Leveraging Query Logs and Machine Learning for Parametric Query Optimization

Kapil Vaidya*
MIT
kapilv@mit.edu

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) must address two problems: identify a relatively small number of plans to cache for a parameterized query (populateCache), and efficiently select the best cached plan to use for executing any instance of the parameterized query (getPlan). Our approach decouples these two decisions. We formulate populateCache as an optimization problem with the goal of identifying a set of plans that minimizes the optimizer estimated cost of queries in the log, and present an efficient algorithm. For getPlan, we leverage query logs to train machine learning (ML) models to choose the lowest optimizer-estimated cost plan from the cached plans. We conduct extensive experiments using complex parameterized queries from benchmarks and real workloads. Our algorithm for populateCache achieves low geometric mean sub-optimality (1.2) even for complex queries using relatively few plans, and scales well to large query logs. The mean latency of our ML model based getPlan technique ($\sim 210\mu sec$) is between one to four orders of magnitude faster compared to prior PQO techniques. The mean sub-optimality is low (1.05), and the 95th percentile sub-optimality (1.3) is between 1.1× and 25× lower compared to prior techniques. Finally, we present an efficient algorithm for getPlan that leverages execution time information in query logs to circumvent inaccuracies of the query optimizer's cost estimates.

1 INTRODUCTION

Database applications extensively use *parameterized queries*, where the same SQL statement is executed repeatedly with different parameter bindings (e.g., a stored procedure). For complex SQL queries, the straightforward approach of optimizing every query instance (Opt-Always), can consume significant CPU and memory resources. In fact, many commercial relational database systems [4, 26] adopt the other extreme approach by optimizing only the first (or user-specified) query instance of a parameterized query and caching the plan for reuse in subsequent query instances. While this technique (Opt-Once) minimizes optimization time, the cached plan may be arbitrarily sub-optimal for subsequent query instances.

Parametric query optimization (PQO) is a middle-ground approach that aims to sharply reduce optimization overheads compared to Opt-Always while incurring execution sub-optimality as little as possible compared to Opt-Always. For a given parameterized query, a PQO technique must identify a set of execution plans to be cached. When a new query instance arrives, the PQO technique must efficiently pick the best plan to use for this query instance from among the cached plans. Using terminology from prior work [10, 22], we use the term populateCache for the former task and the term getPlan for the latter task. The sub-optimality of any populateCache technique for a given query instance is the ratio of the cost of the best plan available in the cache to the cost of the plan obtained by optimizing the query instance (i.e., Opt-Always). The sub-optimality of getPlan for the given query instance is the ratio of the optimizer estimated cost of the plan selected by getPlan to the optimizer estimated cost of the best plan in the cache. Note that the overall sub-optimality of a PQO technique is the combined sub-optimality of populateCache and getPlan. Since getPlan is on the critical path of query execution, another crucial metric is efficiency, i.e., the time taken for an invocation of getPlan.

There has been extensive prior work on PQO (see Section 7) most of which are *online* techniques [9, 10, 22, 32]. With respect to getPlan these techniques often fall short in at least one of the above metrics of sub-optimality or efficiency. Specifically, their decision to reuse a cached plan for a new query instance is based on assumptions (e.g., monotonicity) on how cost of the plan changes over multi-dimensional space of selectivity of parameterized predicates. Since these assumptions often fail for complex SQL queries [37], getPlan sub-optimality is often very high. Some techniques make more conservative assumptions on how the cost of a plan changes, and thereby achieve better sub-optimality, but these techniques frequently need to fall back to making expensive optimizer or re-cost calls [22], and therefore suffer from high mean and tail latency for getPlan. With respect to populateCache, the online techniques adaptively add and remove plans from the plan cache, but: (a) are operationally challenging due to lack of predictability and debuggability arising from a continuously changing set of plans for a query, and (b) often require a large number of plans to be cached.

1.1 Decoupling populateCache and getPlan

We propose to architecturally decouple populateCache and getPlan into independent modules. In this approach, getPlan only selects the best among the cached plans and does not fall back to the optimizer, immediately resolving the high latency issue. populateCache, which is run as an offline step, identifies a set of plans to be cached which are then installed in the plan cache. This results in greater predictability and debuggability and is simpler to operationalize. The decoupled architecture has an additional benefit that it allows DBAs to manually include a set of plans to cache based on their experience. This architecture requires invoking the offline step when the data or workload characteristics change significantly.

1.2 Leveraging query logs for PQO

We observe that today's DBMSs come equipped with efficient technology for logging query instances [6, 7]. An application developer or DBA can also explicitly provide a workload of query instances for a parameterized query. In our approach to PQO both populate-Cache and getPlan leverage query logs as described below.

We present a novel formulation for populateCache that uses query logs in an offline step to identify plans to cache such that the optimizer-estimated cost of the workload is minimized (Section 5). Query logs help populateCache focus on workload-relevant regions in an otherwise very large selectivity space. We show that a simple, greedy search technique scales well with the number of instances in the query log. Even when restricted to use only a small number of plans to cache, usually no more than 6, our algorithm achieves a relatively low geometric mean sub-optimality (~ 1.2) over queries in the workload (Section 6.3). Finally, we observe that due to the decoupling of populateCache and getPlan in our architecture, one could alternatively replace our algorithm with a prior technique for populateCache [20, 21, 25, 28–30, 37].

For getPlan– the main focus of this paper – given a parameterized query, we use the query logs and a set of cached plans as input to train a supervised machine learning (ML) model that can efficiently identify the best cached plan to use for an incoming instance of that parameterized query (Section 3). Observe that, unlike prior techniques [9, 10, 22, 25, 28, 32], our approach works for any complex SQL query and also makes *no assumptions* about the cost behavior of the plans. We show that using decision trees [1] or tree-ensembles [15], we can generate ML models of small sizes (e.g., 16KB) that are very accurate for the task. Our ML model based technique has low mean latency ($\sim 210\mu sec$) and 95th percentile latency ($\sim 350\mu sec$), which is one to four orders of magnitude smaller compared to prior techniques. Our getPlan technique also achieves low sub-optimality (geometric mean of ~ 1.05), and the 95th percentile sub-optimality (~ 1.3), which is 1.1$\times$ to 25$\times$ lower compared to prior PQO techniques for getPlan.

1.3 Exploiting execution cost for getPlan

Previous work on getPlan, including the formulation above, frames the goal of getPlan in terms of minimizing the optimizer-estimated cost of query instances, and may lead to sub-optimal selection of plans in terms of execution cost (e.g., CPU time or execution time). This mismatch is not surprising due to well-known issues such as simplified cost models that are used by query optimizers and errors

in cardinality estimation [13, 33]. Indeed, such behavior is common in parameterized queries that we evaluated (Section 6.5).

We observe that the problem of designing execution-cost aware getPlan for a given parameterized query and a given set of cached plans is much simpler than the general problem of making cost estimation used by the query optimizer consistent with execution cost for arbitrary queries. A straightforward adaptation of the supervised ML approach described in Section 1.2 is, however, impractical due to the prohibitive cost of acquiring labeled training data which includes obtaining execution costs for query instances for multiple cached plans.

We propose an *execution-cost aware* getPlan formulation based on the principle of *contextual bandits* [42] in the reinforcement learning literature (Section 4). For training, the bandit learner uses the execution cost obtained after each query instance has executed. Unlike supervised ML techniques, no additional executions for other cached plans, which were not chosen to execute the query instance, are required for training. We find that a contextual bandit learner implemented using *Bootstrapped Thompson-sampling* [19] with tree-based regression models [1, 15] has low-latency and leads to plans with low execution cost. For several parameterized queries, the geometric mean of execution sub-optimality improves by 2$\times$ and the 95th percentile execution sub-optimality improves by an order of magnitude when compared to choosing the best plan according to optimizer estimated cost. Since a standard implementation of bandit learners results in highly sub-optimal plans during the initial learning phase, we develop a technique to mitigate this limitation by leveraging the query optimizer for bootstrapping.

1.4 Summary of contributions

- Unlike many prior PQO techniques where populateCache and getPlan are intertwined [9, 10, 22, 32], we propose decoupling the modules. This leads to greater flexibility in choosing plans to cache and improves predictability and debuggability, which is crucial in production environments.

- We present a formulation of populateCache as an offline search problem based on query logs. Evaluation of our algorithm shows that across different parameterized queries, the algorithm achieves low aggregate optimizer estimated cost with relatively few plans, and scales to a large number of input query instances.

- For getPlan, for the optimizer-estimated cost metric, we show that simple ML models with small memory footprint can achieve both: (i) fast average and tail getPlan times *and* (ii) low sub-optimality. Furthermore, unlike prior work, our techniques work for arbitrary SQL queries without requiring assumptions about the behavior of optimizer cost models.

- We introduce a novel framing of the getPlan problem based on execution cost where the goal is to select from among the cached plans, the plan with the lowest execution cost. We formulate this as a contextual multi-armed bandit problem whose rewards are based on execution cost feedback.

- We present a comprehensive empirical evaluation using multiple synthetic and real-world parameterized queries, comparing the proposed techniques against several previous PQO techniques from the research literature and approaches adopted by commercial DBMSs.

2 QUERY LOG DRIVEN PQO ARCHITECTURE

Applications extensively rely on parameterized queries, using database mechanisms such as stored procedures and prepared statements. Different instances of a parameterized query can vary only in the parameter bindings. Some commercial systems [4, 26] cache a *single* plan for each parameterized query, whereas other DBMSs allow storing multiple plans [32]. The advantage of caching plans for a parameterized query is that the overhead of invoking the query optimizer can be avoided. In Figure 1, we outline the components for a DBMS that: (i) architecturally decouples populateCache and getPlan and (ii) leverages query logs for PQO. Below, we describe the functionality of each key module shown in the figure.

Collecting query logs: The functionality of logging query instances is standard in most commercial DBMSs [6, 7]. A query log contains information about each query instance including its parameter bindings, execution time, and CPU time. We expect that the DBMS bootstraps the PQO process by using an existing PQO technique, e.g., Opt-Once [4], or Merging-Ranges [32], or even Opt-Always, and then utilizes the proposed techniques once sufficient instances have been logged[1].

populateCache: The offline populateCache module is responsible for identifying a set of plans to cache for a given parameterized query Q. It takes as input a set of query instances of Q. This input can be obtained from the query log, or can be provided directly by a DBA or application developer. populateCache takes as input a budget on the number of plans (K) to cache. It also takes and an additional optional constraint as input: a set of plans that must be included in the set of plans to cache. During its search, the populateCache module consults the query optimizer using two APIs: (a) A regular optimizer call, which returns a plan for a query instance or (b) A recost call, which returns the cost of a given plan for a specific query instance [8]. The set of K plans returned can be added to the plan cache, e.g., using an API similar to plan guides [5].

Training models for getPlan: For a given parameterized query Q, this offline module takes as input a set $\mathcal{P}$ of K plans and a log of query instances of Q, and trains an ML model. Given an arbitrary query instance as input, the ML model selects one of the plans in $\mathcal{P}$ to use for executing that query instance. To obtain labeled data necessary to train the ML models, this module consults the query optimizer using the recost interface to obtain the matrix of (plan, cost) values. Note that even when the plans to be cached are not changed, we may wish to re-train getPlan, e.g., when data or workload characteristics have changed sufficiently. Finally, the ML model is installed for use by getPlan.

getPlan: The plan cache contains one entry for each parameterized query, which points to the set of cached plans for that query and the ML model to select among them. For a new query instance, getPlan first parses and then matches it to its parameterized query Q. It extracts the bindings for the query instance and uses them to compute the selectivity for each parameterized predicate, which forms the input to the ML model. The plan returned by invoking the ML model used to execute the query instance.
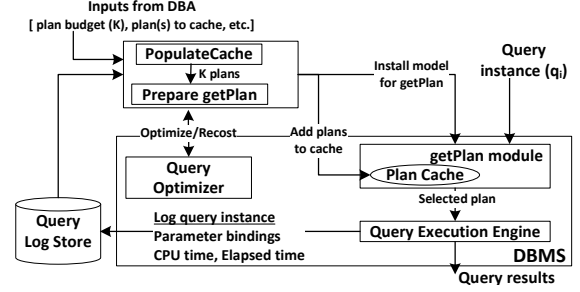

Figure 1: Architecture Overview of Query-Log driven PQO

2.1 Problem description

For a parameterized query (a.k.a. query template) Q, with d parameterized predicates, we use q_e to denote an instance of Q. Each instance of Q is defined by the set of bindings of the parameterized predicates, e.g., $V=(v_1, \ldots, v_d)$. We also associate a derived selectivity vector $S=(s_1, \ldots, s_d)$ with each query instance, consisting of the selectivity of each parameterized predicate. For a given query instance q_e, the plan obtained by invoking the query optimizer on it is denoted by $P_{opt}(q_e)$. Further, for a given plan P and query instance q_e, the optimizer estimated cost is denoted with $Cost(P, q_e)$ and the actual execution cost is denoted by $ExecCost(P, q_e)$. Next, we define a set of n query instances (a.k.a workload) of Q as $W = \langle q_1, q_2, \ldots, q_n \rangle$. Below we formally define the two goals of PQO: populateCache and getPlan, along with the relevant metrics for their evaluation.

Problem definition for populateCache: Given a workload W, and a user specified budget K on number of plans, efficiently choose a plan set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ such that the aggregate (e.g., geometric mean or 95th percentile) sub-optimality metric $SO_{\text{populateCache}}$ over all instances in W is minimized, where

$$SO_{\text{populateCache}}(\mathcal{P}, q_e) = \frac{argmin_{P \in \mathcal{P}} Cost(P, q_e)}{Cost(P_{opt}(q_e), q_e)} \quad (1)$$

The term in the numerator assumes a perfect getPlan which chooses the lowest cost plan among all plans in $\mathcal{P}$. The term in the denominator is the cost of the instance assuming an ideal strategy Opt-Always, which optimizes the query instance. We consider the aggregated metric over all instances in W, in the form of either geometric mean or 95th percentile.

Another metric relevant for populateCache evaluation is the total time needed to identify the required plans. We also measure the total number of *optimize* and *recost* calls made by the algorithm.

Problem definition for getPlan: Given a workload W, for parameterized query Q, a set of K cached plans $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$, and a query instance (may not be part of W) select the cached plan that minimizes

Objective 1: $SO_{\text{getPlan}}^{OptCost}$ for a query instance, where

$$SO_{\text{getPlan}}^{OptCost}(P', q_e) = \frac{Cost(P', q_e)}{\arg\min_{P \in \mathcal{P}} Cost(P, q_e)} \quad (2)$$

Objective 2: $SO_{\text{getPlan}}^{ExecCost}$ for a query instance where

$$SO_{\text{getPlan}}^{ExecCost}(P', q_e) = \frac{ExecCost(P', q_e)}{\arg\min_{P \in \mathcal{P}} ExecCost(P, q_e)} \quad (3)$$

Similar to populateCache, we consider an aggregated metric in the form of either geometric mean or 95th percentile. In addition, we also evaluate getPlan with *latency* as the time taken to return the selected cached plan, and report its mean and tail percentile values. We also report the time spent in generating the labeled data and training the ML model for getPlan.

3 OPTIMIZER COST BASED GETPLAN

Given a set of K plans to be cached, and a new incoming query instance q_e, getPlan is responsible for selecting the cached plan to use for executing q_e. In this section, we focus on the setting where the objective is to select the cached plan with the lowest *optimizer estimated cost*. A straightforward method to accurately determine the cost-minimizing plan for q_e is to compute estimated cost for each plan and return $argmin_{P \in \mathcal{P}} Cost(P, q_e)$, however, it requires invoking a recost method that must work for arbitrarily complex SQL queries. Prior work [22] evaluated an existing feature in a commercial database (USE PLAN query hint in Microsoft SQL Server), and found that a recost call incurs at least 10 ms per plan and often 100s of milliseconds for complex queries[2], which is too expensive to use in practice.

We observe that identifying the best cached plan does not necessarily require computing accurate costs of cached plans, and instead use machine learning (ML) techniques to design an efficient and accurate implementation for getPlan. In particular, we propose using a supervised ML model for getPlan requiring small memory footprint and fast inference time. Each ML model is trained to deal with a single parameterized query, by leveraging a set of query instances labeled with the best plan choices. The effectiveness of such a model arises from its ability to identify patterns in the best plan labels from the training data. In the rest of this section, we describe the input features to the ML model, present two different supervised ML formulations: classification and regression, followed by discussions on model training and a comparison of the two formulations.

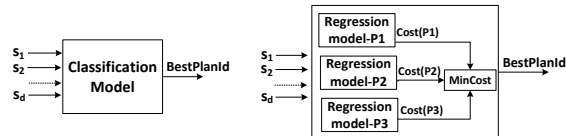| s_1	s_2	...	s_d	Cost(P1)	Cost(P2)	Cost(P3)	BestPlanId
0.05	0.22	...	0.14	10	28	20	1
0.29	0.13	...	0.32	75	70	50	3
0.11	0.47	...	0.05	30	12	35	2
0.7	0.32	...	0.42	140	110	70	3

Figure 2: ML Model designs for a given parameterized query

---

[2]The overhead is dominated by the cost of creating the optimizer's internal memo data structure. Even if the internal data structure were to be cached [22], a recost call takes up to 10 ms per plan to invoke cardinality and cost computation functions across all operators in a complex plan.

3.1 Features

We use the vector of selectivities of each parameterized predicate as input features to the model. Intuitively, our models use these features to divide the high dimensional selectivity space into *partitions*, and map each partition to one of the cached plans. This is a significant benefit compared to prior work [9, 10, 22, 32] since getPlan does not need to invoke the query optimizer. Note that most prior work in PQO also rely upon the same information to make their getPlan decisions. These features can be efficiently obtained ($\approx 25~\mu sec$ per selectivity) using statistics such as histograms that are available in database systems.

While we found that selectivity features deliver sufficient accuracy to outperform prior PQO techniques, we also consider alternative input features in our experimental evaluation (see Section 6.2.3). In particular, we use parameter bindings of the parameterized predicates parsed from the query instance as input features. We use the same feature encoding for parameter bindings as proposed recently in the context of cardinality estimation [23] for commonly occurring range and categorical predicates (IN clause). Interestingly, we find that the models with only parameter bindings can also yield sufficiently accurate models to match the best of the prior PQO techniques, although selectivity features deliver even better accuracy. We observe that for both feature alternatives, the getPlan overhead is much smaller than prior techniques that support accurate plan selection.

3.2 Classification formulation

The getPlan problem can naturally be viewed as a *multi-class classification* problem, where the given K plans correspond to K-classes. The model learns to partition the feature space (e.g., selectivities) based on the best plan id labels. Hence, the training data consists of a set of query instances from W labeled with the id of the *best* among the K plans. The training data and design of ML models is depicted in Figure 2.
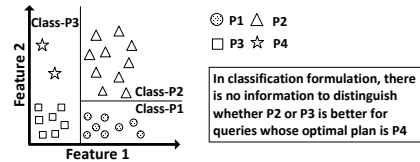

Figure 3: Example to illustrate limitation of classification formulation.

While the classification formulation appears to fit well for our problem, its objective of aiming to maximize classification accuracy may not align well with our sub-optimality metric for getPlan, as defined in Section 2.1. This is because the cost impact of misclassifying a query instance q_e to plan P_i, rather than the actual cost-optimal choice plan P_j, can be very high depending on P_i, P_j and q_e. Such classification problems where the mis-classification penalty depends on the particular example (rather than only the class label) are known as *example-dependent cost-sensitive* classification. While there exist research proposals to handle cost-sensitivity in binary classification scenarios, it is difficult to handle multi-class problems [43].

We present an example to illustrate this limitation of the classification formulation in Figure 3. It shows a set of training examples in a 2-dimensional selectivity space represented using markers for the corresponding best plan choice among P_1, P_2, P_3 and P_4. The figure also shows class boundaries based on a reasonably accurate classifier. Observe that, the examples with the best plan label P_4 have been classified as plan P_3, which may lead to higher misclassification penalty (i.e., sub-optimality) compared to an equally accurate alternative classifier that assigns the same examples to class P_2.

3.3 Regression formulation

In this formulation, we train individual regression models, one for each cached plan, to approximate its *optimizer estimated cost* given the input features of the query instance. To determine the best plan for a new incoming query instance, we invoke each regression model to predict the cost of each plan and select the minimum cost plan – as shown in Figure 2. Observe that, this formulation is a low overhead (one to two orders of magnitude faster) approximation of the accurate method of re-costing each of the cached plan to determine the best plan[3]. We expect the regression formulation to perform better since it has full cost information about all cached plans, unlike classification that utilizes only best plan labels.

3.4 Discussion

3.4.1 Model training. Interestingly, in our context, the labeled input data required for training both classification and regression formulation is the same. The cost of each of the cached plans on the query instances in the workload, which directly serve as labels in the regression formulation[4], and are also needed to generate the best plan label for the classification formulation (Figure 2).

We have experimented with several ML models including simple linear models, support vector models, as well as tree-based non-linear models such as Decision Trees and XGBoost [15] models. The overall cost of training a model is typically dominated by the labeling cost, i.e., using re-cost calls for each of the cached plans for the given representative set of queries. The actual model training time given the labeled data is relatively small (typically around 100ms in our evaluation). Finally, we note that this training step is performed offline and can be re-invoked when data or workload characteristics change significantly, or when plans are added or removed from the plan cache (see discussion in Section 5.3).

3.4.2 Classification vs. regression. While both classification and regression models are significantly better in sub-optimality compared to prior work, we also empirically compare these two techniques (Section 6.2). We found that for small memory budgets and training data sizes, regression models result in better sub-optimality, particularly at higher percentiles such as 95th percentile, and its getPlan (and model training) time is only slightly higher than classification models.

---

[3]While an approximate and efficient recost method could potentially be implemented inside a DB engine as well, use of regression models has the advantage of being non-intrusive, supporting a trade-off between accuracy and memory overhead, and adapt to different cost metric such as execution cost, as discussed later in Section 4.

[4]We use log-transformed version of cost values to emphasize regions where the plan has small cost values and is expected to be the best among the cached plans.

4 EXECUTION COST BASED GETPLAN

The supervised ML techniques for getPlan proposed in Section 3, like all prior PQO work, aim to select the cached plan with the lowest optimizer-estimated cost. This can, however, lead to execution sub-optimality since it is well-known ([13, 33]) that even when a plan P_1 is lower in terms of optimizer-estimated cost than plan P_2, it can sometimes be worse in terms of execution cost metrics such as CPU time or elapsed time. This can happen either due to erroneous cardinality estimates for intermediate query expressions [33] or inability of optimizer cost model to represent the current execution environment. Thus, a getPlan that can choose the best plan according to execution cost can be beneficial. Note that while cardinality estimation and cost modeling for arbitrary SQL queries are very difficult problems, selecting the best among K cached plans in terms of execution cost is a simpler problem since it involves a fixed query template and a small, fixed set of execution plans.

One straightforward approach is to reuse the supervised learning method in the previous section by simply replacing optimizer cost based label with *execution cost labels*. However, this approach is impractical in terms of generating labeled training data since it would require executing every training query instance with each of the K cached plans, some of which can be highly sub-optimal[5].

4.1 Approach: Leveraging execution feedback

We propose a novel formulation for adapting getPlan module to execution costs by leveraging the execution cost information (CPU time, elapsed time) obtained when previous query instances of the same parameterized query execute. Such information can be captured through standard logging capabilities in commercial DBMSs [6, 7].

Contextual multi-arm bandits: We formulate getPlan for execution costs as a *contextual multi-arm bandit (CMAB)* problem. The CMAB framework falls under reinforcement learning (RL) paradigm[6]. While supervised learning involves offline model training using a set of pre-labeled training examples, a bandit technique learns from feedback across multiple iterations, as we process more examples. In each iteration: (a) it processes input (called *context*) by taking an *action* that involves selecting one out of a fixed set of candidates (called *arms*), (b) it observes the *reward* generated for that action, and (c) it uses the reward to influence the future actions for similar inputs (contexts) - the last step may be done in batches. Considering plan selection as *action* and execution cost feedback as the *reward*, this technique is well suited for our application. It is worth highlighting that the technique neither requires taking optimal actions for initial inputs, nor do sub-optimal actions need to be explicitly corrected. We show that the bandit technique can be implemented in practice by making simple extensions to the regression formulation described in the previous section.

4.2 getPlan formulation

CMAB is a generalization of the popular *multi-arm bandit* technique [40] due to the extra context information as input. It has

---

[5]The best plan label for classification formulation can be collected by executing the K plans in parallel until one of them finishes. However, the absolute overhead can still be quite large.

[6]An intuitive difference is that generic RL has to deal with *delayed* rewards for actions while bandit formulations work with *immediate* reward for their actions.

many real-world applications including website optimization and clinical trials [42]. Next, we provide formal definition of CMAB and then describe how to formulate the getPlan problem.

Bandit: Bandit B can be defined as a set of arms $k \in \{1, 2, ..K\}$, where each arm is defined by some reward function that maps context vector x_t to $r_{t,k}$ for each time step t until horizon T.

Policy: Policy π seeks to maximize its cumulative reward $\sum_{i=1}^{T} r_{t,a_t}$ by sequentially selecting one of the bandits arms, defined as taking action $a_t \in \{1, 2, ..K\}$ for each time step $t \in \{1, 2, ...T\}$. Policy π does this selection by using one or more ML models and updating their parameters (θ_t) at every time step t.

Arm Selection Process: In each round $t = \{1, 2, .., T\}$, policy π
- observes the $d-$dimensional context vector x_t
- uses models with parameters θ_{t-1} to select an action $a_t \in \{1, 2..k\}$
- receives a reward r_{t,a_t} for action a_t
- updates the model parameters to θ_t using $\{x_t, a_t, r_{t,a_t}\}$

Adaptation to getPlan problem: For a given query q_e, the K cached plans correspond to the K arms, the selectivity vector S (the parameter binding vector V) corresponds to the context vector, and (negative of) the reward for choosing i^{th} arm is given by the execution cost of q_e using the plan P_i. As more instances are processed, the bandit based getPlan can be expected to select plans with lower execution cost. Observe that we could adapt the contextual multi-arm bandit formulation for our K plan selection only because each of the K cached plans is a *valid alternative* for any instance of the parameterized query Q, and the only difference among the cached plans is their execution cost.

4.3 Thompson sampling for getPlan

Various arm (plan) selection strategies have been proposed in literature for bandit learners (see [42] for a survey). One important difference between strategies is how they balance the exploration vs exploitation trade-off. Intuitively, exploitation refers to making the best decision based on the data observed so far while exploration refers to deviating from the observed data to find potentially better decisions. Specifically, ϵ-greedy explores by selecting a random plan with a $\epsilon-$probability and exploits the feedback by selecting minimum cost plan with (1-ϵ) probability. Other strategies, such as Thompson Sampling (TS) and Upper Confidence Bound (UCB), implicitly balance the exploration vs. exploitation trade-off by choosing an action with the highest expected reward w.r.t. randomly sampled belief [11, 38]. We use *Thompson sampling* (TS) due to its efficiency as well as empirical success across problem domains [12]. Specifically, we use the *online bootstrapped version* [19].

Pseudo-code for Bootstrapped TS: The algorithm to process query instances based on the online bootstrapped TS strategy is given in Algorithm 1. The strategy maintains t supervised regression models for each of the K arms (i.e., plans). For every incoming instance q_e, we start with iterating over each plan, and choosing a model *at random* (out of t models) to predict the plan's execution cost for q_e. Then, the plan P_a with minimum predicted execution cost (i.e., maximum reward) is chosen to execute the query instance q_e. Next, the algorithm updates the getPlan method using the execution cost feedback. After execution of q_e using P_a, we obtain the true execution cost r_a^e, and the new observation $\{S^e, r_a^e\}$ is added to the training data of models corresponding to the plan P_a, where S^e

is the selectivity vector of q_e. Here, it is important to note that, the new observation is added to each model's training data (associated with plan P_a) but with a different weight factor decided by a gamma distribution. It is easy to see that the algorithm *explores* by using a randomly chosen model from the multiple models available per plan. Further, it *exploits* by selecting the plan with the minimum predicted execution cost, after the models have been trained with sufficient query examples.

Algorithm 1 getPlan using online bootstrapped TS

1: **for** each query instance q_e with context (selectivity) S^e **do**
2: //getPlan
3: **for** plan i in 1 to K **do**
4:    Select model j uniformly at random from [1,t]
5:    set $\hat{r}_i = \hat{f}_{i,j}(S^e)$
6: **end for**
7: Select action a = $\arg\min_i \hat{r}_i$
8: Execute q_e with P_a to obtain true execution cost r_a^e
9: //adapt getPlan using the execution cost feedback
10: add observation $\{S^e, r_a^e\}$ to the history of plan P_a
11: **for** model j in 1 to t **do**
12:    Sample observation weight w = Gamma(1,1)
13:    update $\hat{f}_{a,j}$ with new observation $\{S^e, r_a^e\}$ with weight w
14: **end for**
15: **end for**

4.4 Discussion

Avoiding the initial sub-optimal decisions: The bandit based algorithm makes randomized plan selection decisions in the initial phase. We empirically observed (Section 6.5) that the initial plan selections are worse when compared to selecting the cached plan with the lowest optimizer estimated cost. After the initial phase however, the contextual bandits quickly improve to a significantly better plan selection policy. We can avoid the sub-optimal decisions in the initial phase by using a simple mitigation strategy that replaces the randomized decision with the decision based on the optimizer-estimated costs (similar to approaches in the CMAB literature [39, 41]). While this simple modification reduces the sub-optimality in the initial phase of contextual bandit learning, it also has a side effect that the initial exploratory phase of the contextual bandits is now *biased*. Although there is a risk of delay in convergence due to this bias, our empirical results (reported in Section 6.5) indicate that this mitigation strategy largely retains the benefit of bandits while avoiding the initial risk.

Training and inference overheads: In terms of overhead spent on model training, the contextual bandit formulation: (1) trains multiple regression models per plan, and (2) it needs to retrain the models after each batch of queries. It is important to note that we use tree-based (decision tree or XGBoost [15]) models that are very fast in terms of training and the additional overhead spent on model training is worth the saving in execution times due to selection of better plans. Further, the overhead in terms of making the plan selection decision is the same as the supervised regression formulation, both require evaluating one model per cached plan.

Handling changes in the set of cached plans: Unlike supervised ML techniques that require re-training when the plan gets added, a contextual bandit technique automatically adapts when a new plan (or set of plans) is cached. While the sub-optimality may increase temporarily, we observe that the bandit learner adapts

quickly. Similarly, dropping a cached plan can be handled by a bandit learner in a straightforward manner by ignoring the dropped arm (plan). We empirically evaluate these scenarios in Section 6.5.

5 POPULATECACHE **ALGORITHM**

The goal of the populateCache method is to efficiently identify a small set of plans that can together help achieve small sub-optimality across a given workload W of query instances (see Section 2.1 for the exact definition). Our query log based approach has the following phases:

(1) `Plan-Collection` phase: We collect candidate plans by invoking the query optimizer for a subset of n queries in W, with worst case overhead of $n \times O_t$, where O_t is the average optimization time. Suppose m distinct plans are collected.

(2) `Plan-Recost` phase: We recost each candidate plan for each of the n queries, with worst case overhead of $n \times m \times R_t$, where R_t is the average plan recosting time. We use the term *plan-recost matrix* to refer to this information.

(3) `K-set identification` phase: We use the information in the *plan-recost matrix* to determine the set of plans to be cached for a given plan budget K. This phase has a worst case overhead of $\binom{m}{K} \times C_t$, where C_t is the time needed for computing the optimization metric for each candidate set of K plans.

For a query log of size n = 2000, and assuming m = 100 plans, and K = 5, the brute force implementation can take more than 10 hours! This analysis is based on our empirical observation that each optimizer call takes $\approx$100ms up to a few seconds, and each re-costing call takes $\approx$10ms -100ms. However, the largest fraction of time is needed to evaluate the $\approx$75 million candidate K-sets. In fact, for the maximum cost suboptimality metric, the K-set identification problem has been shown to be NP complete [20]. This overhead is clearly impractical. Therefore, in the rest of this section we discuss heuristics for reducing the running time in each phase above.

5.1 K-set identification: greedy search

Given the hardness of the problem for certain objective functions, we propose to use a simple greedy strategy for this phase (similar to [20]). The algorithm requires a candidate plan set, the plan-recost matrix, final set size K and metric function. It starts with an empty plan set and greedily adds a plan to this set in every iteration that minimizes the metric. This procedure continues until the desired set size (K) is reached. The complexity of the algorithm is $O(K \times m \times C_t)$ where C_t is the time needed to compute the metric value for a set of candidate plans, which is efficient once the plan-recost matrix is computed. In our evaluation, for 200 plans ($m = 200$), 2000 instances ($n = 2000$), $K = 10$ with metric as geometric mean sub-optimality, the greedy algorithm run time was less than a second. Details of our empirical evaluation of populateCache are described in Section 6.

5.2 Plan collection and recost phases

To reduce the overheads in these phases without hurting the quality of the solution significantly, we propose using only a small subset of query instances instances in each phase. We find that selecting around 100-200 instances *at random* from the query log works well in practice. This is based on the empirical observation that while picking query instances to optimize in a *random* order, the size of

collected plan set as well the best plan quality achievable using them stabilize quickly.

In the re-cost phase, the aim is to have enough re-costings so that the search strategy can make an informed decision about which plans to be included in the small budget of K. Similar to the plan collection phase we can selectively or randomly choose a smaller subset of instances to work with. Once again, we find empirically that selecting a set of around 200-400 instances *at random* yields good results for populateCache.

5.3 Handling changes to data and workload

The supervised ML techniques (see Section 3) assume that the data and workload, remain unchanged. When there are significant changes in data or workload, we need to re-invoke populateCache as well as the module for training ML models for getPlan. For the latter, we need to first generate the training data followed by model retraining for the supervised models, whereas bandit learners can adapt by re-training continuously, as empirically validated in Section 6.5. Data changes can be detected by leveraging the mechanism that exist in most DBMSs to trigger updates to statistics, whereas workload drift can potentially be tracked by adapting techniques from the literature [36]. Evaluation of these proposals for PQO is an interesting area of future work.

6 EXPERIMENTS

In this section, we evaluate: (a) getPlan for optimizer estimated cost using supervised ML models (described in Section 3). (b) populateCache algorithm (Section 5). (c) getPlan for execution cost using the contextual multi-armed bandit approach (Section 4).

6.1 Setup

Databases and parameterized queries: We work with two synthetic databases TPC-DS (100 GB) [2], TPC-H (skewed, 10 GB) [3] and one real-world customer database *REAL1* (97GB). All experiments are conducted on Microsoft SQL Server 2017 DBMS. In total, we evaluate 25 distinct parameterized queries across the databases.[7] More than half of these parameterized queries contain complex SQL constructs such as multiple SELECT clauses with UNION OR INTERSECT operator, sub-queries including those with EXISTS and NON-EXISTS clauses, and outer joins. Specifically for REAL1, the queries have between 5 to 10 joins (including outer joins). The rest of the parameterized queries are single block queries with joins of four or more tables.

Query instance generation: The number of parameterized predicates per query template vary from 3 to 16 (with a median of 9). For REAL1, the number of parameterized predicates vary between 8 and 16. For each parameterized query, we generate 2000 *distinct* query instances by instantiating the parameterized predicates with binding values. For each parameterized predicate, we randomly choose a value from a pre-determined space of bindings with exponentially distributed selectivities. Since all instances are distinct, the training and test sets for ML models have no overlap.

---

[7]In particular, we experimented TPC-DS queries [16,17,18,24,25,35,38,69,91] and TPC-H queries [2,5,7,8,9,10,16,20,21] and 7 queries from REAL1.

Model setup: Our experiments use XGBoost [15] models with 16 KB memory constraint and predicate selectivities as input features, unless specified otherwise.
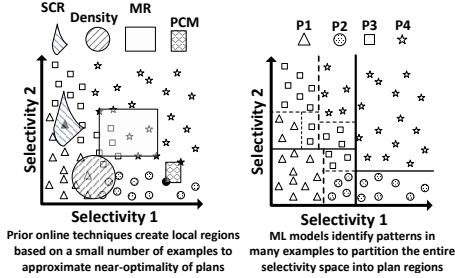

Figure 4: Past online techniques vs. ML based getPlan

Alternative PQO techniques: We compare with the following prior online PQO techniques: (1) Merging-Ranges (MR) [32], (2) PCM, (3) SCR [22] (4) Density[8]. Unlike our approach, these baselines interleave getPlan and populateCache in an online manner as query instances are processed. Figure 4 provides a visual comparison of plan regions created by the proposed ML based getPlan as well as the prior online techniques. To ensure fair comparison, we adapt evaluation of these techniques as follows (except for the end-to-end evaluation in Section 6.4): (a) we restrict all methods to the same fixed set of plans, (b) existing techniques refer to optimizer when their getPlan fails to pick one of the cached plan, we replace this optimizer call with a method that recosts the cached plans to return the best plan - it takes less time than optimizer call and retains the semantics of optimal plan choice.

Machine: All the experiments were conducted on a machine with a 2-socket, Intel(R) Xeon(R) E5-2660 CPU @ 2.60GHz, with 20 total cores, 192GB RAM and a 2TB local SSD.

Viewing Figures: We recommend using a colour printout since the figures use colours.

6.2 Optimizer cost based getPlan

We have three main parts of the experiment that demonstrate:

- Our getPlan method that uses supervised ML models can achieve cost sub-optimality equivalent to the best of the prior techniques while being orders of magnitude faster than them.
- While both classification and regression formulations produce getPlan with high accuracy, the regression formulation achieves better tail sub-optimality than classification when training data size is small.
- The sensitivity of ML models to memory budget, training data size, features, and model type.

The two metrics we use are the getPlan time and the optimizer estimated cost sub-optimality for getPlan (see Section 2.1).

Evaluation methodology: For each query template we found a set of good plans using the populateCache algorithm described in Section 5. The size of this set varies from 4-7 depending on the

---

[8]Our implementation of the density based plan selection [9] where its getPlan module finds the count of all plans in a selectivity neighborhood, and if there is a plan that appears more frequently than all other plans combined, return the plan. Otherwise, the best among all cached plans is returned.

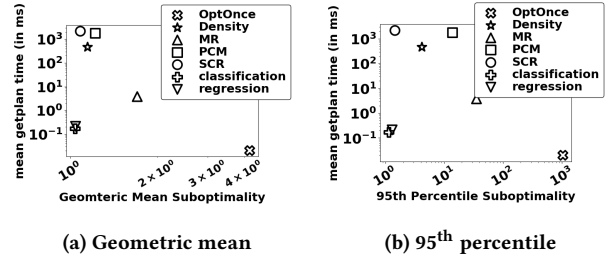| (a) Geometric mean | (b) 95ᵗʰ percentile |

Figure 5: Aggregate sub-optimality and mean getPlan time for various PQO techniques

query template. We then re-costed this plan set on all the 2000 instances of a query template. We use this plan-recost matrix for the experiment. Each PQO technique is run on 1600 instances per query template during which they construct their getPlan logic. Similarly, the supervised ML methods are given the same set of 1600 instances to train. We then evaluated the getPlan time and cost sub-optimality of each method on the remaining 400 instances per query template. We repeat this experiment 10 times for each query template with different subset of 1600 out of 2000 instances. In total, we collect the cost sub-optimality values and getPlan times over 400 instances × 10 repetitions × 25 query templates.

6.2.1 Sub-optimality vs. getPlan time. Figure 5a plots the mean getPlan time (in milliseconds) vs. the geometric mean of cost sub-optimality aggregated over all experiments across all parameterized queries. We see that both classification and regression based supervised ML techniques are significantly better in one or both dimensions when compared to all techniques evaluated, sometimes by orders of magnitudes. We observe similar behavior even with 95th percentile sub-optimality as shown in Figure 5b. As expected, Opt-Once has the worst sub-optimality, but the getPlan is fastest since the logic is trivial. Among other prior techniques, (i) Merging Ranges (MR) has relatively small mean getPlan time, but suffers from a significantly higher mean and tail cost sub-optimality, and (ii) SCR achieves near-optimal getPlan at the expense of very high getPlan time, due its reliance on invocation of the optimizer (and re-cost) calls during getPlan. The remaining techniques such as PCM and Density fall in-between the two, and are significantly worse than the ML methods in both dimensions. Classification and regression based getPlan are comparable in both 95th percentile sub-optimality (more details in Section 6.2.2) as well as getPlan time. Note that regression models have getPlan times comparable to classification version since the same memory constraint is enforced for both.

Next, in Figure 6a, for each technique we plot different percentiles ranging from 75th to 99th, of the sub-optimalities aggregated over all experiments. Supervised ML techniques are shown in red, while other techniques are shown in black. Both Opt-Once and MR suffer from sub-optimality >2 for more than 80% of the queries. While PCM, Density, and SCR pick near optimal plans for ≈ 90% of the queries, this comes at the expense of significantly higher mean getPlan time as noted earlier. Supervised ML methods outperform all prior techniques in terms of sub-optimality of plan choices.
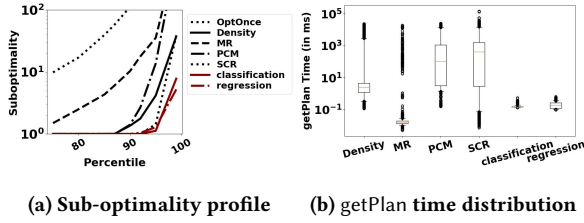
(a) Sub-optimality profile   **(b) getPlan time distribution**

Figure 6: Detailed comparison among PQO techniques

Figure 6b drills down into getPlan times, and uses boxplots to summarize the distribution for each technique. The getPlan technique using supervised ML models leads to low mean *and* variance. Prior techniques have much higher variance, with high getPlan time when they fall back to invoking the query optimizer. The techniques, in terms of the increasing order of optimizer invocations for getPlan are: supervised models (none), MR, Density, SCR/PCM. Note that SCR and PCM incur larger optimizer invocations due to their conservative approach that is reflected in their better sub-optimality profile in Figure 6a. Overall, supervised models have the smallest getPlan time as well best sub-optimality profile. Observe that, all the optimizer interaction required for the supervised ML methods happens in the offline training phase.

6.2.2 Classification vs. regression. As discussed in Section 3.4.2, when classification mis-classifies an instance it is not equipped to consider the sub-optimality of the replacement plan. Such misclassifications happen more often when training data is small, leading to sparse regions in a large selectivity space. With 100 training examples, classification models exhibit tail sub-optimality as large as 100, while regression models produce 5× better plans.

6.2.3 Sensitivity experiments for ML models. Next, we evaluate the sensitivity of supervised ML models with regard to cost sub-optimality metric. We vary model-size, training-size, model-type and features from their default values. We use regression models in these experiments, classification models show similar trends.
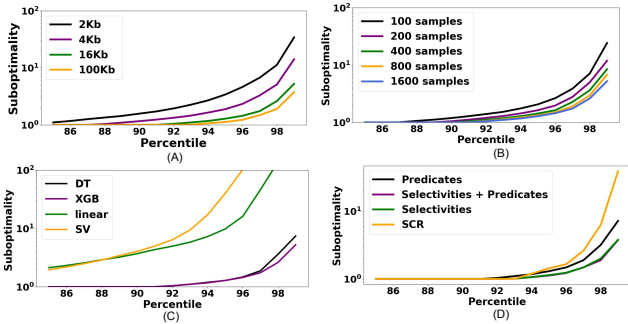

Figure 7: Sub-optimality impact of regression model parameters [(A) Model memory (B) Training size (C) Model types (D) Features]

Memory: Figure 7(A) shows that the sub-optimality improves as we increase the model-size from 2 KB to 100 KB with diminishing
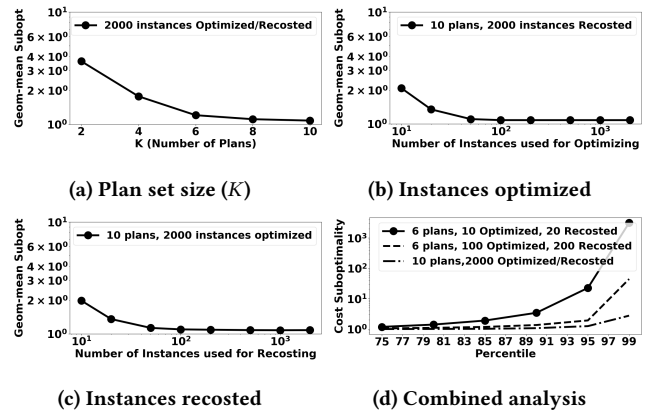

(a) Plan set size (K)   **(b) Instances optimized**

(c) Instances recosted   **(d) Combined analysis**

Figure 8: Sub-optimality impact of populateCache **parameters**

returns. Since a cached plan typically takes at least 100KB of memory, and often more for large plans, we expect our default value of 16KB to be acceptable in practice.

Training size: When the training size is increased from 100 to 1600, we find that the sub-optimality improves significantly as shown in Figure 7(B). We believe that 400 examples can provide a reasonable balance between training cost and sub-optimality, as more examples bring gains only in the very high percentiles.

Model type: With regard to model type, we found that tree based models, i.e., regression trees and their boosted versions (XGBoost), significantly outperform other simpler models such as linear and support vector models, as shown in Figure 7(C).

Features: We also evaluated multiple alternative feature sets: (a) selectivities only, (b) predicate bindings only, and (c) both – the results are summarized in Figure 7(D). We find that selectivity features are compact, accurate, available in most DBMSs and fast to compute. While we observe that only predicate bindings can also lead to reasonably accurate models[9], our evaluation did not find any significant advantage of using them as additional features.

6.3 Evaluation of populateCache

In this section, we evaluate the quality and efficiency of our offline populateCache module (Section 5). Our claim is that a relatively simple set of heuristics enables our technique to scale well to large input workloads while achieving low sub-optimality for the workload. Recall that the trade-off that populateCache navigates is the time spent in terms of optimizer and re-cost calls vs. the quality of set of plans identified. The parameters to the algorithm is the set size K, number of query instances to be optimized(N_{opt}), number of instances to be re-costed (N_{recost}). We perform a controlled study of the sensitivity of each parameter on quality and running time of the algorithm. We vary one parameter at a time, and set the others to a high default value. The default values for the parameters are $K = 10$, $N_{opt} = 2000$ and $N_{recost} = 2000$. Recall from Section 2.1 that for any populateCache algorithm we can define its sub-optimality with respect to an *ideal* (potentially unattainable)

---

[9]It may be interesting to further explore scenarios where selectivity computation is not feasible, e.g., parameterized predicates on views or user-defined predicates.

cost, i.e., the cost of the plan obtained if the query instance were optimized. We use this metric in our evaluation.

Evaluation methodology: For each of the 25 query templates, we collect the 2000 query instances. We optimized the 2000 query instances to obtain a set of distinct plans. In order to generate the complete plan-recost matrix, we re-costed each distinct plan over all 2000 instances. We use this data to run our algorithm while varying the set size(K), number of instances to optimize/re-cost(N_{opt}, N_{recost}). Since our algorithm chooses instances randomly, we repeat the experiment 10 times. We then evaluate the plan set returned by our algorithm on the entire set of 2000 instances. We report the geometric mean over all these points.

Varying number of plan to cache (K): Here, we vary K while fixing both N_{opt} and N_{recost} to 2000. Since we provide the entire plan re-cost matrix, in effect this experiment evaluates the quality of the greedy algorithm. Given the entire plan re-cost matrix, the greedy search algorithm runs within a second. The plan re-cost matrix generation time dominates the greedy run time. Hence, we focus on the sub-optimality of the plan set chosen by the algorithm. Figure 8a shows that the greedy algorithm significantly improves sub-optimality when we vary the set size (K) from 2-10, and is able to achieve a geometric mean sub-optimality below 1.2, with around 6 plans.

Varying number of instances for re-costing/optimizing: In the two experiments below, we fix K to 10. In Figure 8b, we vary the N_{opt} from 10 to 2000, while fixing the N_{recost} at 2000. We find that choosing 100-200 instances to optimize for obtaining candidate plans works well across all parameterized queries we evaluated, and there are diminishing returns of optimizing more instances. In Figure 8c, we vary N_{recost} from 10 to 2000, while fixing N_{opt} to 2000. Once again, we observe diminishing returns in sub-optimality. Choosing 200-400 instances for re-costing works well across all parameterized queries we evaluated.

Combining parameters: Finally, Figure 8d shows that the sub-optimality of our algorithm with the parameter choices ($K = 6$, $N_{opt} = 100$, $N_{recost} = 200$) is comparable with the maximum parameter values possible ($K = 10$, $N_{opt} = 2000$, $N_{recost} = 2000$) until 95th percentile, and significantly better than ($K = 6$, $N_{opt} = 10$, $N_{recost} = 20$). The greedy search time is negligible compared to the optimizer and recost calls.
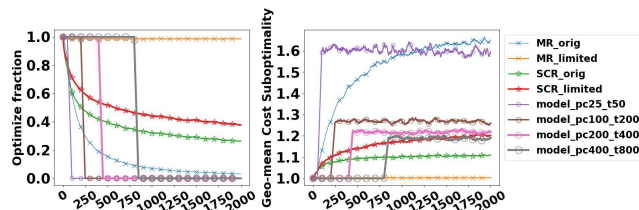

Figure 9: End to end comparison of optimizer cost based getPlan **with prior online PQO techniques**

6.4 End-to-end evaluation

We compare the proposed PQO techniques with SCR [22] and Merging Ranges [32], for the same sequence of query instances and report

the sub-optimality for each technique with respect to Opt-Always. Therefore, it captures the *combined* sub-optimality introduced by populateCache and getPlan. Since prior techniques are allowed to invoke the optimizer to obtain a new plan, we also report the optimizer overhead (left chart), i.e., fraction of instances for which an optimizer call is made, in addition to the sub-optimality metric (right chart), in Figure 9. We use the terms *MR_limited* and *SCR_limited* to denote adaptions of prior techniques that are allowed to make optimizer calls, but are restricted to use K cached plans only to ensure a fair comparison.

We report multiple variants of our approach where *model_pcX_tY* represents regression models trained using X and Y instances for populateCache and getPlan training, respectively. For bootstrapping, we use Opt-Always before the supervised regression model for getPlan is trained, however any other technique with lower optimizer overhead could also be used. We make the following observations: (a) *SCR_limited* is a conservative approach, and leads to lowest sub-optimality. However, it invokes the optimizer for more than 40% of the queries. (b) *MR_orig* invokes the optimizer much less often compared to *SCR_orig*, but the optimize fraction rises close to 1.0 for *MR_limited* due to the limit on plan cache. Even while using more cached plans, *MR_orig* faces steep increase in mean sub-optimality because its getPlan is based on simplifying assumptions that often fails to hold. (c) Regression based models lead to even worse sub-optimality than *MR_orig* when we use a small number of instances ($X = 25$ and $Y = 100$) for populateCache and getPlan training. However, it improves quickly with increase in size of query logs used for training, achieving reasonable accuracy with $X = 100$ and $Y = 200$. Further increasing training data leads to only marginal gains in mean sub-optimality. The initial optimizer overhead amortizes quickly as no optimizer invocations happen after getPlan is trained. Note that offline overhead spent in populateCache and training models are not shown in Figure 9. Overall, our techniques deliver low sub-optimality compared to Opt-Always with small offline training cost.

6.5 Execution cost based getPlan

The key claims of this experiment are:

- The bandit learners converge to picking plans with lower execution cost compared to when the plan with the lowest optimizer cost is picked.
- The sub-optimality of bandit learner in the initial phase can be largely mitigated by choosing the best plan according to optimizer estimated cost for the initial phase, and then switching to execution cost based getPlan.
- The bandit learner adapts gracefully to workload drift, addition and removal of plans, as well as noise in execution time.

Evaluation methodology: Similar to the previous section, for each query template we found a set of good plans using the populateCache algorithm described in Section 5. The size of this set varies from 4-7 depending on the query template. We generate 2000 query instances for evaluating the getPlan policy. We provide a sequence (a random permutation) of the 2000 query instances to the getPlan policy, and for each instance it chooses a plan from the fixed set of plans. This chosen plan is then executed, and the feedback provided to the policy is this execution cost. The getPlan

policy may use this feedback to improve its future decisions. We repeat this process 50 times using different random sequences.

Training models for contextual bandits: We use exponential batch sizes (10,20,40,80,160..) for training the bandit learner, i.e., train after processing 10,30,70,150... query instances. The training time for the models is approximately 10ms per batch, and even smaller for the initial batches. In practice, the cost of retraining models for the bandit learner does not add significant overheads.

Sub-optimality metric: Recall from Section 2.1 that execution cost sub-optimality of a getPlan policy is the ratio of execution cost (i.e., CPU time or elapsed time) of the plan chosen by the policy divided by the execution cost of the *best* plan in the set. We report the execution cost sub-optimality metric using CPU time metric.

We compare among the following strategies: (a) **optimizer-cost-baseline**: It picks the best among the K cached plans based on the optimizer estimated costs. (b) **bandit-random**: It uses bandit based getPlan for all query instances, therefore, makes randomized decisions for the initial queries in the sequence. (c) **bandit-mitigated**: It uses the *optimizer-cost-baseline* for the first 50 instances in the sequence and then switches to the bandit based getPlan learned with the execution cost feedback from first 50 instance as initial data. (d) **bandit-exec-bootstrap**: It picks the best plan in terms of *execution cost* for the first 50 instances, which is found by executing all K plans in parallel until the best plan finishes, and then switches to bandit based getPlan.

Sub-optimality evaluation: The bandit learners use execution cost feedback obtained from a sequence of query executions to improve their decisions. For each experiment, we evaluate the strategies with a random sequence of 500 query instances from a specific query template and collect the execution cost sub-optimality of the 500 getPlan choices. We repeated this experiment with 25 Query templates and 50 different random sequence of instances. In total, there are 25×50 such experiments and in each experiment each strategy makes 500 plan choices. We aggregated the geometric mean and tail sub-optimality for each of the 500 plan choices across the 25×50 experiments. We plot the aggregated metrics in Figure 10. To observe the trends clearly, we report the sliding window average over 25 instances.

Note that the execution cost sub-optimality of *optimizer-cost-baseline* is large, and does not improve even after processing many queries. The *bandit-random* strategy starts with significantly suboptimal decisions, however starts converging to a better policy than *optimizer-cost-baseline*, as more execution feedback is collected. The *bandit-mitigated* strategy is able to largely avoid the initial phase of sub-optimal decisions, and exhibits similar sub-optimality profile as the *bandit-random* strategy after ~100 queries, despite the *biased* start. Both the bandit strategies converge to small sub-optimality profiles than *optimizer-cost-baseline*. The trends are similar for both the geometric mean and tail (95th percentile) sub-optimality. Finally, while the expensive baseline *bandit-exec-bootstrap* converges to a slightly better sub-optimality than *bandit-mitigated*, the mean sub-optimality in the initial phase is too high to be practical.

6.5.1 Evaluating convergence of bandit learners. We now evaluate factors that may impact convergence pattern of bandit learners.

Impact of workload drift: Figure 11a demonstrates the ability of bandit learners to adapt to changes in workload distribution.
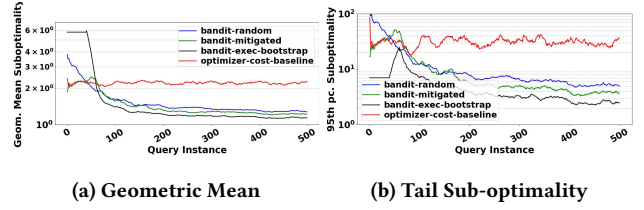

(a) Geometric Mean   (b) Tail Sub-optimality

Figure 10: Execution sub-optimality of bandit learners

(a) Impact of workload drift   (b) Impact of plan set drift

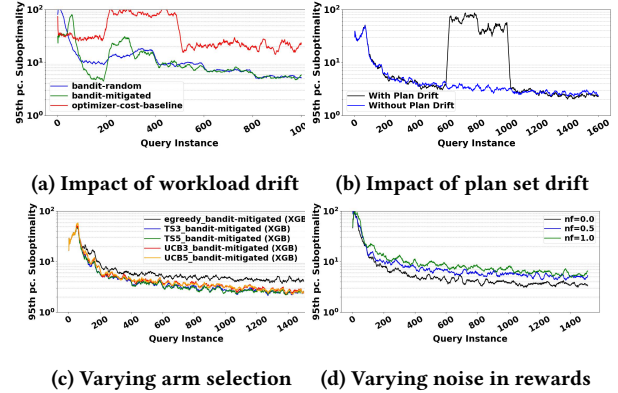(c) Varying arm selection   (d) Varying noise in rewards

Figure 11: Impact of (a) workload drift (b) plan set drift (c) arm selection methods and (b) noise in execution rewards, on the convergence of bandit learners

In this experiment, we evaluate bandit learners using a specialized sequence of query instances constructed to simulate workload drift. We first picked the top-3 selectivity dimensions that have largest variance in our workload and construct the query sequence by picking the first 200 queries with smallest selectivities, next 300 queries with medium selectivities, and then queries with largest selectivities towards the end. The sub-optimality profile of optimizer-cost-baseline provides evidence that the above design leads to abrupt and significant changes in the workload distribution, providing an adversarial challenge to the bandit learners. We observe that the bandit learners already start improving the sub-optimality of getPlan choices for the first 200 queries but the tail sub-optimality profile degrades quickly as the workload distribution shifts from small to medium selectivities. The bandit learns start adapting again and converge to small sub-optimalities as more queries are processed. While *bandit-random* faces less disruption than *bandit-mitigated*, we believe that the *bandit-mitigated* is still the preferred variant since it converges quickly when the workload is stable and adapts reasonably well to the workload shift.

Handling plan changes: We evaluate how the contextual bandit adapts to changes in the set of plans cached. Figure 11b shows the sub-optimality impact of dropping half of the plans in the cache (chosen at random) after 500 queries, and later adding the same set of plans after 1000 queries. As expected, the sub-optimality degrades after 500 queries, however, after adding the plans back *bandit-mitigated* strategy quickly converges to its original performance, by leveraging the previously collected execution cost feedback.

Importance of Thompson Sampling: Next, we compare different arm (plan) selection strategies for bandit learners: (1) ϵ-greedy (with $\epsilon = 0.1$), (2) thompson-sampling (TS), and (3) upper-confidence-bound (UCB). In terms of getPlan overhead, both ϵ-greedy and TS are efficient and comparable to supervised regression models. Both TS and UCB incur relatively larger overhead in terms of memory and training, as they maintain multiple regression models per plan, e.g., TSk denotes TS with k models per plan. The tail sub-optimality performance of each bandit variant is reported in Figure 11c. We found that both TS and UCB outperform ϵ-greedy in terms of sub-optimality performance, demonstrating the advantage of TS or UCB. We pick TS due to its getPlan efficiency advantage over UCB.

Handling noise in execution time: The execution time of a query instance can vary depending on other queries that are executing in the system (this variation is less for the CPU time metric). Hence the feedback available to the bandit learner can be noisy. In order to evaluate the effectiveness of the bandit learner with noisy rewards, we scaled the actual execution time by a randomly generated value before providing it to the bandit learner. We multiply a constant noise factor(nf) with a randomly generated value from gamma distribution($Gamma(\alpha = 2, \beta = 2)$)[10]. Thus, the final execution time is $TrueExecCost \times nf \times Gamma(2.0, 2.0)$. We achieve different amounts of noise by changing this noise factor nf over the values $\{0.0, 0.5, 1.0\}$. Figure11d shows that as we increase despite the injection of significant noise the bandit learner improves over time, however its rate of convergence slows down.

7 RELATED WORK

PQO has been studied for over three decades leading to a large body of relevant research work. We start by discussing the techniques relevant to getPlan, which is the primary focus of this paper.

Prior work on getPlan: There have been several proposals with focus on efficient getPlan: Merging Ranges [32], PCM, Ellipse [10], density based clustering [9] and SCR [22]. However, they suffer from a few important limitations. First, they assume that the plan cost functions are linear and/or monotonic; these assumptions fail for complex SQL queries [37] and leads to sub-optimal plan selection - our empirical evaluation validates this limitation. Second, both getPlan and populateCache modules are intertwined during the online processing. As a result: (a) getPlan often falls back to an expensive optimizer call (and recost calls in [22]) causing large getPlan time especially for high-dimensional parameterized queries, and (b) the getPlan is continuously adapted as more queries are executed, making debugging difficult. In contrast, we propose to decouple populateCache from getPlan and use getPlan based on supervised ML models trained offline using data derived from query logs, thereby making no assumption about the plan cost functions. Empirically, we find these models make good decisions for getPlan. Our decoupled architecture also ensures that the getPlan time is consistently small.

Prior work on populateCache: A subset of prior works that aims to find a single *robust plan* across the selectivity space [14, 16–18, 27] is inherently limited in its ability to achieve near-optimal plan quality. Hence, most proposals focus on finding the set of

optimal plans across the full space of selectivities of parameterized predicates [25, 28, 30]. Research on *plan diagrams* and their reduction [29, 37] demonstrated that a small number of plans (~10) are sufficient to achieve near-optimality across the selectivity space. Follow up work on plan diagrams proved that the problem of finding a small set of near-optimal plans from the set of optimal plans across the selectivity space is NP-hard, and an efficient greedy algorithm is a reasonable alternative to brute force search [20]. Our empirical findings corroborate these observations, and we leverage a greedy search algorithm for populateCache. However, the prior works identified the small set of plans after enumerating all plans in a large multi-dimensional selectivity space, which has prohibitive cost beyond 5-6 dimensions [21]. In contrast, populateCache in online PQO techniques [9, 10, 22, 32] is opportunistic: a query is optimized only when the getPlan method finds that none of the existing plans is near-optimal for the new incoming query. Our approach can be viewed as a middle-ground between the above mentioned extremes: we utilize the query logs to guide our candidate plan generation in the relevant parts of the large selectivity space, and identify a small set of plans to seed the plan cache.

ML for query optimization: There have been many recent attempts to improve query optimization using ML techniques (see survey [31]), including techniques for index selection, knob-tuning, cardinality estimation etc. The closest prior work aims to use ML to improve plan selection for ad-hoc queries [34, 35], by using a custom designed neural network model to predict execution cost of an arbitrary execution plan. In contrast, we propose low overhead models to pick one of a small set of plans for an *individual* parameterized query - a much simpler problem compared to [34, 35]. Our work is also similar to prior work [23, 24] that emphasize use of *low overhead* ML methods to improve various aspects of query optimization.

8 CONCLUSION

We revisit the important problem of parametric query optimization in databases and propose three key ideas. First, we advocate decoupling the problem of identifying which plans to cache (populateCache), from the problem of selecting, for a given instance of the parameterized query, which plan to use from the set of cached plans (getPlan). Second, we develop techniques that leverage historical query logs for both populateCache and getPlan problems. Unlike prior work, our approach is applicable for arbitrary complex SQL without having to make any assumptions on the cost behavior of plans. For populateCache we present an efficient and scalable algorithm to choose a small set of plans to cache to minimize the aggregate sub-optimality across queries in the log. In contrast to prior techniques, for getPlan we are able to achieve both low latency and low sub-optimality using machine learning models trained on query logs. Finally, we also introduce a novel formulation of getPlan for PQO that leverages the knowledge of execution time information in query logs instead of using optimizer estimated cost to circumvent inaccuracies of the query optimizer's costing. Our solution adapts the contextual bandit formulation from reinforcement learning, and preliminary experimental results show that this approach results in lower execution cost when compared to the best solution for the traditional PQO formulation.

---

[10]The percentile values of $Gamma(\alpha = 2.0, \beta = 2.0)$ are 5th =0.7 ,25th=1.9 ,50th=3.35 ,75th=5.38 ,95th=9.4.

REFERENCES

[1] [n.d.]. https://scikit-learn.org/stable/modules/tree.html.
[2] [n.d.]. http://www.tpc.org/tpcds/.
[3] [n.d.]. https://www.microsoft.com/en-us/download/confirmation.aspx?id=52430.
[4] [n.d.]. Plan Caching in SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-plan-cache-object?view=sql-server-ver15.
[5] [n.d.]. Plan guides in SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance/plan-guides?view=sql-server-ver15.
[6] [n.d.]. Query Store in SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-query-store-object?view=sql-server-ver15.
[7] [n.d.]. SQL Server XEvents. https://docs.microsoft.com/en-us/sql/relational-databases/extended-events/extended-events?view=sql-server-ver15.
[8] [n.d.]. USE PLAN hint. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver15.
[9] Guenes Aluc, David DeHaan, and Ivan Bowman. 2012. Parametric Plan Caching Using Density-Based Clustering. In *ICDE*.
[10] P. Bizarro, N. Bruno, and D. J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE TKDE* 21, 4 (2009), 582–594.
[11] Alexandra Carpentier, Alessandro Lazaric, Mohammad Ghavamzadeh, Rémi Munos, and Peter Auer. 2011. Upper-confidence-bound algorithms for active learning in multi-armed bandits. In *International Conference on Algorithmic Learning Theory*. Springer, 189–203.
[12] Olivier Chapelle and Lihong Li. 2011. An Empirical Evaluation of Thompson Sampling. In *NeurIPS*.
[13] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *ACM SIGMOD*.
[14] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance Aware Optimization of Parameterized Queries. In *ACM SIGMOD*.
[15] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System *(ACM KDD)*. New York, NY, USA.
[16] Francis Chu, Joseph Halpern, and Johannes Gehrke. 2002. Least Expected Cost Query Optimization: What Can We Expect?. In *ACM PODS*.
[17] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. 1999. Least Expected Cost Query Optimization: An Exercise in Utility. In *ACM PODS*.
[18] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *ACM SIGMOD*.
[19] David Cortes. 2018. Adapting multi-armed bandits policies to contextual bandits scenarios. *CoRR* abs/1811.04383 (2018). arXiv:1811.04383 http://arxiv.org/abs/1811.04383
[20] Harish D, Pooja Darera, and Jayant Haritsa. 2007. On the Production of Anorexic Plan Diagrams. In *VLDB*.
[21] Atreyee Dey, Sourjya Bhaumik, Harish D, and Jayant R. Haritsa. 2008. Efficiently Approximating Query Optimizer Plan Diagrams. *PVLDB* 1, 2 (Aug. 2008), 1325–1336.
[22] Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2017. Leveraging Re-Costing for Online Optimization of Parameterized Queries with Guarantees. In *ACM SIGMOD*.

[23] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently Approximating Selectivity Functions Using Low Overhead Regression Models. *PVLDB* 13, 12 (2020).
[24] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates Using Lightweight Models. *PVLDB* 12, 9 (2019).
[25] Sumit Ganguly. 1998. Design and Analysis of Parametric Query Optimization Algorithms. In *VLDB*.
[26] Ahmad Ghazal, Dawit Seid, Bhashyam Ramesh, Alain Crolotte, Manjula Koppuravuri, and Vinod G. 2009. Dynamic Plan Generation for Parameterized Queries. In *ACM SIGMOD*.
[27] G. Graefe and K. Ward. 1989. Dynamic Query Evaluation Plans. In *ACM SIGMOD*.
[28] Arvind Hulgeri and S. Sudarshan. 2002. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *VLDB*.
[29] Arvind Hulgeri and S. Sudarshan. 2003. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *VLDB*.
[30] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1997. Parametric Query Optimization. *The VLDB Journal* 6, 2 (May 1997), 132–151.
[31] H Lan, Z Bao, and Y Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering* (2021), 86–101.
[32] Allison W. Lee and Mohamed Zait. 2008. Closing the Query Processing Loop in Oracle 11g. *PVLDB* 1, 2 (Aug. 2008), 1368–1378.
[33] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.
[34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *ACM SIGMOD*. 1275–1288.
[35] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019).
[36] Stephan Rabanser, Stephan Günnemann, and Zachary C. Lipton. 2019. Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift. *CoRR* abs/1810.11953 (2019). arXiv:1810.11953 [stat.ML]
[37] Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *VLDB*.
[38] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2017. A tutorial on thompson sampling. *arXiv preprint arXiv:1707.02038* (2017).
[39] Rajat Sen, Karthikeyan Shanmugam, and Sanjay Shakkottai. 2018. Contextual Bandits with Stochastic Experts. In *AISTATS*.
[40] Joannès Vermorel and Mehryar Mohri. 2005. Multi-Armed Bandit Algorithms and Empirical Evaluation *(ECML)*. Springer-Verlag.
[41] Chicheng Zhang, Alekh Agarwal, Hal Daumé Iii, John Langford, and Sahand Negahban. 2019. Warm-starting Contextual Bandits: Robustly Combining Supervised and Bandit Feedback. In *ICML*.
[42] Li Zhou. 2015. A Survey on Contextual Multi-armed Bandits. *CoRR* abs/1508.03326 (2015). arXiv:1508.03326 http://arxiv.org/abs/1508.03326
[43] Zhi-Hua Zhou and Xu-Ying Liu. 2006. On Multi-Class Cost-Sensitive Learning, In AAAI. *Proceedings of the 21st National Conference on Artificial Intelligence, Boston, MA*.