Parametric Plan Caching Using Density-Based Clustering

Güneş Aluç *, David E. DeHaan *, Ivan T. Bowman *

#University of Waterloo, *Sybase, an SAP Company

galuc@uwaterloo.ca, ddehaan@sybase.com, ibowman@sybase.com

Abstract—Query plan caching eliminates the need for repeated query optimization; hence, it has strong practical implications for relational database management systems (RDBMSs). Unfortunately, existing approaches consider only the query plan generated at the expected values of parameters that characterize the query, data and the current state of the system, while these parameters may take different values during the lifetime of a cached plan. A better alternative is to harvest the optimizer’s plan choice for different parameter values, populate the cache with promising query plans, and select a cached plan based upon current parameter values. To address this challenge, we propose a parametric plan caching (PPC) framework that uses an online plan space clustering algorithm. The clustering algorithm is density-based, and it exploits locality-sensitive hashing as a pre-processing step so that clusters in the plan spaces can be efficiently stored in database histograms and queried in constant time. We experimentally validate that our approach is precise, efficient in space-and-time and adaptive, requiring no eager exploration of the plan spaces of the optimizer.

I. INTRODUCTION

Query plan caching has been widely adopted by RDBMSs to eliminate the overhead of repeated optimization [1], [2]. For frequently executed queries in which query optimization consumes a significant portion of total query execution time, it is beneficial to cache and reuse the optimal query plan [1]. A query plan is a tree of relational algebra operators, each encapsulating some information about choice of algorithm and resource allocation [3]. The optimal plan is computed via cost-based evaluation of candidate plans, taking into account recent statistics on the data and the current system state. Unfortunately, optimization time can become a performance bottleneck, particularly for queries that are cheap to execute [4]. Plan caching addresses this problem; that is, query optimization can be bypassed if plans of frequently executed queries are cached and reused. A high level overview of plan caching is depicted in Figure 1: part (A).

In conventional plan caching, the generated plan is chosen based on the expected values of certain parameters that characterize the query, data and the current state of the system (e.g., predicate selectivities, data distribution, available memory, etc.); this is known as the least specific cost plan [5]. Unfortunately, if these parameters do not take their expected values during the lifetime of a cached plan, at the two extremes, either (i) a plan may be prematurely dropped from the cache due to a temporary change in parameter values, or (ii) a sub-optimal plan may be executed until the RDBMS discovers that the cached plan is not optimal for the current parameter values.

The former case yields poor caching efficiency, whereas the latter causes performance degradation due to executing poor plans. An alternative is to populate the cache with promising plans and select a cached plan based upon similarity of parameter values. This forms the main objective of our paper.

We achieve our objective by harvesting information about the optimizer’s plan choice for different values of parameters that characterize the query, data and the state of the system, known as the plan space of the query [6]. Even though previous work on robust query processing (RQP) [5], [7] and parametric query optimization (PQO) [8], [9], [10], [11] present some valuable ideas for plan caching, we note that plan caching needs to be precise, efficient and adaptive, and it should avoid redundant pre-processing. Precise decisions are essential because a bad decision could cause execution of an arbitrarily-bad query plan, which could cost much more than the alternative of invoking the optimizer. Plan caching must operate on a very limited space budget, and it is profitable only when its time overhead is negligible. Adaptivity is required because the optimizer’s plan choices may change due to variations in the workload, data or system state. Finally, eager approaches that fully construct plan spaces in advance are not practical. As discussed in Section VI, existing approaches are not tailored for these requirements.

This paper makes the following contributions to address these problems. We perform a qualitative comparison of candidate clustering approaches with respect to parametric plan caching (PPC). Based on this evaluation, we propose a PPC framework that uses a novel online plan space clustering algorithm. The clustering algorithm is density-based, exploiting locality-sensitive hashing as a pre-processing step so that clusters in the plan spaces can be efficiently stored in database histograms and queried in constant time.

Figure 1 shows the workflow in our proposed PPC framework. The RDBMS executes a workload consisting of instantiations of a fixed set of query templates. During execution,
the PPC framework obtains information about the plan spaces of the query templates in a lazy manner by mapping each query instance to a point in the plan space. Each plan space point is labeled with both the optimal query plan and that plan’s execution cost at that point; points are clustered by plan identifier. For each new query instance, the algorithm decides whether the corresponding plan space point belongs to any of the clusters. If so, the associated plan is reused from the cache (if present). If not, the query is optimized and its plan space point is fed back to the algorithm. Throughout this process, performance of the clustering algorithm is monitored to help decide which plans to evict from a full cache, as well as to detect significant changes in the underlying plan spaces due to fluctuations in workload, data characteristics, or system state.

II. FUNDAMENTALS AND PROBLEM DEFINITION

A. Fundamentals

A query template is a SQL query with explicit and implicit parameters [12] where the parameter degree of a query template is the total number of parameters. Explicit template parameters appear as placeholders in the query text and are replaced with application-supplied values at execution time. Implicit template parameters represent characteristics of the data or system state that the optimizer uses when selecting an execution plan. As examples, the optimizer might consider base table sizes, data distribution, cache size, and multiprogramming level. A query instance is an instantiation of a query template including the values of all explicit and implicit parameters.

Definition 1. Query Instance: An instance of query template $Q_1$ of parameter degree $m$ is an $m$-tuple $q = (v_1, \ldots, v_m)$ consisting of the actual values of the template parameters of $Q_1$.

By definition, the parameters of query template $Q_1$ fully determine the execution plan the RDBMS selects for any instantiation $q$. More formally, let $\phi_t$ denote the set of all instances of the query template $Q_1$, and let $\mathbb{P}$ denote the set of query plans. Then, the optimizer’s plan selection for instances of template $Q_1$ can be modeled as the function $\Omega_1 : \phi_1 \rightarrow \mathbb{P}$.

As outlined elsewhere [11], the query optimizer does not directly use the values in $\phi_1$. For example, when a query template containing parameterized predicate $\phi(v_1)$ is instantiated with $v_1 = 35$, the optimizer computes the best plan based on the selectivity estimation of the predicate $\phi(35)$, rather than directly examining the value 35. We use the term optimizer parameters to describe quantities that are directly used by the optimizer when selecting an execution plan.

Without loss of generality, we assume each optimizer parameter is normalized on $[0, 1]$. Then, $\Omega_1$ can be decomposed into two functions $f : \phi_1 \rightarrow [0, 1]^r$ and $\text{plan} : [0, 1]^r \rightarrow \mathbb{P}$ such that $\Omega_1 = \text{plan}(f(q))$. Here, $f$ represents a normalization pre-step mapping from the template parameters to $r$ optimizer parameters, while $\text{plan}$ is a model of the optimizer’s plan selection given particular values of the $r$ optimizer parameters.

Definition 2. Plan Space: For a query template $Q_1$, $r$ different optimizer parameters and a choice of plan $f(q) : \phi_1 \rightarrow \mathbb{P}$ such that $f : \phi_1 \rightarrow [0, 1]^r$ and plan : $[0, 1]^r \rightarrow \mathbb{P}$, the plan space of $Q_1$ for the selected optimizer parameters is the set of mappings induced by plan.

Consider $Q_1$ in Appendix A and the selectivities of predicates “s_date $\leq \langle v_1 \rangle$” and “l_partkey $\leq \langle v_2 \rangle$” as the optimizer parameters of interest. Figure 2 depicts the plan space of $Q_1$. Each color represents a unique query plan.

A workload history is a sequence of query instance executions, where each query instance belongs to a particular query template. For each execution, the workload history also tracks the optimizer’s plan choice and the cost of execution. This is formalized in the following definition.

Definition 3. Workload History: A workload history is a sequence of tuples from $\mathbb{Q} \times \phi \times \mathbb{P} \times \mathbb{R}^+$, where $\mathbb{Q} = \{Q_1, \ldots, Q_k\}$ is the set of query templates in the workload, $\phi$ is the union of the possible query instances of each query template in $\mathbb{Q}$, $\mathbb{P}$ is the set of query plans and $\mathbb{R}^+$ is the set of positive real numbers that denote the cost of query execution (with respect to some arbitrary cost metric).

B. Problem Definition

For brevity, we will assume that the workload consists of a single query template. The following definitions can easily be extended to the case with multiple query templates. Given a workload history and $r$ independent functions $f_1, \ldots, f_r : \phi \rightarrow [0, 1]$ that map the query instances in the workload history to some distinct optimizer parameters, the goal of parametric plan caching is to predict which query plan the query optimizer will pick for a new query instance $q$.

Predicate selectivities are critical inputs in modeling the costs of query plans and, hence, also in plan selection [13]. Consequently, even though our approach is applicable to any optimizer parameter, we focus here on predicate selectivities. For a given query instance the framework computes the predicate selectivities in the same way that the query optimizer makes its selectivity estimations, that is, by exploiting the formerly generated statistics on data. Incorporating other optimizer parameters is a topic of future investigation.
Hence, assume that each \( f_i \) maps query instances in the workload to the selectivity of a query predicate. Then, the plan space can be represented as an \( r \)-dimensional Euclidean space, where each point is annotated with both a query plan and the cost of execution (cf. Figure 2). The plan selection problem is then reduced to an online clustering problem in which each cluster represents a set of points in the plan space sharing the same query plan. When a new query instance \( q \) arrives, it is mapped to a point in the plan space and the algorithm then decides whether the new point belongs to any of the formerly identified clusters; if so, that cluster is returned.

The output of the algorithm is either a query plan \( P_i \in \mathbb{P} \) or NULL if a prediction cannot be made. Even though the algorithm should make as many predictions as possible, there is a trade-off in accuracy due to potentially incorrect predictions, and so the algorithm may choose to not make a prediction. Consequently, our algorithm tries to maximize recall for a given lower bound on precision, where precision and recall are defined as follows:

**Definition 4. Precision and Recall:** The precision of a series of plan caching predictions, where each prediction is a member of \( \mathbb{P} \cup \{ \text{NULL} \} \), is the ratio of the number of correct predictions to the total number of NULL-free predictions. Furthermore, recall is the ratio of the number of correct predictions to the total number of predictions.

There are only a finite number of points in the workload history. On the other hand, there are potentially infinitely many points that belong to a cluster. Therefore, cluster membership is predicted according to the following assumptions.

**Assumption 1. Plan Choice Predictability:** For every \( d \in \mathbb{R}^+ \), there exists a sufficiently large \( \chi \in (0, 1] \) such that for any pair of points \( (\vec{x}_1, \vec{x}_2) \) in the plan space, where \( d(\vec{x}_1, \vec{x}_2) \leq d \), \( Pr(\text{plan}(\vec{x}_1) = \text{plan}(\vec{x}_2)) \geq \chi \) holds\(^1\).

**Assumption 2. Plan Cost Predictability:** For every \( d \in \mathbb{R}^+ \), there exists a sufficiently small \( \epsilon \in \mathbb{R}^+ \) such that for every pair of points \( (\vec{x}_1, \vec{x}_2) \) in the plan space, where \( d(\vec{x}_1, \vec{x}_2) \leq d \), if \( \text{plan}(\vec{x}_1) = \text{plan}(\vec{x}_2) \), then \( \cos((\vec{x}_1, \text{plan}(\vec{x}_1)) \leq (1 + \epsilon)\cos((\vec{x}_2, \text{plan}(\vec{x}_2))) \) holds\(^2\).

Informally, plan cost predictability implies that optimal plan cost forms a relatively smooth surface over the optimizer parameter space. In regions where a particular plan is optimal, that plan’s cost is often a monotonic function of its parameter values; therefore, a small change in parameter values is expected to cause only a small change in cost. Similarly, plan choice predictability implies that plan spaces—such as the one illustrated in Figure 2—contain many points that are not close to any plan boundaries. The intuition follows from cost predictability—the relative smoothness of cost surfaces for the best plans in a region tends to limit surface intersections.

Predictability of plan choices and costs are simplifying assumptions that may not hold for all plan spaces. For example, the space of predicate selectivities could contain multiple points where a predicate is expected to be either a contradiction or a tautology; in regions near such a singularity point the plan choices and costs could vary significantly. Nevertheless, similar assumptions have been exploited in the past by other authors [11], [13]. In Appendix B we experimentally demonstrate that these assumptions hold over plan spaces generated from predicate selectivities of TPC-H-style queries.

### III. Preliminary Evaluation of Candidate Clustering Methods

Centroid, single linkage and density-based clustering are alternative clustering methods that could be used in parametric query plan prediction [14]. In this section, we highlight strengths and weaknesses of each and how they can be extended to achieve high precision and controllable recall, which are desired properties in plan caching. This evaluation helps us formulate our online plan space clustering algorithm.

Centroid-based clustering assigns a new point to the cluster with the nearest centroid, which is best suited for data consisting of spherical clusters [14]. Spheres are likely poor approximations for plan clusters since, as pointed out by Reddy et al., the plan diagrams of modern query optimizers tend to be very complex with plans spanning multiple non-contiguous regions or forming embedded regions within another [6].

Single linkage-based clustering assigns a new point to the cluster of the nearest point. Density-based clustering assigns a new point to the cluster with the most points within a predefined radius \( d \). Both single linkage and density-based clustering can handle clusters of arbitrary shapes [14]; however, single linkage clustering is more sensitive to outliers because it does not distinguish cases where the point of interest falls in the middle of a cluster versus near a cluster boundary. In contrast, when amended with some simple sanity checks on plan densities, density-based clustering becomes more conservative near cluster boundaries where prediction is unsafe. This property becomes crucial in our framework.

#### A. Quantitative Comparison

To experimentally validate the above observations, we selected a representative clustering algorithm for each clustering method and extended it with sanity checks that permit varying levels of trade-offs in recall for gains in precision. Each algorithm is initialized with a set of sample plan space points denoted by \( X \), where \( |X| = 1000 \) and the algorithm is asked to produce a prediction for a test point \( \vec{x} \). The results are reported after the experiment has been repeated 20 times over 1000 such test points. Following are the details of the algorithms:

a) **K-Means Predict:** Sample points in \( X \) are grouped by their query plan labels. Each group is independently clustered into \( c \) clusters using the k-means algorithm [15], where \( c \) is a user-specified integer. To predict the plan for point \( \vec{x} \), the algorithm finds the nearest cluster centroid and returns that cluster’s plan label. The algorithm returns NULL if the distance exceeds some user-specified radius \( d \).
b) Single Linkage Predict: To predict the plan for point \( \vec{x} \), the algorithm finds the nearest point in \( X \) and returns its plan label. The algorithm returns NULL if the distance exceeds some user-specified radius \( d \).

c) Density Predict: To predict the plan for point \( \vec{x} \), the algorithm identifies the subset of points in \( X \) within some user-specified radius \( d \) of \( \vec{x} \), and returns the highest-frequency plan label in the subset. The algorithm returns NULL if a user-specified confidence threshold \( \gamma \) is not met. We defer details on the confidence model until Section IV-A, but \( \gamma \) is translated into a lower bound on relative frequencies between plan labels in the subset. The intuition is that relative plan frequencies are an indicator for how close \( \vec{x} \) lies to a plan boundary.

The characteristics of the data and query templates used in the experiments are summarized in Appendix A. Figure 3 depicts the precision achieved by the three algorithms for different choices of radius \( d \), where \( c = 40 \) for k-means predict, and density predict is tested at \( \gamma = \{0.5, 0.75, 0.95\} \). Similar graphs of recall were omitted for space reasons.

Observe that the precision of k-means predict is relatively poor and tends to decline as radius \( d \) is increased. Increasing the cluster count in k-means predict improves precision, but as \( c \to \infty \) the algorithm behaves like single linkage predict.

Compared to k-means predict, both single linkage predict and density predict achieve both much higher precision and higher recall. In these experiments, precision and recall for single linkage predict are comparable to those for density predict with a low confidence threshold \( \gamma \). Increasing \( \gamma \) allows density predict to achieve much better precision—albeit with a decrease in recall—by avoiding unsafe decisions.

In summary, of the three clustering approaches we studied, density-based clustering consistently achieved the highest precision because its density-based sanity checks proved much more effective at avoiding bad decisions than the distance-based sanity checks of the other two methods. For this reason, we formulate our parametric plan caching framework based on density predict. The precise algorithm, denoted BASELINE, is given in Algorithm 1.

IV. DENSITY-BASED PLAN PREDICTION

In this section we describe how density-based clustering can be used as the basis for a practical plan caching algorithm. Algorithm BASELINE described in the previous section exhibits excellent classification precision, but it is neither efficient nor adaptive. The memory and computational budgets of an RDBMS plan caching component are small, yet BASELINE requires storing all of the points in sample set \( X \). Because the quality of the predictions depend on \( |X| \), the sample set may need to be large in order to yield acceptable recall for a high confidence threshold. Furthermore, BASELINE assumes that this sample set is provided a priori, and it has no mechanism to maintain the sample set over time; in particular, it lacks a mechanism to detect when characteristics of the sampled plan

![Fig. 3](image-url) Quantitative comparison of k-means predict, single linkage predict and density predict.

![Algorithm 1](image-url) BASELINE

Input: query plans \( P \), sample plan space points \( X \), unlabeled plan space point \( \vec{x} \), radius \( d \), confidence threshold \( \gamma \)
1: for all \( x_i \in X \) do
2: if \( |x_i - \vec{x}| \leq d \) then
3: \( \text{density}[\text{plan}(x_i)]++ \)
4: end if
5: end for
6: for all \( p_i \in P \) do
7: totalDensity \( \leftarrow \) totalDensity + density\[p_i\]
8: if density\[p_i\] > density\[max\] then
9: max \( \leftarrow p_i \)
10: end if
11: end for
12: ratio \( \leftarrow \) totalDensity/density\[max\]
13: if sin(getConfidenceAngle\(\text{(ratio)}) > \gamma \) then
14: return max
15: end if
16: return NULL

![Fig. 4](image-url) Probabilistic model describing plan optimality regions (left) and the confidence model used in plan prediction (right).
A. Confidence Model

Consider a two-dimensional plan space containing $n$ plans that satisfies the plan choice predictability assumption as follows: given any two points within distance $d = 0.1$, with probability at least $\chi = 0.9$ the two points have the same query plan $P \in \mathbb{P}$. Around each point we imagine a circle of radius 0.1 over which we evenly distribute the probability weight 0.9 that other points in that circle share plan $P$; this is illustrated in Figure 4(a) for point sets $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \vec{x}_5\}$ and $\{\vec{x}_6, \vec{x}_7, \vec{x}_8\}$ labeled by query plans $P_1$ (blue) and $P_2$ (red), respectively. Given a new point $\vec{x}$, our confidence in predicting the plan label for $\vec{x}$ should depend upon the intensities of the two colours at $\vec{x}$, with mixed colours demarking regions where it is not safe to make a prediction. We formalize this next.

For test point $\vec{x}$ and each plan $P_i \in \mathbb{P}$, let $c_i(\vec{x}, d)$ denote the number of sample points of plan $P_i$ within distance $d$ of $\vec{x}$. Let $\max \{1, \ldots, n\}$ denote the index of a plan with the highest sample count. For a sufficiently small $d \in \mathbb{R}^+$, if $\sum_{i=1}^n c_i(\vec{x}, d) = c_{\max}(\vec{x}, d) = \alpha \geq 1$, we assume that $\vec{x}$ is inside the optimality region of $P_{\max}$, where a larger $\alpha$ implies a greater confidence in plan prediction. Otherwise, if $\sum_{i=1}^n c_i(\vec{x}, d) > c_{\max}(\vec{x}, d) \geq 1$, then $\vec{x}$ may lie near a plan boundary and so prediction may be unsafe.

To assess prediction safety we consider the relative plan frequencies within fixed radius $d$. Let $\text{area}_i(\vec{x}, d)$ denote the true area of the optimality region of the query plan $P_i$ that lies inside the circle centered at $\vec{x}$. We assume that for small $d$, samples of different query plans are uniformly distributed within the circle; hence, $\text{area}_i(\vec{x}, d) \propto c_i(\vec{x}, d)$. Furthermore, we assume that the plan space boundary separating $P_{\max}$ from the other query plans is a line segment splitting the circle into two regions: one consisting of $P_{\max}$ and the other consisting of all the other query plans, as depicted in Figure 4(b). This is a reasonable simplification when $d$ is much smaller than the average width of the optimality regions. According to this model, $\vec{x}$ lies within the optimality region of $P_{\max}$ if and only if ratio $c_{\max}(\vec{x}, d)/\sum_{i=0,j\neq\max}^n c_i(\vec{x}, d) \geq 1$. When this condition holds, we use this ratio to compute angle $\theta \in [0, \frac{\pi}{2}]$ in Figure 4(b), and the confidence is then defined as $\sin(\theta)$.

B. Efficiently Representing Plan Space Points

A naïve approach for reducing the space and time overhead of BASELINE is to partition the plan space into a grid and within each grid bucket record both the total sample point count (a 32-bit integer) and the average plan cost (a 32-bit float) for each query plan. The density of plan $P_i \in \mathbb{P}$ at $\vec{x}$ can be approximated by (i) locating the grid bucket that contains $\vec{x}$, (ii) obtaining the total number of samples of $P_i$ in the bucket (and in the neighboring buckets if necessary), and (iii) dividing the total number of samples by the area of the bucket(s). We denote this algorithm as NAÏVE.

For a total of $n$ query plans and a choice of $b_q$ buckets per grid, the space overhead of NAÏVE is $n \times b_q \times 8$ bytes, while the algorithm performs $O(1)$ computations per plan prediction. Even though this is a significant improvement over BASELINE, NAÏVE has certain drawbacks. The density within a circular region may not be approximated accurately from grid buckets, especially since the point of interest may not lie at the center of the bucket. In Section V-A, we experimentally demonstrate that the prediction error of NAÏVE is too high to justify its space and time efficiencies.

Instead of relying on a single grid with fixed orientation, if one can produce multiple grids whose orientations are all randomized, the aggregate density computed from the grids will be more accurate. We achieve this behaviour by applying randomized locality-preserving geometrical transformations to the plan space points, which may alter the actual coordinates of the points, but not their relative orientation.

Consider a set of sample points from the plan space of $Q_1$ (Figure 2) after some randomized locality-preserving geometrical transformations in Figures 5(a)–(c). In each subfigure, the rectangular bucket that contains the reference point $A$ is highlighted. In the original plan space, these rectangular regions correspond to some arbitrary-shaped polygons (Figure 5(d)). Although individually each polygon is a poor approximation of the circular region around $A$, intersecting several such polygons yields a more precise approximation.

To this end, we extend a locality-sensitive hashing technique that Tao et al. proposed for nearest-neighbor search [16]. Let $\lambda$ be the radius of hypersphere $S$ whose volume is equal to the volume of an $r$-dimensional hypercube with coordinates $[-1,1]^r$. Every point in an $r$-dimensional plan space is first translated by the $r$-dimensional vector $\vec{m} = (-0.5, \ldots, -0.5)$ and then scaled by $\frac{\lambda}{\sqrt{r}}$ so that the new point lies in the hypercube $[-\frac{\lambda}{\sqrt{r}}, \frac{\lambda}{\sqrt{r}}]^r$. The vertices of this second hypercube lie on the surface of $S$. The points are then stretched until they span the volume of $S$, which minimizes the shrinking effects of the locality-preserving transformation. Next, $s$ different unit vectors $\vec{a}_1, \ldots, \vec{a}_s$ are constructed, where $\vec{a}_j = \frac{1}{\sqrt{a_{1j}^2 + \ldots + a_{sj}^2}}(a_{1j}, \ldots, a_{sj})$ and each component $a_{ij}$ of $\vec{a}_j$ is drawn independently from a normal distribution. Furthermore, $s$ variates of translation $b_1, \ldots, b_s$ are constructed such that
each \( b_j \) is drawn independently from \([0, \frac{1}{\Delta}]\), where \( \Delta \) is the grid resolution along a single axis. The new coordinates of the points in the \( s \)-dimensional space are computed by projecting the points onto each unit vector \( \vec{a}_j \) and shifting them by the corresponding variate of translation \( b_j \). Finally, the points are assigned to buckets based on their \( \vec{a}_j \cdot \vec{x} \) values.

While the translation described above is similar to the one used by Tao et al. [16], we note a few distinctions. Tao et al. desire the transformations to produce points that are as dispersed as possible, because in nearest neighbor search it is important that only neighboring points are hashed to the same value. To effect this, they select \( s \) such that \( s \gg r \), and \( b \) is generated over a large interval. In contrast, the plan caching context is more tolerant of hashing non-nearby plan space points to the same bucket, particularly if they belong to the same query plan. Therefore, we use \( s = r \) for low dimensions, and \( s \ll r \) when dimensionality reduction is necessary. Additionally, we generate \( b_1, \ldots, b_s \) over a much smaller interval, which suffices to inject some randomness without violating the plan choice predictability assumption.

In summary, for a particular query template, we apply \( t \) randomized transformations to the points in the plan space, producing \( t \) intermediate \( s \)-dimensional data spaces, namely, \( \mathbb{I} = \{I_1, \ldots, I_t\} \). Each \( I_j \in \mathbb{I} \) is partitioned by a grid of fixed resolution. Given point of interest \( \vec{x} \), the \textsc{Naive} algorithm is applied independently to each intermediate space \( I_j \) to obtain \( t \) different estimations for each of the plan densities surrounding \( \vec{x} \), and the median estimation is selected for each plan. We denote this algorithm as \textsc{Approximate-LSH}.

For \( t \) randomized transformations, \( n \) query plans and \( b_g \) buckets per grid, \textsc{Approximate-LSH} consumes \( t \times n \times b_g \times 8 \) bytes, which is \( t \) times as much as \textsc{Naive}. As demonstrated in the experiments in Section V-A, the improvement in precision justifies this increase in space.

**C. Creating Synopses of Plan Space Points**

If the total plan space point count varies smoothly across neighboring grid buckets, the space overhead of \textsc{Approximate-LSH} can be reduced further without significantly compromising prediction error. This is crucial for a query template with a high parameter degree, as the total number of buckets grows exponentially with the dimensionality of the plan space. To this end, we store in database histograms the distributions of the total number of points across the buckets induced over the plan space. Because database histograms are unidimensional data structures, we require a method to map multi-dimensional distributions to a single dimension.

To address this problem, we exploit \textsc{z-ordering}[17], a space-filling curve that preserves the locality of the data which was used similarly by Tao et al. [16]. Consequently, the data points in each \( I_j \in \mathbb{I} \) are linearized on \([0,1]\) according to their \textsc{z-orders}, and the distribution and plan costs are stored within database histograms. Figure 6 shows some sample distributions. A separate histogram is created for every query plan in the plan space. Therefore, for a plan space with \( n \) distinct query plans, a total of \( t \times n \) histograms are allocated.

For a particular plan space point \( \vec{x} \), let \( T_{ij}(\vec{x}) \) denote the linearized \textsc{z-order} coordinates of \( \vec{x} \) over the grid induced on space \( I_j \). We define the algorithm variant \textsc{Approximate-LSH-Histograms} as similar to \textsc{Approximate-LSH}, but when estimating the density (or plan cost) for each plan \( P_t \) in space \( I_j \), rather than looking up a count in a grid cell, the corresponding database histogram is queried on the range \([T_{ij}(\vec{x}) + \delta, T_{ij}(\vec{x}) - \delta]\). Quantity \( \delta \) is computed from the user-provided radius \( d \) such that \( 2\delta \) is equal to the volume of a hypersphere with radius \( d \).

For \( t \) randomized transformations and \( n \) query plans, the space overhead of \textsc{Approximate-LSH-Histograms} is \( t \times n \times b_h \times 12 \) bytes, where \( b_h \) denotes the maximum number of buckets in a database histogram. In comparison to the fixed cells in \textsc{Approximate-LSH}, each histogram bucket requires an additional 4 bytes to store the value of the bucket boundary. However, by using standard histogram construction techniques that choose boundaries to minimize estimation error, many similar adjacent cells can be consolidated into a single bucket. Hence, from the histogram storage we expect to realize substantial savings in space with only a small loss in precision, which is demonstrated in Section V-A.

The use of \textsc{z-order} introduces two changes. First, the ordering may place two distant points from the multi-dimensional space next to each other. To address this issue, \textsc{Approximate-LSH-Histograms} requires the plan space point density obtained from the histograms to be above a constant factor of the total number of plan space points in \( \mathbb{X} \). This sanity check is called \textsc{noise elimination}. In our evaluations, we use a fixed threshold.

Second, when the ordering splits a contiguous region into non-contiguous intervals, more histogram buckets are required to efficiently store the interval boundaries. For practical reasons, one would like to control the space overhead of the clustering algorithm by limiting the number of histogram buckets, which inherently places restrictions on precision and recall. The \textsc{Approximate-LSH-Histograms} algorithm relies on the \textsc{confidence sanity check} described in Section IV-A to maintain high precision.

Consider the coordinates \([0.65, 0.95]\) in Figure 6 for which a \textsc{z-ordering} has produced two non-contiguous intervals for plan \( P_3 \), where the gap is filled up by \( P_9 \). The problem arises
when a histogram bucket of $P_3$ spans the two non-contiguous intervals due to space restrictions. If queried for values outside the actual interval boundaries—say, at 0.8—the histogram will return positive values for $P_3$, which is not accurate. However, with the confidence sanity check, a plan prediction is made if and only if the plan’s density is significantly greater than the sum of densities of all other plans in the region. In this example, the presence of $P_3$ around 0.8 will prevent the algorithm from making such incorrect predictions.

D. Online Query Plan Prediction

In an online version of the problem, $X$ is empty when execution of workload $W$ begins. For each query instance $q \in W$, the prediction algorithm decides whether a cached plan can be used. If not, the query is sent to the optimizer, after which the generated query plan is saved in the plan cache and the corresponding plan space point $\vec{x}$ for $q$ is added to $X$.

In this execution model, plan predictions are delayed until the algorithm has obtained sufficient input. From a practical point of view, this window is short; nevertheless, we propose a small extension to shorten the warm-up period further and to achieve higher precision. The idea is to invoke the optimizer with some probability even if the algorithm produces a prediction. This probability is a function of a user-defined mean invocation probability and the confidence value computed by the algorithm for that query instance. As experimentally demonstrated in Section V-B, even a small mean invocation probability such as 10% can shorten the warm-up period.

Whether or not to use the algorithm’s own plan predictions as input is non-trivial. Positive feedback—inserting predictions believed to be correct into sample pool $X$—shortens the warm-up period but has the potential to produce an avalanche of false positive input, and so our algorithm does not use it. Negative feedback—correction of the underlying prediction models to remove support for an erroneous prediction—is safer and improves precision and possibly recall, but it begs the question how the algorithm can identify its own mistakes, since the existence of the prediction implies the confidence sanity check was satisfied. We exploit the plan cost predictability assumption to identify prediction errors, with details given in the next section. When a prediction error is identified at plan space point $\vec{x}$, the optimizer is immediately invoked to obtain the true query plan at $\vec{x}$. The newly-optimized point is added to sample pool $X$ by inserting it into the database histograms, which changes the relative plan densities computed for the vicinity of $\vec{x}$, thereby reducing support for the previously predicted plan.

E. Estimating Precision and Recall

Whenever a plan prediction $P_i \in P$ is made for query instance $q$ corresponding to plan space point $\vec{x}$, we track the execution cost of $P_i$ at $\vec{x}$, denoted $cost(\vec{x}, P_i)$. The histograms corresponding to $P_i$ store the average execution cost for points within each bucket, and so the average execution costs of the sample points around $\vec{x}$ can be estimated via a histogram range query, similar to the estimation of plan densities described previously. Because we do not exploit positive feedback in our approach, the costs summarized within the histograms are for only truly optimal plans. According to the plan cost predictability assumption, if $P_i$ is the optimal query plan at $\vec{x}$, the computed average cost should be within an error bound of $cost(\vec{x}, P_i)$. If the cost difference is beyond a threshold, we use the contrapositive argument to conclude that a false prediction has been made.

To this end, three sets of estimations are maintained: $prec_k[P_i]$ keeps track of the precision of the last $k$ predictions of each query plan $P_i \in P$, while $prec_k[Q_i]$ and $rec_k[Q_i]$ store the overall precision and recall, respectively, of the last $k$ predictions made over each query template $Q_i \in Q$. Note that by definition, $rec_k[Q_i] = \beta(Q_i) \cdot prec_k[Q_i]$, where $\beta(Q_i)$ is the ratio of the number of NULL-free predictions to the total number of predictions made for the last $k$ instances of $Q_i$, which can easily be measured.

These estimations enable us to both identify query plans with caching potential and to detect changes in the optimizer’s plan selections and, hence, to abort plan caching if necessary. If precision estimations for a query template falls below a threshold, we drop all histograms created for that query template and start accumulating sample points from scratch.

V. Evaluation

We start by evaluating the trade-offs of the approximations made over the BASELINE algorithm (Section V-A). Next, the online performance of the algorithm is evaluated with respect to precision and recall (Section V-B), which is followed by a simulation of the end-to-end performance improvements (Section V-C). The section is concluded with the potential extensions to handle changes in sampled plan spaces (Section V-D). In the experiments, a modified TPC-H schema is used, while the parameter degrees of the query templates range from 2–6. The full experimental setup is described in Appendix A.

There are two types of workflows. In the offline workflow, the algorithm is first “warmed-up” with points that are uniformly sampled from a plan space ($X$). An independent set of test points are produced in the same fashion ($T$).

In the online workflow, the algorithm slowly uncovers the plan space of a query template as it executes a test workload consisting of a sequence of 1000 query instances. To generate the test workload, which is called random trajectories, a cursor is moved along 10 independent, randomly produced trajectories over the plan space. The test points are selected
such that their distance to the cursor follows a Gaussian distribution with $\mu = 0$ and $\sigma = r_d$, where $r_d$ is enumerated with values from $\{0.01, 0.02, 0.04, 0.08\}$. Figure 7 visualizes a sample workload.

A. Efficient Approximation of the BASELINE Algorithm

One needs to increase the sample point count exponentially to maintain a desired level of accuracy at higher dimensions. For this reason, the BASELINE algorithm, whose performance is a function of $|X|$, cannot scale. On the other hand, the three approximations over BASELINE—namely, NAÏVE, APPROXIMATE-LSH and APPROXIMATE-LSH-HISTOGRAMS—have complexity which solely depends on $|T|$; hence, they are scalable (cf. Table I).

NAÏVE, APPROXIMATE-LSH and APPROXIMATE-LSH-HISTOGRAMS achieve the desired efficiencies in time by varying levels of compromise in plan prediction accuracy. In this section, we demonstrate that with locality-sensitive hashing and the summarization of plan spaces in database histograms, prediction error can be reduced to tolerable levels.

We first compare NAÏVE and APPROXIMATE-LSH against BASELINE in an offline workload with $|T| = 1000$, where $\gamma = 0.7$ is used as the confidence threshold, the query radius is selected as $d = 0.05$, and each algorithm is allocated a controlled space budget.

As depicted in Table I, the space consumption of each algorithm depends on various parameters. For example, given a budget of $M$ bytes, BASELINE can only use a sample of size $|X| = \frac{M}{T}$. The space consumption of the remaining algorithms is independent of $|X|$, hence, we test them at $|X| = \{200, 400, 800, 1600, 3200, 6400\}$. Furthermore, NAÏVE can use more grid buckets than APPROXIMATE-LSH (depending on the number of randomized transformations $t$ used in APPROXIMATE-LSH). In this experiment, $t$ is enumerated over $\{3, 5, 7, 9, 11\}$. For each algorithm, the mean values of precision and recall are reported.

The above experiment demonstrates that plan prediction errors of NAÏVE and APPROXIMATE-LSH are—indeed—higher than BASELINE. For queries with larger number of parameters, the prediction error of NAÏVE is too high to be practical for plan caching, whereas APPROXIMATE-LSH has tolerable error. Consider the two extremes in Figure 8. NAÏVE and APPROXIMATE-LSH achieve very similar precision for $Q_1$; on the contrary, in $Q_7$, the precision of NAÏVE can be as bad as $30\%$, while APPROXIMATE-LSH portrays a performance much closer to that of BASELINE. Despite the fact the grid buckets in APPROXIMATE-LSH are less granular for a fixed space budget, this algorithm is making predictions over multiple, randomly selected regions around the query point, hence, it is less sensitive to bucket misalignments. In fact, APPROXIMATE-LSH is choosing not to make plan predictions whenever there are conflicting density values. Clearly, this implies a compromise in recall, which is illustrated in Figure 8.

Next, we validate that using standard database histograms, precision can be improved even further. The experiment compares APPROXIMATE-LSH and APPROXIMATE-LSH-HISTOGRAMS using the same experimental configuration as before. For $Q_5$, Figure 9 depicts the improvement in precision, while there is a seemingly unexpected compromise in recall. The precision improvement stems from the fact that standard database histograms rely on techniques that choose boundaries to minimize estimation error, whereas in the fixed grid partitioning, the buckets may not be aligned with the clusters. The drop in recall is a consequence of $z$-ordering—specifically, the “false negatives” phenomenon discussed in Section IV-C—and the fact that the confidence sanity check may impose NULL predictions when histogram buckets have to span multiple non-contiguous regions due to space limitations. This trade-off in recall can be controlled, as discussed shortly.

Table II depicts how precision increases as the confidence threshold is increased. The experiment is performed over $Q_1$ with $|X| = 3200$ samples, where $b_h = 40$, $t = 5$ and the

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Space (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>$O(</td>
<td>T</td>
</tr>
<tr>
<td>NAÏVE</td>
<td>$O(</td>
<td>T</td>
</tr>
<tr>
<td>APPROXIMATE-LSH</td>
<td>$O(</td>
<td>T</td>
</tr>
<tr>
<td>APPROXIMATE-LSH-HISTOGRAMS</td>
<td>$O(</td>
<td>T</td>
</tr>
</tbody>
</table>
results are averaged over query radii \( d = \{0.05, 0.1, 0.15, 0.2\} \). In the subsequent experiment, the effects of increasing the randomized transformations \((t)\) are investigated. Here, the confidence threshold is fixed at \( \gamma = 0.7\); otherwise, the configuration remains unchanged. We have observed that increasing \( t \) does not cause a significant change in recall. However, it improves precision (cf. Figure 10(a)), which is aligned with our expectations from locality-sensitive hashing. Furthermore, the improvement in precision at higher dimensions is more substantial. Finally, we study the effects of increasing the number of histogram buckets \((b_h)\), for which we fix \( t = 5\). As shown in Table 10, when the number of histogram buckets is increased, recall increases. Although not displayed here for space considerations, precision remains constant. This is an important feature of \textsc{approximate-LSH-histograms} because it implies that one can control the space utilization of the algorithm largely by reducing recall.

**B. Online Performance**

Performance of \textsc{online-approximate-LSH-histograms} is evaluated over random trajectories generated at \( r_d = \{0.01, 0.02, 0.04, 0.08\} \), where \( b_h = 40\), \( t = 5\), \( \gamma = 0.8\). In this experiment, noise elimination and random optimizer invocations (5% mean invocation probability) have both been enabled. Results are averaged over \( d = \{0.05, 0.1, 0.15, 0.2\} \). We observe that for most queries the precision is excellent (e.g., at \( r_d = 0.08\), precision is greater than 90% for \( Q_0-Q_3 \) and for \( Q_6-Q_7 \) and recall is acceptably high (e.g., at \( r_d = 0.08\), for \( Q_0-Q_3 \) recall is above 70%, for \( Q_6-Q_8 \) recall is above 55% and for \( Q_4-Q_5 \) recall is greater than 35%).

Let us now focus on \( Q_8 \) where the following properties of the algorithm can be observed more easily (cf. Figure 11). First, the algorithm has a learning phase in which recall increases before it plateaus. Second, precision and recall decrease as the average distance between the sample points in the workload is increased, which is reasonable because the algorithm is forced to make predictions over a larger radius, for which the plan choice predictability assumption is less tight. Next, the improvements due to noise elimination, negative feedback and random optimizer invocations are investigated. For consistency, each variant of \textsc{approximate-LSH-histograms}—with and without these improvements—is executed on the same 25 workloads.

Figure 12 demonstrates that precision gradually decreases without noise elimination because as more points are inserted into \( \mathbb{X} \), there are more points in total that have been falsely placed into the same buckets. With noise elimination, a fixed ratio of samples are assumed to be false positives, hence, they are not included in plan predictions. With this scheme, precision follows a steady pattern. Finally, we observe that negative feedback can improve both precision and recall. The effects of randomized optimizer invocations are evaluated independently. We observe that precision increases by \( \approx 0.02 \) for each 10% increase in mean invocation probability. However, making too many optimizer invocations may wipe out the gains of plan caching. For this reason, low invocation rates should be targeted.

**C. Runtime Performance**

To evaluate the end-to-end runtime performance of our approach, we have implemented an out-of-the-engine Java prototype that simulates the benefits of plan caching. Fig. 1 provides an overview of our prototype with respect to the existing components of a commercial DBMS. Here, the query optimizer is treated as a black box. In our simulations, we use the timings of our prototype as an upper bound on the overhead of the techniques proposed and measurements with a
raman (100\% - 20\%) by fixing the cost error bound at the predictability assumption to classify plan predictions as correct substantial, a different mechanism is needed. Section IV-E towards recent history. However, when the changes are more substantial, a different mechanism is needed. Section IV-E introduced a binary estimator that relies on the plan cost predictability assumption to classify plan predictions as correct or incorrect. In our evaluations, we have found out that by fixing the cost error bound at $\epsilon = 0.25$, the estimator can achieve 72\% accuracy. Consequently, online precision estimations can be used to detect when the characteristics of the sampled plan spaces change. This idea was validated in a second experiment (details omitted), where mid-way through the workload the plan space of $Q_1$ was artificially manipulated to violate the plan choice predictability and plan cost predictability assumptions. A sudden drop in precision estimation was observed shortly after the manipulation.

VI. RELATED WORK

A. Robust Query Processing

Robust query processing tries to find a single plan that has a minimum average cost over the whole parameter distribution [3]. As opposed to the plan with the least specific cost [5], a robust plan is not necessarily the optimal plan at the mean or modal values of parameters. Hence, there is a trade-off between choosing a query plan that has the minimal cost at the expected parameter values and choosing a plan that has the most predictable cost over the whole parameter distribution. This often requires manual intervention [18]. In a more recent work, Chaudhuri et al. propose a skyline algorithm to automate the configuration process [12]. Finally, Abhirama et al. present a principled set of modifications to the dynamic programming model that accounts for the robustness of query plans [13]. Unfortunately, these RQP approaches cannot be applied to plan caching because the space-and-time overhead of computing the skyline of plans, in addition to performing the extra steps in dynamic programming, is very high. In plan caching, such an overhead is difficult to justify, especially for queries that are—otherwise—cheap to execute.

B. Parametric Query Optimization

Parametric query optimization considers a set of alternative plans for different ranges of parameter values [8], [9]. Based on the actual values of parameters, the appropriate plan is selected during query execution [3]. In work by Markl et al., first the ranges of selectivity/cardinality values for which a query plan is optimal are computed [19]. Next, the actual selectivity/cardinality values are compared against the precomputed validity ranges at various query execution checkpoints to determine whether query re-optimization is necessary. Unfortunately, pre-computing the validity ranges is an expensive process.

In work on parametric plan caching by Bizarro, Bruno and DeWitt, the regions of optimality of cached plans are computed eagerly by making optimizer calls using various values in the parameter space [11]. To reduce the number of optimizer invocations, these regions are approximated by two adverse geometrical models: a hyper-rectangle and an ellipsoid. This idea is similar to earlier work by Hulgeri and Sudarshan where various heuristics are used to decompose the parameter space into polytopes that approximate plan boundaries [20], [10]. At higher dimensions, though, the trade-off in quality due to these approximations can be substantial. Our approach requires no eager exploration of the parameter space. Furthermore, it relies on locality-sensitive hashing to

![Fig. 13. Runtime performance of online-lsh-histograms (with noise elimination) at $r_d = 0.01$, $b_o = 40$, $t = 5$, $\gamma = 0.8$ and $d = 0.01$. The performance of a hypothetical predictor with 100\% precision and recall is also reported (IDEAL).](image-url)
support very precise plan predictions, and it can gracef ully indicate when it is not “safe” to make a prediction.

C. Query Clustering for Plan Caching

Ghosh et al. demonstrate that query clustering has potentially useful applications in plan caching [21]. In their approach, similar queries are grouped into clusters. Later on, these clusters are used in determining which formerly generated query plan is applicable to a new query, hence bypassing query optimization whenever possible. Even though our work shares similar incentives with that of Ghosh et al. in that it also relies on query clustering, there are some fundamental differences. As demonstrated in Section 3, the work by Ghosh et al. suffers from the weaknesses of centroid-based clustering, which cannot efficiently handle clusters of arbitrary shapes [14]. The quality of their clustering algorithm is very sensitive to some user-defined similarity thresholds and the initial choice of cluster centers, which—as the authors point out—is highly correlated with the query sequence in the workload.

D. Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) is an efficient technique developed for nearest-neighbor search [16] and it has been applied to other problems such as join size estimation [22]. A function is said to be locality-sensitive if it maps nearby points to the same hash value with some probability greater than or equal to $Pr_1$, and the probability of it mapping non-nearby points to the same hash value is less than or equal to $Pr_2$, where $Pr_2 < Pr_1$.

In a recent work on nearest neighbor search [16], randomized $m$-to-$n$ dimensional transformations that rely on LSH and space-filling curves are used. Even though we also use LSH and space-filling curves, we are addressing a substantially different problem. In nearest neighbor search, every data point (i.e., record) is potentially a nearest neighbor, therefore, it needs to be stored and indexed. On the other hand, we are only interested in storing information about the clusters but not the individual points; hence, for our context it is sufficient to use and maintain only a synopsis of the points, for which we use database histograms. This lossy summarization of the data permits much greater space and time efficiencies, with a controllable trade-off in prediction error.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have described an online density-based clustering algorithm for parametric plan caching. The algorithm exploits locality-sensitive hashing as a pre-processing step allowing clusters in the plan spaces to be efficiently stored in database histograms and queried in constant time. As demonstrated in our experiments, this lossy summarization permits greater space and time efficiencies, with a controllable trade-off in prediction error. Furthermore, our approach is adaptive and it avoids redundant pre-processing of plan spaces.

There are two notable extensions to the architecture we have described which deserve further investigation. First, it would be desirable to extend the space of modeled optimizer parameters beyond predicate selectivities. Specifically, modeling the system context as optimizer parameters would make the system more robust and adaptive to context changes. While it would be straightforward to append additional parameters when defining the parameter space of a query template, naively doing so could pollute the parameter space with irrelevant parameters that reduce the precision of the decision models; hence, further research into parameter modeling and selection is needed (and would be directly applicable to parametric query optimization as well). Second, it would be desirable to incorporate positive feedback into the decision algorithm to shorten the training period and improve recall. Using positive feedback comes with the risk that the importance of some information is unduly amplified, and so a system of checks and balances would be needed to prevent a feedback spiral that destroys precision. We hope to address these problems as future work.

ACKNOWLEDGMENT

Work described herein was performed while Güneș Aluç was at Sybase, an SAP Company. We thank Anisoara Nica, Glenn N. Paullley, Anil K. Goel, Dan Farrar, Eygenienia Eflov, and Ray Liu for their feedback, as well as the anonymous reviewers for their constructive suggestions.

REFERENCES

APPENDIX

A. Experimental Setup

A slightly modified TPC-H schema and the query templates in Table III have been used. The parameter degree of the query templates range from 2–6. The plan counts in Table III are estimated by probing the optimizer at a finite number of plan space points, hence, these numbers show a lower bound on the number of plans. A date column has been added to each TPC-H table, which has been populated by values following a Gaussian distribution. The database is generated using a TPC-H scale factor of 1. Indexes have been created over the primary and foreign key attributes described in the original TPC-H benchmark as well as the date columns that have artificially been added.

The experiments are executed on a PC with 5.8GB of memory and an AMD Phenom™1035T hexacore processor. A commercial DBMS is used, whose buffer pool size is set to 2GB.

B. Experimental Evaluation of Assumptions

To experimentally validate Assumption 1 (plan choice predictability), 200 test points within the plan spaces of $Q_0$–$Q_5$ are paired with 1000 points that are at most $d$ from each test point. The probability that any pair of points have the same optimal query plan is computed over 200 x 1000 pairs. Figure 14 depicts the probabilities at the statistical lower bound at the 95% confidence interval as $d$ is varied. Similar results validating Assumption 2 (plan cost predictability) have been omitted for space reasons.