Review on Query Processing For Cost Functions with Different Parameters

[1]SP.Chandrakanth, [2]B.RamaSubba Reddy

[1]Department of Computer Science & Information Technology, Jyothishmathi Institute of Tech & Sciences, JNTU, Hyderabad, AP, INDIA.
[2] Department of Computer Science & Information Technology, Jyothishmathi Institute of Tech & Sciences, JNTU, Hyderabad, AP, INDIA.

Abstract-
This paper(Review On Query Processing For Cost Functions With Different Parameters) will explain about applications how they usually depends on precompiled parameterized procedures to interact with a database. But, executing a procedure with a set of parameters different from those used at compilation time may be arbitrarily suboptimal. Adaptive query processing has been used to detect and correct optimizer errors due to incorrect statistics or simplified cost metrics; it has been applied to long-running continuous queries over data streams whose characteristics vary over time Parametric query optimization attempts to solve this problem by exhaustively determining the optimal plans at each point of the parameter space at compile time. For a number of reasons, even the best query optimizers can very often produce sub-optimal query execution plans, leading to a significant degradation of performance. This is especially true in databases used for complex decision support queries and/or object-relational databases. This paper will used to calculate the overall cost for generating the parameters at run time. The cost of a query plan depends on many parameters, such as predicate selectivity's and available memory, whose values may not be known at optimization time. Parametric query optimization optimizes a query into a number of candidate plans, each optimal for some region of the parameter space. We first propose a solution for the PQO problem for the case when the cost functions are linear in the given parameters. This solution is minimally intrusive in the sense that an existing query optimizer can be used with minor modifications: the solution invokes the conventional query optimizer multiple times, with different parameter values.

1 INTRODUCTION

The values of runtime parameters of the system, data, or queries themselves are unknown when queries are originally optimized. The cost of a query plan depends on many parameters, such as predicate selectivity's and available memory, whose values may not be known at optimization time In these scenarios, there are typically two trivial alternatives to deal with the optimization and execution of such parameterized queries. One approach, termed here as Optimize-Always, is to call the optimizer and generate a new execution plan every time a new instance of the query is invoked. Another trivial approach, termed Optimize Once, is to optimize the query just once, with some set of parameter values, and reuse the resulting physical plan for any subsequent set of parameters. Both approaches have clear disadvantages. Optimize-Always requires an optimization call for each execution of a query instance. These optimization calls may be a significant part of the total query execution time, especially for simple queries. In addition, Optimize-Always may limit the number of concurrent queries in the system, as the optimization process itself may consume too much memory. On the other hand, Optimize-Once returns a single plan that is used for all points in the parameter space. The chosen plan may be arbitrarily suboptimal for parameter values different from those for which the query was originally optimized.

1.1Parametric Query Optimization

An alternative to Optimize-Always and Optimize-Once is Parametric Query Optimization (PQO). At optimization time, PQO determines a set of plans such that for each point in the parameter space, there is at

least one plan in the set that it is optimal. The regions of optimality of each plan are also computed. Later, when an instance of the query is submitted, PQO chooses the best precompiled plan for the query instance and executes it without making a new optimization
call. PQO proposals often assume that the cost formulas of physical plans are linear or piecewise linear with respect to the cost parameters and that the regions of optimality are connected and convex. However, in reality, the cost functions of physical plans and regions of optimality are not so well behaved. A more important problem results from the fact that PQO has a much higher start-up cost than optimizing a query a single time (PQO usually requires several invocations
Of the optimizer with different parameters). When a previously unseen query arrives, it is therefore not clear to determine whether PQO should be used: it may not be cost-effective to solve the full PQO problem if the query is not executed frequently or if it is repeatedly executed with values covering a small subspace of the entire parameter space. Most previous work (see Section 6) ignores this dilemma and instead solves the full PQO problem, potentially wasting more resources than necessary.

1.2 Contributions

In this paper, we propose an alternative approach to handle parametric queries that addresses the shortcomings de-scribed above. Our contributions are listed as follows In Section 2, we propose Progressive Parametric Query Optimization (PPQO), a novel framework to improve the performance of processing parameterized queries. We also propose the Parametric Plan (PP) interface as a way to incorporate PPQO in DBMS .In Sections 3 and 4, we propose two implementations of PPQO with different goals. On one hand, Bounded has proven optimality guarantees. On the other hand, Ellipse results in higher hit rates and better scalability.

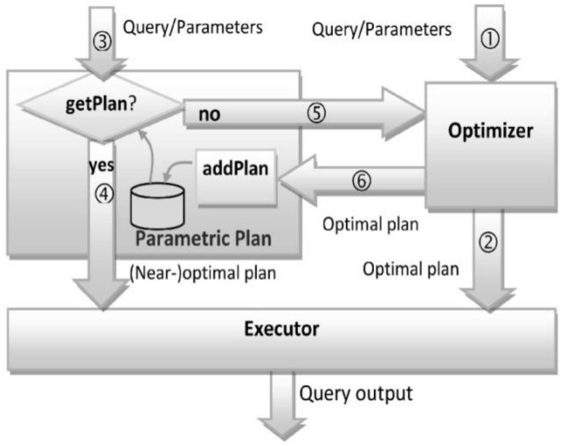
2 RELATED WORKS

Dynamic query plans include more than one physical plan choice. The plan to use is determined at runtime by the choose-plan operator after it costs the alternatives given the now known parameter values. How to enumerate dynamic query plans was proposed only later with the concept of incomparability of costs: in the presence of unbound parameters at optimization time, plan costs are represented as intervals, and if intervals of alternative plan overlap, none is pruned. At runtime, when parameters are bound to values, the choose-plan selects the right plan. This approach may enumerate a large number of plans, and all those plans may have to be recost at runtime. Ioannidis et al. Coined the term PQO and proposed using randomized algorithms to optimize in parallel the parametric query for all possible values of unknown variables. This approach is unfeasible for continuous parameters, gives no guarantees on finding the optimal plan for a query, and places no bounds on the optimality of the plans produced.

A closely related piece of work is PLASTIC. Like PPQO, PLASTIC incrementally maintains clusters of incoming queries and avoids optimizing a new query if it is “close enough” to a previously seen cluster. At a high level, we can see PLASTIC as an instance of PPQO, where getPlan compares an incoming query against each of the previously saved ones and reuses an old query plan if it is “close enough” to the current query, and addPlan adds a plan as a new cluster representative. In contrast to Bounded and Ellipse, query similarity in PLASTIC is measured as a distance between feature vectors that describe the queries (such as the number of relations in the query, the number and type of predicates, and estimated sizes of tables and intermediate relations). For that reason, PLASTIC has the potential to detect similarities between queries with similar structure but touching different tables (like “SELECT R.a FROM R JOIN S” and “SELECT T.b FROM T JOIN U”). In our work, we do not attempt to reuse plans across different queries, so a direct implementation of PLASTIC would always compare instances of the same query with different parameters. As a consequence, the distance metric between queries would result in the sum of differences in the cost parameters, and PLASTIC would reduce to performing nearest neighbor searches on the parameter space with a threshold that determines when a new cluster should be created. As such, PLASTIC cannot give worst case quality guarantees on the resulting plans (as Bounded does) nor is able to model long and narrow optimality regions (as Ellipse does). It is, however, an interesting implementation of PPQO that might be useful in certain scenarios.

Finally, the recent work in coins the term “plan diagram” to denote a pictorial enumeration of the execution plan choices of a query optimizer over the selectivity space. These work shows, using plan diagrams, that assumptions commonly held by PQO (plan convexity, plan uniqueness, and plan homogeneity) do not hold. These discoveries do not affect Bounded-PPQO, which provides optimality guarantees. On the other hand, Ellipse-PPQO results in higher hit rates but gives no optimality guarantees on returned plans and may produce poor results for large $_3$ -acceptable regions. Very recently, in a follow-up to ,the authors propose to reduce the plan diagram for a given query by "collapsing" plans whose costs are close enough to each other This work shares with ours the notion that in many cases, obtaining near-optimal plans is sufficient and might lead to dramatic reductions in the number of plans to consider without sacrificing the quality of the optimization process. A crucial difference with our work is that proceeds a posteriori, after optimizing the input query for all possible parameters (specifically, over a fine grid that is laid out over the parameter space). In contrast, PPQO is to progressively build a PP data structure with no long start-up costs.

3 PROGRESSIVE PARAMETRIC QUERY OPTIMIZATION



The main idea of PPQO is to incrementally solve (or approximate) the solution to the PQO problem as successive

Fig. 1. Using PPs to process a query.

Query execution calls are submitted to the DBMS. Fig. 1 shows a high-level architecture of our approach. Given a query and its parameter values, a traditional optimizer returns the optimal execution plan along with its estimated cost . In contrast, a PPQO-enabled optimizer introduces a data structure called PP, which incrementally maintains plans and optimality regions, allowing us to reuse work across optimizations. As the PP data structure becomes populated, it is possible to completely bypass the optimization process without hurting the quality of the resulting execution plans.

When a new instance of a parametric query arrives (in Fig. 1), PPQO tries to obtain an optimal (or near-optimal) plan by consulting the PP data structure. If it is successful, it returns such plan, and a full optimization call is avoided in Fig. 1). Otherwise, it makes an optimization call in Fig. 1), and both the resulting optimal plan and cost are added to the PP for future use in Fig. 1). Due to the size of the parameter space, PPs should not be implemented as exact lookup caches of plans because there would be too many "cache misses." Also, due to the nonlinear and discontinuous nature of cost functions, PPs should not be implemented as nearest neighbor lookup structures as there will be no guarantee that the optimal plan of the nearest neighbor is optimal or close to optimal for the point in the parameter space being considered. We now describe the PPQO problem in more detail, borrowing notation and definitions from the classic parametric optimization problem.

2.1 Definitions and Preliminaries

A parametric query Q is a text representation of a relational query with placeholders for m parameters vpt =(v_1,... . . ,$v_{m)}$. Vector vpt is called a Value Point. Examples of parameter values are system parameters (e.g., available memory) and query-dependant parameters (e.g., constants in parametric predicates). In the rest of the paper, we focus on query-dependant parameters since they cover the most common scenarios. We note, however, that our techniques can also be adapted to other kinds of parameters.

Using $vpt = (v1 \ldots\ldots v_m$ directly to model the parameter space and characterize regions of optimality for plans is in general difficult (see below for an example). To address this problem, we use a transformation function φ', which is optimizer. specific and transforms Value Points into what we call Cost Points. A Cost Point is a vector $cpt = (c1, c2 \ldots \ldots \ldots c_n)$ where each c_i is a cost parameter with an ordered domain. A well-known implementation of ', which we justify below and use in the rest of the paper, is transforming parametric predicate values into the corresponding predicate selectivity's. For instance, consider predicate age < \$X\$, with parameter \$X\$. Function' would then map a specific constant c for <\$X\$ into the selectivity of the nonparametric predicate age.

Let p be some execution plan that evaluates query Q for a given vpt. The cost function of p, denoted takes a Cost Point cpt as an input and returns the cost of evaluating plan p under cpt. For every legal value of the parameters, there is some plan that is optimal. Given a parametric query Q, the maximum parametric set of plans (MPSP) is the set of plans, each of which is optimal for some point in the n-dimensional cost-based parameter space. The region of optimality for plan p, denoted r(p), is defined as

$$r(p) = \{(t_1, \ldots, t_n) \mid p \text{ is optimal at } (c_1 = t_1, \ldots, c_n = t_n)\}$$

Finally, a parametric optimal set of plans (POSP) is a minimal subset of MPSP that includes at least one optimal plan for each point in the parameter space. Having introduced this basic terminology, we next justify the need for the transformation function' and then define the PPQO framework in detail.

2.2 The Parameter Transformation Function '

Recall that a value parameter refers to an input value of the parametric SQL query to execute. On the other hand, a cost parameter is an input parameter in the formulas used by the optimizer to estimate the cost of a query plan. Cost parameters are estimated during query optimization from value parameters and from information in the database catalog. (Physical characteristics that affect the cost of plans but do not depend on query parameters, such as the average tuple size or the cost of a random I/O, are considered physical constants instead of cost parameters.)

A crucial cost parameter that is used during optimization is the estimated number of tuples in (intermediate) relations processed by the query plan: most query plans have cost formulas that are monotonic in the number of tuples processed by the query. On the other hand, there is no obvious relationship between the value parameters and the cost of the query plans. Thus, it becomes much easier to characterize the regions of optimality using a cost-based parameter space than using a value-based parameter space. In Example 1 below and in what follows, we use a cost-based parameter space whose dimensions are predicate selectivity's. (Note that the estimated number of tuples of each relation processed by a query is typically derived from selectivity's of sub expressions computed during query optimization.)

Example 1*.Table FRESHMEN (NAME, AGE) succinctly de-scribes first-year graduate students.*
The age distribution of students is showed in Fig. 2. Consider queries of the following form:

```
SELECT * FROM
FRESHMEN
WHERE AGE =$X$ OR AGE $ X$
```

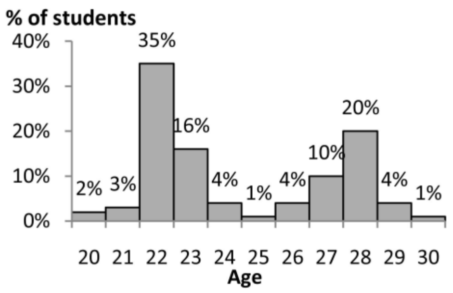


Fig. 2.Age distribution in table FRESHMEN.

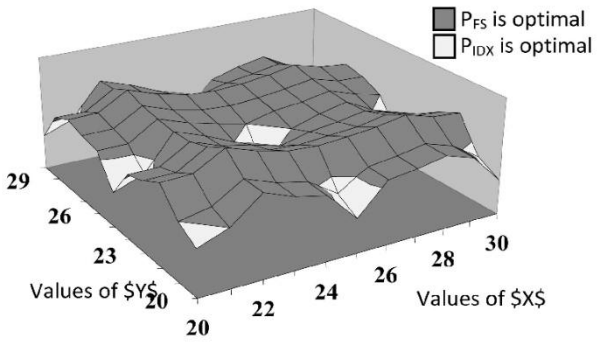


Fig. 3. Value-based parameter space

Assume that the optimal plan for queries that retrieve less than 5 percent of FRESHMEN tuples is P_{IDX}, a plan using an index on column AGE. For all other queries, the optimal plan is P_{FS}, a full-table scan on FRESHMEN. The parameters of this query can be represented as the absolute values used for parameters \$X\$ and \$Y\$ or as the selectivity of predicate age = \$X\$ and predicate age = \$Y\$. Accordingly, the costs of physical P_{IDX} and P_{FS} can be represented in value-based parameter spaces, shown in Fig. 3, or in selectivity-based (also referred to as cost-based) parameter spaces, shown in Fig. 4. Clearly, the selectivity-based representation results in a much more manageable parameter space than the (seemingly chaotic) value-based representation. The reason is that selectivity-based representations are better aligned to the optimizer cost model and tend to be represented by monotonic cost functions, and therefore, the regions of optimality of plans tend to cluster together.

In the rest of this paper, we assume that function φ' takes query Q and its SQL parameters, vpt, and returns cpt as a vector of selectivity. Computing the selectivity's in cpt corresponds to the task of selectivity estimation, a subroutine inside of query optimization. Other components of query optimization—e.g., plan enumeration, rule transformation, and costing—need not be part of the implementation of function φ'. In general, computing selectivity values from actual values is done by manipulating in-memory histograms, which is very efficient, and a negligible fraction of the full query optimization task.

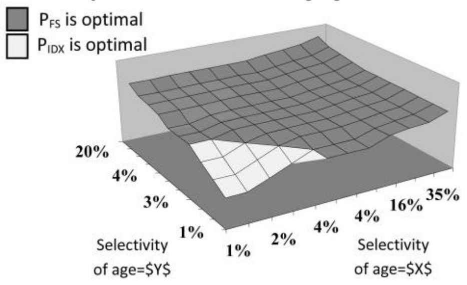


Fig. 4. Selectivity-based parameter space.

```
function processQuery (
   inputs: Query Q, ValuePoint vpt
   input/output: ParametricPlan pp )
CostPoint cpt ← φ(Q, vpt);        // ValuePoint to CostPoint
Plan p ← pp.getPlan(Q, cpt);     // what plan to use?
if (p == NULL)
  Cost cost;                      // cost is output param below
  p ← optimize(Q, vpt, cost);//finds optimal plan & cost
  pp.addPlan(Q, cpt, p, cost);// stores plan & cost in pp
execute(p);
```

Fig. 5. Using PPs.

We note that the artily of the value-based parameter space and that of the selectivity-based parameter space are not necessarily the same. On one hand, it is possible to have predicates of the form age > X and age < Y, where two value predicates are collapsed into a single selectivity value for the combined predicate. Similarly, a query that contains. a predicate of the form R:age < X and also a join between tables R and S might require two selectivity parameters to capture the optimizer's cost model: one for the selectivity of the predicate on the base table and another for the selectivity of the predicate on the join. In our prototype and experimental evaluation, we use a simple one-to-one mapping between parametric predicates and selectivity values (i.e., we do not consider join predicates nor combine atomic predicates over the same column). The reasons behind our choice are the following: 1) this is the mapping used in previous work on parametric optimization, 2) it can be implemented without deep knowledge about the under-lying query optimizer, and 3) our experiments show that this simple model is very competitive.

2.3 The Parametric Plan Interface

We now give an operational description of the PP component of PPQO by describing its two main operations (also see Fig. 1):

1. addPlan (Q,cpt, p,c). This operation registers that plan p, with estimated cost c, is optimal for query Q at Cost Point cpt.

2. getPlan (Q, cpt). This operation returns the plan that should be used for query Q and cost values cpt or returns null if no plan is considered good Enough for Q.

Implementations of the PP interface are used during query processing, as shown in Fig. 1 and in the pseudo code in Fig. 5. When parametric query parameter instances are required to execute, the DBMS calls the PP's getPlan method. If getPlan returns plan p_1, then p_1 is used for execution, and an optimization call is avoided. If getPlan returns null (we call this situation a getPlan miss), then the optimizer is called, and a potentially new plan, p_2, is obtained from the optimizer. Plan p_2 is then executed. The parameter

```
Optimize-Always implements PP
   addPlan(inputs: Query Q, CostPoint cpt,
                   Plan p,  Cost cost)
   return; // does nothing

   getPlan(inputs:  Query Q, CostPoint cpt;
           outputs: Plan p)
   return null;
```

Fig. 5. Optimize-Always implementation.

```
Optimize-Once implements PP
   private Plan p = null;

   addPlan(inputs: Query Q, CostPoint cpt,
                   Plan plan, Cost cost)
   if (p == null) p = plan;   // saves first plan

   getPlan(inputs:  Query Q, Cost-Point cpt;
           outputs: Plan plan)
   return p;                  // returns first plan
```

Fig. 6. Optimize-Once implementation.

Values, plan p_2, and its cost are then added to the PP using addPlan.

As we show in Sections 3 and 4, the PP interface can be used to implement various PPQO policies. However, it can also implement simple policies like Optimize-Always and Optimize-Once. Fig. 6 shows the Optimize-Always implementation of the PP interface, in which addPlan is empty and getPlan always returns null, forcing an optimization for every query. Fig. 7 shows the Optimize-Once implementation of the PP interface, in which addPlan saves the first plan it is given as input and getPlan returns such plan in all subsequent calls.

2.4 Para metrics Plans: Requirements and Goals

The main trade-off in PPQO is to avoid as many optimization calls as possible as long as we are willing to execute suboptimal—but close to optimal—plans (note that this goal has also been proposed in and in the context of classical PPQO). Thus, PP implementations must obey the
Inference Requirement below.

Inference Requirement. After a number of addPlan calls, there must be cases where getPlan returns an (near-)optimal plan p for query Q and parameter point cpt, even if addPlan(Q,cpt, p, cost) was never called.

Given a sequence of execution requests of the same query with potentially different input parameters, PPQO has therefore two conflicting goals:

1. Goal 1. Minimize the number of optimization calls.

2. Goal 2. Execute plans with costs as close to the cost of the optimal plan as possible.

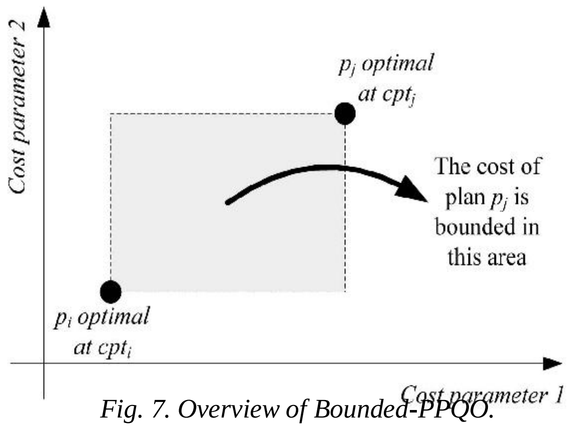


Fig. 7. Overview of Bounded-PPQO.

3 THE BOUNDED IMPLEMENTATION

We now describe the first of two proposed PPQO implementations, termed Bounded-PPQO or simply bounded. This implementation provides guarantees on the quality of the plans returned by getPlan (Q; cpt), thus focusing on Goal 2 of PPQO (see previous section). Either the returned plan p is null (or an optimization call cannot be avoided) or p has a cost guaranteed to be within a user-specified bound of the cost of the optimal plan. Specifically, the cost of plan p returned by getNext is guaranteed to be bounded by Opt Cost*M + A, where Opt Cost is the cost of the optimal plan, and $M \geq 1$ and $A \geq 0$ are user-defined constants. Both M and A can be used to specify different bounds on sub optimality and are generally application specific. (We report, however, the effects of varying parameters M and A in Section 5.)

The intuition for the Bounded-PPQO implementation is given as follows: Consider a parametric query with two parameters. If plans p_i and p_j are optimal in some Cost Points cpt_i and cpt_j, which delimit a box as shown in the two-dimensional example in Fig. 8, then we can provably bound the cost of plan p_j in all points within that box if the cost functions are monotonic along all dimensions (e.g., if the cost of the query increases whenever the selectivity of any parameter increases). Specifically, the cost of plan p_j in the box will be between the cost of plan p_i at cpt_i and the cost of plan p_j at cpt_j.

3.1 Preliminaries

We now introduce some definitions required to describe the Bounded-PPQO implementation

1. Relationship equal (=)Given $cpt_1 = c_1c_2c_3 \dots \dots .c_{1,n}$ and $cpt_2 = c_1c_2c_3 \dots \dots .c_{1,n}$¼$c_{1,n} \dots c_{2,n}, cpt_1 \equiv cpt_2$ iff $\forall i c_{1,i} = c_{2,i}$
2. Relationships below ( ) and above ( ). Given $cpt_1 = c_1c_2c_3 \dots \dots .c_{1,n}$ and $cpt_2 = c_1c_2c_3 \dots \dots .c_{1,n}$, $cpt_1 < cpt_2, \; cpt_1 > cpt_2$ iff $\forall i c_{1,i} > c_{2,i}$.
3. Opt(cpt). It is the cost of an optimal plan at cpt.

```
getPlan(inputs: Query Q, Cost-Point cpt;
        outputs: Plan plan)

List T_Q ← getList(Q);        // gets list of triples for Q
for each (t1, t2) in T_Q      // look any pair of triples
  if (t1.cost≤t2.cost≤t1.cost*M+A and
      t1.cpt ‹ cpt ‹ t2.cpt)
      return t1.p;
return null;
```

Fig. 8. Bounded's getPlan implementation.

$$plan_i(cpt_i) \leq plan_j(cpt_j) \leq plan_i(cpt_i)^* M + A,$$

where M and A are, respectively, any user-defined multiplicative and additive factors, with $M \geq 1$ and $A \geq 0$. The pair (t_i, $t_{j)}$ is also said to bound cpt if $cpt_1 < cpt \; < cpt_i$ We additionally rely on the intuitive Monotonic Assumption (or MA), stated as follows: given plan p and

3.2 Implementation of addPlan for Bounded

Function addPlan (Q,cpt, p, cost) shown in Fig. 9, associates with each parametric query Q a list TQ of triples (cpt, p, cost) ordered by cost, where p is an optimal plan at cpt with an estimated execution cost (at cpt) of cost =cpt

3.3 Implementation of getPlan for Bounded

Theorem 1. $if\ t_i = (cpt_i, plan_i cost_i) and\ t_j = (cpt_j, plan_j, cost_j$ are a bounding for some $m \geq 1\ A \leq 1$ then under the MA
, the cost of plan
can be tightly bounded such that, $opt(cpt) \leq plan_j(cpt) \leq opt(cpt) * M$
$= A\ for\ all\ cpt\ that\ cpt_1 < cpt \; < cpt_i$

Proof. By Lemma 1 and $cpt_1 < cpt < cpt_i$, it follows that $cost_i \leq opt(cpt) \leq cost_j$, Also ,by lemma 2 and $cost_i \leq cost_j \leq cost_i * M + A$, we get $opt(cpt) \leq cost_j \leq opt(cpt) * M + A$.

Example 2. For some query Q, assume that addPlan was already called for the points (and associated triples) shown in Fig. 11 (i.e., assume that the PP stores information about the optimal plans and costs for the triples in $T_Q = (t_1 t_2 t_3 t_4 t_5 t_6))$.Given cpt (shown as a black circle) in the cost-based parameter space, M = 1.5, and A =0, which plan would getPlan (Q, cpt) return? There
are six pairs $cpt_i cpt_j$ such that

$cpt_1 < cpt < cpt_i$: $(cpt_1, cpt_6), (cpt_1, cpt_7), (cpt_1, cpt_5), (cpt_3, cpt_6), (cpt_3, cpt_7)$
$c_3 \leq c_5 \leq c_3 * 1.5 + 0 \leftrightarrow 6 \leq 8 \leq 9$ *and pair* (t_3, t_6) because $c_3 \leq c_6 \leq c_3 * 1.5 + 0 \leftrightarrow 6 \leq 9 \leq 9.$

Thus, either plan p_5 and plan p_6 can be safely returned by getPlan

4 THE ELLIPSE IMPLEMENTATION

Bounder's getPlan provides strong guarantees on the cost of plans returned. However, we expect low hit rates of Bounder's getPlan for small values of M and A or before Bounder's T_Q has been populated. In this section, we propose the Ellipse-PPQO (or simply Ellipse) implementation of the PP interface, designed to address Goal 1 in Section 2.2 (i.e., having high hit rates). For that purpose, Ellipse's getPlan returns Δ-acceptable plans rather than guaranteed near-optimal costs.

Definition (Δ-acceptable plans). For Δ €[0; 1], if plan p is known to be optimal at points cpt_1 and cpt_2 in the cost-based parameter space, then plan p is Δ -acceptable at point cpt in the cost-based parameter space if and only if

$$\frac{\| cpt_1 - cpt_2 \|}{\| cpt - cpt_1 \| + \| cpt - cpt_2 \|} \geq \Delta$$

where ||p – q|| is the Euclidean distance between p and q.

It follows from the definition of Δ -acceptable that if p is optimal at cpt_1 and cpt_2, then p is 1-acceptable only on points between cpt_1 and cpt_2 and p is 0-acceptable at all points. Note that in a two-dimensional space, the area where p is acceptable is equivalent to the definition of an ellipse; if p is optimal for cpt_1 and cpt_2, then p is Δ -acceptable at cpt if cpt is on or inside an ellipse of foci cpt_1 and cpt_2 such that the distance between the foci, $||cpt_1 - cpt_{2||}$, over the sum of the distances between cpt and the foci, $||cpt - cpt_{1||} + ||cpt - cpt_{2||}$, is at least Δ. Fig. 13 shows the areas where p is 0.5-acceptable, 0.8-acceptable, and 1-acceptable if p is optimal at cpt_1 and cpt_2.

Ellipse-PPQO encodes the heuristic that if a plan p is optimal in two points cpt_1 and cpt_2, then p is likely to be optimal or near-optimal in a convex region that encloses cpt_1 and cpt_2. Note that a nearest neighbor algorithm could be used as an alternative to Ellipse-PPQO. However, since regions of optimality are frequently long and narrow, for any given cpt point, the closest known plan could very well be from another region of optimality (which we verified in practice). In addition, Δ -acceptable areas can easily encode both small and large regions of optimality.

6 CONCLUSIONS

Before PPQO, processing parameterized queries was an all-or-nothing approach: either the optimizer explores all the parameter space and computes the full PQO solution (traditional PQO) or it relies on luck and uses the very first plan it gets for a query. PPQO is able to progressively construct information about the parametric space and approximate optimality regions, being able to bypass the optimizer up to 99 percent of the times, while still returning plans within 5 percent of the cost-optimal plan for 99 percent of the cases. Unlike PQO, PPQO does not perform extra optimizer calls or extra plan-cost evaluation calls. At execution time, PPQO selects which plan to execute by using only the input cost parameters without recosting plans. PPQO is an adaptive technique that works prior to execution (and assumes the optimizer to be correct just like any other PQO approach). Query reoptimization and other adaptive query processing (AQP) approaches work during optimization and execution and assume that the optimizer can make mistakes or that the system characteristics change significantly during the execution of a single query. Also, PPQO is an interquery adaptive approach, while AQP are frequently intraquery optimization approaches.

PPQO is also amenable to be implemented in a complex commercial database system as it requires no changes in the optimization or execution processes. In fact, our PPQO prototype ran outside the DBMS server. For technical reasons, we did not implement function ’ ourselves but instead used SQL Server’s cost model to transform value into cost parameters. For that reason, we did not evaluate the impact of such function in our experimental evaluation. However, it is important to note that function ’ can be implemented by simply manipulating in memory histo-grams (i.e., 200-int arrays), which is a negligible fraction of optimization time and would not have resulted in any noticeable difference in our experimental evaluation.

PPQO was evaluated in a variety of settings, with queries joining up to eight tables, with multiple sub queries, up to four parameters, and in plan spaces with close to 400 different optimal plans. PPQO yielded good results in all scenarios except for the Bounded algorithm in complex queries using a four-dimensional parameter space. How-ever, even in this challenging scenario, Ellipse on the average executed plans just 3 percent more costly than the optimal, while avoiding 87 percent of all optimization calls.

REFERENCES

[1] S. Babu and P. Bizarro, “Adaptive Query Processing in the Looking Glass,” Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR), 2005.
[2] R.L. Cole and G. Graefe, “Optimization of Dynamic Query Evaluation Plans,” Proc. ACM SIGMOD, 1994.
[3] D. Harish, P. Darera, and J. Haritsa, “On the Production of Anorexic Plan Diagrams,” Proc. 33rd Int’l Conf. Very Large Data Bases (VLDB), 2007.
[4] Deshpande, Z. Ives, and V. Raman, “Adaptive Query Processing,” Foundations and Trends in Databases, vol. 1, no. 1, pp. 1-140, 2007.
[5] S. Ganguly, “Design and Analysis of Parametric Query Optimiza-tion Algorithms,” Proc. 24th Int’l Conf. Very Large Data Bases (VLDB), 1998.
[6] Ghosh, J. Parikh, V.S. Sengar, and J.R. Haritsa, “Plan Selection Based on Query Clustering,” Proc. 28th Int’l Conf. Very Large Data Bases (VLDB), 2002.
[7] G. Graefe and K. Ward, “Dynamic Query Evaluation Plans,” Proc. ACM SIGMOD, 1989.
[8] Hulgeri and S. Sudarshan, “Parametric Query Optimization for Linear and Piecewise Linear Cost Functions,” Proc. 28th Int’l Conf. Very Large Data Bases (VLDB), 2002.
[9] Hulgeri and S. Sudarshan, “AniPQO: Almost Non-Intrusive Parametric Query Optimization for Nonlinear Cost Functions,” Proc. 28th Int’l Conf. Very Large Data Bases (VLDB), 2003.