star}(\mathbf{s}), \mathsf{Cost}^{\star}(\mathbf{s}) \rangle$ (whose purpose will become apparent later), and register each unique optimal plan in the pool.

Finally, in the third step, for all unique plans in the candidate pool, PARQO estimates their expected penalties and returns the one with the lowest expected penalty. Note that this estimation is done using cache populated in the previous step, since its entries were sampled from $f(\mathbf{s}; \hat{\mathbf{s}})$ in the first place. Specifically, we estimate $\mathrm{E}[\mathsf{Penalty}(\pi, \mathbf{S})|\hat{\mathbf{s}}]$ as $\frac{1}{S}\sum_{\text{cached } \langle \mathbf{s}^{\star}, \_, c^{\star} \rangle} \mathsf{Penalty}(\pi, \mathbf{s}^{\star})$, where each $\mathsf{Penalty}(\pi, \mathbf{s})$ is evaluated using cached $c^{\star}$ plus a call for $\mathsf{Cost}(\pi, \mathbf{s}^{\star})$.

Overall, not including sensitivity analysis (whose complexity was given in Section 4), the process takes $O(S)$ calls to Opt and $O(S \times \grave{S})$ to Cost, where $\grave{S}$ denotes the number of unique candidate plans. We show the practical S and $\grave{S}$ values we used for the JOB benchmark in Section 6. Finally, note that opportunities also exist for caching and reusing the samples acquired during sensitivity analysis.[3] We did not explore these opportunities in this paper because we did not want to introduce extra dependencies across components that may complicate understanding of performance.

5.2 Parametric Robust Query Optimization

PQO works by caching several plans for the same query template as candidates. Given an incoming query with the same template, PQO would select one of the cached candidates instead of invoking the optimizer, which is far more expensive. It is natural for PARQO to combine robust query optimization and PQO, not only because PQO helps amortize the overhead of robust query optimization across multiple queries, but also because robust query optimization involves significant effort beyond optimizing for a single point in the selectivity space, which intuitively should help PQO as well. This combination allows PARQO to both reduce the optimization overhead and deliver better plans than a traditional optimizer.

Suppose that PARQO has already done the work of optimizing a query with estimated selectivity $\hat{\mathbf{s}}$. Now consider an incoming query with the same template but different parameters and hence different estimated selectivity $\hat{\mathbf{s}}'$. Many opportunities exist to reuse earlier work: we could assume the same set of sensitive dimensions; we could reuse the cached optimal plans and costs collected while finding the most robust plan for $\hat{\mathbf{s}}$ (Section 5.1); or we could go as far as returning the same robust plan. While the last option is the cheapest, it would either require a stringent reuse condition that limits its applicability, or give up any form of guarantee on the actual robustness under the new setting. Hence, PARQO takes a more measured approach, as described below.

First, it would be unrealistic to assume that set of sensitive dimensions always stays the same. Recall from Section 4 that sensitivity analysis is done for an optimizer plan at a particular setting of selectivity estimates. We can only expect sensitivity analysis to yield same or similar results if the penalty "landscapes" around $\hat{\mathbf{s}}$ and $\hat{\mathbf{s}}'$, induced by estimation error, are similar. We use the KL-divergence between the distributions $f(\mathbf{s}|\hat{\mathbf{s}})$ and $f(\mathbf{s}|\hat{\mathbf{s}}')$, denoted $\mathsf{KL}(f(\mathbf{s}|\hat{\mathbf{s}}) \parallel f(\mathbf{s}|\hat{\mathbf{s}}'))$ as a test.[4] (Importantly, these distributions include *all* dimensions, not merely the sensitive dimensions selected for $\hat{\mathbf{s}}$.) If the KL-divergence is low (we will discuss how to set this threshold shortly), we allow the set of sensitive dimensions for $\hat{\mathbf{s}}$ to be reused for $\hat{\mathbf{s}}'$ and continue with other reuse opportunities. Otherwise, we look for an different $\hat{\mathbf{s}}$ to reuse, analyze/optimize $\hat{\mathbf{s}}'$ from scratch, or simply fall back to the traditional optimizer. We argue that the KL-divergence between distributions of true selectivities (conditioned on the estimates) is a more principled and effective reuse test than those based on surrogates such as similarity among query parameter values.

Now, assuming $\hat{\mathbf{s}}'$ has passed the KL-divergence test for reusing $\hat{\mathbf{s}}$, we reuse the S cached samples and the $\grave{S}$ candidate plans when we optimized for $\hat{\mathbf{s}}$. One complication is that the cached samples were drawn from $f(\mathbf{s}|\hat{\mathbf{s}})$ instead of $f(\mathbf{s}|\hat{\mathbf{s}}')$. Hence, when computing expected penalty for a candidate plan π at the $\hat{\mathbf{s}}'$, we apply *importance sampling* [31], which lets us evaluate properties of a target distribution using samples drawn from a different distribution. Specifically, the expected penalty of candidate plan π can be estimated as: $\frac{1}{S}\sum_{\text{cached }\langle \mathbf{s}^\star,\_,c^\star\rangle} \frac{f(\mathbf{s}^\star|\hat{\mathbf{s}}')}{f(\mathbf{s}^\star|\hat{\mathbf{s}})} \mathsf{Penalty}(\pi, \mathbf{s}^\star)$, where the fraction $\frac{f(\mathbf{s}^\star|\hat{\mathbf{s}}')}{f(\mathbf{s}^\star|\hat{\mathbf{s}})}$ reweighs the sample to account for the difference between distributions. Among the $\grave{S}$ candidates, we then pick the one with the lowest expected penalty. With this technique, no Opt or Cost calls are needed to find the robust plan for $\hat{\mathbf{s}}'$.

If the two distributions are very different, however, importance sampling will require more samples to provide a reasonable estimate. The lower bound of the sample size required to ensure estimation accuracy through importance sampling is discussed in detail in [7]. This lower bound is indeed determined by the KL-divergence between the two distributions. According to this lower bound, we derive the maximum KL-divergence under which S samples are able to provide acceptable accuracy. We use this threshold for the reuse test described earlier in this subsection, ensuring that it is safe to also reuse the same samples for expected penalty calculation.

6 EXPERIMENTS

We have implemented PARQO on top of the latest version of PostgreSQL, V16.2. We modified PostgreSQL to expose Opt and Cost calls, with no changes to its optimizer or executor otherwise; plan and selectivity injection is done as hints to PostgreSQL, with help of [20] and [38]. We have open-sourced our implementations in [56].

We use three benchmarks in evaluation. **JOB** (Join Order Benchmark) [33] contains 33 query templates and 113 query instances, with real-world data from IMDB. This benchmark includes skewed and correlated data distributions as well as and diverse join relationships, all of which contribute to selectivity estimation errors. **DSB** [12] is an industrial benchmark that builds upon TPC-DS [39] by incorporating complex data distributions and join predicates. **STATS**(-CEB) [20] features a real-world dataset from the Stats Stack Exchange. This paper will focus on evaluation results on JOB, and only summarize the results on DSB and STATS; additional details are available in [57]. Each experiment setup involves two query workloads. First, a *profile workload* is used by PARQO to build error profiles as described in Section 3; example result error distributions can be found in [57]. Second, a separate *evaluation workload* contains queries that are targets of our evaluation. We will describe these workloads when discussing specific experimental setups.

Unless otherwise specified, PARQO uses the penalty function in Equation (1) with $\tau = 1.2$, a setting that is widely used in the robust query optimization literature, e.g., by [11, 14, 21, 52]. When using Morris and Sobol for sensitivity analysis (Section 6), we sample until convergence, so the parameter K varies across queries; the number of sensitive dimensions depends on the distribution of scores and also varies. To find robust plans (Section 5.1), we use $S = 100$ samples to build the candidate plan pool for each Q; the number of unique candidate plans per query varies. We report summary statistics on these varying quantities in Table 1, along with other useful measures such as the memory footprint of the error models, the number of relevant dimensions per query, etc.

All experiments were performed on a Linux server with 16 Intel(R) Core(TM) i9-11900 @ 2.50GHz processors. To reduce noise

[3]Strictly speaking, there is a slight difference in their distributions: samples in Morris and Sobol (Section 4) were drawn from the original $f(\mathbf{s};\hat{\mathbf{s}})$ with all dimensions, whereas samples in this subsection are drawn from $f(\mathbf{s};\hat{\mathbf{s}})$ restricted to only the sensitive dimensions. This difference can be corrected if needed.

[4]These distributions are conditioned on the estimates; even if the error profiles relative to $\hat{\mathbf{s}}$ and $\hat{\mathbf{s}}'$ are the same, the distributions of true selectivities will have little in common if $\hat{\mathbf{s}}$ and $\hat{\mathbf{s}}'$ are far away.

when measuring execution time, we execute each plan multiple (no fewer than 5 and up to 101) times and record the median latency.

Traditional vs. Robust Plans on Current Database Instance. As a warm-up, consider a setup where given each query, we compare the actual execution times for the following plans: *PostgreSQL* denotes the plan found by the PostgreSQL optimizer with its default selectivity estimates $\hat{\mathbf{s}}$ (after refreshing all statistics on the current database instance); *WBM* denotes the plan obtained using the approach of [52][5]; *Recentering* refers to the baseline introduced in Section 3, where we correct PostgreSQL's estimates using the expectation of $f(\mathbf{s}|\hat{\mathbf{s}})$ derived from the error profiles; *PARQO-Morris* and *PARQO-Sobol* refer to the plans chosen by PARQO for $\hat{\mathbf{s}}$, using the Morris and Sobol's Methods for picking sensitive dimensions, respectively. Before proceeding, we note that this setup is not ideal for evaluating robust plans. The advantage of robust plans should be their overall performance over a range of possibilities, but the current database instance only reflects one of these possibilities. Nonetheless, given a benchmark database, users inevitably wonder how different plans perform on it, so this setup is natural. Instead of fixating on one particular query's performance, however, we can get better insight on robustness with an overall comparison over all queries in the workload. We also will follow up with additional experiments later to examine each plan's performance under different errors and different database instances.

Figure 1 summarizes the results on JOB[6]: the x-axis is labeled by queries; on top we show execution times on a log-scale y-axis; on bottom we additionally show speedup/regression factors on the y-axis. Among the 33 queries in the evaluation workload, PARQO-Sobol outperforms PostgreSQL in 19 of them (with an overall speedup[7] of 4.51×) but underperformed in 5 of them. For clarity, we group them into (a) and (b) in Figure 1. The remaining 9 queries are omitted because PostgreSQL and PARQO-Sobol plans have nearly identical times, and WBM is no faster either.

From Figure 1, we see that PARQO-Sobol outperforms others in most cases. The most notable improvements are in Q17, where PARQO-Sobol takes 620ms while PostgreSQL and WBM take more than 5,000ms, and in Q20, where PARQO-Sobol achieves a speedup of 12× over PostgreSQL. PARQO-Morris, although not as effective as PARQO-Sobol, still surpasses WBM in most cases.

WBM fails to offer much improvement over PostgreSQL here, because it by design avoids plans that cost much higher than PostgreSQL for the original estimates $\hat{\mathbf{s}}$, but this cutoff overlooks the (sometimes likely) possibility that $\hat{\mathbf{s}}$ is far off from reality. As an example, for Q17, the PostgreSQL plan costs 4.6k (in PostgreSQL cost unit) at $\hat{\mathbf{s}}$ while PARQO-Sobol's plan costs 12.5k; therefore, WBM did not consider PARQO-Sobol's plan at all, but instead picked a plan similar to PostgreSQL. However, it turns out $\hat{\mathbf{s}}$ is really off: in reality, PostgreSQL and WBM ran more than an order of magnitude slower than their cost predicted at $\hat{\mathbf{s}}$, while PARQO-Sobol ran > 8× faster than them. This example highlights the need to consider errors instead of relying purely on decisions local to $\hat{\mathbf{s}}$.

As for Recentering, it sometimes provides impressive speedups (e.g., Q2, 26, and 30), which indicates that our error profiling, despite its simplicity, can already deliver some benefits by correcting biased estimates. However, Recentering is still far less effective than PARQO-Sobol overall (e.g., Q7, 18, 20, and more), which is evidence that bias correction along is not sufficient — other components of PARQO also play a significant part in its overall effectiveness.

We now turn to queries where PARQO-Sobol underperforms PostgreSQL. As argued above, a better way of evaluating robust plans is to examine their performance over a range of situations. Indeed, in later experiments such as PQO, we will see that PARQO-Sobol plans are robust despite their misfortune on the current database instance. For example, Q6 is the worst case for PARQO-Sobol in this experiment, but in the PQO setting we are able to achieve a 2.57× speedup (Figure 6). Nonetheless, it is instructive to study why the robust plans underperform in these particular cases. Delving deeper, we believe the reason lies in the uncertain nature of selectivity estimates. For instance, Q1 has a sensitive dimension matching $it^{\sigma} \bowtie mi\_idx$ with *it.info = 'top 250 rank'*; it happens that the estimate was not bad, but our error profiling thought the error would be large. Similarly, for querylet $mk \bowtie k^{\sigma}$ in Q3, 4, and 6, and $cn^{\sigma} \bowtie mc^{\sigma}$ in Q15, the actual errors were somewhat unlikely according to our error profiles. Such discrepancies could arise due to uncertainty, which is inevitable, or due to poor error profiling; it is hard to tell which on the basis of a few particular cases. As our later experiments that examine many possibilities in aggregate generally produce good results, we believe our error profiling is adequate if still imperfect.

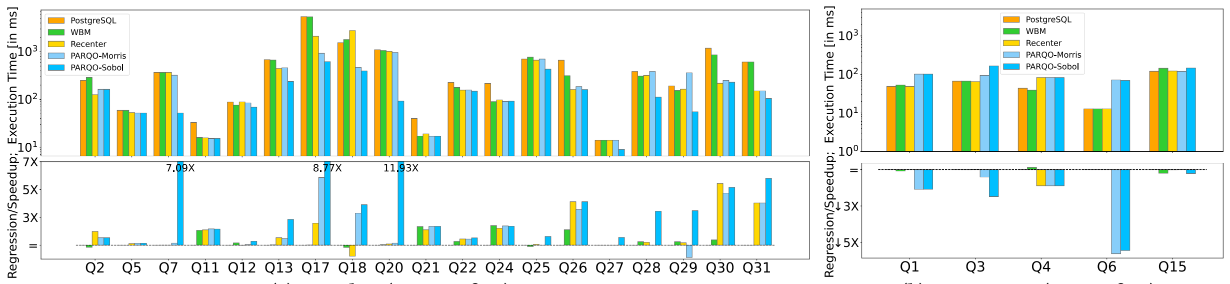
The overall speedup for the entire evaluation workload of JOB is 3.23×. Detailed results for DSB and STATS can be found in [57]. As a brief summary here, the overall speedups for all queries in DSB and STATS are 2.01× and 1.36×, respectively. For DSB, PARQO outperforms PostgreSQL in 8 out of 15 queries, with a maximum speedup of 8.8×; for STATS, PARQO outperforms PostgreSQL 10 out of 26 queries, with the highest speedup of 425.7× observed. The only regression across these benchmarks occurs in S120 (↓ 1.87×); however, the benefits can still be evident in the PQO experiments.

Demonstrating Robustness over Error Distribution. Next, we demonstrate the robustness of various plans by showing their cost penalties over possible errors in selectivity estimates. Continuing with the previous setup, for each plan and the initial selectivity estimates $\hat{\mathbf{s}}$, we sample true selectivities $\mathbf{s}$ according to Equation (3) obtained by our error profiling (hence, the results here do not validate the quality of error profiling itself), and cost the plan at $\mathbf{s}$. The resulting costs are shown as a cumulative density function. Figure 2 summarizes the results for JOB. There are too many queries to show, so we choose four as representative examples: Q2, 17, 26, and 15. They represent a range of complexities, from simpler 5-table joins to more complex 12-table joins, and Q15 is intentionally chosen as it showed a regression in our first experiment (Figure 1). Since PARQO-Sobol is generally more effective than PARQO-Morris, we focus on PARQO-Sobol here. As shown in Figure 2, PARQO plans indeed demonstrate robustness: they have substantially lower chance of incurring large cost penalties compared with plans selected by

[5] [52] proposed three robustness metrics; we show only the plan with the fastest execution time. WBM sets a threshold (120%) relative to the cost of the optimizer plan at $\hat{\mathbf{s}}$; it would not consider robust plans with cost higher than this threshold.

[6] In JOB, 33 instances labeled "(a)" are used as the evaluation workload, while all other 80 instances are used to collect possible predicate literals and construct error distributions.

[7] Note here and after that we calculate the "overall" speedup/regression for a collection of queries as the speedup/regression in the *total execution time* over all queries (as opposed to the arithmetic mean of the speedup/regression factors of individual queries), so speedup/regression in slower queries contribute more than faster queries.



(a) Speedup (19 out of 33) **(b) Regression (5 out of 33)**

Figure 1: Actual execution times for different plans selected by various methods on JOB. (a) and (b) separate queries for which PARQO-Sobol outperforms or underperforms PostgreSQL; queries for which they perform similarly (9 out of 33) are omitted.

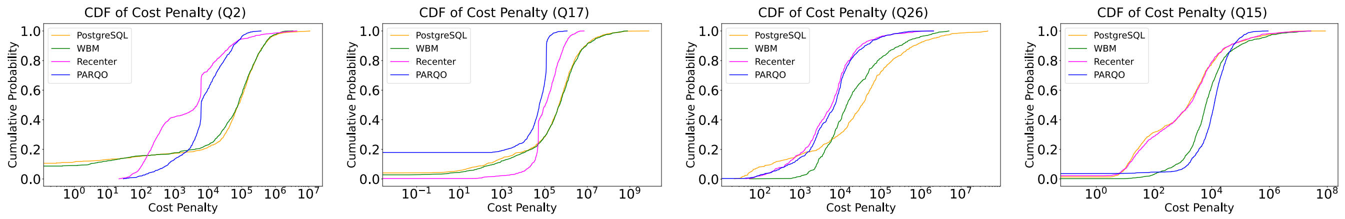


Figure 2: Cumulative density of cost penalty incurred by plans for Q2, 17, 26 and 15 in JOB.

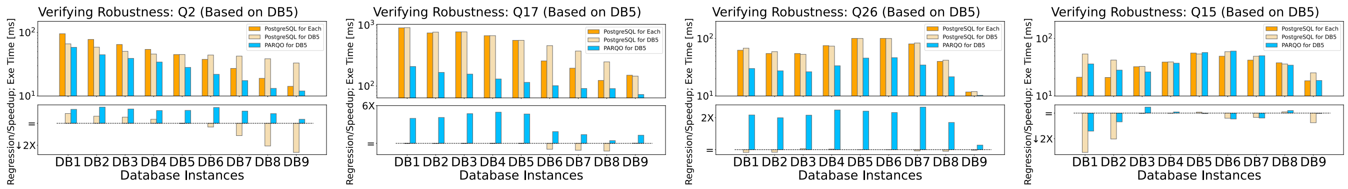


Figure 3: Actual execution times of JOB queries on multiple instances by time-slicing IMDB. DB5 is the base instance.

alternative methods. For example, for Q17, PostgreSQL and WBM plans incur $\geq 10^6$ penalty 30% of the time, while the worst-case penalty of PARQO is 10^6. Notably, for Q15, we do see that robustness comes with a price in the low-penalty region: 30% of the time, PostgreSQL and WBM have penalties $\leq 10^2$, while PARQO can reach 10^4. However, the protection offered by PARQO shines in the high-penalty region: PostgreSQL and WBM plans have non-trivial probabilities of incurring catastrophic penalties between 10^6 and 10^8, while PARQO only reaches 10^5 to 10^6 in the worst case.

Verifying Robustness using Multiple Database Instances. The above demonstration assumes that estimation errors follow the distribution obtained using our profiling method, but we also wish to test robustness in less controlled settings encountered in real-world scenarios where additional errors arise as databases evolve. To simulate such settings, for JOB, which has a static snapshot of the IMDB dataset, we create multiple database instances by slicing the original dataset into smaller pieces. We choose one of these as the *base instance*, and apply PARQO (and alternatives) to choose the best plan using information on this instance alone (e.g., error profiling is done only on this instance). Then, we execute and time the same plan on the other instances, without knowledge of or regard to selectivities or estimation errors on these instances. For comparison, we also run the same PostgreSQL plan chosen for the base instance on these instances, as well as the PostgreSQL plan optimized specifically for each instance (which has the advantage of seeing its statistics). We use the latter as the reference for speedup/regression factors.

We consider two ways to slice the IMDB dataset used by JOB. The first is ***time-slicing***, where we sort the the title (t) rows by *production_year* and use a sliding window on them to create 9 instances labeled DB1–DB9. Each instance contains 20% of the title rows along with associated data from other tables, and two consecutive instances have 10% of the title in common. The results for the same four representative queries from JOB are shown in Figure 3, with DB5 as the base instance. For Q2, 17, and 26, we see that the plan chosen by PARQO with the knowledge of DB5 outperforming PostgreSQL plans not only for DB5 but also for all other instances; it even outperforms the instance-optimized PostgreSQL plans, which were obtained with access to better information on their corresponding instances. For Q15, PARQO is just slightly worse than the base PostgreSQL plan (for DB5) on 3 instances (DB5, DB6, and DB7, which are consecutive in time and may have similar statistics) out of the 9; however, it is much more robust overall, avoiding the significant performance degradation experienced by the base PostgreSQL plan on DB1, DB2, and DB9. It is able to outperform the instance-optimized PostgreSQL plans on DB3, DB4, and DB9, despite not having any knowledge about these instances.

The second way to create multiple instances for JOB is ***category-slicing***, where we partition the IMDB dataset by item categories (*kind_type.kind*) such as "Movie" and "TV Series", and name each of the 6 result instances by the category. We intended this partitioning to create more challenging scenarios than time-slicing, because items in these categories follow very different distributions. The results, detailed in [57], point to similar conclusions as above.

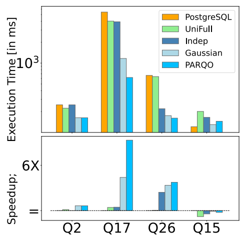


Figure 4: Comparison of error profiling methods.

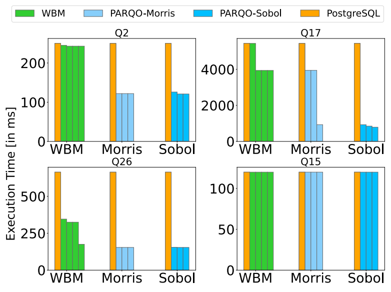


Figure 5: Improvements by correcting estimates for sensitive dimensions.

Impact of Error Profiling Strategies. We further explore the impact of various approaches to error profiling on PARQO's performance. First, the traditional PostgreSQL optimizer can be seen as taking an extremely simple approach of assuming no estimation error. The second approach, *UniFull*, encodes the assumption in [1, 8, 10] that true selectivities are drawn uniformly at random from the entire selectivity space. The third approach, *Indep*, assumes that join and selection selectivities are estimated independently and their estimation errors are also independent; hence, we only need to profile errors for each selection and join condition in isolation. The fourth approach, *Gaussian*, is identical to PARQO's method described in Section 3, except that it fits a single Gaussian distribution to each bucket of collected errors instead of using kernel density estimation. We run PARQO-Sobol with error models obtained under these strategies to optimize the four representative JOB queries, and the resulting plans are timed. The results, shown in Figure 4, indicate that *UniFull* yields marginal improvement over PostgreSQL, underscoring the importance of incorporating better knowledge on errors in robust query optimization. *Indep* does better than *UniFull* but still much worse than *Gaussian* and PARQO's default method, highlighting the need to profile dependencies among selectivities as we described in Section 3. Finally, *Gaussian* further improves upon *Indep* but sometimes underperforms PARQO's default, because its single-Gaussian model is crude compared with PARQO's default. For this experiment and all other experiments including those on DSB and STATS, the memory footprint of PARQO's error model is always under 15KB. This low memory usage leaves considerable room for improving model accuracy; it will be interesting future work to investigate how much additional improvement can be gained with more sophisticated error modeling.

Effectiveness of Sensitive Dimensions in Prioritizing Corrections of Estimates. To show that PARQO can identify a good set of sensitive dimensions (Section 4), we consider the following setup, motivated by [9, 32]. Given a plan optimized by PostgreSQL with estimates $\hat{\mathbf{s}}$, and a list of sensitive dimensions recommended by different methods, we would acquire the true selectivity values for these dimensions[8] and ask PostgreSQL to reoptimize the query based on the accurate selectivities instead of their estimates. We process the list of sensitive dimensions iteratively and obtain a new plan after correcting one additional dimension at a time; all plans are executed and timed. We consider the three most sensitive dimensions found by the Morris and Sobol's Methods, sorted by their sensitivity. We compare them with WBM's choice of sensitive dimensions, which include all non-key-foreign-key join selectivities; these dimensions are ordered using $\partial\,\mathsf{Cost}(\pi, \mathbf{s})/\partial s_i$, based on one of their robustness metrics. We also note that WBM's behavior in this experiment is not affected by the 120% threshold.

Figure 5 shows the results on the full IMDB database, with the progression of bars showing how quickly query performance is improved by following each recommendation. The extra last bar for WBM shows the final plan after processing *all* of WBM's sensitive dimensions. For Q2, 17, and 26, we see that correcting the top three sensitive dimensions with both Sobol and Morris results in significant speed improvements, but Sobol "converges" quicker. WBM is only able to match the same improvement for Q26 after correcting all its sensitive dimensions; for Q2 and 17, it never reaches the level of Sobol and Morris. Finally, for Q15, none of the methods improves upon the original PostgreSQL plan, indicating that this plan is already very good for the given database instance.

There is also a trade-off in how expensive these methods are. The slope-based metric in WBM only require d-1 calls to Opt, as it is local and OAT. Morris and Sobol perform better but require far more Opt calls. As an example, for Q17, Morris requires 520 calls ($K = 40$) to its solution, while Sobol requires 1,664 calls ($K = 64$). In fact, the cost of finding sensitive dimensions dominates that of robust query optimization — once the sensitive dimensions are identified, finding the robust plans only requires additional $S = 100$ Opt calls (we also experimented with $S = 1{,}000$ but did not find obvious improvement in overall performance). The total overhead is considerable, averaging at several minutes per query, which renders the approach applicable only to very slow queries. Luckily, the complexity of robust query optimization depends only on query complexity and not on data complexity, so it is more appealing to massive databases. For faster queries, instead of sacrificing solution quality and principality, we argue for combining robust query optimization with parametric query optimization, such that the overhead of optimization is amortized over many queries sharing the template. Next, we present results from the PQO experiments, along with a more detailed analysis of overhead.

Parametric Query Optimization. The PQO experiment setup for JOB requires a bigger evaluation workload of queries beyond the 113 included in the benchmark. Here, we use all 113 queries as our profiling workload and collect all literals therein by their attribute domains. The multiset of literals from the same domain defines the distribution to be used when generating new queries requiring literals from this domain. For each of the 33 templates, we generate 1000 random query instances for the evaluation workload, where each literal is replaced with one randomly drawn from the same domain. We treat the 33 JOB queries labeled (a) as anchors, perform robust query optimization on each, and populate the PQO plan cache with 100 samples and up to 3 robust plan candidates obtained when optimizing the anchor (Section 5.2). While it is certainly possible to use more than one anchor per template or to cache more per anchor, we have found this modest level is already sufficient to achieve satisfactory performance.

Overall, the total time for PostgreSQL to execute the entire evaluation workload of 33,000 queries is 6.59 hours. PARQO's PQO

[8]For the purpose of this experiment, we simply run a COUNT subquery for each selectivity of interest, but in practice one can instruct the database system to refresh statistics relevant to the selectivities or use sampling method to answer the COUNT subqueries quickly but approximately.

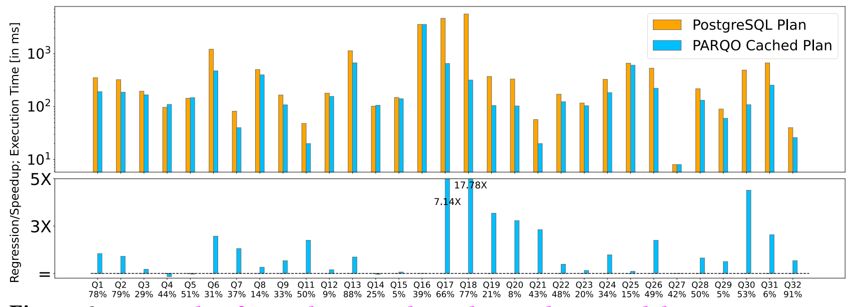


Figure 6: PQO results of JOB. The x-axis shows the template ID and the average reuse fraction of each template.

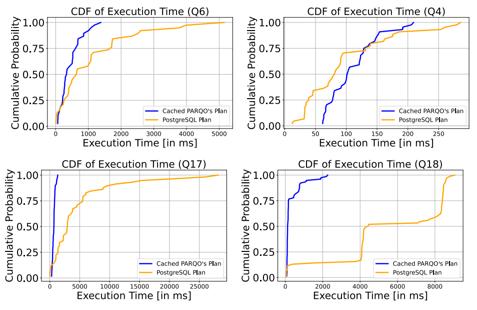


Figure 7: Cumulative density function of execution time in PQO: for Q6, Q4, Q17 and Q18

setup reduces this time to 4.34 hours (not yet including the upfront overhead of populating the cache, which we discuss later). Figure 6 summarizes the results by query template. We use the term *reuse fraction* to represent the proportion of queries that trigger reuse (by passing the KL-divergence test with respect to the anchor associated with its template). The average reuse fraction over all templates is 37%. For each template, Figure 6 compares the average execution time over queries that reused a cached PARQO plan against plans that PostgreSQL chooses. We omit templates with a reuse fraction below 5% (Q10 and 33) because the numbers are too low to draw reliable conclusions.[9] Among the remaining 31 templates, PARQO plans achieve a speedup in 28 of them. Notably, template Q18 has a 17.8× speedup. For the 3 templates with no speedup, Q4, 5, and 14, the worst regression is only ↓ 1.1×. Recall that besides Q4, in the earlier experiment in Figure 1b, PARQO also underperformed PostgreSQL on Q1, 3, 6, and 15; however, here with PQO, we see that queries with templates Q1, 3, 6, and 15 have an overall speedup.

Additionally, Figure 7 shows the distribution of query execution times for queries in four representative templates. The four are chosen for different reasons. Q6 was the "worst" query for PARQO in the earlier experiment in Figure 1b. Here, we see that while PostgreSQL is slightly faster than PARQO for 20% of the queries with running time below 250ms, PostgreSQL causes 15% of the queries to experience substantially longer running times than PARQO; overall PARQO in fact provides a significant speedup (Figure 6). Next, Q4 is the "worst" query template for PARQO in Figure 6. Even in this case, its execution time distribution provides protection against some long-running times incurred by PostgreSQL. Finally, Q17 is also one of the representative queries chosen in earlier experiments, and Q18 is the "best" query template for PARQO in Figure 6. As can be seen from Figure 7, PostgreSQL oftentimes makes disastrous choices for queries with these two templates, yet PARQO's robust plans help avoid these situations.

In closing, we present a detailed analysis of the various overheads incurred by PARQO in robust PQO compared with traditional query optimization. First, for JOB, PARQO incurs a one-time, upfront cost of 2.13 hours to populate its PQO cache for all 33 query templates, which averages at about 4 minutes per template. While a considerable amount of overhead, it depends on the complexity of the query rather than the size of data, and if the query template is used often, the initial investment pays off quickly. Recall that

[9]For these and other templates with relatively low reuse fractions, it seems that their anchors' selectivities are quite different from most of the queries from the evaluation workload. A smarter way of picking anchors that adjusts to the query workload should be a helpful future work direction.

	JOB	DSB	STATS
# of samples S	100	100	100
Physical size of $f(\mathbf{s}\mid\hat{\mathbf{s}})$	13.8 KB	13.66 KB	5.84 KB
# of relevant dimensions d	6-34	8-25	6-20
# of sensitive dimensions	2-6	2-4	1-4
# of seeds K of Morris	20-160	10-80	20-80
# of seeds K of Sobol	8-128	8-64	8-64
Avg # of unique plans $\hat{S}$	18	14	10
Up-front overhead of PARQO	2.13 h	0.99 h	0.74 h
Overall speedup per query	3.23×	2.01×	1.36×
Workload size in PQO	33,000	15,000	22,000
Average reuse fraction	37%	93%	43%
Execution time saved by PQO	2.25 h	0.68 h	11.47 h
Optimization time saved by PQO	0.03 h	0.09 h	0.01 h

Table 1: Summary of PARQO on three benchmarks.

PostgreSQL runs the entire evaluation workload in 6.59 hours while PARQO runs it in 4.34 hours; the saving of 2.25 hours is already more than the initial overhead of 2.13. A simple calculation reveals that it takes on average about 934 JOB query instances per template to break even the upfront optimization cost. This target is not difficult to reach in production settings where there is a database application with a limited set of query templates and many users, or when queries are more expensive than the benchmark setting we experimented with. Delving deeper, PARQO saves time not only by having better plans, but also by reducing runtime optimization overhead. PARQO's runtime overhead depends on the number of cached plans and samples. Under our experimental setting, over all query instances that triggered reuse, PARQO has an average optimization overhead of 5.58ms per query, which is much lower than PostgreSQL's optimization overhead of 14.9ms. Over all queries instances, PARQO has an average optimization overhead of 55.6ms (including the time spent on KL-divergence testing and the time to fall back to PostgreSQL when the test fails), which is still better than PostgreSQL's average optimization time of 58.8ms. There are ample opportunities for future work on selectively picking anchors for PGO to maximize reuse and avoid those with low reuse.

Finally, we briefly summarize the results of PQO experiments on DSB and STATS; futher details are available in [57]. The average reuse fraction over all templates is 93% for DSB and 43% for STATS. Overall, PARQO achieves improvements of 0.68 hours and 11.47 hours in executing the entire evaluation workload for DSB and STATS respectively. Additionally, it reduces optimization overhead by 0.09 hours for DSB and 0.01 hours for STATS.

7 RELATED WORK

Robust Query Optimization How to improve the ability of error resistant and avoid sub-optimal risks has been widely discussed [6, 19, 22]. RQO can be regarded as part of robust query processing and is classified by the number of plan provided by a recent survey [58]. For single-plan based approach, LEC [10] was among the first to utilize probability density for estimating selectivity, aiming to identify plans with the lowest expected cost. However, LEC restricts the search space of plans and lacks a clear methodology for constructing probability measures. Similarly, [1, 8] pick a plan that has low variance and minimum average cost over extremes for the entire parameter space. Since the error can not be captured in an arbitrary large selectivity space, their effectiveness is limited. In contrast, RCE [4] tries to quantify the uncertainty, but requires random samples from real data at runtime to infer the distribution of actual selectivity. Rio [5] and its extension [3, 17] leverage bounding boxes or intervals to quantify selectivities, and collect the plan as a candidate if the cost is close to optimal over the bounding box. When executing the query, these candidate plans can be utilized as re-optimization alternatives. The idea of adaptive processing, i.e. collecting running time observations and then switching the current plan to another is also leveraged in [14, 16, 47, 53]. [53] identifies cardinality "ranges" where the original plan remains optimal. When the "ranges" are broken during execution, re-optimization will be triggered. This line of work generally requires runtime adaptation and is complementary to our approach, which focuses on compile-time optimization of standard execution plans. These interval-based approaches need to assume that predicate selectivity is known with in a narrow intervals, which is often violated in practical situation [24, 33]. Besides, research on plan diagrams [21, 28] aims to identify a fine-grained set of plan candidates for a query template across the selectivity space, each candidate can be regarded as a robust plan for certain area. Subsequent works [11, 40] present methods to reduce the complexity of the diagram, but they still necessitate time-consuming offline training through repeated invocations of Opt. Additionally, their plan selection is still reliant on $\hat{\mathbf{s}}$, which may lead to sub-optimal outcomes. Risk Score [26] employs the coefficient of variation to measure the robustness of execution plans. However, this metric requires real execution times under various actual selectivity values. MSO [14, 28, 29, 40] is widely used in robust query processing that quantifies the worst-case sub-optimality ratio across the entire selectivity space. It relies on the availability of the real optimal plan, which is typically only known to the optimizer after the query execution has begun. [34, 35] learn from real executions to make the *optimizer* more robust by improving the mapping between "plan" to "execution time", and DbET [34] shares a similar idea that predicts the latency of a plan as a distribution. Our approach differs in that we aim to identify sensitive cardinality uncertainties and select robust plans in a more explainable manner. [32] analyzes the tolerance of a plan to cardinality errors posteriorly, requiring true cardinality for all sub-queries and extensive real execution. In paper, instead, we present a principled approach to measure sensitivity. WBM [52] presents three alternative metrics (based on the slope or integral of the cost function) that only based on estimation to measure the robustness. The robustness metrics in PARQO follow this direction that are only based on estimation without executing the query or subquery and independently evaluate each plan.

Parametric Query Optimization PQO has been a subject of study for three decades [25, 27]. The primary focus is to minimize the optimizer's invocation by utilizing cached plans while ensuring the plan's cost remains acceptable. According to [15], current PQO methods can be classified by the plan identification phase, which includes online and offline-based methods. Online-based methods are widely used in commercial DBMS [36]. [2, 18] build a density map by clustering executed queries to select stored historical plans for new query. Idea of storing the optimality ranges for plans [53] can also be applied. A recent study [15] introduces "re-cost" to efficiently calculate $\mathsf{Cost}(\pi, \mathbf{s})$, thereby reducing overhead. "re-cost" is demonstrated effective [11] and also employed in PARQO. For offline-based methods, the objective is to identify a set of plans work for the entire selectivity space [25, 27]. Plan diagram [21, 23] is applicable in this setting. A novel framework [13, 48] uses actual execution times for all candidate plans to train a model for each template and predict the best plan for new queries. Candidate plans are searched from executed queries [48], or generated by randomly perturbing different dimensions [13], which is similar to our candidate plans searching. However, PARQO focuses on sensitive dimensions and samples from $f(\mathbf{s}|\hat{\mathbf{s}})$ directly. [13, 48] demonstrate that learning from real executions can accelerate PQO, offering a promising avenue for future research. As shown in section 6, PARQO can serve as an offline plan caching and online plan selection method, and it can readily be extended to cover the entire selectivity space using techniques such as clustering [2, 18] or plan diagrams [23]. Our experiments demonstrate that robust plans are effective in PQO settings. Even without necessitating real execution times of query instances like [13], PARQO enhances the overall performances.

8 CONCLUSION AND FUTURE WORK

Besides various future work directions already mentioned earlier (such as better error profiling and visualization/interfaces aided by sensitive dimensions), we outline several more below. First, we still do not have a theoretical guarantee on PARQO's solution optimality with respect to robustness. We feel that principled sensitivity analysis proposed by PARQO is a promising approach to the problem from the angle of reducing dimensionality, but more work is still needed in this direction. Another angle that needs to be further investigated in order to achieve any optimality guarantee is the identification of robust plan candidates. Our current approach intuitively looks for candidates among optimal plans at different points in the selectivity space, but what if the most robust plan is not optimal (or even among the top optimal) for any single point? New methods are needed for surfacing such elusive candidates and/or determining that they are unlikely to exist. To make progress, we may need to reconsider the limited ways of interacting with existing query optimizers (which was done by PARQO for practicality), and instead seek tighter integration with the optimizer core. Finally, while having an error distribution enables stochastic optimization, what if the error distribution changes? It can be argued that whenever we notice a significant change in error distribution, the first course of action should be to refresh statistics and retrain models. If that first line of defense is able to restore the error distribution back to acceptable levels, it will help make changes to error distribution smaller or less frequent. Some of the techniques we already employ in PARQO (e.g., KL-divergence tests, caching, and importance sampling) can help detect changes and lower the cost of adaptation, but a comprehensive solution for handling such changes still needs to be developed and evaluated.

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. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1137–1148.

[2] Gunes Aluç, David E DeHaan, and Ivan T Bowman. 2012. Parametric plan caching using density-based clustering. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 402–413.

[3] Khaled Hamed Alyoubi. 2016. *Database query optimisation based on measures of regret.* Ph.D. Dissertation. Birkbeck, University of London.

[4] Brian Babcock and Surajit Chaudhuri. 2005. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 119–130.

[5] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 107–118.

[6] Renata Borovica-Gajic, Goetz Graefe, and Allison Lee. 2017. Robust performance in database query processing (Dagstuhl seminar 17222). In *Dagstuhl Reports*, Vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Sourav Chatterjee and Persi Diaconis. 2018. The sample size required in importance sampling. *The Annals of Applied Probability* 28, 2 (2018), 1099–1135.

[8] Surajit Chaudhuri, Hongrae Lee, and Vivek R Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 531–542.

[9] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2009. Exact cardinality query optimization for optimizer testing. *Proceedings of the VLDB Endowment* 2, 1 (2009), 994–1005.

[10] Francis Chu, Joseph Y Halpern, and Praveen Seshadri. 1999. Least expected cost query optimization: An exercise in utility. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 138–147.

[11] Atreyee Dey, Sourjya Bhaumik, and Jayant R Haritsa. 2008. Efficiently approximating query optimizer plan diagrams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1325–1336.

[12] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[13] 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. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[14] 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.

[15] Anshuman Dutt, Vivek 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*. 1539–1554.

[16] 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.

[17] 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)*. IEEE, 227–235.

[18] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. 2002. Plan selection based on query clustering. In *VLDB'02: Proceedings of the 28th international conference on very large databases*. Elsevier, 179–190.

[19] Goetz Graefe, Wey Guy, Harumi Anne Kuno, and Glenn Paullley. 2012. Robust query processing (dagstuhl seminar 12321). In *Dagstuhl Reports*, Vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[20] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[21] D Harish, Pooja N Darera, and Jayant R Haritsa. 2007. On the production of anorexic plan diagrams. (2007).

[22] Jayant R Haritsa. 2019. Robust query processing: Mission possible. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2072–2075.

[23] Naveen Reddy Jayant R Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st international conference on Very large data bases. VLDB Endowment*. 1228–1239.

[24] Axel Hertzschuch, Guido Moerkotte, Wolfgang Lehner, Norman May, Florian Wolf, and Lars Fricke. 2021. Small Selectivities Matter: Lifting the Burden of Empty Samples. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3448016.3452805

[25] Arvind Hulgeri and S Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 167–178.

[26] Fabian Hüske. 2016. *Specification and Optimization of Analytical Data Flows.* Ph.D. Dissertation. https://login.proxy.lib.duke.edu/login?url=https://www.proquest.com/dissertations-theses/specification-optimization-analytical-data-flows/docview/2314065425/se-2

[27] Yannis E Ioannidis, Raymond T Ng, Kyuseok Shim, and Timos K Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6 (1997), 132–151.

[28] Harish D Pooja N Darera Jayant and R Haritsa. 2008. Identifying robust plans through plan diagram reduction. In *VLDB*, Vol. 24. Citeseer, 25.

[29] Srinivas Karthik, Jayant R Haritsa, Sreyash Kenkre, and Vinayaka Pandit. 2018. A concave path to low-overhead robust query processing. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2183–2195.

[30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[31] Teun Kloek and Herman K Van Dijk. 1978. Bayesian estimates of equation system parameters: an application of integration by Monte Carlo. *Econometrica: Journal of the Econometric Society* (1978), 1–19.

[32] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2871–2883.

[33] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[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] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[36] Microsoft. 2023. Monitoring Performance By Using the Query Store. https://learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver16. Accessed: 2024-05-28.

[37] Max D Morris. 1991. Factorial sampling plans for preliminary computational experiments. *Technometrics* 33, 2 (1991), 161–174.

[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 *VLDB*, Vol. 6. 1049–1058.

[40] Sanket Purandare, Srinivas Karthik, and Jayant Haritsa. [n.d.]. Dimensionality Reduction Techniques for Robust Query Processing. ([n. d.]).

[41] Andrea Saltelli, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. 2010. Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index. *Computer physics communications* 181, 2 (2010), 259–270.

[42] Wolfgang Scheufele and Guido Moerkotte. 1997. On the complexity of generating optimal plans with cross products. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 238–248.

[43] I.M Sobol. 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Mathematics and Computers in Simulation* 55, 1 (2001), 271–280. https://doi.org/10.1016/S0378-4754(00)00270-6 The Second IMACS Seminar on Monte Carlo Methods.

[44] James C Spall. 2005. *Introduction to stochastic search and optimization: estimation, simulation, and control.* John Wiley & Sons.

[45] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. 2006. Isomer: Consistent histogram construction using query feedback. In *Proc. 22th Annu. IEEE Int. Conf. Data Eng.* 39–39.

[46] Jess Tan, Desmond Yeo, Rachael Neoh, Huey-Eng Chua, and Sourav S Bhowmick. 2022. MOCHA: a tool for visualizing impact of operator choices in query execution plans for database education. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3602–3605.

[47] 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.

[48] Kapil Vaidya, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2021. Leveraging query logs and machine learning for parametric query optimization. *Proceedings of the VLDB Endowment* 15, 3 (2021), 401–413.

[49] Haruko M Wainwright, Stefan Finsterle, Yoojin Jung, Quanlin Zhou, and Jens T Birkholzer. 2014. Making sense of global sensitivity analyses. *Computers & Geosciences* 65 (2014), 84–94.

[50] Hu Wang, Hui Li, Sourav S Bhowmick, and Baochao Xu. 2023. ARENA: Alternative Relational Query Plan Exploration for Database Education. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 107–110. https://doi.org/10.1145/3555041.3589713

[51] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743* (2020).

[52] 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 (jul 2018), 1360–1372. https://doi.org/10.14778/3236187.3236191

[53] 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*. 663–675.

[54] Peizhi Wu and Zachary G Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[55] Yi-Leh Wu, Divyakant Agrawal, and Amr El Abbadi. 2001. Applying the golden rule of sampling for query estimation. *ACM SIGMOD Record* 30, 2 (2001), 449–460.

[56] Haibo Xiu. 2024. GitHub Repository of PARQO. https://github.com/Hap-Hugh/PARQO

[57] Haibo Xiu, Pankaj K. Agarwal, and Jun Yang. 2024. (Full Version Paper) PARQO: Penalty-Aware Robust Query Optimization. https://github.com/Hap-Hugh/PARQO

[58] 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.

A SENSITIVITY ANALYSIS: MORRIS METHOD

Given a function $h : [0,1]^d \to \mathbb{R}$, the *Morris method* [37] uses a collection of K seeds from the input domain and calculates, for each seed, a derivative-based *elementary effect* for each input dimension in a small neighborhood around the seed; these elementary effects are then aggregated over all seeds to provide a sensitivity measure for each input dimension. Specifically, starting with each seed $\mathbf{x}$ (think of it as a point location in $[0,1]^d$), the method picks a random ordering ς of the d dimensions and generates a path $\mathbf{x} = \mathbf{x}^{[0]} \to \mathbf{x}^{[1]} \to \cdots \to \mathbf{x}^{[d]}$ with d steps, one for each dimension according to ς. Let $\varsigma(i)$ denote the ordinal position for dimension i in ς. The step for dimension i, corresponding to $\mathbf{x}^{[\varsigma(i)-1]} \to \mathbf{x}^{[\varsigma(i)]}$, would move the input point along dimension i by some small step size Δ_i, while keeping all other coordinates unchanged; in other words, $\mathbf{x}^{[\varsigma(i)-1]}$ and $\mathbf{x}^{[\varsigma(i)]}$ differ only in $x_i^{[\varsigma(i)-1]} + \Delta_i = x_i^{[\varsigma(i)]}$. The elementary effect of dimension i on seed $\mathbf{x}$ is calculated as $\mathsf{EE}_i(\mathbf{x}) = (h(\mathbf{x}^{[\varsigma(i)]}) - h(\mathbf{x}^{[\varsigma(i)-1]}))/\Delta_i$. Finally, from the collection of K seeds $\mathbf{x}_1, \ldots \mathbf{x}_K$, we calculate the *Morris-sensitivity* for dimension i as $\frac{1}{K}\sum_{j=1}^{K} |\mathsf{EE}_i(\mathbf{x}_j)|$.

To apply this method in our setting to understand the plan π obtained under selectivity estimates $\hat{\mathbf{s}}$, we analyze the function $h(\mathbf{s}) = \mathsf{Penalty}(\pi, \mathbf{s})$ by drawing the K seeds randomly by $f(\mathbf{s}|\hat{\mathbf{s}})$. This approach directs Morris to focus more on the more relevant regions of the selectivity space around $\hat{\mathbf{s}}$. Even though each individual elementary effect is derivative-based and local, tallying them over all paths explored by Morris intuitively paints an overall picture of variability over the relevant regions. There is a chance that Morris may still miss some critical features of $\mathsf{Penalty}(\pi, \mathbf{s})$; we set the step size ($\Delta_i$) and K large enough to mitigate this issue. Our current implementation adopts the same setting of $\Delta_i = 0.05 \times \hat{s}_i$ for all dimensions, but an interesting alternative worth investigating as future work would be to set step size for each dimension according to the marginal error distribution for that dimension.

We denote the *Morris-sensitivity* for dimension i as $\xi_i^{\mathsf{morris}}(\pi, \hat{\mathbf{s}})$. Overall, this analysis uses K seeds, each requiring evaluating $\mathsf{Penalty}$ $d+1$ times. Each invocation of $\mathsf{Penalty}(\pi, \mathbf{s})$ requires calling Opt to obtain the optimal plan (and its cost) at $\mathbf{s}$, calling Cost to re-cost π at $\mathbf{s}$. Hence, the total cost of Morris is $O(Kd)$ Opt and Cost calls. We show practical K values to reach convergence in Section 6.

B ADDITIONAL RESULTS OF VERIFYING ROBUSTNESS

Figure 8 summarizes the results when using “Movie” as the base instance[11] (additional results lead to similar conclusions and can be found in [57]). PARQO does well for Q2 and 17, even outperforming the instance-optimized PostgreSQL plans across instances. For Q26 and 15, PARQO is slower than instance-optimized PostgreSQL plans in most cases, by not by much. In comparison, the base PostgreSQL plan regresses much further in most cases; Q15 on the “Video Movie” instance is particularly illustrative of PARQO’s robustness.

Besides using “Movie” as the base instance, which has the largest average number of rows among the instances partitioned by category, we additionally regard "Video Game" as the base instance, representing the smallest one with the smallest average number of rows. The other instances except “Video Game” serve as testing instances. From the results shown in Figure 9, we draw similar conclusions: for Q2 and Q17, PARQO consistently outperforms both instance-optimized PostgreSQL plans and base PostgreSQL plans across instances. For Q15 and Q26, although not as effective as instance-optimized PostgreSQL plans, PARQO ’s plan successfully avoids significant regressions.

[11] Q26 includes a predicate “*kt.kind = ‘movie’*” on the *kind_type* table. Under category-slicing, we modify it to “*kt.kind IS NOT NULL*” to avoid returning empty results for some instances, which would not be useful.

C EXPERIMENTS ON DSB AND STATS

DSB. DSB [12] is a recently published industrial benchmark, positioned as an enhanced version of TPC-DS [39]. It incorporates more complex distributions and correlations, both for single columns and multi-dimensional data, within a table and across tables. As in [54], we focus on the 15 SPJ query templates, use a scale factor of 2 to populate the data, and apply the default settings to generate the query workload. For each query template, we generate 100 queries as the profiling workload and use a different random seed to create the workload for evaluation.

STATS. The data for STATS(-CEB) [20] is sourced from the Stack Exchange dataset. Its query workload includes 146 handpicked queries. Since there are no explicitly defined templates, we first randomly select 26 queries as the evaluation set, with the constraint that these query templates range from simple 3-table joins to more complex 7-table joins to ensure generalizability. We extract 22 unique query templates from the evaluation workload. We then use the remaining 120 queries as the profiling workload, constructing error profiles using the same method as for JOB, as discussed in Section 6.

First, to compare the robust plan selected by PARQO with other approaches for a single query, we use the same hyper-parameters introduced in Section 6. Figure 11 shows the results for DSB and STATS. For queries in DSB, PARQO-Sobol outperforms PostgreSQL in 8 out of 15 queries with an overall speedup of 3.43×. For the remaining 7 queries (omitted here), all methods select plans with nearly same runtime performance compared with PostgreSQL. Similar to our observation in JOB, PARQO-Sobol outperforms both WBM and PARQO-Morris in most cases. The overall speedup of PARQO-Sobol across all DSB queries is 2.01×. Notably, for complex queries such as D100, 101, and 102, which contain non-PKFK many-to-many and non-equi joins, PARQO-Sobol performs well, demonstrating its ability to discover a more robust plan for complex selectivity uncertainties by successfully capturing and penalizing them.

We note similar behavior in the STATS benchmark. PARQO-Sobol accelerates 10 out of 26 queries with an average of 1.36× speedup than PostgreSQL. Since the execution time per query ranges from 10ms to 20 minutes, here we highlight the speedup achieved for each individual query as shown on the bottom of Figure 11. Notably, for S56, PostgreSQL and WBM plans take more than 5,500 ms, whereas PARQO-Sobol’s plan only takes 13 ms, resulting in a 425.7× speedup over PostgreSQL and WBM. PARQO-Sobol shows regression on only one query, S120, with a ↓ 1.87×, however we can still demonstrate the benefits later in the PQO experiments. PostgreSQL takes nearly 15.9 minutes to execute the entire testing query set, while our method completes it in 11.7 minutes, achieving

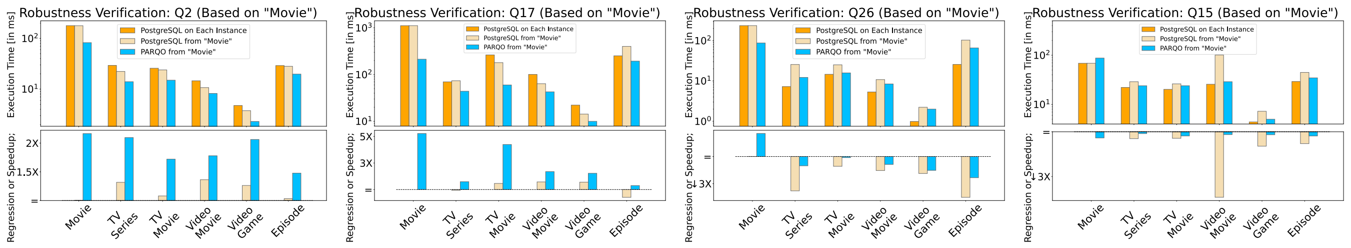


Figure 8: Actual execution times of JOB queries on multiple instances by category-slicing IMDB. "Movie" is the base instance.

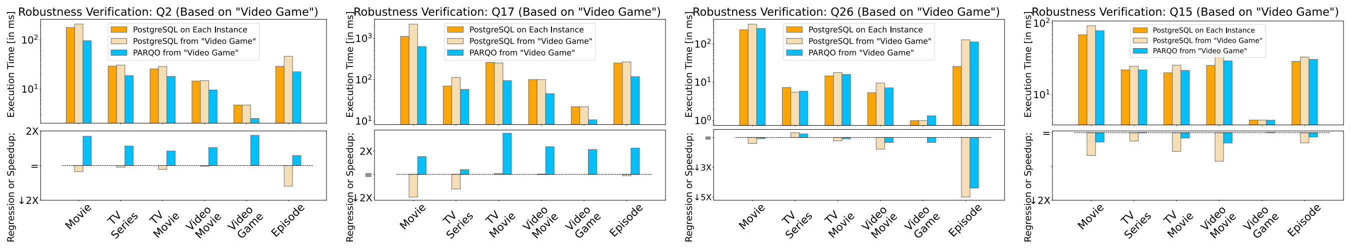


Figure 9: Actual execution times of JOB queries on multiple instances by category-slicing IMDB. "Video Game" is the base instance.

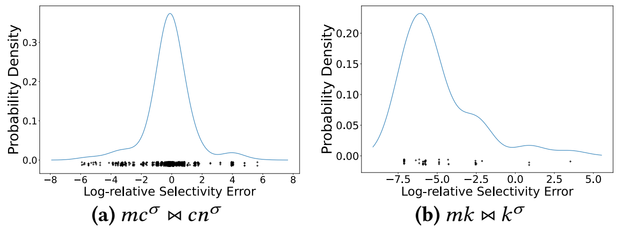


Figure 10: Examples error distributions for querylets in JOB, construct based on the profiling workload, which includes 80 instances from JOB. The dots in the plot represent the collected log-relative selectivity errors.

an overall 1.36× speedup. Again, since we are calculating the average speedup as mentioned in section 6, this number is contributed more by those slower queries.

We use the same experimental setup for PQO as described in Section 6 and present the execution times in Figure 12. The other performance measurements can be found in 1. In DSB, across all query instances, PARQO achieves a 0.68-hour improvement compared with PostgreSQL, which takes 1.23 hours running all queries, resulting in a 2.24× speedup. We also observe that the average reuse fraction in DSB is much higher than in the other benchmarks. After optimized one single query, PARQO can benefit 93% of new queries on average. This is because DSB defines the workload distribution for each query template, leading to more *similar* queries generated from the same distribution. In contrast, most predicates in JOB and STATS are distinct and divergent. Therefore, the selectivities of some anchors differ from the majority from the evaulation workload. For STATS, the average reuse fraction is 43%[12], and the execution time saved across all 22,000 query instances is 11.47 hours, which is an 1.14× speedup compared with PostgreSQL's 91.6 hours. [13] For template S120, despite previous regression in RQO settings, PARQO achieves a 2.02× speedup in 52% of generated new instances. Additionally, for templates S26, 24, 28, and 135, although the robust plans did not improve the single query performance as in Figure11, PARQO achieves significant performance gains when applied to the query templates. The optimization time also shows a speedup for these two benchmark. PARQO achieves 0.09-hour and 0.01-hour improvements in the query optimization time for DSB and STATS respectively. Combining the improvements made by PARQOin execution and optimization with the up-front overhead of PARQO, which is 0.99 hours for DSB and 0.74 hours for STATS, the benefit becomes evident.

[12] Since there can be at most two queries per template in STATS, in Figure 12, we only present results using the query with the higher reuse fraction as the anchor. But this reported average reuse fraction is calculated by the rates of triggering reuse for all anchors.

[13] This execution time does not include those long-running queries where both the robust plan and PostgreSQL's plan exceeded the timeout threshold, set at 20 minutes in our experiments. When PostgreSQL's plan ran over 20 minutes while the robust plan finished within 20 minutes, we simply recorded PostgreSQL's latency as 20 minutes. We did not observe any cases where PostgreSQL's plan finished within 20 minutes but PARQO's plan did not.

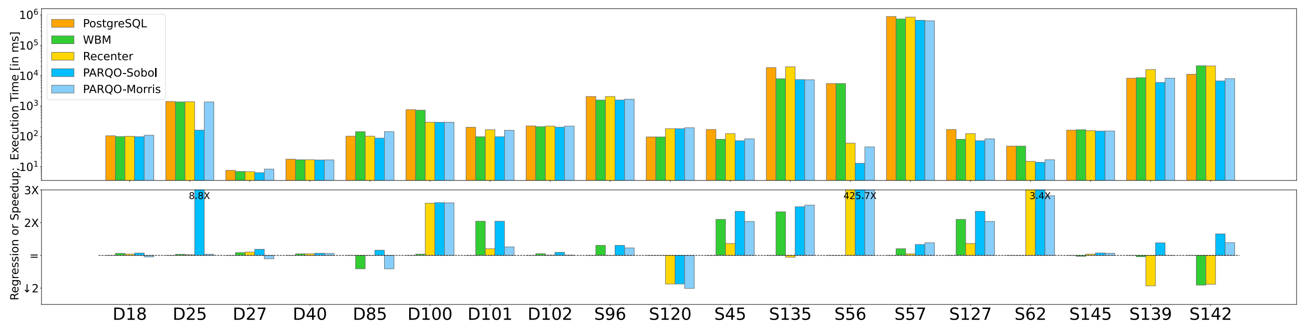


Figure 11: Actual execution times for different plans selected by various methods on DSB and STATS. Queries for which they perform similarly (7 out of 15 for DSB, and 15 out of 26 for STATS) are omitted.

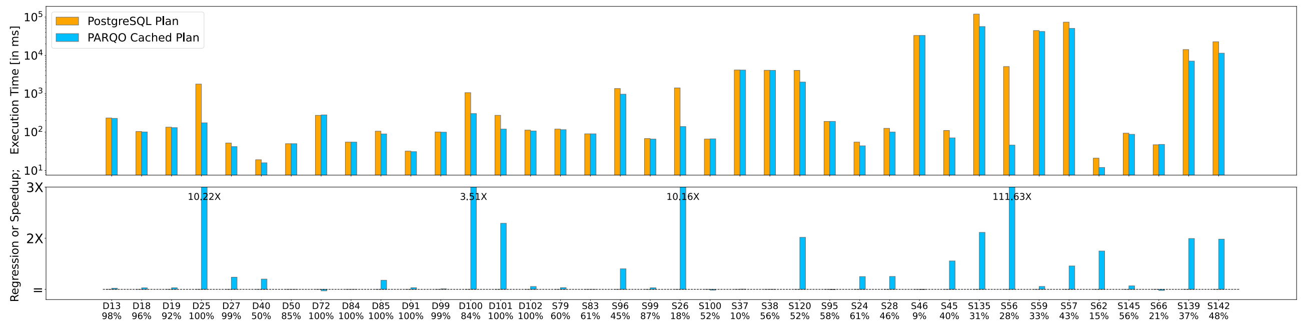


Figure 12: PQO results for DSB and STATS benchmarks. The horizontal axis displays the query template ID and the reuse fraction of the template.