Progressive Parametric Query Optimization

Pedro Bizarro, Nicolas Bruno and David J. DeWitt

Abstract— Commercial applications usually rely on pre-compiled parameterized procedures to interact with a database. Unfortunately, executing a procedure with a set of parameters different from those used at compilation time may be arbitrarily sub-optimal. Parametric query optimization (PQO) attempts to solve this problem by exhaustively determining the optimal plans at each point of the parameter space at compile time. However, PQO is likely not cost-effective if the query is executed infrequently or if it is executed with values only within a subset of the parameter space. In this paper we propose instead to progressively explore the parameter space and build a parametric plan during several executions of the same query. We introduce algorithms that, as parametric plans are populated, are able to frequently bypass the optimizer but still execute optimal or near-optimal plans.

Index Terms— Parametric Query Optimization, Adaptive Optimization, Selectivity Estimation.

1 Introduction

In many applications, the values of run-time parameters of the system, data, or queries themselves are unknown when queries are originally optimized. In these scenarios there are typically two trivial alternatives to deal with the optimization and execution of such parameterized queries. One approach, termed here *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 sub-optimal for parameter values different from those for which the query was originally optimized.

1.1 Parametric 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 pre-computed 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 piece-wise 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 startup cost than optimizing a query a single time (PQO usually requires several invocations of the optimizer with different parameters [8, 9]). 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 sub-space 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 described above. Our contributions are 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

- *Pedro Bizarro is with the CISUC/DEI, University of Coimbra, DEI – Polo 2, 3030-290 Coimbra, Portugal. E-mail: bizarro@dei.uc.pt*
- *Nicolas Bruno is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: nicolasb@microsoft.com*
- *David J. DeWitt is with the University of Wisconsin – Madison, 1210 W Dayton Str, Madison, WI 53706 E-mail: dewitt@cs.wisc.edu*

Manuscript received (insert date of submission if desired).

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.
- Finally, in Section 5 we present an extensive performance evaluation of PPQO using a prototype implementation on Microsoft SQL Server 2005.

2 Progressive Parametric Query Optimization

The main idea of *Progressive Parametric Query Optimization,* or PPQO for short, is to incrementally solve (or approximate) the solution to the PQO problem as successive 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 (① and ② in the figure). In contrast, a PPQO-enabled optimizer introduces a data structure called *Parametric Plan* (PP), which incrementally maintains plans and optimality regions, allowing us to reuse work across optimizations. As the Parametric Plan data structure becomes populated, it is possible to completely bypass the optimization process without hurting the quality of the resulting execution plans.

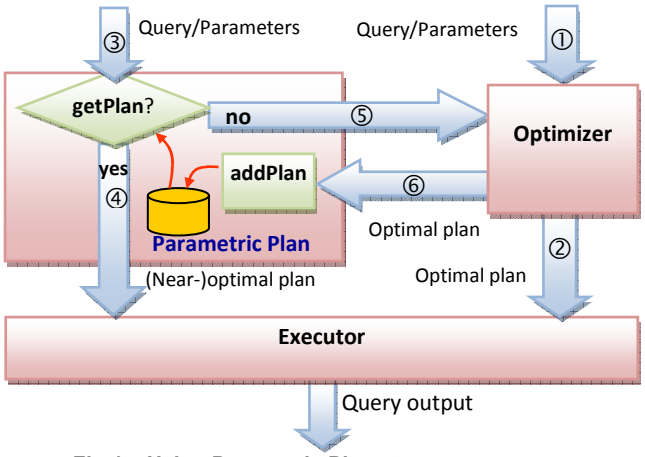

Fig 1 – Using Parametric Plans to process a query.

When a new instance of a parametric query arrives (③ in Fig 1), PPQO tries to obtain an optimal (or near-optimal) plan by consulting the parametric plan data structure. If it is successful, it returns such plan and a full optimization call is avoided (④ in Fig 1). Otherwise, it makes an optimization call (⑤ in Fig 1) and both the resulting optimal plan and cost are added to the parametric plan for future use (⑥ in Fig 1). Due to the size of the parameter space, parametric plans should not be implemented as exact lookup caches of plans because there would be too many "cache misses". Also, due to the non-linear and discontinuous nature of cost functions, parametric plans 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 [3, 16]. 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 $vpt = (v_1, \ldots, v_m)$. Vector vpt is called a *ValuePoint*. 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 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 *ValuePoints* into what we call *CostPoints*. A *CostPoint* is a vector $cpt = (c_1, \ldots, c_n)$ where each c_i is a cost parameter with an ordered domain. A well-known implementation of φ, which we justify below and use in the rest of the paper, is transforming parametric predicate values into the corresponding predicate selectivities. For instance, consider predicate `age<$X$`, with parameter `$X$`. Function φ would then map a specific constant `c` for `$X$` into the selectivity of the non-parametric predicate `age<c`.

Let p be some execution plan that evaluates query Q for a given vpt. The cost function of p, denoted $p(cpt)$, takes a *CostPoint* 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 φ, and then define the PPQO framework in detail.

2.2 The Parameter Transformation Function φ

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 selectivities. (Note that the estimated number of tuples of each relation processed by a query is typically derived from selectivities of sub-expressions computed during query optimization.)

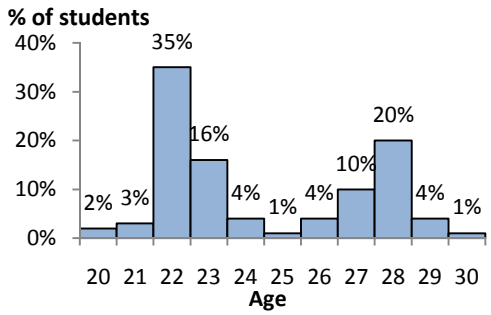

Fig 2 – Age distribution in table FRESHMEN

Example 1: Table `FRESHMEN(NAME, AGE)` succinctly describes 1st-year graduate students. The age distribution of students is showed in Fig 2. Consider queries of the form:

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

Assume that the optimal plan for queries that retrieve less than 5% 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 selectivities 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 Figure 3, or in selectivity-based (also referred to as cost-based) parameter spaces, shown in Figure 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, tend to be represented by monotonic cost functions, and therefore the regions of optimality of plans tend to cluster together.

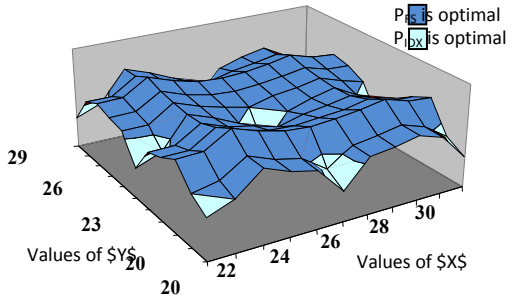

Fig 3 – Value-based parameter space

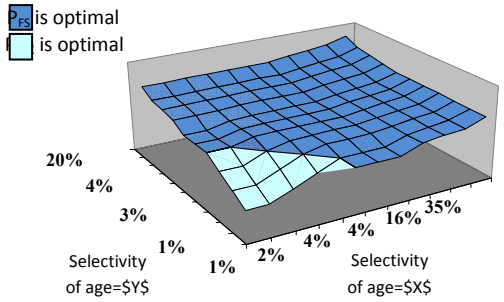

Fig 4 – Selectivity-based parameter space

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 selectivities. Computing the selectivities 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.

We note that the arity 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 (i) this is the mapping used in previous work on parametric optimization, (ii) it can be implemented without deep knowledge about the underlying query optimizer, and (iii) our experiments show that this simple model is very competitive.

2.3 The Parametric Plan Interface

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

- **addPlan**(*Q*, *cpt*, *p*, *c*): registers that plan *p*, with estimated cost *c*, is optimal for query *Q* at *CostPoint cpt*.
- **getPlan**(*Q*, *cpt*) – 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*.

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

Fig 5 – Using Parametric Plans

Implementations of the PP interface are used during query processing as shown in Figure 1 and in the pseudocode in Figure 5. When parametric query parameter instances are required to execute, the DBMS calls the parametric plan'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 values, plan p_2 and its cost are then added to the Parametric Plan using *addPlan*.

```
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 6 – Optimize-Always implementation

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.

```
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 7 – Optimize-Once implementation

2.4 Parametric Plans: Requirements and Goals

The main tradeoff in PPQO is to avoid as many optimization calls as possible as long as we are willing to execute sub-optimal -but close to optimal- plans (note that this goal has also been proposed in [5] and [11] in the context of classical PQO). Thus, PP implementations must obey the *Inference Requirement* below.

Inference Requirement: After a number of *addPlan* calls, there must be cases where *getPlan* returns a (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:

Goal 1: Minimize the number of optimization calls.

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

Consider a trivial cache implementation of the PP interface, which stores (*Q*, *cpt*) pairs as the lookup key and (*p*, *cost*) as the inserted value. This implementation cannot fulfill the inference requirement because it would return

hits only for previously inserted (*Q*, *cpt*) pairs. In the next sections we propose two PPQO implementations, each giving priority to one of the above goals. *Bounded-PPQO*, described in Section 3, gives priority to Goal 2. *Ellipse-PPQO*, described in Section 4, gives priority to Goal 1.

3 The Bounded-PPQO 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 (and 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 *OptCost***M*+*A*, where *OptCost* is the cost of the optimal plan, and *M*≥1 and *A*≥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 the Experimental Evaluation.)

The intuition for the *Bounded-PPQO* implementation is as follows. Consider a parametric query with two parameters. If plans p_i and p_j are optimal in some *CostPoints* cpt_i and cpt_j, which delimit a box as shown in the 2-dimensional example of 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.

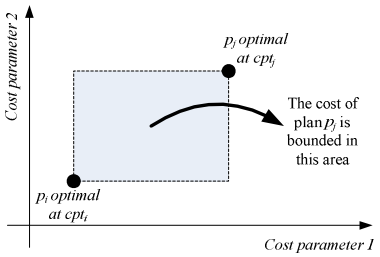

Fig 8 – Overview of Bounded-PPQO

3.1 Preliminaries

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

- *Relationship equal* (**≡**)**:** Given $cpt_1=(c_{1,1}, \ldots, c_{1,n})$ and $cpt_2=(c_{2,1}, \ldots, c_{2,n})$, $cpt_1 \equiv cpt_2$ iff $\forall i\ c_{1,i}=c_{2,i}$.
- *Relationships below* (◂) *and above* (▸): Given $cpt_1=(c_{1,1}, \ldots, c_{1,n})$ and $cpt_2=(c_{2,1}, \ldots, c_{2,n})$, $cpt_1 \triangleleft cpt_2$ ($cpt_1 \triangleright cpt_2$) iff $\forall i$, $c_{1,i} \le c_{2,i}$ ($c_{1,i} \ge c_{2,i}$), and $\exists i$, $c_{1,i} \ne c_{2,i}$. Note that both ◂ and ▸ are transitive. That is, if $cpt_1 \triangleleft cpt_2$ ($cpt_1 \triangleright cpt_2$) and $cpt_2 \triangleleft cpt_3$ ($cpt_2 \triangleright cpt_3$) then $cpt_1 \triangleleft cpt_3$ ($cpt_1 \triangleright cpt_3$).
- *Opt*(*cpt*): It is the cost of an optimal plan at *cpt*.
- Triples $t_i=(cpt_i, plan_i, cost_i)$ and $t_j=(cpt_j, plan_j, cost_j)$ are a *bounding pair* if plan $plan_i$ ($plan_j$) is an optimal plan at cpt_i (cpt_j) with cost $cost_i$ ($cost_j$), $cpt_i \triangleleft cpt_j$ and $plan_i(cpt_i) \le plan_j(cpt_j) \le plan_i(cpt_i)*M+A$, where *M* and *A* are, respectively, any user-defined multiplicative and additive factors, with $M \ge 1$ and $A \ge 0$. The pair (t_i, t_j) is also said to *bound cpt*, if $cpt_i \triangleleft cpt \triangleleft cpt_j$.

We additionally rely on the intuitive *Monotonic Assumption* (or *MA*), stated as follows: given plan *p* and *CostPoints* cpt_1 and cpt_2, if $cpt_1 \triangleleft cpt_2$ then $p(cpt_1) \le p(cpt_2)$.[1]

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 T_Q 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*=*p*(*cpt*).

```
addPlan (inputs: Query Q, CostPoint cpt,
                 Plan p, Cost cost) {
01 List T_Q ← getList(Q);      // Gets the list of triples for Q
02 if (T_Q ==null)
03    T_Q = new List();        // If no list, create one
04 T_Q.insert(cpt, p, cost); // Inserts triple in cost order
05 setList(Q, T_Q);}           // adds/replaces TQ into catalog
```

Fig 9 – Bounded's addPlan Implementation

3.3 Implementation of *GetPlan* for Bounded

For user-defined constants $M \ge 1$ and $A \ge 0$, Bounded's *getPlan*(*Q*, *cpt*) searches for a pair $t_i=(cpt_i, plan_i, cost_i)$ and $t_j=(cpt_j, plan_j, cost_j)$ that bounds *cpt* (i.e., with $cost_i \le cost_j \le cost_i*M+A$ and with $cpt_i \triangleleft cpt \triangleleft cpt_j$). If it finds no such bounding pair, *getPlan* returns null. Otherwise, it returns such plan (see Fig 10 for a high-level description).

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

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

[1] All cost parameters we use are selectivities. Since higher selectivities imply more tuples to process, the monotonic assumption follows the intuition that plans that process more tuples likely cost more than plans that process less tuples. Although not true for all queries–e.g., queries using SQL clause NOT EXISTS may have non-monotonic costs–plans with non-monotonic costs are less common than plans with costs monotonic with the number of processed tuples.

Fig 10 – Bounded's getPlan Implementation

We next show that if *getPlan* returns plan p, it guarantees under the *Monotonic Assumption*, that the cost of executing p at cpt satisfies $Opt(cpt) \le p(cpt) \le Opt(cpt)*M+A$. We first show in Lemma 1 that if the Monotonic Assumption holds for every plan considered, then the cost of the optimal plan at any point (regardless of what the optimal plan is at any single point) also increases monotonically with the parameters.

Lemma 1: If $cpt_1 \lhd cpt_2$, $cost_1=p_1(cpt_1)=Opt(cpt_1)$, and $cost_2=p_2(cpt_2)=Opt(cpt_2)$ then $cost_1 \le cost_2$.
Proof: We note that if p_2 is optimal at cpt_1, then $cost_1=p_2(cpt_1)$. Otherwise, p_2 is not optimal at cpt_1, and therefore $cost_1<p_2(cpt_1)$. In any case, we have that $cost_1 \le p_2(cpt_1)$, which, coupled with the monotonic assumption and $cp_{t1} \lhd cp_{t2}$, it implies that $p_2(cp_{t1}) \le p_2(cp_{t2})=cost_2$. Puting the last two inequalities together, we obtain $cost_1 \le cost_2$. ■

Lemma 2: If $M \ge 1$, $cost_x \le cost_z \le cost_x*M+A$, and $cost_x \le cost_y \le cost_z$, then $cost_y \le cost_z \le cost_y*M+A$.
Proof: Since $M \ge 1$ and $cost_x \le cost_y$, it follows that $cost_x*M+A \le cost_y*M+A$. Also, since $cost_x \le cost_z \le cost_x*M+A$ it follows that $cost_z \le cost_x*M+A \le cost_y*M+A$. Finally, since $cost_x \le cost_y \le cost_z$ it follows that $cost_y \le cost_z \le cost_y*M+A$. ■

Finally, Theorem 1 establishes our desired result.

Theorem 1: If $t_i=(cpt_i, plan_i, cost_i)$ and $t_j=(cpt_j, plan_j, cost_j)$ are a bounding pair for some $M \ge 1$ and $A \ge 0$, then, under the Monotonic Assumption, the cost of $plan_j$ can be tightly bounded such that $Opt(cpt) \le plan_j(cpt) \le Opt(cpt)*M+A$, for all cpt such that $cpt_i \lhd cpt \lhd cpt_j$.
Proof: By Lemma 1 and $cpt_i \lhd cpt \lhd cpt_j$ it follows that $cost_i \le Opt(cpt) \le cost_j$. Also, by Lemma 2, and $cost_i \le cost_j \le cost_i*M+A$, we get $Opt(cpt) \le cost_j \le Opt(cpt)*M+A$. ■

Example 2: For some query Q, assume that *addPlan* was already called for the points (and associated triples) showed in Fig 11 (i.e., assume that the parametric plan stores information about the optimal plans and costs for the triples in $T_Q=(t_1, t_2, t_3, t_4, t_5, t_6, t_7)$). Given cpt (showed 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 $cpt_i \lhd cpt \lhd cpt_j$: (cpt_1, cpt_5), (cpt_1, cpt_6), (cpt_1, cpt_7), (cpt_3, cpt_5), (cpt_3, cpt_6), and (cpt_3, cpt_7). From those pairs, only two triples bound cpt: pair (t_3, t_5), because $c_3 \le c_5 \le c_3*1.5+0 \Leftrightarrow 6 \le 8 \le 9$, and pair (t_3, t_6), because $c_3 \le c_6 \le c_3*1.5+0 \Leftrightarrow 6 \le 9 \le 9$. Thus, either plan p_5 and plan p_6 can be safely returned by *getPlan*. ■

3.4 Efficient Implementation of *getPlan*

The naïve implementation of *getPlan* in Fig 10 enumerates all pairs of tuples $(t_i, t_j) \in T_Q \times T_Q$, $t_i \ne t_j$ that were introduced by *addPlan* and tests if any pair bounds cpt. If some pair (t_i, t_j) bounds cpt, then plan p_j can be returned as the answer to *getPlan*. The complexity of this procedure is clearly quadratic in the size of T_Q. To avoid the enumeration of all of pairs of triples that have to be checked, we apply an optimization that allows us to choose a single pair of triples (t_1, t_2) to be checked.

Definition [◂ (below) and ▸ (above) operators]. Given a list, T_Q, of k triples $(cpt_i, p_i, cost_i)$ ordered by $cost_i$, with $i=0...k-1$, where cpt_i is a *CostPoint* and $cost_i$ represents the cost of executing the optimal plan p_i at cpt_i and given cpt, another *CostPoint* we define the following operations:

1- $T_Q \lhd cpt$ is the list of triples $(cpt_i, p_i, cost_i)$ from T_Q, ordered by $cost_i$, such that $cpt_i \lhd cpt$.
2- $T_Q \rhd cpt$ is the list of triples $(cpt_i, p_i, cost_i)$ from T_Q ordered by $cost_i$, such that $cpt_i \rhd cpt$.

Example 3: Let $T_Q=(t_1, t_2, t_3, t_4, t_5, t_6, t_7)$, be triples shown in a 2-dimentional cost-based parameter space of Fig 11. Then $T_Q \lhd cpt=(t_1, t_3)$ (the triples in the light gray area) and $T_Q \rhd cpt=(t_5, t_6, t_7)$ (the triples in the dark gray area). ■

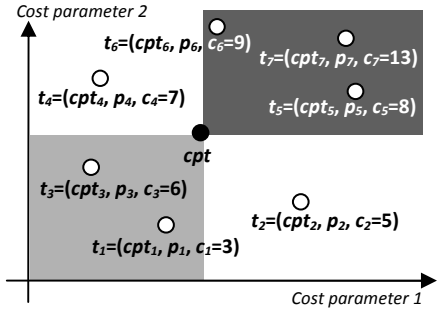

Fig 11 – T_Q=(t1, t2, t3, t4, t5, t6, t7)

As shown in Example 2 in the previous section, there is potentially more than one solution to *getPlan*(Q, cpt). We next show that, if there is a solution, we only need to check if $cost_{last} \le cost_{first} \le cost_{last}*M+A$, where c_{first} is the cost of the first triple in $T_Q \rhd cpt$ and c_{last} is the cost of the last triple in $T_Q \lhd cpt$. In such situation, then the plan in the first triple of $T_Q \rhd cpt$, p_{first} is returned. Theorem 2 proves the correctness of this approach.

Theorem 2: If $\exists cpt_b : t_b=(cpt_b, p_b, cost_b)$, $t_b \in T_Q \lhd cpt$, $\exists cpt_a : t_a=(cpt_a, p_a, cost_a)$, $t_a \in T_Q \rhd cpt$, and $cost_b \le cost_a \le cost_b*M+A$, then $cost_{last} \le cost_{first} \le cost_{last}*M+A$, where $cost_{first}$ is the cost of the first triple in $T_Q \rhd cpt$ and $cost_{last}$ is the cost of the last triple in $T_Q \lhd cpt$.

Proof: By definition, the *CostPoint* of any triple that belongs to the below list is below the *CostPoint* of any triple that belongs to the above list. Formally, $\forall cpt_b{:}t_b{=}(cpt_b, p_b, cost_b) \in T_Q{\lhd}cpt$, $\forall cpt_a{:}t_a{=}(cpt_a, p_a, cost_{ba}){\in} T_Q{\rhd}cpt$ we have that $cpt_b \lhd cpt \lhd cpt_a$. Then, by Lemma 1 we have that $cost_b{\leq}cost_{last}{\leq}Opt(cpt){\leq}cost_{first}{\leq}cost_a$. By $cost_b{\leq}cost_a{\leq}cost_b{*}M{+}A$ and Lemma 2, it follows that $cost_{last}{\leq}$ $cost_a{\leq}cost_{last}{*}M{+}A$. Also, if $cost_x{\leq}$ $cost_z{\leq}cost_x{*}M{+}A$ and $cost_x{\leq}cost_y{\leq}cost_z$, then $cost_x{\leq}$ $cost_y{\leq}cost_x{*}M{+}A$. Putting all together, it follows that $cost_{last}{\leq}cost_{first}{\leq}cost_{last}{*}M{+}A$. ■

The optimized implementation of *getPlan* is shown in Fig 12. We can see that given the properties of $T_Q{\lhd}cpt$ and $T_Q{\rhd}cpt$, it is possible to select a single triple t_1 from $T_Q{\lhd}cpt$ and a single triple, t_2 from $T_Q{\rhd}cpt$ such that only pair (t_1, t_2) needs to be checked. Note that the implementation of *getPlan* in Fig 12 makes at most a single pass over T_Q; thus, it has O($|T_Q|$) time complexity, where $|T_Q|$ is the number of elements in T_Q. (Note that the search condition depends on multiple attribute values –the cost parameters- and therefore more sophisticated search procedures such as binary search are not applicable) Before *addPlan* is called the first time, any *getPlan* call returns null. As new triples are added, the hit rate of *getPlan* is expected to increase. Intuitively, as more triples are added, the more likely it is that *getPlan* returns a plan because it is more likely that any two triples fulfill the requirements of Theorem 2. Note also that the lower the values of M and A, the less likely it is to find pairs of triples that fulfill the requirements of Theorem 2, and thus, more added triples are needed to obtain higher hit rates.

```
getPlan(inputs: Query Q, Cost-Point cpt;
        outputs: Plan plan)
01 List T_Q ←getList(Q);          // gets list of triples for Q
02 if (T_Q ==null) return null;
03 Triple last=null;              // last triple of TQ◁cpt
04 for Triple t in T_Q            // in cost order
05    if (t.cpt ≡ cpt) return t.p; // exact match
06    else if (t.cpt ◁ cpt) // keep track of last triple of TQ◁cp
07       last = t;
08    if (t.cpt ▷ cpt)            // first triple of TQ▷cpt
09       if (last == null) return null;
10       if (last.c ≤ t.c ≤ last.c*M+A)
11          return t.p;
```

Fig 12 – Bounded's getPlan Implementation

4 THE ELLIPSE-PPQO IMPLEMENTATION

Bounded's *getPlan* provides strong guarantees on the cost of plans returned. However, we expect low hit rates of Bounded's *getPlan* for small values of M and A, or before Bounded'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 of 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 $\Delta{\in}[0, 1]$, 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:

$$\frac{||cpt1 - cpt2||}{||cpt - cpt1|| + ||cpt - cpt2||} \geq \Delta$$

where $||p - q||$ is the Euclidian 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 2-dimentional space, the area where p is Δ-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, $||cpt1 - cpt2||$, over the sum of the distances between cpt and the foci, $||cpt - cpt1|| + ||cpt - cpt2||$, 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.

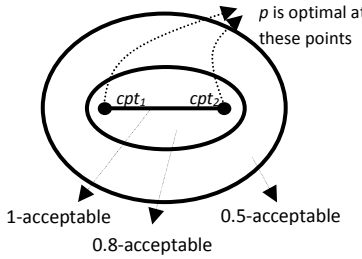

Fig 13 - Areas where p is Δ-acceptable

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 [16], 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.

4.1 Implementation of *addPlan* for Ellipse

The implementation of *addPlan* for Ellipse proceeds as follows. For each query Q and for each plan p that is optimal in some point of the parameter space, Ellipse's

addPlan(Q, cpt, p, cost) essentially maintains a list of (*cpt, cost)* pairs where *p* is optimal for *Q* (see *Fig 14*).

```
addPlan(inputs: Query Q, CostPoint cpt,
                Plan p, Cost cost)
01 PointList L ← getPointList(Q, p);// where is p optimal?
02 if (L==null) // if no PointsList
03    L = new PointList(); // create new one
04    PlanList P ← getPlanList(Q); // optimal plans for Q
05    if (P==null)P=new PlanList(p);
06    else         P.insert(p); // add new optimal plan to list
07    setPlanList(Q, P);      // adds/replaces list P in catalog
08 L.insert(cpt, cost);       // adds information about p.
09 setPointList(Q, p, L)      // adds/replaces list L in catalog
```

Fig 14 – Ellipse's addPlan Implementation

4.2 Implementation of *getPlan* for Ellipse

Ellipse's *getPlan* (see Fig 15) consists in the following. For each optimal plan *p*, it iterates over pairs of points where *p* is optimal for the given query, *Q*. For each pair of points (cpt_1, cpt_2), it tests if *p* is Δ-acceptable at the given point *cpt*. If it is, *getPlan* returns *p*, otherwise *getPlan* keeps trying other points and plans. If all pairs of plans for *Q* are exhausted without an Δ-acceptable plan being found, *getPlan* returns null. Note that we return the first Δ-acceptable plan and therefore *getPlan* depends on the order on which points are enumerated. Instead of returning the first match, we can consider all Δ-acceptable plans and return the one with the largest distance from Δ, which might improve the quality of the resulting plans at the cost of a slower implementation of *getPlan.*

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

01 PlanList P ←getPlanList(Q);    // gets optimal plans
02 if (P ==null)                  // tests for empty list
03    return null;
04 for Plan plan in P
05    PointList L ← getPointList(Q, plan);
06    for PointPair (cpt1, cpt2) in L // enums point pairs
07       if (Δ <= dist(cpt1, cpt2) /
             (dist(cpt, cpt1) + dist(cpt, cpt2)))
08          return plan;          // found Δ-acceptable plan
09 return null;
```

Fig 15 – Ellipse's getPlan Implementation

5 Experimental Evaluation

In this section we report an experimental evaluation of PPQO using Microsoft SQL Server 2005. The client application implements the pseudo-code described in sections 3 and 4 , and Microsoft SQL Server is used to obtain estimated optimal plans and estimated costs of plans.

5.1 Dataset, Metrics, and Setup

The TPC-H benchmark [17] was used to evaluate the PPQO implementations. Table 1 shows which tables are joined by each query. The tables are lineitem (L), orders (O), customer (C), supplier (S), part (P), partsupp (T), nation (N), and region (R).

Query	Tables Joined	Column 1	Column 2
7	LOCSNN	c_acctbal	o_totalprice
8	LOCPSNNR	s_acctbal	l_extendedprice
9	LOTPSN	s_acctbal	l_extendedprice
18	LLOC	c_acctbal	l_extendedprice
21	LLLOSN	s_acctbal	l_extendedprice

Table 1 – Description of TPC-H queries used

As in Reddy and Haritsa [16], and unless otherwise noted, we added two extra selections to the TPC-H queries to more easily explore the parameter space (see Section 5.7 for experiments with more than two selection predicates). The two selections are of the form $col_i \leq val_i$, i=1,2, where, for each query, col_i is one of the two columns shown in Table 1 and val_i is a random value from the domain of the column.

For each query tested, we generated 10,000 random val_1 and val_2 values. (A (val_1, val_2) pair is a *ValuePoint*.) To guarantee that random parameter values uniformly explore the parameter space, we altered the values in the columns subject to the extra selections to such that those values are uniformly distributed in their domains instead of using the non-uniform TPC-H generated distributions.

For each query and each ValuePoint *vpt* we make a *getPlan* lookup call (see Fig 5), where PP is either Optimize-Once, Optimize-Always, Bounded, or Ellipse. If *getPlan* returns a plan we call it a hit and check if the plan is optimal; if it is not optimal we check how its estimated cost compares with the estimated optimal cost. These give rise to the following metrics:

- **HitRate**: Fraction of *getPlan* calls that return a plan.
- **OptRate**: The percentage of plans that are optimal.
- **SO**: Measure of suboptimality: $p_{hit}(cpt)/Opt(cpt)$, with p_{hit}= getPlan(Q, cpt). SO≥1.
- **AvgSO**: The average of all SO values.
- **MaxSO**: The maximum of all SO values; reflects how risky a PP implementation can be.
- **Number of points**: Number of (*cpt*, *plan*, *cost*) triples stored in a ParametricPlan (i.e., number of misses).
- **Number of plans**: Number of distinct optimal plans.
- **QP**: Number of queries processed.

The experiments were run on a lightly loaded Pentium M at 1.73GHz with 1GB of RAM and using TPC-H scale

factor 1. Indexes and statistics were built on all columns subject to selections and on all primary and foreign key columns. To estimate the cost of sub-optimal plans returned by PPQO, each sub-optimal plan was forcibly cost by SQL Server [13]. Unless otherwise stated, Bounded used M=1.1 and A=0 and Ellipse used Δ=0.95.

5.2 Variation on HitRate and OptRate

The first experiment consisted in processing queries using 10,000 different random ValuePoints (value vectors) for each query and observing how HitRate and OptRate varied for Bounded and Ellipse. This experiment was performed for the five TPC-H queries listed in Table 1 and the results for three are shown in Figures 16–18 . Several trends can be observed:

- Ellipse always has a higher HitRate than Bounded;
- Except for Query 8 (more on this below), Bounded always has a higher OptRate than Ellipse.
- HitRate converges quickly, but OptRate converges slightly faster.
- HitRate monotonically increases as a function of QP (more processed queries imply more misses and each miss adds information to the ParametricPlan, therefore increasing the likelihood of future hits).
- OptRate naturally varies up and down, as the initial random (*cpt*, *plan*, *cost*) triples are added to the ParametricPlan object, until it converges.

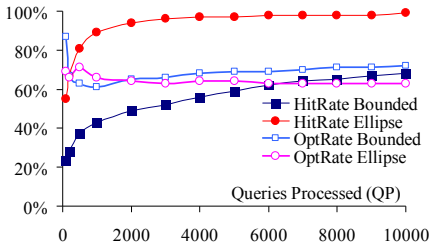

Fig 16 – HitRate and OptRate for Query 7

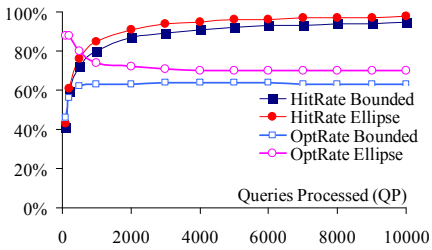

Fig 17 – HitRate and OptRate for Query 8

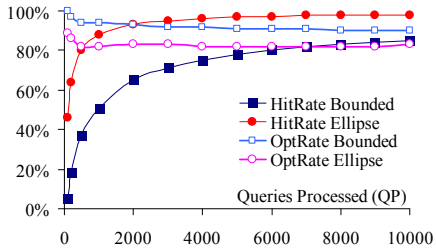

Fig 18 – HitRate and OptRate for Query 21

5.3 Number of Plans, of Points, Space, and Time

Fig 19 shows the number of plans and number of points for the experiments of the previous section. Bounded has a higher number of plans and number of points because it has a lower HitRate; for every miss there will be a new point stored in the ParametricPlan object.

Storing the number of plans and the number of points took only between ~600Kbytes to ~1300Kbytes using the original uncompressed XML plan representations provided by SQL Server. Storing zip-compressed XML plans instead would decrease the size of the plan representation by a factor of 10. (Plans do not need to be understood, zipped, or unzipped by *addPlan* or *getPlan* functions.)

Fig 20 reports the time and space taken by the Bounded and Ellipse approaches during optimization. Time (in seconds) includes time elapsed during optimization (if there is a miss), during *addPlan*, and during *getPlan*, but not execution time nor time consumed by function φ. For comparison purposes, the time taken for Optimize-Once and Optimize-Always is also included.

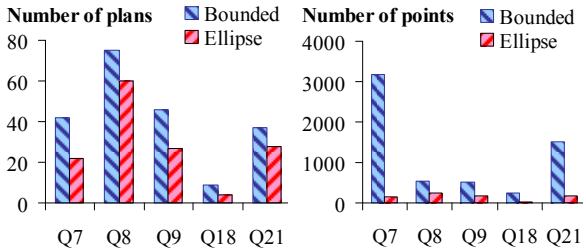

Fig 19 – Number of plans and points for 10,000 QP

After 10,000 queries have been processed, Optimize-Always took between 5.2 and 13.6 times longer than Bounded and between 10.7 and 18.5 times longer than Ellipse. Thus, although Bounded only used between 7% and 20% of the optimization time, it still returned plans that were, as shown in section 5.4, on average just 1% more costly then the optimal plan. Ellipse used between 5% and 9% of the optimization time and returned plans that were 6% more costly than the optimal plan. Ellipse was always faster than Bounded because it had less

optimize and *addPlan* calls (due to higher HitRates) and faster *getPlan* calls (because it has less information stored in its parametric plans).

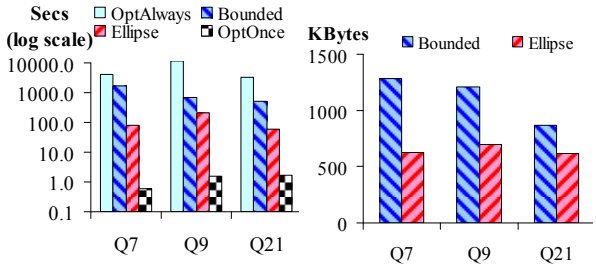

Fig 20 – Optimization time and space for 10,000 QP

Note that although Optimize-Once spends the least optimization time, it is not the best overall approach (as seen in Fig 24 below). In fact, the entire Parametric Query Optimization research area aims to overcome the performance problems of using Optimize-Once.

5.4 Quality of Returned Plans

The quality of the returned plans is described in this section. The sub-optimality of each plan returned by Bounded, Ellipse, and Optimize-Once was measured in the same experiments of the previous two sections.

Figures 21–23 show the quality of the returned plans (hits) for Bounded, Ellipse, and Optimize-Once in the form of cumulative distributions. The X axis represents how much the cost of a returned plan is above optimal, and the Y axis represents the cumulative percentage of plans that correspond to that sub-optimality level. For example, about 77% of the plans returned by Bounded for Query 7 are within 1% of the cost of optimal and 99.9% are within 10% the cost of optimal. The quality of most plans returned by Ellipse and Bounded is very good, and the quality of the plans returned by Bounded is higher.

To complete the picture, Figure 24 shows the average and maximum sub-optimality for the three policies and five queries. While both Bounded and Ellipse have very good average cases, Ellipse, can have as bad worse cases as Optimize-Once (but less frequently). Overall, Bounded's most sub-optimal plan was 5 times worse than the optimal plan, while the most sub-optimal plan chosen by both Ellipse and Optimize-Once was 412 times more costly than the optimal plan (MaxSO graph of Fig 24).

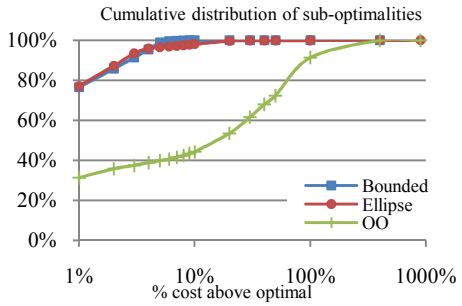

Fig 21 – Quality of returned plans (Q7)

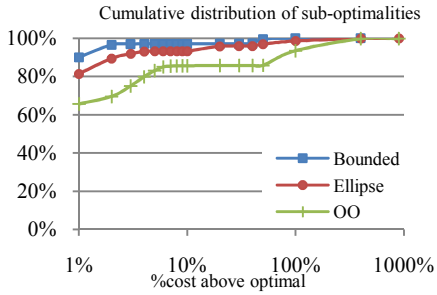

Fig 22 – Quality of returned plans (Q9)

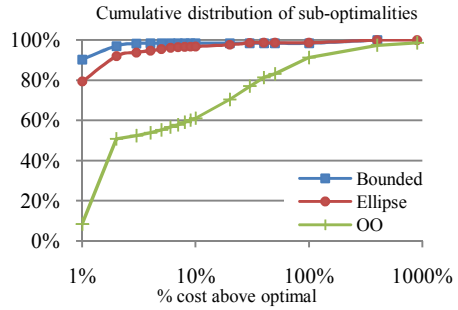

Fig 23 – Quality of returned plans (Q21)

An interesting observation is that although Bounded (with M=1.1) is supposedly guaranteed to return plans no more than 110% the cost of the optimal plan, in some experiments that guarantee was violated. Indeed, for queries 7, 8, 9, and 21, the most sub-optimal plan returned by Bounded was, respectively, 155%, 499%, 172%, and 177% the cost of the corresponding optimal plan. Further analysis showed that the problem lied with the tool that forces plans and that obtains the estimated cost of those plans. In some very rare cases, for a specific *CostPoint cpt*, the tool returned a plan, say, p_1 with cost c_1 at *cpt*, as if it was optimal, but some other plan, say, p_2, had an estimated cost c_2 at *cpt* lower than c_1. This lead to two problems: 1) Bounded stored plans and costs in its data structures that were not optimal; 2) the costs of the (presumed) optimal plan appeared non-monotonic. Other than those very rare occasions, Bounded guaranteed its sub-optimality specifications. (Arguably, this issue affected Ellipse less because the Ellipse implementation does not rely on monotonic cost functions.)

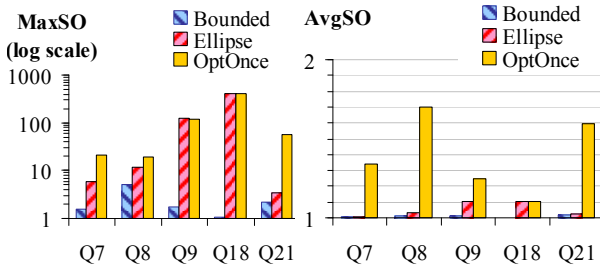

Fig 24 – MaxSO and AvgSO

Another surprise was how well Optimize-Once did in the AvgSO metric. On average, across all queries, Optimize-Once returned plans with costs ~140% the cost of optimal (the same average was ~101% for Bounded and ~106% for Ellipse). One possible explanation is the following. Optimize-Once obtains the optimal plan for the first of the 10,000 random parameter values and reuses that plan for all other values. If that first plan is also the plan with less cost variation in the plan space, then there is a high chance that that plan will do well in many other points in the space. Consider Fig 25, which shows a conceptual representation of the costs of four different plans, each optimal in different regions of the parametric space.

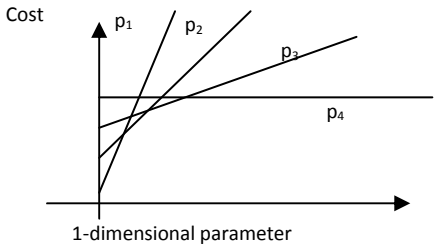

Fig 25 – Typical costs of optimal plans

Executing either plan p_3 or plan p_4 for all points of the parameter space would yield costs, on average, not much higher than the cost of optimal. Coincidently, the likelihood that any given point lies in the space where either p_3 or p_4 are optimal is very high, and thus, by random chance, Optimize-Once is likely to use a plan that is not catastrophic. However, Optimize-Once can and will return catastrophic plans eventually. We will explore this issue further in Section 5.6 - Vary Query Order.

5.5 Vary Bounded's *M* and Ellipse's Δ

In this experiment the value *M* of Bounded was varied from 1.1 to 3 for query 21. The values of OptRate and HitRate are shown in Fig 26 and Fig 27. As expected, a lower value for *M* (tighter optimality bound) results in a higher OptRate (because returned plans cannot be much worse than the corresponding optimal plans due to the tight optimality bound M) but a lower HitRate (because tight values of M result in small regions with quality guarantees and therefore a larger number of calls do not return any plan). Because the HitRate for M=1.5 is already so close to 100% (Figure 27), increasing M to 3 barely improves HitRate or change change OptRate much. Alternatively, it could have resulted in a small change in HitRate but a larger change in OptRate (as it does not happen, there might be a correlation between HitRate and OptRate in this scenario). The same query 21 with the same random parameter values was run using Ellipse while varying Δ from 0.9 to 0.99 (see Figures 28 and 29). As expected, a higher Δ results in a higher OptRate but a lower HitRate (the reasons are similar to those above). Due to space constraints, we do not report experiments varying Bounded's *A* parameter. (Results, however, were similar to the ones for M, i.e., larger values of A increase HitRate and decrease OptRate).

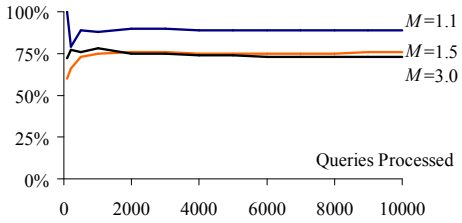

Fig 26 – OptRate for Bounded, vary M, Q21

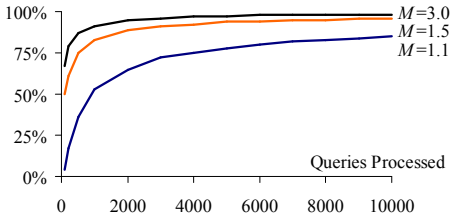

Fig 27 – HitRate for Bounded, vary M, Q21

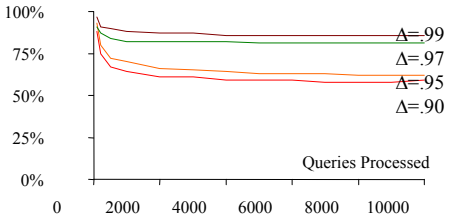

Fig 28 – OptRate for Ellipse, vary Δ, Q21

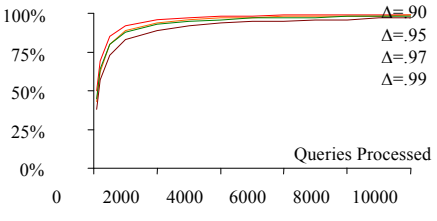

Fig 29 – HitRate for Ellipse, vary Δ, Q21

5.6 Vary Query Order

This experiment assessed the impact of the order of the incoming queries on the performance of the algorithms.

The same 10,000 random values used for Query 21 were used again, but the order in which those 10,000 queries were processed was chosen randomly. Six random orders were generated and processed with Bounded (M=1.1, A=0), Ellipse (Δ=0.9), and Optimize-Once.

The results are shown in Figures 30–33 and summarized in Table 2. Note that in Figures 30–33 it is not possible to tell apart which line is which. That is precisely the point: except for Ellipse's OptRate, query order essentially had no effect on the values of HitRate or OptRate.

Table 2 – Effects of Different Query Orders

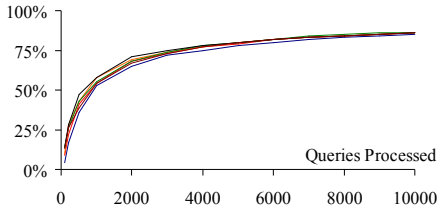

Fig 30 – HitRate for Bounded, vary query order, Q21

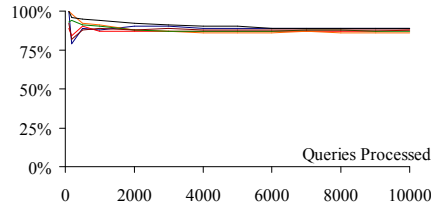

Fig 31 – OptRate for Bounded, vary query order, Q21

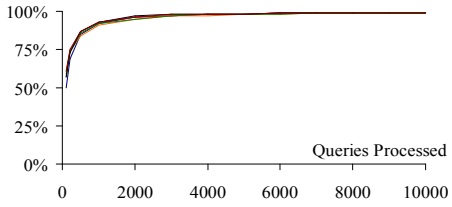

Fig 32 – HitRate for Ellipse, vary query order, Q21

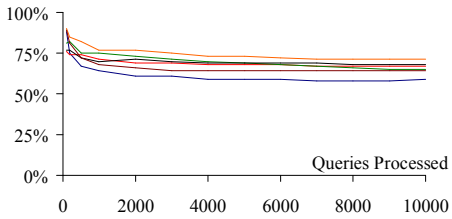

Fig 33 – OptRate for Ellipse, vary query order, Q21

Note that although query order had no impact in the final values of Bounded's OptRate, Bounded's HitRate, and Ellipse's HitRate, query order did have a medium impact on the final value of Ellipse's OptRate.

On the other hand, for Optimize-Once, query order had a very significant impact on OptRate, with final values ranging from 3% to 48%. An interesting observation is that the performance of Optimize-Once was exactly the same for four out of those six random orders. Further analysis showed that, although the very first value of each of the six random orders were all different, for four of them, the corresponding optimal plan was the same. This follows the observation (Section 5.4, Fig 25, and [16]) that some plans have very large optimality areas.

5.7 Vary Number of Dimensions

In all the experiments so far, the parameter space was 2-dimensional. The next experiment varies the number of dimensions, from 1 to 4. Query 8 is used (with extra parametric selections as needed) because it was the one with the highest number of plans and thus, more likely to suffer from the "curse of dimensionality": an exponential growth of complexity with a linear increase in the number of dimensions. The query was then run for 10,000 random values for Bounded (M=1.1, A=0) and Ellipse (Δ=0.95). The results, showed in Figures 34-37 are summarized in Table 3.

	Final OptRate			Final HitRate		
	Max	Min	Avg	Max	Min	Avg
Bounded	89.0%	86.0%	87.8%	86.0%	85.0%	85.8%
Ellipse	71.0%	59.0%	65.7%	99.0%	99.0%	99.0%
OptOnce	48.0%	3.0%	35.2%	-	-	-

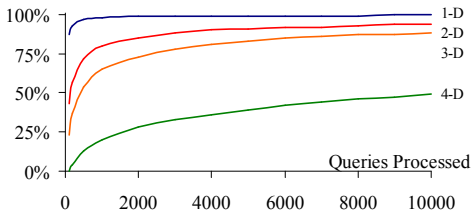

Fig 34 – Vary dimensions, HitRate for Bounded, Q8

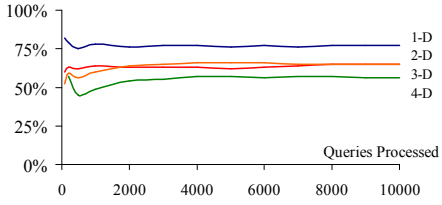

Fig 35 – Vary dimensions, OptRate for Bounded, Q8

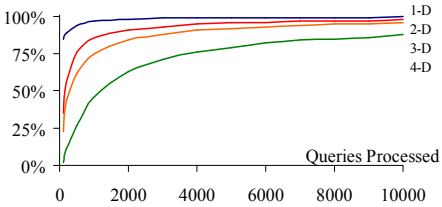

Fig 36 – Vary dimensions, HitRate for Ellipse, Q8

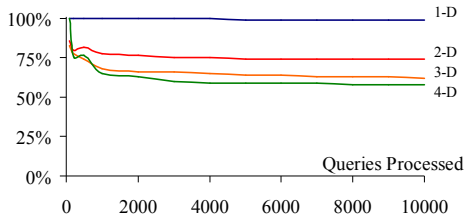

Fig 37 – Vary dimensions, OptRate for Ellipse, Q8

Table 3 – Variation of Number of Dimensions

It is clear that the more dimensions the parameter space has, the lower are the OptRate and HitRate. Some of the reasons that contribute to this effect are twofold. First, given a point *cpt* centered in the middle of the parameter space, the percentage of space ◂ *cpt* (or ▸ *cpt*) decreases exponentially with the number of dimensions (affects Bounded). That is, the larger the number of dimensions, the less likely it is that any two random points are above or below some other point. For example, for a 1-dim space, 50% of space is below (above) the mid-point. For 2-dim, 25% of the parameter space is below (above) the mid-point (12.5% for 3-dim, ~6% for 4-dim). Fig 38 shows the number of plans and points for Bounded. Second, the number of unique optimal plans increases exponentially with the dimensionality of the parameter space. This issue affects Ellipse because this approach relies on finding two close-by points where the same plan is optimal. Fig 39 shows the number of plans and points for Ellipse.

The number of plans and points increase exponentially for both Ellipse and Bounded, but slower for Ellipse. For each of the experiments above (which use 1, 2, 3 and 4 cost parameters), the returned plans were on average 7%, 8%, 45%, and 35% respectively more expensive than the optimal plans when using Ellipse and 0.2%, 2%, 24%, and 10% respectively more expensive than the optimal plans when using Bounded (not shown in the graphs).

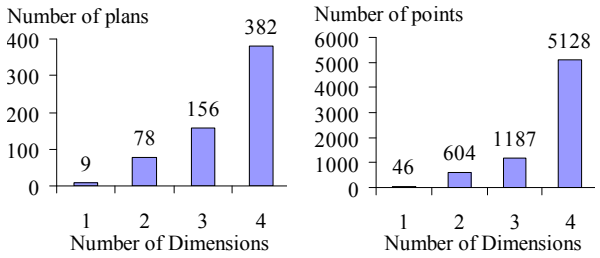

Fig 38 – Number of plans and points, Bounded, Q8

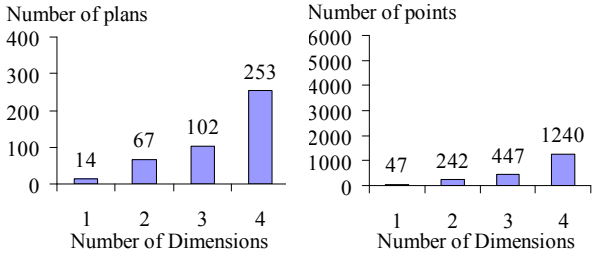

Fig 39 – Number of plans and points, Ellipse, Q8

	OptRate				HitRate			
	1-D	2-D	3-D	4-D	1-D	2-D	3-D	4-D
Bounded	77%	65%	65%	56%	100%	94%	88%	49%
Ellipse	99%	74%	62%	58%	100%	98%	96%	88%

6 Related Work

Parametric query optimization was first mentioned by Graefe [7] and Lohman [12]. This pioneering early work also proposed *dynamic query plans* and a new meta-operator, the *choose-plan* [7]. Dynamic query plans include more than one physical plan choice. The plan to use is determined at run-time 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 [2] 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 plans overlap, none is pruned. At run-time, when parameters are bound to values, the choose-plan selects the right plan. This approach may enumerate a large number of plans (see [15]), and all those plans may have to be re-cost at run-time. Ioannidis et al [10] coined the term Parametric Query Optimization 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. Ganguly [5] uses a geometric approach to solve the PQO problem for one and two parameters under the assumption that cost functions are linear and that regions of optimality of plans are convex. Ganguly solves PQO for restricted forms of non-linear, one-parameter, cost functions. Prasad [14] extends the geometric approach to solve PQO for ternary linear cost functions and binary non-linear functions. Hulgeri and Sudarshan [8] propose a solution to PQO that handles piecewise linear cost functions for an arbitrarily number of parameters but requires substantial changes to the query optimizer. AniPQO [9] is a recent technique that

approximates the solution to PQO for non-linear functions and for an arbitrary number of parameters. AniPQO approximates optimality regions to *n*-dimensional convex polytopes and finds its solution to PQO by calling the optimizer multiple times and evaluating plan costs up to thousands of times. Unlike AniPQO, PPQO never calls the optimizer or costs plans more often than what a traditional non-PQO approach would.

A closely related piece of work is PLASTIC [6]. 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 number of relations in the query, 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, recent work [16] coins the term "plan diagram" to denote a pictorial enumeration of the execution plan choices of a query optimizer over the selectivity space. This 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 Δ-acceptable regions. Very recently, in a follow-up to [16], the authors propose to reduce the plan diagram for a given query by "collapsing" plans whose costs are close enough to each other [3]. 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 [3] 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 builds a parametric plan data structure with no long startup costs.

7 Conclusions

Before Progressive Parametric Query Optimization (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% of the times, while still returning plans within 5% of the cost optimal plan for 99% 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 re-optimization [11] and other adaptive query processing (AQP) approaches [1, 4] 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 inter-query adaptive approach while AQP are frequently intra-query 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 φ 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 φ can be implemented by simply manipulating in memory histograms (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 4-D parameter space. However, even in this challenging scenario, Ellipse on average executed plans just 3% more costly than the optimal, while avoiding 87% of all optimization calls.

REFERENCES

[1] S. Babu, P. Bizarro. Adaptive Query Processing in the Looking Glass. In Proceedings of *CIDR 2005*.

[2] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In Proceedings of *SIGMOD 1994*.

[3] Harish D, P. Darera and J. Haritsa. On the Production of Anorexic Plan Diagrams. In Proceedings of *VLDB 2007*.

[4] A. 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 Optimization Algorithms. In Proceedings of *VLDB 1998*.

[6] A. Ghosh, J. Parikh, V. S. Sengar, J. R. Haritsa. Plan Selection Based on Query Clustering. In Proceedings of *VLDB 2002*.

[7] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In Proceedings of *SIGMOD 1989*.

[8] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In Proceedings of *VLDB 2002*.

[9] A. Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In Proceedings of *VLDB 2003*.

[10] Y. E. Ioannidis, R. T. Ng, K. Shim, and T K. Sellis. Parametric Query Optimization. In Proceedings of *VLDB 1992*.

[11] N. Kabra, D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In Proceedings of *SIGMOD* 1998.

[12] G. M. Lohman. Is Query Optimization a 'Solved' Problem? Workshop on Database Query Optimization. *Oregon Graduate Center Tech. Rep. 89-005*, 1989.

[13] Microsoft Corporation. Plan Forcing Scenario: Create a Plan Guide That Uses a USE PLAN Query Hint. *SQL Server 2005 Books Online*.

[14] V. G. V. Prasad. Parametric Query Optimization: A Geometric Approach. *MSC Thesis. IIT, Kampur*, 1999.

[15] S. V. U. Maheswara Rao. Parametric Query Optimization: A Non-Geometric Approach. *Master Thesis. IIT, Kampur*, 1999.

[16] N. Reddy and J. R. Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. In Proceedings of *VLDB 2005*.

[17] Transaction Processing Performance Council. *The TPC-H Benchmark*. http://www.tpc.org/. Accessed March 2006.