Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees

Anshuman Dutt
Microsoft Research
andut@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Parametric query optimization (PQO) deals with the problem of finding and reusing a relatively small number of plans that can achieve good plan quality across multiple instances of a parameterized query. An ideal solution to PQO would process query instances *online* and ensure (a) tight, bounded cost sub-optimality for each instance, (b) low optimization overheads, and (c) only a small number of plans need to be stored. Existing solutions to online PQO however, fall short on at least one of the above metrics. We propose a plan re-costing based approach that enables us to perform well on all three metrics. We empirically show the effectiveness of our technique on industry benchmark and real-world query workloads with our modified version of the Microsoft SQL Server query optimizer.

Keywords

Parameterized Queries; Online; Workload; Cost sub-optimality

1. INTRODUCTION

Applications often interact with relational database systems through parameterized queries, where the same SQL statement is executed repeatedly with different parameter instantiations. One approach for processing parameterized queries is to optimize each query instance, thereby generating the best plan for that instance (referred to as Optimize-Always). However, the drawback of this approach is that it can incur significant optimizer overheads, particularly for frequently executing or relatively inexpensive queries. Another simple approach, that is commonly used in today's commercial database systems [22, 25], is to optimize the query for only one instance (e.g. the first query instance or an application specified instance), and reuse the resulting plan for all other instances (referred to as Optimize-Once). While the latter approach greatly reduces optimization overheads, the chosen plan may be arbitrarily sub-optimal for other query instances. Furthermore there is no way to quantify the sub-optimality resulting from Optimize-Once.

Parametric query optimization (PQO) techniques approach this problem by attempting to find a middle-ground between Optimize-Always and Optimize-Once. They store a small set of carefully chosen execution plans for a parameterized query rather than only one as in Optimize-Once. When a new query instance arrives they judiciously select one of these plans to use such that the cost of the selected plan is not much worse when compared to the cost of the plan if that query instance had been optimized (as in Optimize-Always). *Online* techniques for PQO such as [4, 2, 17] make decisions *progressively* – as each new query instance arrives – on which stored plan to use for that instance (or to optimize the instance), and whether to update the set of plans stored.

The effectiveness of any online solution to PQO can be measured using three metrics: (a) *cost sub-optimality*: cost of the selected plan relative to the cost of the optimal plan for each query instance, (b) *optimization overheads*: fraction of query instances that are optimized, (c) *number of plans stored*. An ideal online PQO solution would guarantee a tight *bound* on cost sub-optimality, optimize only a small fraction of query instances, and store only a few plans.

Existing online approaches to PQO fall short on one or more of the above three metrics. Specifically, the only prior online approach that guarantees bounded cost sub-optimality is PCM [4]. To provide this guarantee, it assumes that the cost of a plan increases monotonically with selectivity. The drawbacks of the PCM technique are that it optimizes a large fraction of query instances and requires a large number of plans to be stored. In contrast, heuristic approaches to online PQO such as [4, 2, 17] are successful in significantly lowering the optimization overhead. However, they are susceptible to incur unbounded cost sub-optimality, and require a large number of plans to be stored.

We propose a new technique for online PQO that, given a bound on cost sub-optimality that can be tolerated, effectively addresses all three metrics above, based on the following key ideas.

Selectivity check: Our first contribution is a check that can determine if the optimal plan for a previously optimized instance q_a can also be used for a new instance q_b, while *guaranteeing* that its cost sub-optimality is within the specified bound. This check is very efficient as it only requires comparing the selectivities of parameterized predicates of q_a with the corresponding selectivities of q_b. Its soundness is based on a conservative assumption on how rapidly the cost of a plan changes with selectivity. We will explain why this assumption is realistic for most relational operators. If the

SIGMOD '17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064040

selectivity check is successful, we can reuse a stored plan for q_b and thereby reduce optimization overheads (since we avoid an optimizer call).

Cost check: For query instances where the selectivity check fails, we invoke a cost check that relies on a new *Recost* API that we have implemented in the database engine. We use this API to compute the cost of using the optimal plan associated with an already optimized instance q_a, for a new query instance q_b. Using this cost, and the selectivities of q_a and q_b, the cost check determines whether the optimal plan for q_a can be used for q_b while still guaranteeing the same bound on plan cost sub-optimality. Although the *Recost* API is more expensive than a selectivity check, it is much faster than the traditional optimizer call (up to two orders of magnitude in our implementation). Hence, if the cost check passes, we still achieve significant reduction in optimization overheads. In practice, many query instances that fail the selectivity check typically pass the cost check, and can therefore still use a stored plan with bounded cost sub-optimality.

Redundancy check: This check applies when both the above checks fail. In this case we need to optimize the new query instance q_b. If this optimization results in a new plan, we first check whether it is *redundant* with respect to the existing stored plans, i.e., whether one of the stored plans could ensure bounded cost sub-optimality for q_b. Otherwise, this plan is added to the set of stored plans. Thus, only non-redundant plans are stored, thereby reducing the memory required. This check also leverages the *Recost* API.

We refer to our technique as SCR for its characteristics of exploiting three checks: **S**electivity check, **C**ost check and **R**edundancy check. Finally, we note that our technique can also guarantee an upper limit on number of stored plans without compromising guarantees on cost sub-optimality. Although limiting the number of stored plans increases optimization overheads, we empirically observe that this increase is not significant for practical settings.

We have implemented our techniques in the Microsoft SQL Server 2016 database engine. Our experiments are conducted on TPC-DS and TPC-H industry benchmark queries as well as two real-world workloads. We observe that on each of the three metrics, the SCR technique performs *similar or significantly better* than the best existing techniques as discussed in Section 7. The following are some of the empirical conclusions from our evaluation:

- **Cost sub-optimality:** In terms of average cost sub-optimality across instances of a given query, the performance of SCR is typically much better than PCM, while all heuristic techniques perform much worse. For instance, when configured to guarantee a maximum sub-optimality of 2, SCR achieves 95th percentile value of 1.22 as compared to 1.92 of PCM, and > 6 for even the best heuristic technique. For lower values of the bound, e.g. 1.1, the performance for SCR at 95th percentile is quite close at 1.09.
- **Optimizer overheads:** For optimizer overheads, SCR is significantly better than PCM and comparable to heuristic techniques. At the 95th percentile, SCR incurs optimizer calls for 13.9% of the query instances, which is comparable to 10.9% for the best heuristic technique – the average is much better at 3.7% and 3.2%, respectively. In contrast, for PCM, even the average overheads are >30%.
- **Number of plans:** SCR is significantly better than all previous techniques. The 95th percentile values are 15 for SCR in our experiments, 93 for the best heuristic technique and 219 for PCM.

The rest of the paper is organized as follows. We formalize the online PQO problem in Section 2. In Section 3, we discuss the limitations of existing online PQO techniques and show a glimpse of SCR performance, using an example query sequence. We present a solution overview in Section 4 and our technical results and algorithms in Sections 5 and 6. Section 7 discusses experimental results. We provide a detailed comparison of related work in Section 8 and conclude in Section 9.

2. PROBLEM DESCRIPTION

Given a parameterized query (a.k.a query template) Q, we use the term *dimensions* for the number of parameterized predicates and denote it with d. We use q_e to denote an example instance of Q and a vector $sVector_e$ for compact representation of instance q_e that captures corresponding selectivities for the parameterized predicates, e.g. $(s_1, s_2, ..., s_d)$. Further, for example query instance q_e, the optimal plan, as determined by the query optimizer, is denoted with $\mathrm{P}_{opt}(\mathrm{q}_e)$. For a given plan P and query instance q_e, the optimizer estimated cost is denoted with $\mathrm{Cost}(P, \mathrm{q}_e)$. Next, we define a workload W to be a sequence of query instances for the query template Q, i.e., $\mathrm{W} = <\mathrm{q}_1, \mathrm{q}_2, ..., >$. Finally, let $\mathcal{P}$ denote the set of all plans that are optimal for at least one query instance in W, and n denote the cardinality of $\mathcal{P}$.

Online PQO setting. An online PQO technique needs to decide which plan to use for each incoming query instance in an online fashion. It does so by storing a set of plans in a *plan cache* and then deciding whether to pick one of the cached plans or make an optimizer call. This is usually done by associating each plan with a *inference region*, i.e., selectivity region where it can be reused. At any intermediate state, we denote the query instance currently being processed with q_c and the set of previously processed instances with W_{past}.

2.1 Metrics

We consider following set of metrics for comparison and performance evaluation of online PQO techniques,

1. **Cost sub-optimality.** For query instance q_e, let $\mathrm{P}(\mathrm{q}_e)$ denote the plan used by the online technique. Then sub-optimality for q_e is defined as

$$\mathrm{SO}(\mathrm{q}_e) = \left(\frac{\mathrm{Cost}(\mathrm{P}(\mathrm{q}_e),\, \mathrm{q}_e)}{\mathrm{Cost}(\mathrm{P}_{opt}(\mathrm{q}_e),\, \mathrm{q}_e)} \right)$$

Any plan P, for which $1 < SO(\mathrm{q}_e) \leq \lambda$, is termed as λ-*optimal plan* for q_e. Further, we measure the worst case sub-optimality across the workload sequence using MSO defined as follows:

$$\mathsf{MSO} = \max_{\mathrm{q}_e \in W} (SO(\mathrm{q}_e))$$

Since MSO captures only worst case performance and does not reflect whether such cases are frequent or rare, we also measure aggregate performance of the technique over the given workload using TotalCostRatio defined as:

$$\mathsf{TotalCostRatio} = \frac{\sum_{q_e \in W} \text{Cost}(\text{P}(q_e),\, q_e)}{\sum_{q_e \in W} \text{Cost}(\text{P}_{opt}(q_e),\, q_e)}$$

Observe that TotalCostRatio falls in the range $[1, \mathsf{MSO}]$, and lower values indicate better performance.

2. **Optimization overheads.** We use numOpt to denote the number of optimizer calls made across the workload. We also consider the average overhead for picking a plan from the cache, whenever the optimizer is not invoked.
3. **Number of plans cached.** We denote the maximum number of plans stored in the plan cache with numPlans. We also consider other bookkeeping memory overheads required to maintain a set of plans and support the decision of plan picking for a new query instance.

For the purpose of evaluation, we generate fixed length workloads, such that $|W| = m$, in which case numPlans $\leq$ numOpt $\leq m$. Also, we use only optimizer estimated costs in our evaluation since the execution times may suffer high variability in dynamic execution environments (system load, concurrency, available memory etc.), which is an orthogonal problem.

Optimize-Always and Optimize-Once are alternative techniques that presume extreme settings, i.e., numPlans $=0$ and numOpt $=1$, respectively but may be highly wasteful or sub-optimal otherwise. The existing online PQO techniques try to minimize numOpt, either while guaranteeing an upper bound on the MSO (PCM [4]), or with no bound on MSO at all (Ellipse [4], Density [2], Ranges [17][1]). Also, none of the existing techniques supports a limit on numPlans, an important metric in practice.

2.2 Problem definition

In this work, we focus on designing an online PQO technique that aims to minimize optimizer overheads while ensuring that every processed query instance q satisfies $SO(q) \leq \lambda$, where $\lambda \geq 1$ is an input parameter.[2]

We also study a variant of the above problem with an additional constraint that the number of plans stored in the plan cache cannot exceed $k \geq 1$.

3. EXISTING TECHNIQUES

Previous techniques for online PQO have been successful to various degrees in terms of reducing optimizer calls. We first provide a summary of their plan inferencing criteria in Table 1. Then, we discuss a few shortcomings of these techniques with regard to the three metrics of Section 2.1 using the example workload shown in Figure 1. This workload consists of 13 query instances which are marked in the figure as $i[j]$, where i represents i^{th} query instance q_i and j denotes j^{th} plan P_j which is optimal plan for q_i, i.e., $P_{opt}(q_i)$. The comparative performance of all existing techniques including a glimpse of proposed technique SCR, is visually captured in Figure 1 (except for Density which is self explanatory).

PCM	The current query instance q_c lies in a rectangular region created by a *pair* of previously optimized query instances such that one dominates the other in the selectivity space and their optimal costs are within λ-factor
Ellipse	q_c lies in an elliptical neighborhood of a pair of previously optimized instances with the same optimal plan
Ranges	q_c lies in a rectangular neighborhood enclosed by a minimum bounding rectangle for all previously optimized instances with the same optimal plan
Density	q_c has sufficient number of instances with the same optimal plan choice in a circular neighborhood

Table 1: Criteria that must be satisfied by existing online techniques to skip optimizer call for a new query instance q_c

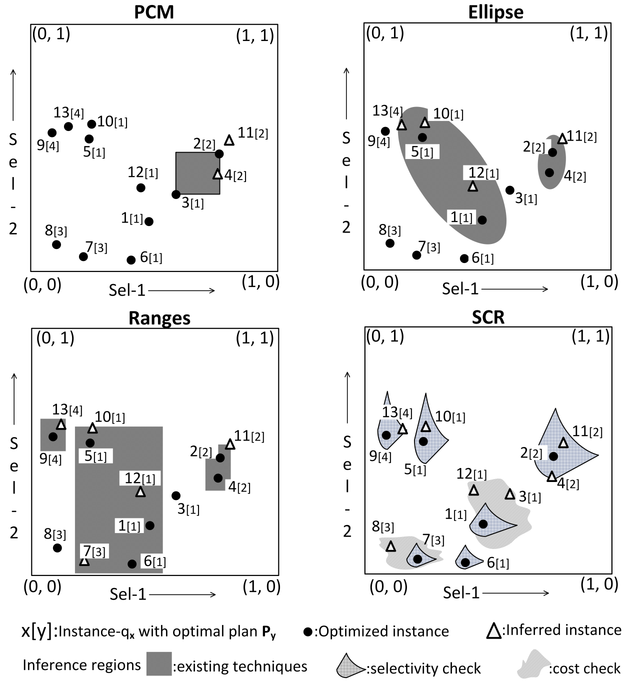

Figure 1: Inference regions for various techniques while processing example workload for a 2-dimensional query

- **Limitations affecting optimizer overhead:** For all existing techniques, the reuse of a stored plan P is possible only after workload has provided two or more instances that satisfy certain pre-conditions (Table 1). Such restrictions may prevent reuse of a suitable plan that already exists in plan cache. For instance, PCM or Density cannot make any inference using plan P_1 even after it was known to be optimal for many instances (q_1, q_3, q_5, etc.).
- **Limitations affecting cost sub-optimality:** All plan inference techniques that use selectivity-based *neighborhoods* can seriously compromise MSO, since they do not take into account the cost behavior of the inferred plan or optimal plan – as explained in Appendix A. In the example workload, while the elliptical region around q_1 and q_5 provides optimal plans for q_{10} and q_{12}, it also results in a sub-optimal plan (P_1) for q_{13}. Similarly for Ranges, the rectangular neighborhood for plan P_1 containing instances q_1, q_5 and q_6 provides an optimal plan for q_{10} and q_{12} but also leads to selection of sub-optimal plan (P_1) for q_7. Further, the absence of any mechanism to detect sub-optimality of inferred plan choices, may lead to repeated mistakes in plan inferences and hence high values of TotalCostRatio. For example, any instance close to q_7 would be

[1] We refer to this technique as Ranges as it stores a plan with a selectivity range around the corresponding optimized instance and reuses the plan whenever any new instance falls within the selectivity range.

[2] This goal is identical to PCM [4].

assigned plan P_1 by Ranges. Finally, Density would also wrongly assign P_1 to q_{13}.

- **Limitations affecting number of plans required:** Existing techniques mostly use trivial policies for managing the plan cache, e.g. store every new plan and never drop a plan. As a result, they are prone to storing a large number of plans. For example, the techniques in [4] were found to store hundreds of plans (for d $\geq$ 3). Note that, the example workload does not highlight this limitation of existing techniques (all technique store 4 plans).

Overall, none of the existing techniques provide the ability to control cost sub-optimality *and* number of plans while also achieving significant reduction in optimizer overheads.

Performance of SCR: In comparison to existing techniques, SCR invokes optimizer calls for only 6 instances while PCM required 12 and best heuristic technique required 8 optimizer calls. For every optimized instance q_i in Figure 1, the surrounding region (smaller region with dark boundary) represents the set of all possible instances that can satisfy the selectivity check with respect to q_i and is termed as *inference region due to selectivity check.*[3] Similarly, for some instance (q_1 and q_7) we highlight the *inference region due to cost check* (shaded without boundary). For q_4 and q_{11}, it infers plan P_2 since they satisfy the selectivity check for q_2 and hence avoids optimizer calls. The plans for q_{10} and q_{13} are also inferred due to the selectivity check. Further, plan P_1 is chosen for q_3 because the cost check with q_1 succeeds even though selectivity check fails. Likewise, plans for q_8 and q_{12} are inferred due to the cost check. In short, SCR saves significantly more optimizer calls than PCM and does not pick sub-optimal plans like the heuristic techniques.

4. SOLUTION OVERVIEW

We now present an overview of the architecture and modules in our solution to the online PQO problem. We also discuss the requirements that a database engine needs to satisfy for effective implementation of our solution. We follow the same basic framework as first proposed in [4], but with important extensions as described next.

4.1 Architecture

The architecture of our solution is shown in Figure 2. At any intermediate stage, the plan cache stores a set of plans and extra information to capture their inference regions – we denote the set of plans with $\mathcal{P}_\mathcal{C}$.

When a new query instance q_c arrives, the first step is to compute its selectivity vector ($sVector_c$) which is then passed to the getPlan module. This modules interacts with the plan cache and makes the decision to *optimize or not* for q_c. Let P(q_c) denotes the plan chosen for q_c. getPlan tries to find the plan P(q_c) from the plan cache failing which an optimizer call is made to obtain the optimal plan for q_c. Note that getPlan occurs on the critical path of query execution, and must therefore be efficient.

If q_c is optimized, then we get the plan $P_{opt}(q_c)$, which is then fed to the manageCache module. This module makes the following decisions regarding the state of plan cache: (a) If $P_{opt}(q_c)$ is found to already exist in plan cache, then how to modify its inference region. (b) If $P_{opt}(q_c)$ is not found in the plan cache, then whether to *store or not.* (c) For each of the existing plans in plan cache, *drop or not.* Since manageCache does not need to occur on the critical path of query execution, it can be implemented asynchronously on a background thread.

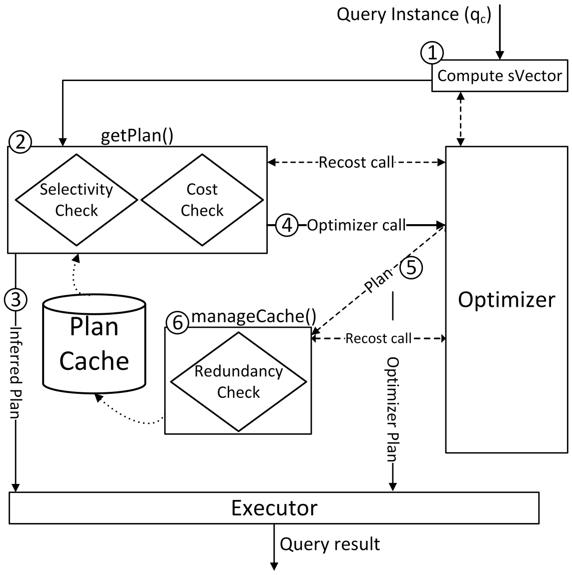

Figure 2: SCR architecture

4.2 Requirements from the database engine

To support implementation of proposed solution technique, the database engine needs to support following two APIs in addition to the traditional optimizer call.

1. **Compute selectivity vector:** Given a query instance q_c, efficiently compute and return $sVector_c$.
2. **Recost plan:** Given a plan P and a query instance q_c, efficiently compute and return Cost(P, q_c).

Note that the selectivity vector is a generic requirement across all existing online PQO techniques [17, 4, 2], and even *Recost* has been used in previous offline PQO techniques [14, 9]. It is however critical that in our case that the *Recost* API is much more efficient compared to an optimizer call. An implementation outline of both routines for Microsoft SQL Server can be found in the Appendix B.

4.3 Outline for getPlan and manageCache

getPlan *module.* Our technique utilizes a two step check to make the decision *optimize or not* for new instance q_c. The first check, called the Selectivity check, takes two $sVectors$ for q_e, q_c and the sub-optimality requirement λ as input and returns true only if plan $P_{opt}(q_e)$ can be inferred to be λ-optimal at q_c, purely on the basis of $sVectors$ and an assumption on plan cost functions – the details of this check are provided in Sections 5.3 and 6.2. Only when this check fails, the Cost check is invoked. The *Recost* feature is used to compute the cost of one or more stored plans for query instance q_c to check whether one of these plans can be inferred to be λ-optimal for q_c. If both checks fail to identify a plan, an optimization call is made for q_c.

[3] Note that, the regions due to selectivity check have a specific geometrical shape, which is explained in Section 5.

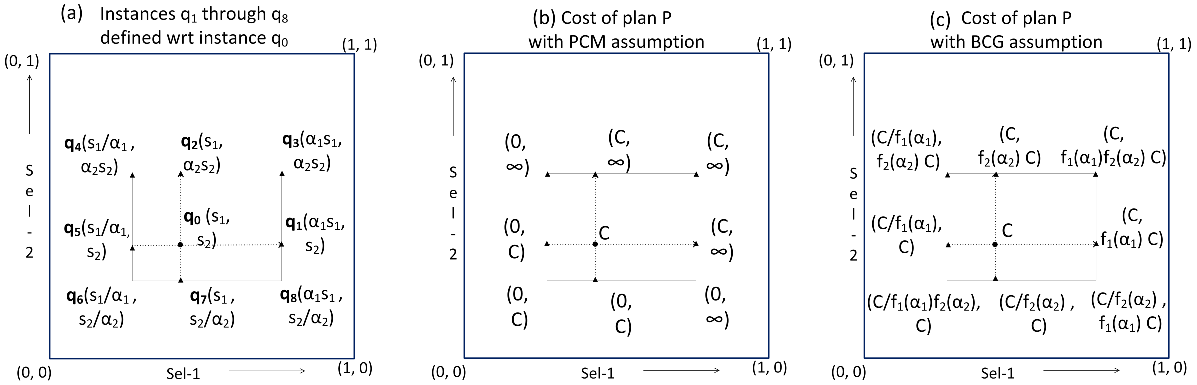

Figure 3: Neighboring instances of q_0 with assumption specific bounds on cost of P in terms of C (cost of P at q_0)

manageCache module. If the optimization of q_c results in a plan that already exists in the plan cache, we modify its inference region to include q_c. Even if the plan is not present in the plan cache, it is possible that one of the existing plans is λ-optimal for q_c as both checks in getPlan are conservative. In such a case, we consider the new plan to be *redundant* and discard it from the cache. Instead, we modify the inference region of the existing λ-optimal plan. Otherwise, we add the new plan $P_{opt}(q_c)$ to the cache.

Discussion We note that any online PQO technique would benefit a query workload only if the query optimization overheads are not trivial compared to the execution time [11]. For instance, a query for which the average execution time is ≈ 30 sec against average optimization time of only 50 msec, would not get any noticeable benefit. In contrast, a different query that has an average optimization time of 350 msec against average execution time 200 msec, can achieve significant benefits using online PQO.

5. DESIGN OF getPlan MODULE

We first describe our approach to decide if a plan that already exists in the plan cache can be reused for a new query instance while guaranteeing λ-optimality. This forms the basis of inferring *λ-optimality regions* around each optimized instance and leads to the construction of selectivity and cost checks of the getPlan module. The λ-optimality guarantee is based on the assumption of bounded cost growth for plans, whose validity we examine for typical physical operators in a relational database system.

5.1 Plan reuse approach

When a query instance q_e is optimized, we get plan $P_{opt}(q_e)$. The sub-optimality of $P_{opt}(q_e)$ when used for another query instance q_c is given by:

$$\text{SubOpt}(P_{opt}(q_e),\, q_c) = \frac{\text{Cost}(P_{opt}(q_e),\, q_c)}{\text{Cost}(P_{opt}(q_c),\, q_c)} \qquad (1)$$

The challenge is that computing exact value of sub-optimality is not possible without making optimizer call for q_c, which would defeat the very purpose of plan reuse.

Our approach is to infer an *upper bound* on the value of $\text{SubOpt}(P_{opt}(q_e),\, q_c)$ by utilizing an upper bound on the cost of the numerator and a lower bound on the cost of the denominator. We show in Section 5.2, that such cost bounds can be computed by using the selectivity ratio's between q_e and q_c under bounded cost growth assumption on plan cost functions.

We also exploit the fact that the above sub-optimality bound can be further tightened if we can obtain the *exact* value of numerator. Interestingly, computing the numerator requires only *re-costing* of plan $P_{opt}(q_e)$ for query instance q_c, which, in our implementation is up to two order of magnitude faster than an optimizer call.

Finally, the above plan reuse approach is not restricted only to the *optimal plan* for q_e, and can be utilized for any generic plan P as long as its sub-optimality at q_e is known. We exploit this property (in manageCache) to retain only a subset of optimal plans without violating λ-optimality.

5.2 Bounded cost growth (BCG) assumption

Our assumption on plan cost functions can be viewed as an extension of the *Plan Cost Monotonicity* (PCM) assumption that has been used in past work [4].[4]

Consider a plan P whose cost at query instance $q_0 = (s_1, s_2)$ of a 2-dimensional query is C. Also consider query instances $q_1 = (\alpha_1 s_1, s_2)$ and $q_2 = (s_1, \alpha_2 s_2)$, with $\alpha_1, \alpha_2 > 1$. A visual representation of such instances is provided in Figure 3a. Note that, we use 2-dimensional query template for the sake of presentation and the arguments can be generalized for n-dimensions in a straightforward manner.

PCM assumption: This assumption intuitively means that the cost of a plan increases with increase in each individual selectivity, i.e., (a) $\text{Cost}(P,\, q_1) > C$, (b) $\text{Cost}(P,\, q_2) > C$. Thus, it provides a *lower* bound on cost of P at q_1 and q_2.

BCG assumption: This is an extension to PCM assumption where it is also assumed that for every individual selectivity dimension, if the selectivity value increases by a factor α, the resulting increase in cost of P is *upper* bounded by a known function of α. That is,
(a) $C < \text{Cost}(P,\, q_1) < f_1(\alpha_1)C$
(b) $C < \text{Cost}(P,\, q_2) < f_2(\alpha_2)C$

[4] Also, similar to past work on PQO problem, we continue to assume the following: (a) plan cost functions are smooth, (b) other factors that may influence plan costs, e.g. join-selectivities, main memory etc., remain the same across all query instances and (c) selectivity independence between base predicates.

where both f_1 and f_2 are increasing functions defined over the domain $(1, \infty)$. Similar assumption has also been used previously in [16, 20] in different contexts. We present arguments for our choice of f_i's in Section 5.4.

5.2.1 Cost implications for neighboring instances

We analyse the cost implications of PCM and BCG assumptions on query instances q$_3$ through q$_8$, whose selectivity vectors are shown in Figure 3a.

For $\mathrm{q}_3 = (\alpha_1 s_1, \alpha_2 s_2)$, PCM assumption implies cost lower bound, i.e., $\mathrm{Cost}(P,\ \mathrm{q}_3) > C$, while successive application of BCG assumption implies both lower and upper bounds on cost of P, i.e., $C < \mathrm{Cost}(P,\ \mathrm{q}_3) < f_1(\alpha_1) f_2(\alpha_2) C$.

For $\mathrm{q}_5 = (\frac{s_1}{\alpha_1}, s_2)$, PCM provides an *upper* bound on the cost of P by considering q$_5$ as the reference for q$_0$, that is, $\mathrm{Cost}(P,\ \mathrm{q}_0) > \mathrm{Cost}(P,\ \mathrm{q}_5) \Rightarrow \mathrm{Cost}(P,\ \mathrm{q}_5) < C$. On the other hand, BCG again provides lower as well as upper bounds on cost:

$$\Rightarrow \frac{C}{f_1(\alpha_1)} < \mathrm{Cost}(P,\ \mathrm{q}_5) < C$$

The case of $\mathrm{q}_4 = (\frac{s_1}{\alpha_1}, \alpha_2 s_2)$ is interesting since selectivity increases in one dimension and decreases in the other. In this case, PCM *cannot* provide any bound on cost of P. In contrast, BCG still provides both lower and upper bounds on the cost of P as follows:

$$\mathrm{Cost}(P,\ \mathrm{q}_5) < \mathrm{Cost}(P,\ \mathrm{q}_4) < f_2(\alpha_2)\mathrm{Cost}(P,\ \mathrm{q}_5)$$

We can use the cost bounds for q$_5$ to get,

$$\frac{C}{f_1(\alpha_1)} < \mathrm{Cost}(P,\ \mathrm{q}_4) < f_2(\alpha_2) C$$

In a similar manner, the cost lower and upper bounds for q$_6$, q$_7$ and q$_8$ can also be inferred using BCG assumption. All the cost bounds are shown in Figure 3b for PCM and Figure 3c for BCG.

5.3 Constructing λ-optimal region

Next, we formalize the cost implications of the BCG assumption to define cost and sub-optimality bounds for plan $\mathrm{P}_{opt}(\mathrm{q}_e)$ for a generic neighboring query instance q$_c$. We use these bounds to construct λ-optimal region for $\mathrm{P}_{opt}(\mathrm{q}_e)$ around q$_e$. In this analysis, we follow the assumption $f_i(\alpha) = \alpha$ and examine its validity in the following section.

Consider two query instances q$_e$ and q$_c$, let P_e and P_c denote their optimal plan choices and $(\alpha_1,\ \alpha_2,\ ...,\ \alpha_d)$ be the vector of selectivity ratios between their $sVector$s with each $\alpha_i = \dfrac{s_i(\mathrm{q}_c)}{s_i(\mathrm{q}_e)}$. Further, let $L = \prod_{\alpha_i < 1} \frac{1}{\alpha_i}$, denote the net cost decrement factor due to selectivity ratios. Similarly, let $G = \prod_{\alpha_i > 1} \alpha_i$ denote the net cost increment factor between q$_e$ and q$_c$. The following lemma gives cost bounds for P_e at arbitrary instance q$_c$,

Lemma 1 (Cost Bounding Lemma). *Under the assumption that the bounding functions are known to be $f_i(\alpha_i) = \alpha_i$, Cost(P_e, q$_c$) satisfies the following bounds,*

$$\frac{Cost(P_e,\ q_e)}{L} < Cost(P_e,\ q_c) < G \times Cost(P_e,\ q_e)$$

Further, following theorem bounds the sub-optimality of P_e at generic instance q$_c$,

Theorem 1 (Sub-optimality Bound). *Under the assumption that the $f_i(\alpha_i) = \alpha_i$ holds for both plans P_e and P_c, SubOpt(P_e, q$_c$) $< GL$.*

Proof for the above theorem can be found in Appendix C. Note that similar results can derived for other possible bounding functions, e.g. for $f_i(\alpha) = \alpha^2$, we get the following bound: SubOpt(P_e, q$_c$) $< (GL)^2$.

Improved upper bound on SubOpt(P_e, q$_c$). Next, let R denote that multiplicative cost factor for plan P_e between q$_c$ and q$_e$, i.e. $R = \dfrac{\mathrm{Cost}(P_e,\ \mathrm{q}_c)}{\mathrm{Cost}(P_e,\ \mathrm{q}_e)}$. R can be easily computed if q$_e$ has been optimized and P_e is re-costed at q$_c$. Since we now know the exact value of numerator cost in terms of R (rather than upper bound in terms of G) in the sub-optimality expression for P_e at q$_c$, the sub-optimality upper bound tightens to RL.

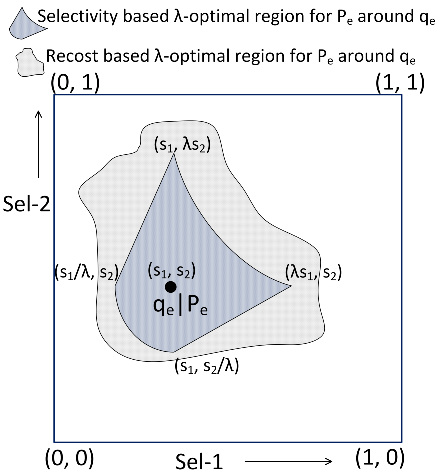

Figure 4: λ-optimal region for P_e around instance q$_e$

A sample representation of the λ-optimal region around q$_e$ is provided in Figure 4. The inner region is the selectivity based λ-optimal region that is defined by $GL \leq \lambda$. Geometrically, it is a closed region bounded by straight lines ($y = \frac{s_2 \lambda}{s_1} x$ and $y = \frac{s_2}{s_1 \lambda} x$) and hyperbolic curves ($y = \frac{s_1 s_2 \lambda}{x}$ and $y = \frac{s_1 s_2}{\lambda x}$). Interestingly, its area coverage is given by: $(\lambda - \dfrac{1}{\lambda}) \ln \lambda \times s_1 s_2$, which is an *increasing* function of λ and selectivities of q$_e$ but independent of the plan choice for q$_e$.[5] Once we know the value of R after using *Recost* for plan P_e at q$_c$, we can use $RL \leq \lambda$ to detect whether q$_c$ lies in *Recost* based λ-optimal region of P_e (outer region in Figure 4 with symbolic shape). Thus, use of *Recost* helps in identifying *extra* opportunities of plan reuse if the cost growth is *slower* than that assumed by the selectivity based inference.[6]

[5] The area of λ-optimal region remains the same even after changes to the underlying cost model as long as the cost growth bounding functions remain the same, i.e., $f_i(\alpha) = \alpha$.

[6] Note that, the true λ-optimal region of plan P_e around a given instance q$_e$ may be even bigger and our checks capture only a conservative portion. Also, the same plan P_e may be optimal at multiple query instances.

5.4 Examining validity of BCG assumption

While BCG promises to provide additional leverage compared to PCM for plan reuse with guarantees, the question of finding valid bounding functions, that is f_i, remains open. To this end, we first discuss bounding functions for standard physical operators in relational databases.

First, the cost of a Scan operator increase linearly with input selectivity. Therefore, if input selectivity is increased by a factor α, we expect the cost to increase at most by a factor α. Next, the cost of a Nested Loops Join operator increases as a function of $s_1 s_2$ where s_1 and s_2 represents the selectivity of the two inputs. We therefore expect that, if any *one* of the input's selectivity increases by a factor α, then the cost of the operator can go up by at most a factor α. If the selectivity increases by a factor α for *both* inputs, the cost can go up by at most a factor of α^2. Thus, assuming $f_i(\alpha) = \alpha$ as the bounding function for *each* selectivity input would suffice for these operators. Observe that for a Hash Join operator, the cost increases as a function of $s_1 + s_2$ (grows slowly compared to Nested Loops Join). Hence, the above bounding function suffices for Hash Join, although the upper bound thus achieved has different degree of tightness for Nested Loops Join and Hash Join. In general, for a series of n binary-joins using the above operators, if selectivity of *each* input selectivity increases by a factor α, then total cost may increase by a factor of at most α^n. Also, we can expect that for any operator that scans each of its input only once, e.g. union, intersection, and even scalar user-defined functions, using $f_i(\alpha) = \alpha$ should be sufficient.

For operators whose implementation may require sorting of input, for example, Sort Merge Join, Sort, sorting-based Group By, the operator cost may vary as $s_1 \log s_1$ and hence its cost may increase super-linearly with input selectivity. Even for such operators, it is possible to choose a polynomial bounding function (of the form α^n with n>1) by using the inequality such as $\ln x \leq \frac{x-1}{\sqrt{x}}$ with $1 \leq x \leq \infty$ [19]. For arbitrary physical operators that fall outside the above set of standard relational operators, our framework can handle such an operator if an appropriate polynomial bounding function can be defined.

Finally, we note that, in practice, the cost models of modern query optimizers can be quite complex. The cost functions may be piece-wise linear, or may even contain discontinuities, e.g. the optimizer might model the transition point from memory based sort to disk based stored due to limited memory. Also, there may be other factors that impact the plan cost, e.g. join selectivities. Despite this, we observed during our extensive experiments that using $f_i(\alpha_i) = \alpha_i$ as bounding functions faces only rare violations. The degree and impact of such violations on MSO and TotalCostRatio metrics are reported in Section 7.2.

6. IMPLEMENTATION DETAILS

6.1 Plan cache data structure

At any intermediate stage when W$_{past}$ has been processed, let W^{opt} denotes the set of instances optimized till now. The plan cache contains a plan list and an instance list. The plan list contains a subset of plans that are optimal across W^{opt} and instance list contains a 5-tuple $I =< V, PP, C, S, U >$ for each instance q$_e$ in W^{opt} where

1. V denotes the selectivity vector for q$_e$
2. PP is a pointer to plan P(q$_e$) in plan list. It may be different from P$_{opt}$(q$_e$) as explained later in Section 6.3
3. C is the optimizer estimated optimal cost for q$_e$
4. S is the sub-optimality P(q$_e$) at q$_e$
5. U is the running count of the number of query instances for which getPlan picks plan P(q$_e$) through instance q$_e$

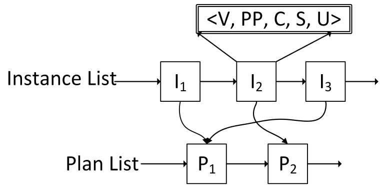

Figure 5: Example contents of Plan cache

The instance list contains one entry for each of the optimized query instances and typically many instances from instance list point to the same stored plan in plan list – a visual representation of the plan cache data structure is provided in Figure 5. In each entry in the instance list: V, C, S are required to capture inference region of plan pointed by PP and hence support getPlan, while S and U are required by manageCache to support ability to reject new plans and drop existing plans, respectively.

Overheads discussion. The instance list is a very small contributor, since: (a) we store an entry *only for* optimized query instances, which is usually a small fraction of all instances and (b) the memory required for each 5-tuple is small (≈100 bytes). In comparison, the memory overheads of plan list can be much larger since we need to store enough information to support execution as well as efficient re-costing for each stored plan. Our implementation suggests these overheads to be few hundred KBs per plan. An alternative implementation of *Recost* may help in reducing the memory overheads but may cause increase in its time overheads.

6.2 getPlan implementation

Based on results in Section 5, we now present the specific implementation for selectivity and cost check to determine whether one of the existing plans in the plan cache is λ-optimal for new instance q$_c$. A pseudocode of the entire getPlan module is given in Algorithm 1.

Selectivity check: This check determines whether q$_c$ lies in the selectivity based λ-optimal region of one of the stored instance q$_e$. For any given q$_e$ from the instance list, we first compute G and L using $sVector_c$ and V and then check whether $GL \leq \frac{\lambda}{S}$. Note that, the above check allows the possibility of P(q$_e$) to be sub-optimal with $S < \lambda$ without violating the λ-optimality guarantee of getPlan. If P(q$_e$) is indeed the same as P$_{opt}$(q$_e$) then the check simplifies to $GL \leq \lambda$.

Cost check: This check determines whether q$_c$ lies in the re-cost based λ-optimal region around any of the stored instance q_e. For this check, we first compute L using $sVector_c$ and V. Then, we compute R by using the *Recost* feature to compute Cost(P(q$_e$), q$_c$) and dividing it by C, i.e. optimal cost for q$_e$. With the above, the cost check is $RL \leq \frac{\lambda}{S}$.

Further, whenever a stored query instance q_e succeeds the selectivity or cost check with new instance q_c, the counter U is increased to keep track of the number of query instances where the stored instance q_e helped in avoiding the optimizer call. Thus, it captures the query instance distribution which is then leveraged by the manageCache module.

```
Data: Q, $\lambda$, $q_c$, $sVector_c$
InstList $\mathcal{I}_Q$ = getList(Q);
//selectivity check ;
for instance $q_e$ in $\mathcal{I}_Q$ do
    G = computeG($sVector_c$, $V$);
    L = computeL($sVector_c$, $V$);
    if $GL \leq \frac{\lambda}{S}$ then
        $U$++ ;
        return plan($PP$);
    end
end
//cost check ;
for instance $q_e$ in $\mathcal{I}_Q$ do
    $newCost$ = doRecost(Q, plan($PP$), $q_c$);
    R = $\frac{newCost}{C}$;
    L = computeL($sVector_c$, $V$);
    if $RL \leq \frac{\lambda}{S}$ then
        $U$++ ;
        return plan($PP$);
    end
end
return getOptPlan($q_c$);
```

Algorithm 1: getPlan algorithm

Overheads discussion. The overheads of getPlan routine include: (1) $sVector$ computation, (2) traversal of instance list during selectivity and cost check, and (3) *Recost* calls during the cost check. Empirically, we found that overheads due to *Recost* calls dominate the getPlan overheads. A single invocation of *Recost* typically requires 2 to 10 milliseconds, depending on the number of parameters and the memory representation used for re-costing plans (see Appendix B). This can easily exceed the overheads of $sVector$ computation and scanning a list of few thousand instances.

For this reason, we use the following heuristic to control the number of *Recost* calls: instances with large values of GL are less likely to satisfy the cost check. To implement this, selectivity check collects potential candidates for cost-check in increasing order of their GL values and rejects all instances beyond a threshold. Also, storing instances with sub-optimality≈1 (with cost-check), will lead to increased coverage by selectivity regions and save future *Recost* calls.

Finally, if the number of instances in the list goes beyond several thousand, overheads of selectivity check may become comparable to that of $sVector$ computation. In such cases, the overheads can also be improved by exploiting similar idea of checking instances with smaller GL values first. This can be achieved by using a spatial index that can provide such instances without scanning the entire list. Also, there are few other alternative heuristics that may help in improving average getPlan overheads by only changing the storage of instance list: (a) decreasing order of area of selectivity region (a function of V and λ), (b) decreasing order of usage counts of instances (U).

Choosing λ. To decide a suitable value of λ, we propose to use Optimize-Always for a small initial subset of query instances and observe the ratio between average optimization overheads and average execution cost. For example, a query where optimization overheads are close to 50% compared to execution cost should use smaller value of λ compared to another query for which optimization overheads dominate the execution cost. The mechanism to keep track of this information is already present in many database engines.

It may also be beneficial to use larger value of λ for cheaper instances and smaller value for expensive instances of the same query template. This is because low cost regions typically have small selectivity regions and high plan density [18]. We explain in Appendix D that our framework can support such dynamic value of λ and it helps in saving optimizer overheads as well as plan cache overheads at the expense of a relatively small increase in TotalCostRatio.

6.3 manageCache implementation

Algorithm 2 gives an outline for the manageCache module.

After $P_{opt}(q_c)$ is obtained by invoking the optimizer, the first step is check whether the new plan is redundant with respect to existing plans in the cache, as described next.

Redundancy check: We iterate over all plans in the plan cache to determine the minimum cost plan and its sub-optimality for query instance q_c – let us denote them with P_{min} and S_{min}, respectively. Next, we check whether $S_{min} \leq \lambda_r$, where $\lambda_r < \lambda$ is a configurable threshold for the redundancy check.[7] This makes sure that the property of λ-optimality is maintained even while storing only a subset of encountered plans. If the above check satisfies, then we infer that $P_{opt}(q_c)$ is *redundant* with respect to plan cache. Otherwise, we add the plan to plan cache. In principle, this inclusion may also help us in discarding existing plans from cache, as discussed in Appendix F.

Next, we describe the possible actions that can be registered to be completed by manageCache in an asynchronous manner, depending on whether the plan already exists in the plan cache or not, and the result of redundancy check:

1. **Plan already exists in plan cache**: The following 5-tuple is added to instance list with a pointer to the plan $P_{opt}(q_c)$ already stored in plan list.

   $< sVector_c$, *pointer*($P_{opt}(q_c)$), Cost($P_{opt}(q_c)$, q_c), 1.0, $1 >$
2. **New plan that failed redundancy check**: First $P_{opt}(q_c)$ is added to plan list and then the above 5-tuple is added to instance list.
3. **New plan that passed redundancy check**: The new plan is discarded but the instance list still gets following 5-tuple: $< sVector_c$, P_{min}, Cost($P_{opt}(q_c)$, q_c), $S_{min}, 1 >$

6.3.1 Enforcing budget on numPlans

Our solution also supports dropping existing plans from the plan cache. This may be required in case a plan cache budget of k plans is enforced. We invoke this routine when the addition of a new plan violates the budget constraint.

Note that, while dropping an existing plan may help us in keeping numPlans in control, if not done correctly, it could result in violation of sub-optimality bound along with increased optimizer overheads for future query instances. First, in order to ensure that dropping a plan does not result in a violation of the bounded sub-optimality guarantee,

[7] We used $\lambda_r = \sqrt{\lambda}$ for reasons explained in Appendix E.

```
Data: Q, $\lambda$, $q_c$, $sVector_c$, $P_{opt}(q_c)$, Cost($P_{opt}(q_c)$, $q_c$)
//create new 5-tuple $T_{new}$;
$V_{new} = sVector_c$, $S_{new} = 1.0$, $U_{new} = 1$;
$P_{new}$ = pointer($P_{opt}(q_c)$);
$C_{new}$ = Cost($P_{opt}(q_c)$, $q_c$);
InstList $\mathcal{I}_Q$ = getList(Q);
if $P_{new} \in \mathcal{P_C}$ then
    //plan already in plan cache ;
    $P_{new}$ = createPointer($P_{opt}(q_c)$, $\mathcal{P_C}$);
    add $T_{new}$ to $\mathcal{I}_Q$ ;
else
    //plan is new for plan cache;
    //use doRecost() to find minimum cost plan in cache;
    $P_{min}$ = getMinCostPlan(Q, $q_c$, $\mathcal{P_C}$);
    $C_{min}$ = doRecost(Q, $P_{min}$, $q_c$);
    $S_{min} = \frac{C_{min}}{C_{new}}$;
    //redundancy check: we used $\lambda_r = \sqrt{\lambda}$;
    if $S_{min} \leq \lambda_r$ then
        $P_{new}$ = getPointer($P_{min}$, $\mathcal{P_C}$);
        $S_{new} = S_{min}$;
        add $T_{new}$ to $\mathcal{I}_Q$ ;
    else
        if $|\mathcal{P_C}| == k$ then
            //drop a plan to enforce plan budget ;
            $P_r$ = findMinUsagePlan($\mathcal{P_C}$) ;
            remove instances from $\mathcal{I}_Q$ that point to $P_r$;
            remove $P_r$ from $\mathcal{P_C}$;
        end
        $P_{new}$ = createPointer($P_{opt}(q_c)$, $\mathcal{P_C}$);
        add $T_{new}$ to $\mathcal{I}_Q$ ;
    end
end
```

Algorithm 2: manageCache algorithm

while dropping plan P, we also remove all instances from instance list that point to plan P. Second, whenever manageCache is required to drop a plan, it drops the plan with minimum aggregate usage count, i.e. sum over U values, across its instances. This heuristic choice is equivalent to least frequently used (LFU) policy and is expected to perform well in the cases when future workload has the same query instance distribution as W_{past}.

7. EXPERIMENTAL EVALUATION

7.1 Databases and Workloads

We have used 90 parameterized queries over two industry benchmarks TPC-H (using data generator with skew [23]), TPC-DS, and two real world databases RD_1 (98 GB), RD_2 (780 GB) to evaluate and compare our proposed solution against various online PQO techniques. Specifically, RD_2 allowed us to create high dimensional query templates (d $\geq$ 5). For a large fraction of our parameterized queries, optimization overheads are significant compared to execution overheads, making them good candidates for online PQO evaluation. For instance, Q18 from TPC-DS had instances with both optimization and execution overheads close to 500 msec each. In particular, the real world databases support queries that are often multi-block statements with large number of relations causing large optimization times. Overall, the optimization overheads of queries on real datasets varied between 0.5 sec to 5 sec, which was often a significant fraction of or even greater than execution times. While our evaluation also include query templates for which optimization overheads were trivially small, they serve to compare the ability and quality of plan inferencing for various techniques.

Workload query instance generation. A workload sequence can be challenging and interesting for evaluating online PQO techniques if it contains instances with (a) widely varying selectivities (b) large number of parameters (c) large number of distinct optimal plan choices and (d) potential for plan reuse across instances. To generate workloads with above properties, (a) we modify queries by adding extra *one-sided range* predicates, i.e. $col_i < val_i$ or $col_i > val_i$, that can support fine grained control large selectivity ranges; (b) we create query templates with up to 10 parameters, $\approx \frac{1}{3}^{rd}$ of them had $d \geq 4$. Finally, to ensure properties (c) and (d) for any given workload length m, we use the following bucketization of the selectivity space. We divide the space into regions such that the selectivities of parameterized predicates are: (a) small for all parameterized predicates (Region$_0$), (b) large for all parameterized predicates (Region$_1$), and (c) large only for i^{th} parameterized predicate (Region$_{di}$). With the above, to construct a sequence of m instances, we generate $\frac{m}{d+2}$ instances from each region and then put them together in random order to get the final sequence.

The ordering of query instances may have different performance impact on different online PQO techniques. To evaluate their performance across different kind of orderings, we construct several orderings from the same set of instances, including random as well as adversarial orderings as described in Appendix H.1 – performance with only random ordering is reported in Appendix H.5.

Overview of evaluation. We evaluate performance of techniques given in Table 2, using 450 workload sequences that are constructed using 5 orderings for each of the 90 query templates – each sequence has 1000 instances (2000 for $d >$ 3). We also present individual experiments with even larger values of number of query instances (m). Finally, we evaluate variants of existing techniques where they have advantage of making *store or not* decision using redundancy check, in Appendix H.6 and present a sample execution time experiment in Appendix H.7.

OptOnce	store and use first plan only
PCMλ	PCM with λ parameter
Ellipse	with parameter Δ= 0.90
Density	radius = 0.1, confidence threshold = 0.5
Ranges	near selectivity range = 0.01
SCRλ	proposed technique with λ parameter

Table 2: Index for evaluated techniques

Next, we present the performance evaluation starting with sub-optimality related metrics before moving on to numOpt and numPlans metrics.

7.2 Evaluation of Cost Sub-optimality

For each of the 450 sequences, we get a value of MSO and TotalCostRatio for each of the techniques. These values for Optimize-Once are plotted in Figure 6 in increasing order of TotalCostRatio values. Here, we find large number of high values for MSO as well as TotalCostRatio indicating that the workload sequences are challenging for Optimize-Once. We also plot the results for best performing existing heuristic

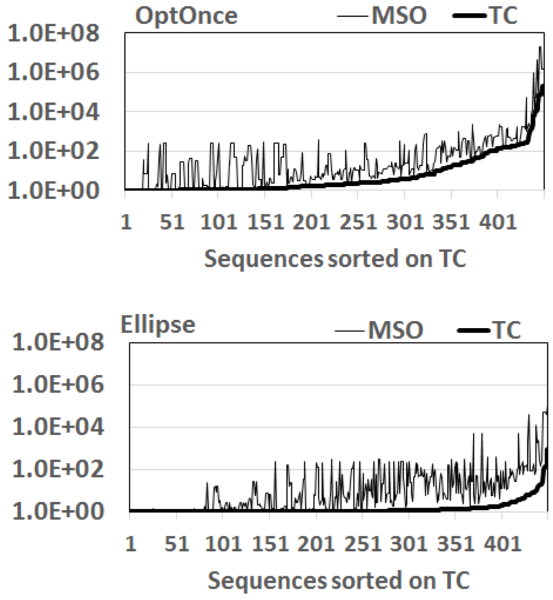

Figure 6: MSO and TotalCostRatio (TC) distribution for Optimize-Once and Ellipse

technique Ellipse in Figure 6. Although Ellipse does reduce TotalCostRatio significantly, it does not avoid the risk of high sub-optimality for certain instances, as is evident from frequently high values of MSO. We also note that it results in TotalCostRatio > 10 for a significant fraction of sequences.

In principle, PCM and SCR with $\lambda = 2$ should have ensured MSO ≤ 2, but we find that in practice that the underlying assumptions about the cost model, i.e., cost monotonicity and bounded cost growth do face occasional violations which result in MSO > 2. Still, as shown in Figure 7, we find the MSO is < 2 for a very large fraction of instances. Furthermore, we observe that such violations are rare; and the resulting values of TotalCostRatio are < 2 for majority of workload sequences.

We also find that violations of sub-optimality bound are much fewer for SCR compared to PCM. One explanation for this behavior is that PCM can potentially infer large selectivity regions based on monotonicity assumption, which when violated can cause large sub-optimality. In contrast, for SCR the inference region for a plan is always localized to a relatively small area around the instance, which means that it has a lower chance of causing large sub-optimality when BCG is violated. Further, our framework supports detection and handling of certain cases of assumption violations, as explained in Appendix G.

Overall, MSO captures only rare sub-optimality violations[8] and in terms of TotalCostRatio, SCR2 processes 99% of the sequences with TotalCostRatio less than 2.16. Aggregate sub-optimality performance is reported in Appendix H.2.

7.2.1 Impact of λ on SCR

Next, Figure 8 shows that the TotalCostRatio values of SCR are consistently lower than allowed values of λ and the difference keeps *increasing* as we allow λ to increase from 1.1 to 2. In fact, with $\lambda = 2$, average TotalCostRatio value for SCR is as low as 1.1. This means that using λ=2 does not hurt much in terms of TotalCostRatio and helps significantly in other metrics (as we will see in other experiments).

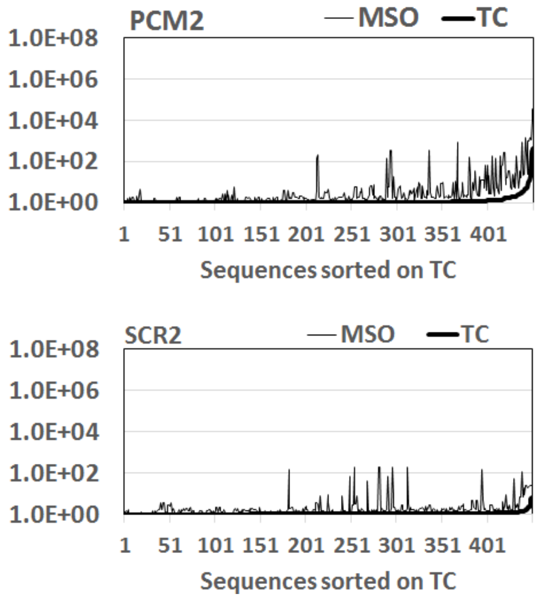

Figure 7: MSO and TotalCostRatio (TC) distribution for PCM2 and SCR2

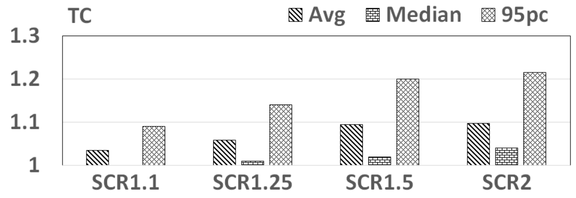

Figure 8: TotalCostRatio (TC) for SCR with varying λ

7.3 Evaluation of Optimizer Overheads

We first show, through a sample experiment, that the getPlan overheads are typically much smaller than the optimizer calls due to our design of getPlan routine. For a 4000 instance workload of TPC-DS Q18, SCR 1.1 (λ_r=0) retains 162 plans out of 264 and a naive implementation getPlan needs to make 162 *Recost* calls. Our heuristic of pruning instances with high *GL* values during the selectivity check, improves the number to just 8. Further, use of redundancy check with $\lambda_r = \sqrt{\lambda}$, helps by retaining only 5 plans and at most 3 *Recost* calls are made by any getPlan call. While few *Recost* calls are quite small compared to optimization overheads, they still dominate the getPlan overheads as the instance list is quite small since only 100 out of 4000 instances are finally stored. The rest of the section discusses overheads due to numOpt.

We highlight in Figure 9 that the optimizer overheads for PCM2 can be very high for certain sequences, e.g. when instances appear in reverse order of optimal cost and it does

[8] In fact, we found that all the significant MSO violations correspond to a few query templates and shown multiple times for different orderings of the same query template.

not get a chance to exploit the created inference regions. With SCR2, the fraction of numOpt is significantly better than most techniques and almost comparable to the best of the heuristic techniques, i.e. Ranges [17].

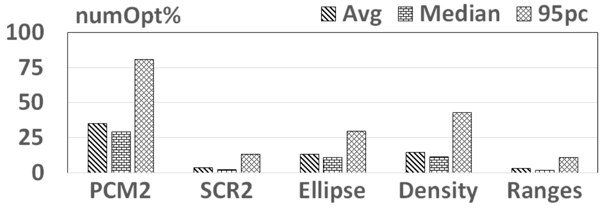

Figure 9: numOpt % for various techniques

7.3.1 Impact of λ

Figure 10 shows that numOpt for SCR improves significantly with increase in λ. Specifically, SCR1.1 could require as large as 35% optimizer calls for a small fraction of workload sequences and it comes down to only 13% with SCR2 and the average numOpt improves from 12% to just 3%.

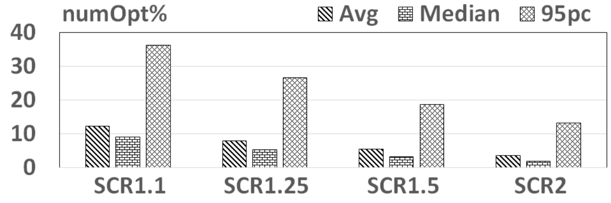

Figure 10: numOpt % for SCR with λ

7.3.2 Impact of number of query instances m

Figure 10 shows that for a small fraction of workloads even SCR2 requires more than 10% optimizer calls and SCR1.1 requires >35%. While such overheads may not be adequate in practice, we note that the required fraction of optimizer calls reduce significantly with increase in the number of query instances – similar behavior was also shown in [4]. To support this claim, we show the performance of an example 4-dimensional query when the number of instances is increased from 1000 to 10,000 in Figure 11. Here, we find that the performance of PCM2 is matched by SCR1.1 itself with increasing length of workload. Further, SCR2 performs significantly better than PCM2 and its numOpt improves from 6.5% to less than 1%. Similarly, for a 10-d example query, the improvement in numOpt % is very similar to Ellipse when it comes down from ≈ 25% for 1000 instances to ≈ 10% for 5,000 instances while PCM2 reduction is much worse at ≈ 35% even for 5000 instances – more details in Appendix H.3.

7.3.3 Impact of dimensions

We study the impact of increase in number of parameterized predicates (d) on the fraction of optimizer calls. We found that optimizer overheads increase rapidly for PCM2 adding ≈10% with each dimension reaching beyond 50% for 10 dimensional query. In contrast, the optimizer overheads as well as the rate of increase is significantly better for SCR2 as shown in Figure 12. Specifically, it starts with 6% and increase only 5% with each dimension. SCR scales better than PCM with increasing dimensions.

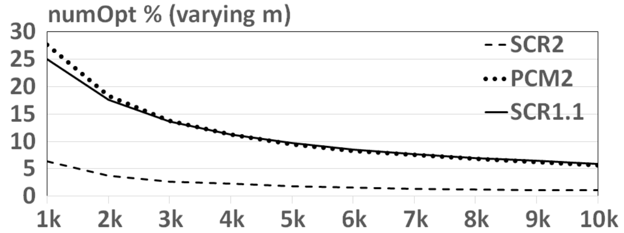

Figure 11: 4-d example query: numOpt % with varying m

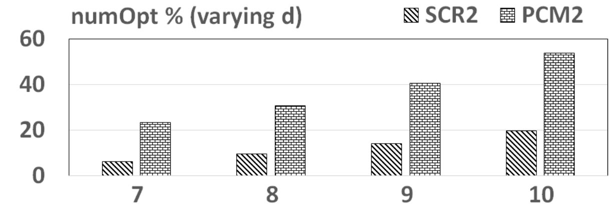

Figure 12: numOpt % for SCR2 and PCM2 with d

We also report the impact of enforcing plan cache budget k on numOpt in Appendix H.4.

7.4 Evaluation of Number of Plans Cached

With regard to the numPlans metric, we first show that SCR2 stores almost an order of magnitude fewer plans compared to other techniques as shown in Figure 13(note that the y-axis is in log-scale).

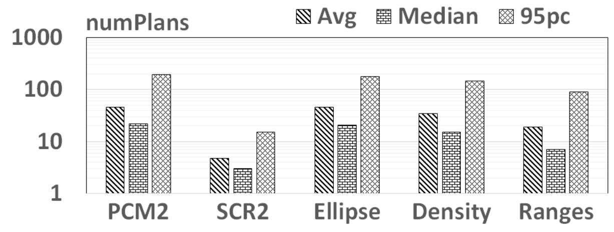

Figure 13: numPlans for various techniques (log scale)

7.4.1 Impact of λ

Next, we find that the number of stored plans improve significantly with increase in λ for SCR, this behaviour is captured in Figure 14.

7.5 Cases when Optimize-Once has MSO < 2

While our choices of workload sequences contained many sequences where Optimize-Once performance was highly suboptimal, it also contained a significant fraction of sequences where Optimize-Once achieved MSO ≤ 2. In other words, the first instance of such sequences results in a plan that can be reused for entire workload sequence without hurting suboptimality much, and therefore there is no significant ad-

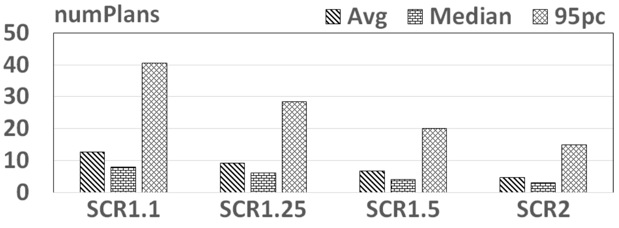

Figure 14: numPlans for SCR with varying λ

vantage in performing more optimizer calls or storing more plans in the plan cache. Thus, such sequences are specifically interesting for evaluation for online PQO techniques as they serve to evaluate the capability of the technique to differentiate simpler workloads from complex ones. We find that SCR performs particularly well in such cases storing less than 2 plans on average while other techniques still need to store tens of plans. Similarly, the fraction of optimizer calls drops to 1.7% for SCR while most other techniques still need 10% or more optimizer calls.

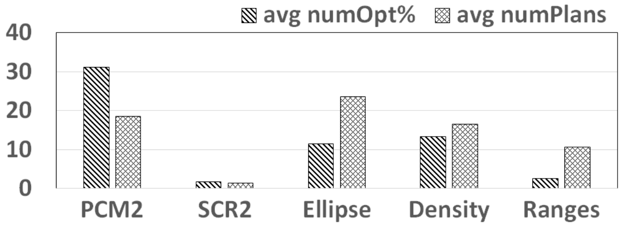

Figure 15: Sequences where Optimize-Once has MSO < 2

8. RELATED WORK

There is a large body of literature that is relevant to plan selection for parameterized queries. We categorize them based on: (a) whether their plan identification phase is offline or online and (b) whether they store a single plan or multiple plans in the plan cache.

Offline or Online, SinglePlan. The online-single plan approach include the solutions like plan caching [22, 24] (equivalent to Optimize-Once), Reopt-Bind [25], Specific-Generic Plans [11]. We refer to these techniques as *online* since they use a only one regular optimization call to identify the plan to be stored and can reuse the stored plan after making simple efficient checks. There have been other approaches where the plan to be stored is identified after an offline compile-time effort that requires more than a single optimization call. These techniques include [6], [5] and [1] and they aim to pick a plan that has least expected cost, low variance and minimum total cost over extremes for the parameter space. While these approaches have different degree of optimization overheads, the entire spectrum of single plan approaches run into the basic limitation that a single plan cannot ensure bounded sub-optimality in a arbitrarily large selectivity space.

This limitation was handled to large extent by the proposal of single *dynamic plan* [7]. However, it was shown in [10] that such a dynamic plan may implicitly store many more plans than required to ensure optimality in the selectivity space. Also, the run-time overheads to resolve the right plan for a given parameter instantiation could be too high to be practical for complex queries.

Offline, MultiPlan. The goal is to identify a set of plans that can ensure *optimal* execution performance for any instance across the selectivity space [15, 10, 13, 18]. Its applicability is limited due to the fact that it may require hundreds of plans to be stored. While there have been many innovative efforts [8, 14, 9] to *significantly* reduce the optimizer or/and plan storage overheads by slightly relaxing the plan quality requirement to be close to optimal – the optimizer overheads may still be wasteful for workloads that originate from small unpredictable regions of the selectivity space. Interestingly, many of these proposals also utilized *Recost* feature for reduction of overheads, the difference is that we use *Recost* feature in online setting where efficiency is even more important. More recently, [21] studied multi-objective PQO problem in *anytime* setting, which is quite different from the focus of this paper, i.e., traditional PQO problem in online setting.

Online, MultiPlan. The earliest known attempt related to online PQO was [12], which aimed to recycle plans across different query templates with 'similar' query structure. Although their approach certainly addresses a wider problem of plan reuse *across* different queries, it does not provide any guarantee on performance sub-optimality. Quite recently, there have been proposals that can handle unpredictable sequences of query instances, i.e., cursor sharing (similar to Ranges in this paper) [17], PCM, Ellipse [4] and Density [2]. These techniques address only one of the three requirements of an online PQO technique (see Section 2.1), while SCR can address all three of them – a detailed discussion about their limitations is provided in Section 3.

9. CONCLUSION

In this paper we present the SCR technique for online PQO that leverages the functionality of plan re-costing and an efficient check based only on selectivities to significantly reduce optimization overheads and number of plans that need to be stored while guaranteeing a tight bound on cost sub-optimality. Efficient plan re-costing functionality and the selectivity check are relatively straightforward to implement in a database engine. We also demonstrate good *average* case sub-optimality on a variety of complex benchmark and real-world queries in the Microsoft SQL Server database engine.

All online techniques including SCR limit themselves to plans that are optimal for at least one query instance observed thus far. It is an interesting area of future work to consider if further reduction in optimization overheads and number of plans are possible without compromising cost sub-optimality by considering plans that may not be optimal for any query instance (e.g., using ideas similar to [1]). Such an approach may be able to combine some of the benefits of offline exploration (e.g., similar to [8]) with those of the online technique.

Acknowledgements. We thank anonymous reviewers, Christian König, Wentao Wu and Bailu Ding for their valuable feedback that helped improve the paper.

10. REFERENCES

[1] M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal, and J. Haritsa, "On the stability of plan costs and the costs of plan stability", *In PVLDB*, 3(1), 2010.

[2] G. Aluc, D. DeHaan, and I. Bowman, "Parametric Plan Caching Using Density-Based Clustering", *ICDE Conf.*, 2012.

[3] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach", *ACM SIGMOD Conf.*, 2005.

[4] P. Bizarro, N. Bruno, D. Dewitt, "Progressive Parametric Query Optimization", *IEEE TKDE*, 21(4), 2009.

[5] S. Chaudhuri, H. Lee and V. Narasayya, "Variance aware optimization of parameterized queries", *ACM SIGMOD Conf.*, 2010.

[6] F. Chu, J. Halpern and J. Gehrke, "Least Expected Cost Query Optimization: What Can We Expect", *ACM PODS Conf.*, 2002.

[7] R. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans", *ACM SIGMOD Conf.*, 1994.

[8] Harish D., P. Darera and J. Haritsa, "On the Production of Anorexic Plan Diagrams", *VLDB Conf.*, 2007.

[9] A. Dey, S. Bhaumik, Harish D, and J. Haritsa, "Efficiently approximating query optimizer plan diagrams", *In PVLDB*, 1(2), 2008.

[10] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", *VLDB Conf.*, 1998.

[11] A. Ghazal, D. Seid, B. Ramesh, A. Crolotte, M. Koppuravuri, and Vinod G, "Dynamic plan generation for parameterized queries", *ACM SIGMOD Conf.*, 2009.

[12] A. Ghosh, J. Parikh, V. Sengar, and J. Haritsa. "Plan selection based on query clustering", *VLDB Conf.*, 2002.

[13] A. Hulgeri and S. Sudarshan, "Parametric query optimization for linear and piecewise linear cost functions", *VLDB Conf.*, 2002.

[14] A. Hulgeri and S. Sudarshan, "AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions", *VLDB Conf.*, 2003.

[15] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis, "Parametric query optimization", *The VLDB Journal 6, 2*, 1997.

[16] L. Krishnan, "Improving Worst-case Bounds for Plan Bouquet based Techniques", *ME thesis, Database Systems Lab, Indian Institute of Science*, 2015. http://dsl.cds.iisc.ac.in/publications/thesis/lohit.pdf

[17] A. Lee and M. Zait, "Closing the query processing loop in Oracle 11g", *In PVLDB*, 1(2), 2008.

[18] N. Reddy and J. Haritsa, "Analyzing plan diagrams of database query optimizers", *VLDB Conf.*, 2005.

[19] F. Topsϕe, "Some bounds for the logarithmic functions", Australian Journal of Mathematical Analysis and Applications, http://ajmaa.org/RGMIA/papers/v7n2/pade.pdf

[20] I. Trummer, C. Koch, "Probably Approximately Optimal Query Optimization", *https://arxiv.org/abs/1511.01782*, 2015.

[21] I. Trummer and C. Koch."An Incremental Anytime Algorithm for Multi-Objective Query Optimization". *ACM SIGMOD Conf.*, 2015.

[22] https://technet.microsoft.com/en-us/library/ms181055(v=sql.105).aspx

[23] https://www.microsoft.com/en-us/download/details.aspx?id=52430

[24] http://www.info.teradata.com/HTMLPubs/DB_TTU_13_10/index.html#page/Database_Management/B035_1094_109A/ch17.19.35.html

[25] http://www.ibm.com/support/knowledgecenter/SSEPEK_10.0.0/comref/src/tpc/db2z_bindoptreopt.html

APPENDIX

A. LIMITATIONS OF SELECTIVITY DISTANCE BASED PLAN REUSE

Most of the existing techniques depend on a selectivity distance based neighborhood to identify opportunities for plan reuse [4, 17, 2]. We discuss the limitations of such techniques with respect to cost sub-optimality performance.

For the purpose of this discussion, let us consider simplistic plan cost functions of the following form,

$$Cost(s_1, s_2) = C_1 s_1 + C_2 s_2 + C_3 s_1 s_2 + C_4$$

First two terms capture the cost of scanning relations with parameterized predicates, third term capture cost of binary join operation between the parameterized relations and last term captures the cost of those operators in the plan that are independent of parameterized predicates. The coefficients are different for different execution plans depending on their join-order and choice of physical algorithms, e.g. Nested Loop vs Hash Join for binary join.

Next, consider query instances $\mathrm{q}_1 = (s_1, s_2)$, $\mathrm{q}_2 = (s_1 + \delta, s_2)$ and $\mathrm{q}_3 = (s_1, s_2 + \delta)$. Both q_2 and q_3 are at a distance δ from q_1 in the selectivity space, but in perpendicular directions. Now, we derive a function for cost difference between for each pair q_1, q_2 and q_1,q_3.

$$Cost(\mathrm{q}_2) - Cost(\mathrm{q}_1) = C_1\delta + C_3\delta s_2$$

$$Cost(\mathrm{q}_3) - Cost(\mathrm{q}_1) = C_2\delta + C_3 s_1\delta$$

If the optimal plan at q_1 is reused at q_2, then its sub-optimality can be unbounded since the increase in cost can be unbounded depending on the values of coefficients C_1 and C_2 which are not considered while making the plan reuse decision. Further, the sub-optimality of reuse for q_2 can be very different compared to sub-optimality at q_3 while they lie at the same selectivity distance from q_1. Finally, the cost difference for the same selectivity distance also changes depending on the selectivities of query instance q_1, i.e., s_1 and s_2. For all the reasons given above, use of selectivity distance to define inference region of any plan can be risky in terms of cost sub-optimality.

Our technique avoids these limitations by using selectivity (multiplication) factors and explicit cost computation using *Recost* to infer a λ-optimal region around optimized query instances. In fact, we can observe that the geometrical shape of our selectivity based λ-optimality region is quite different from the circular, elliptical or rectangular approximations used by the existing techniques. Also, its exact shape as well as size varies with the location in the selectivity space, i.e., s_1 and s_2 values – thereby, adapting to handle cost impact of various combination of individual selectivity factors.

B. REQUIRED API IMPLEMENTATION

In principle, the required features of $sVector$ computation and *Recost* can be implemented by reusing modules that are already present inside the query optimizer, i.e., predicate selectivity computation module and plan cost computation module respectively. Also, their overheads are expected to be much lower than an optimizer call since they require much fewer invocations of these modules compared to a regular optimizer call.

We provide the outline for implementation of required API features in the Microsoft SQL Server database engine, which follows Cascades framework. However, the discussion is applicable to any engine whose query optimizer is based on memoization.

***sVector* computation.** Currently, the query optimizer computes the selectivities of all parameterized base predicates right after parsing stage. That is, during the phase where logical properties of memo groups are determined. Hence, API for *sVector* computation can be efficiently implemented by short-circuiting the physical transformation phase altogether.

***Recost* computation.** Consider that we need to *Recost* plan P_e for query instance q$_c$, where P_e is found after optimization of q$_e$. We propose to compute a cacheable representation for plan P_e at the end of optimization phase of query instance q$_e$. This representation is then stored along with actual execution plan P_e in the plan cache and used to for future *Recost* calls.

At the end of optimization phase, the optimizer choice plan is extracted out of the *Memo* data structure. At this stage, the size of *Memo* can be quite large as it may contain many groups and expressions that were considered during plan search but are no more required for the final optimizer choice plan P_e. These extra groups and expressions are then pruned and we term the resulting data structure as *shrunkenMemo*. We found that such pruning reduces the size of *shrunkenMemo* by around 70% or more for complex queries that access large number of relations. This *shrunkenMemo* can be stored in the plan cache. When *Recost* API is invoked with a pointer to *shrunkenMemo*, the cost for q$_c$ can be computed by replacing the new parameters in the base groups of *shrunkenMemo*, followed by cardinality and cost re-derivation in a bottom-up fashion that typically consists of arithmetic computations.

The overheads of this re-derivation process depends on the number of groups in the *shrunkenMemo* and we found huge savings due to the memo pruning step. Note that, the overheads to create *shrunkenMemo* is only one time per plan and not to be included in the overhead of *Recost* API. Also, there can be alternative implementations of *Recost* that require lesser memory overheads at the cost of increased time overheads for each *Recost* call.

C. PROOF OF SUB-OPTIMALITY BOUND

THEOREM 2 (SUB-OPTIMALITY BOUND). *Under the assumption that the $f_i(\alpha_i) = \alpha_i$ holds for both plans P_e and P_c, SubOpt(P_e, q$_c$) $< GL$.*

PROOF. Using lemma 1 for P_e, we know that

$$\text{Cost}(P_e,\, \text{q}_c) < G \times \text{Cost}(P_e,\, \text{q}_e) \tag{2}$$

Again using lemma 1 for P_c, we know that,

$$\text{Cost}(P_c,\, \text{q}_c) > \frac{\text{Cost}(P_c,\, \text{q}_e)}{L} \tag{3}$$

But since P_e is known to be optimal plan for q$_e$, we can say that, Cost(P_c, q$_e$) > Cost(P_e, q$_e$), which leads to

$$\text{Cost}(P_c,\, \text{q}_c) > \frac{\text{Cost}(P_e,\, \text{q}_e)}{L} \tag{4}$$

Using Eq.2 and Eq.4 together in sub-optimality expression of P_e at q$_c$, we get

$$\text{SubOpt}(P_e,\, \text{q}_c) = \frac{\text{Cost}(P_e,\, \text{q}_c)}{\text{Cost}(P_c,\, \text{q}_c)} < GL$$

D. VARYING λ ACROSS INSTANCES

Our framework can easily support use of a dynamic value of λ that is a function of C value stored in the instance 5-tuple. One possible way to enforce higher values of λ for lower cost queries, is to ask for a range of threshold from the user, i.e., $[\lambda_{min}, \lambda_{max}]$ and use a exponentially decaying function to map the cost C to appropriate λ to be used for a given stored instance. This may result in increased MSO but it is expected to result in reduced numOpt and numPlans without significant increase in TotalCostRatio. To evaluate its benefits, we performed a sample experiment with a workload of 1000 instances (featuring 378 optimal plans) of Q25 of TPC-DS with λ_{max}=10 and λ_{min}=1.1. Compared with static value of λ=1.1, we found that numPlans improved from 148 to 96, numOpt improved from 502 to 310 while TotalCostRatio increased only from 1.03 to 1.08. We also found that, the extra savings due to such dynamic threshold range decrease as the value of λ_{min} increases.

E. CHOOSING $\lambda_{\mathbf{R}}$ PARAMETER

The simplest policy to implement is to store every new plan that is encountered (λ_r=1). But this may not be acceptable if there is limited (per query) memory available in the plan cache. Even when memory is not a concern, we observed that keeping more plans cause more getPlan overheads and there is no significant degradation in sub-optimality even if they are rejected, as highlighted by following sample experiment. For TPC-DS Q18 with 4000 instances and $\lambda = 1.1$, $\lambda_r = 0$ retains 77 plans and requires up to 8 *Recost* calls during getPlan to achieve TotalCostRatio of 1.03. With λ_r=1.01, retained plans reduce to 14 and #*Recost* calls reduce to 5 with no visible impact on TotalCostRatio. Further, with $\lambda_r = \sqrt{\lambda}$, TotalCostRatio increases only to 1.04 but it retains only 5 plans and #*Recost* calls also reduces to 3. Further increase in λ_r do not provide any improvement in getPlan overheads and in fact, cause increase in the number of optimizer calls. This is because, use of sub-optimal plans cause selectivity regions to shrink around many instances as $\frac{\lambda}{S}$ gets closer to 1. For this reason, we used $\lambda_r = \sqrt{\lambda}$ in our evaluation to balance the trade off.

F. REDUNDANCY CHECK FOR EXISTING PLANS IN CACHE

An existing plan P in plan cache is said to be *redundant* if for each of the instances in the instance list that point to P, there exist an alternative λ-optimal plan in the plan cache. Note that, it is not required that all instances should have the same alternative plan.

We have discussed in Section 6.3 that this redundancy check is done for each newly encountered plan. Hence, the existing plans can become redundant only after a new plan satisfies the entry level redundancy check and is added the plan cache. Next, we describe how similar redundancy check can be done for existing plans in cache.

For a given plan P for which we want to check redundancy, we find all the associated instances I_P and temporarily remove the set I_P from instance list and plan P from plan list. Then, we recycle each instance q_e in I_P by invoking getPlan routine (simulated version, does not actually send plan to executor) that gives an alternative λ-optimal plan and its sub-optimality for q_e. If getPlan succeeds for all instances in I_P, then we say that P is a redundant plan and can be discarded. Then, we add the instances I_P back into instance list but with new 5-tuples created using alternative plan choice and its sub-optimality. If getPlan fails for even one instance of I_P, we infer that P was not redundant and we add the instances in I_P and plan P back to the respective lists.

Note that, the overheads of redundancy check increase with the size of I_P. Specifically, it may require re-costing every plan in plan cache (except P) for each of the instances in I_P. For this reason, it may be faster and more likely to find a redundant plan if we do this redundancy check in the increasing order of size of I_P. In our evaluation, we have used redundancy check only for the newly encountered plan.

G. HANDLING SUB-OPTIMALITY DUE TO VIOLATION OF PCM AND BCG

The violation to PCM and BCG assumptions can be detected using *Recost* in the following way: whenever an instance q_e is used to make a plan choice for another instance q_c using cost check, observe the costs $Cost(P, q_e)$ and $Cost(P, q_c)$ where P is the optimizer choice plan for instance q_e. By comparing selectivity ratios between q_e and q_c and the two costs, it can be detected whether any of the assumptions are violated for P at q_e and if yes, q_e can then be marked as removed for future plan reuse using cost-check to avoid further sub-optimal plan inferences.

In addition, an inferred plan can be sub-optimal even if P does not violate the BCG assumption, but it is actually violated by the optimal plan at q_c. There is no way to detect or handle such cases without making an optimizer call and *Recost* calls for q_c. Since this way of detecting the violation defeats the original purpose of plan inference, it cannot be used frequently.

H. ADDITIONAL EXPERIMENTS

H.1 Generating various orderings

The purpose of these orderings is to test the reliability of various techniques against workload sequences having complex and unknown patterns. To construct various orderings from a given set of instances, we first optimize each query instance to obtain its optimal plan and optimal cost. Then we use this information to create orderings with following properties:

1. Decreasing order of optimal cost values.
2. Instances picked from optimality regions of different plans in a round-robin fashion.
3. Inside-out order, i.e., where instances with near-average optimal cost values occur first, and then slowly diverge towards the extreme cost values.
4. Outside-in order, i.e. where instances with extreme optimal cost values occur first, and slowly converge towards average cost values.

H.2 Aggregate sub-optimality performance

The aggregate performance of various techniques for MSO and TotalCostRatio is captured in Figure 16 and Figure 17 respectively. We first notice that most heuristic approaches have average performance either comparable or even higher than their 95 percentile values, indicating a high skew in the performance across the workload. Hence, even if they handle a large number of sequences reasonably well, the associated sub-optimality risk cause their overall average performance to be an order of magnitude worse than SCR2 which is truly close to optimal at average TotalCostRatio as low as 1.1. Even PCM2 has TotalCostRatio value of ≈3, which means that it takes 3 times more optimizer estimated cost compared to Optimize-Always.

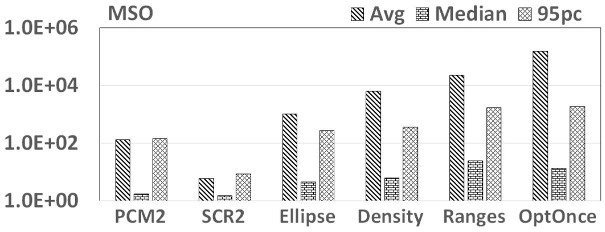

Figure 16: MSO performance

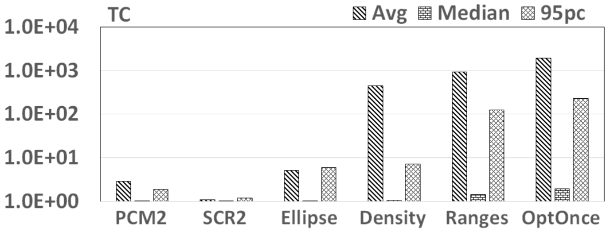

Figure 17: TotalCostRatio (TC) performance

H.3 numOpt with workload size (10-d example)

While the numOpt % is large for the above 10-d example, we show in Figure 18 that numOpt % values comes down significantly as the length of sequence (m) is increased up to 5000 instances. It shows that SCR2 performance is similar to heuristic technique Ellipse while achieving cost sub-optimality comparable to PCM2.

H.4 Impact of k on numOpt of SCR2

SCR supports redundancy check while adding plans and dropping of plans that have become redundant due to addition of a new plan. But sometimes, in order to satisfy a hard budget on numPlans in terms of k, SCR may repeatedly throw and bring back plans that are not redundant, leading to increased optimizer calls. In Figure 19, we show the impact of imposing such budgets on SCR2 over all 450 workloads. It is found that the required number of optimizer calls increase slowly with budgets of 10 and 5 implying that

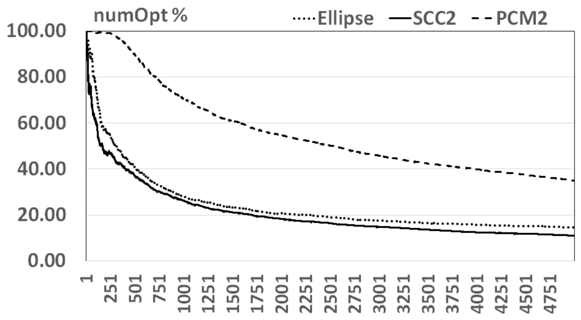

Figure 18: Running numOpt % for a 10d example query

majority of the workloads could be completed within 5 plans without major side-effect on numOpt. Only, when the budget is very tight with $k = 2$, the numOpt values increase significantly which tells that there were significant number of workloads that required more than 2 plans.

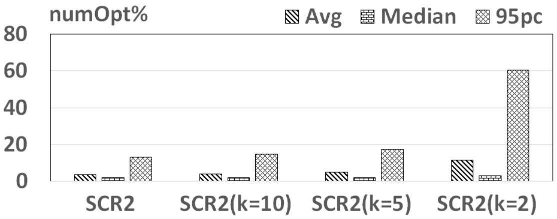

Figure 19: numOpt % variation with plan cache budget(k)

H.5 Performance for only random orderings

The optimizer overheads for most existing techniques improve when considered only for random order sequence. For example, 95 percentile overheads for PCM2 comes down from 81% to 39% and for Density it improves from 30% to 24%. SCR2, on the other hand, has similar performance across all orderings and with evaluation restricted to only random ordering the 95 percentile overheads are 11.9% that is much lower than 39% for PCM2. Further, even in comparison to best heuristic technique 95 percentile performance is 11.9% for SCR2 against 8.1% for Ranges. Overall, we can conclude from this discussion that the advantage of SCR2 over other techniques is not an artifact of the specific orderings. In fact, it shows that SCR2 continues to provide similar performance even when workload follows patterns that hamper performance of other techniques.

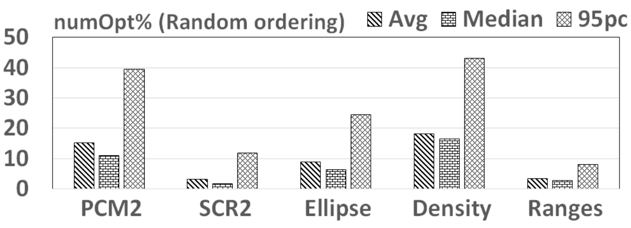

Figure 20: Optimizer overheads (random orderings only)

H.6 Using Recost in existing techniques

While the existing techniques do not support redundancy check in their proposed form, we have evaluated a scenario where they could also use *Recost* feature to implement the redundancy check similar to SCR. We find that this modification helps every existing technique in improving the numPlans. In fact, it even helps in improving optimizer overheads for some of the existing techniques. This is because such modification allows them to make larger selectivity inference regions for each plan as there would now be more instances with same plan choice. But, with regard to cost sub-optimality metrics, we found that the MSO and TotalCostRatio values either remain in the same high range or even degrade further as shown in Figure 21. In contrast, use of *Recost* feature brings advantage in overheads without violating the sub-optimality constraint.

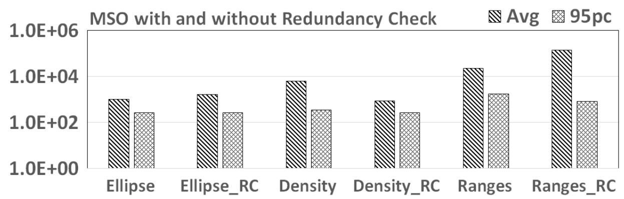

Figure 21: Impact of using *Recost* with existing techniques

H.7 Sample execution experiment

The reason we used optimizer costs for evaluation in the paper is that execution times can be highly variable and may depend heavily on execution environment (CPU load, available memory etc.). Still, we provide one sample execution experiment, for 500 instances of a query based on TPC-DS database, for which we observed optimization overheads to be comparable to execution times (188 seconds compared to 230 seconds for processing the 500 instance sequence) (see Table 3). Note that, PCM1.1 gives minimum execution time but does not save much on optimization overheads. Existing heuristic techniques save significant fraction of optimization overhead but also suffer from sub-optimal executions as their *selectivity neighborhood* parameter is relaxed. Due to a combination of issues that include cost modelling error, SCR1.1 faces sub-optimality beyond the required bound of 1.1 for many individual instances and slightly with regard to ratio of total execution time (≈ 1.2), but saves heavily in the optimization overheads (which include optimization calls and getPlan overheads). Overall, SCR1.1 saves around 40 seconds more than the best performing comparative technique. Also, SCR1.1 retains only 13 out of 101 plans, also much better than best performing comparative technique.

Technique	Opt. Time (sec)	Exec. Time (sec)	Total Time (sec)	Plans
OptAlways	188	230	418	101
OptOnce	0.5	543	543.5	1
Ellipse (Δ =0.9)	99	246	345	77
Ellipse (Δ =0.7)	66	262	328	63
SCR 1.1	19	261	280	13
PCM 1.1	141	238	379	95
Ranges (1%)	22	302	324	31

Table 3: Sample execution performance across techniques