AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions

Arvind Hulgeri * S. Sudarshan

Indian Institute of Technology, Bombay
{aru, sudarsha}@cse.iitb.ac.in

Abstract

The cost of a query plan depends on many parameters, such as predicate selectivities and available memory, whose values may not be known at optimization time. Parametric query optimization (PQO) optimizes a query into a number of candidate plans, each optimal for some region of the parameter space. We propose a heuristic solution for the PQO problem for the case when the cost functions may be nonlinear in the given parameters. This solution is minimally intrusive in the sense that an existing query optimizer can be used with minor modifications. We have implemented the heuristic and the results of the tests on the TPCD benchmark indicate that the heuristic is very effective. The minimal intrusiveness, generality in terms of cost functions and number of parameters and good performance (up to 4 parameters) indicate that our solution is of significant practical importance.

1 Introduction

The cost of a query plan depends on various database and system parameters. The database parameters include selectivity of the predicates and sizes of the relations. The system parameters include available memory, disk bandwidth and latency. The exact values of these parameters may not be known at compile time. For example, in the case of embedded SQL queries containing parameters (unbound variables), and prepared statements in JDBC/ODBC, the values of the parameters are known only at query execution time. Similarly, the memory available for query execution is not known until the query execution time.

Optimizing a query afresh each time it is executed can add substantially to the cost of evaluation. On the other hand, optimizing a query into a single plan at compilation time may result in a substantially sub-optimal plan if the actual parameter values are different from those assumed at optimization time [GW89]. To overcome this problem, *parametric query optimization (PQO)* optimizes a query into a number of candidate plans, each optimal for some region of the parameter space [CG94, INSS92, GK94, Gan98]. At query execution time, when the actual parameter values are known, an appropriate plan can be chosen from the set of candidates, which can be much faster than re-optimizing the query.

Earlier work on PQO by [GK94, Gan98] handled only linear cost functions and some restricted forms of non-linear cost functions. Real world cost functions are often nonlinear and may be discontinuous. Earlier proposals that handle the general case of non-linear cost functions (e.g. [CG94, INSS92]) have severe drawbacks, as outlined in Section 7.

In earlier work [HS02] we proposed an intrusive solution for the case of piece-wise linear cost functions, but that algorithm requires substantial changes to the query optimizer, which may not be feasible in a practical setting.

In this paper, we propose a heuristic solution, which we call *AniPQO* (Almost Non-Intrusive Parametric Query Optimization), for the PQO problem for the general case when the cost functions may be nonlinear in the given parameters. AniPQO has the following desirable properties:

- AniPQO works with arbitrary nonlinear and discontinuous cost functions. An experimental evaluation suggests that it works well for standard cost models for relational operators, which involve non-linearity and discontinuity.
- AniPQO conceptually works for an arbitrary number of parameters. Although the optimiza-

* Work supported by an Infosys Fellowship

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

tion cost (and in fact even the number of parametrically optimal plans) can increase exponentially with the number of parameters, our experimental evaluation suggests that the algorithm is quite practical for up to 4 parameters.

- AniPQO is minimally-intrusive in the sense that it does not need to modify the conventional query optimizer, and can merely use it as a subroutine (invoking it with different parameter values). We also show how a tighter integration can lead to faster optimization.

- AniPQO uses an AND-OR DAG representation of plan alternatives, which gives two benefits: (a) it reduces the run-time overhead of plan choice, and (b) it increases the quality of the heuristic solution; being a heuristic, AniPQO may miss some optimal plans, but the DAG representation allows plan hybrids to be considered at run time, resulting in a better plan, without any extra effort. (See Section 5 for details.)

To the best of our knowledge, AniPQO is the first practical non-intrusive algorithm for the general case of PQO with nonlinear cost functions.

We have implemented the AniPQO algorithm and present a performance study. The study shows that the set of plans found by AniPQO is a "good" subset of the optimal plans i.e. for each point in the parameter space of interest either the optimal plan is in the set of plans found or the minimum cost plan amongst the plans found is only slightly costlier that the actual optimal plan; the maximum performance degradation observed is very small (3.5%). The time taken for optimization is within a reasonable factor of the time taken by ordinary query optimization when the number of parameters is small (up to 4 parameters); although more expensive than single query optimization, the extra effort for PQO can be amortized over a large number of calls with different parameter values.

The rest of the paper is organized as follows. Section 2 formally defines the parametric query optimization problem and provides background material on polytopes. Section 3 describes AniPQO, while Section 4 describes the representation and manipulation of the parameter space decomposition. Section 5 describes the DAG representation of the plans. Section 6 presents the results of the experimental evaluation of AniPQO. Related work is described in Section 7, while Section 8 concludes the paper.

2 Background

In this section we formally define the parametric query optimization problem and provide some background material on polytopes.

2.1 Problem Definition

The parametric query optimization (PQO) problem is defined as follows [Gan98]: Let $s_1, s_2, \ldots, s_n$ denote n parameters, where each s_i quantifies some cost parameter. Let the cost of a plan p be a function of these n parameters and let it be denoted by $C_p(s_1, s_2, \ldots, s_n)$. For every legal value of the parameters, there is some plan that is optimal for that value. Given a query and n parameters, the *maximum parametric set of plans (MPSP)* is the set of plans, each member of which is optimal for some point in the n-dimensional parameter space. The $MPSP$ may be defined as:

$MPSP = \{p \mid p$ is optimal for some point in the parameter space$\}$

For every legal value of the parameters there is a plan in the $MPSP$ that is optimal for that value and vice-versa. The *region of optimality* for a plan p is denoted by $r(p)$ and is the set defined as

$r(p) = \{(s_1, s_2, \ldots, s_n) \mid p$ is optimal at $(s_1, s_2, \ldots, s_n)\}$

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. The parametric query optimization (PQO) problem is to find a $POSP$; a mechanism to find the optimal plan at a given point in the parameter space at run-time is also required. For simplicity in notation, we assume that there is only one $POSP$; however our algorithm does not depend on this assumption.

The *parameter space decomposition* induced by a set of plans is defined as the partitioning of the parameter space into the regions of optimality of the plans in the set.

2.2 Polytopes

In the proposed solution, we need to represent and manipulate parameter space partitions. For parametric query optimization with linear cost functions, the regions of optimality are convex and, if the parameter space of interest is a convex polytope, the regions of optimality are also convex polytopes. For nonlinear cost functions, the regions of optimality are not convex but we approximate them with convex polytopes.

We formally define polytopes below [Mul94].

A *convex polytope* in $\Re^d$ is a nonempty region that can be obtained by intersecting a finite set of closed halfspaces. Each halfspace is defined as the solution set of a linear inequality of the form $a_1x_1 + a_2x_2 + \cdots + a_dx_d \geq a_0$, where each a_j is a constant, the x_j's denote the coordinates in $\Re^d$, and $a_1, a_2, \ldots a_n$ are not all zero. The boundary of this halfspace is the hyperplane defined by $a_1x_1 + a_2x_2 + \cdots + a_dx_d = a_0$. We denote the bounding hyperplane of a halfspace M_i by ∂M_i.

Let $P = \cap_i M_i$ be any convex polytope in $\Re^d$, where each M_i is a halfspace. A halfspace M_i is called *redundant* if it can be thrown away without affecting

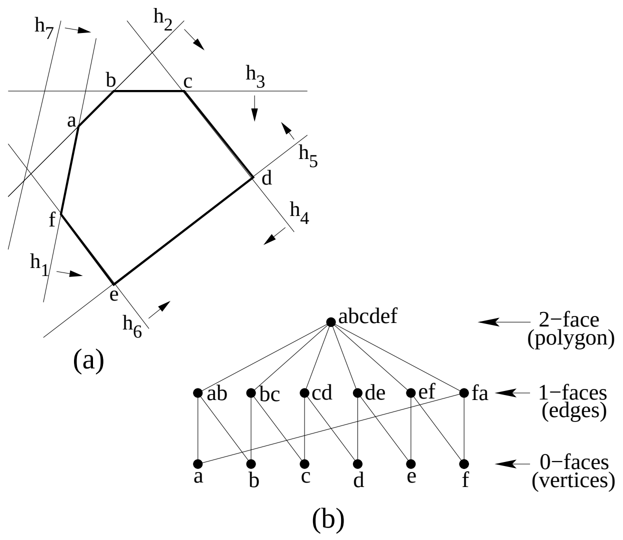

Figure 1: (a) a polytope and (b) its facial lattice

P. This means that the intersection of the remaining halfspaces is also P. Otherwise, the halfspace is called *non-redundant*. The hyperplanes bounding the non-redundant halfspaces are said to be the *bounding* hyperplanes of P. A *facet* of P is defined to be the intersection of P with one of its bounding hyperplanes. Each facet of P is a $(d-1)$-dimensional convex polytope. In general, an *i*-face of P is the (non-empty) intersection of P with $d-i$ of its bounding hyperplanes; a facet is a thus a $(d-1)$-*face*. For example, in three dimensions, a side (facet) of the polytope is a 2-face, an edge of the polytope is a 1-face, and a vertex is a 0-face.

Figure 1(a) shows a polygon *abcdef* in $\Re^2$ (a polytope in $\Re^2$ is a polygon.) It is defined by the halfspaces $h_1, h_2, \ldots, h_7$. On which side of the bounding hyperplane the corresponding halfspace lies is shown by an arrow. Note that the halfspace h_7 is redundant. (Part (b) of Figure 1 is discussed later.)

3 An overview of AniPQO

In this section we give an overview of *AniPQO*.

In general we are not interested in the whole parameter space $\Re^n$ (where n is the number of parameters) as only a part of it would constitute legal combinations of the parameter values. We assume that the parameter space of interest is a closed convex polytope, which we call the *parameter space polytope.*

Conventional query optimizers return an optimal plan along with its cost (at a given point in the parameter space). For parametric query optimization, we also need to find the cost of a given plan at a given point in the parameter space. Generally, a query optimizer does not support this but one can easily extend the statistics/cost-estimation component of the optimizer to do it.

Parametric query optimization involves two steps:

```
Algorithm: AniPQO
Input: $n$ (the number of parameters),
          parameter space polytope /* A polytope in $\Re^n$ */
Output: AniPOSP $\subseteq POSP$

DecompositionVertices =
     vertices of parameter space polytope
$CSOP = \emptyset$ /* Current set of optimal plans */
VerticesOptimized $= \emptyset$
While DecompositionVertices − VerticesOptimized $\neq \emptyset$
     $v$ = a vertex from
          DecompositionVertices − VerticesOptimized
     $p$ = ConventionalOptimizer($v$)
          /* $p$ is one of the optimal plans at $v$ */
     VerticesOptimized = VerticesOptimized $\cup \{v\}$
     If $p \notin CSOP$
          $CSOP = CSOP \cup \{p\}$
          DecompositionVertices = vertices of parameter
               space decomposition induced by $CSOP$
               /* Can be done using algorithm
                    UpdateDecompostionVertices
                    from Figure 4 in Section 4.3 */
return $CSOP$
```

Figure 2: Algorithm to find $POSP$

- Finding the $POSP$ (or as a heuristic, a subset thereof).
- Picking an appropriate plan from this set at run time, when the parameter values are known.

The pseudo code for the AniPQO heuristic for finding $POSP$ is shown in Figure 2. The procedure contains an abstraction for the procedure for finding the vertices of the parameter space decomposition induced by a plan set. It starts with an empty set of plans, $CSOP$ and the parameter space decomposition is the parameter space polytope itself. At each step, a non-optimized vertex of the decomposition is optimized. If the plan returned is not in $CSOP$, it is inserted in $CSOP$ and the parameter space polytope is decomposed afresh based on the new $CSOP$. The procedure is repeated till all the vertices of the decomposition are optimized.

At an abstract level this algorithm is the same as those from [Gan01, HS02][1]. For the case with linear cost functions, the algorithm is exact and finds the complete $POSP$ [Gan01, HS02].

However we propose to use it as a heuristic when cost functions are nonlinear, and hence it may not find all the plans in $POSP$. AniPQO differs from the earlier algorithms in the details of how it performs the parameter space decomposition, taking non-linearity

[1]Algorithms from [Gan01, HS02] mainly differ in that the former operates in $\Re^n$ and the latter operates in $\Re^{n+1}$, where n is the number of dimensions; the abstraction we present here works in $\Re^n$ and, hence, is closer to the algorithm from [Gan01].

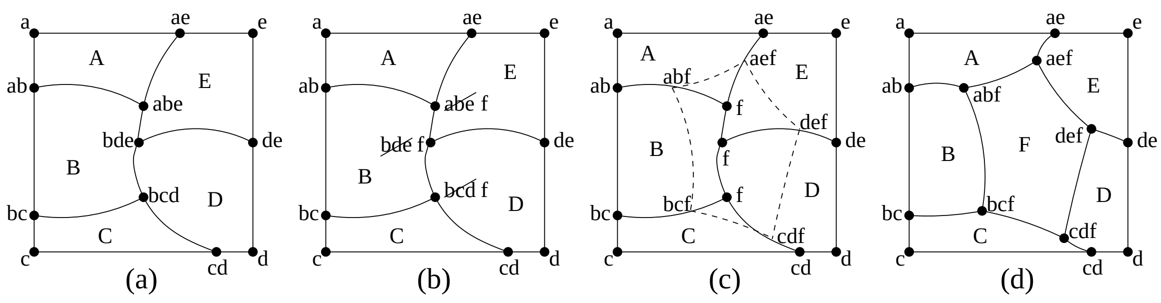

Figure 3: Carving out the region for a new plan in $CSOP$

of cost functions into account; the details are described in Section 4. We use the term *AniPOSP* to refer to the set of plans returned by AniPQO.

Example 3.1 Consider an example presented in Figure 3. The cost functions are nonlinear and the parameter space polytope is a 2-dimensional rectangle. Let $CSOP$ be $\{A, B, C, D, E\}$. Figure 3(a) shows the decomposition of the parameter space induced by $CSOP$. Each vertex of the decomposition is tagged by the set of plans from $CSOP$ that are optimal (within $CSOP$) at the vertex; the regions of optimality of the plans in the set surround the vertex.

We optimize the vertex with tag *bde* and let this return a new plan F. We add this plan to $CSOP$ and the resultant parameter space decomposition is shown in Figure 3(d). We get the new decomposition from the old one by carving out the region of optimality for the new plan F. Figures 3(b), (c) and (d) show the steps in this operation; the steps are explained in detail later. □

Optimality threshold (t): If the cost of a plan at a point is within a small percent of the optimal plan at the point then we may treat the plan to be optimal at the point.

We propose the following modification to the algorithm to bring down the number of calls to the optimizer: Consider an intermediate parameter decomposition induced by $CSOP$. We optimize an unoptimized vertex v and find a new plan $p \notin CSOP$. If the cost of some plan $p' \in CSOP$ is within a small percent t of the cost of p at v then we may discard p and treat p' as optimal at v. We mark v as optimized; $CSOP$ and the intermediate parameter decomposition continue to be the same.

In the case of linear cost functions, the above procedure guarantees that at any point in the parameter space polytope the best plan in the approximate $POSP$ is within t% of the optimal plan; see [HS03] for details.

For the case with nonlinear cost functions the procedure can not guarantee such a bound and can only be used as a heuristic. We adopt it with the following modification: Instead of discarding the new plan p, we do not use it for the further decomposition of the parameter space polytope, but include it in *AniPOSP*. This will help boost the quality of results.

At runtime, when the parameter values are known, the optimal plan has to be chosen. One approach is to index the parameter space decomposition and to use the index to find the appropriate plan from $POSP$; we propose a method for approximate indexing in [HS03]. A simpler approach is to find the optimal plan by evaluating the cost of each plan in $POSP$. We have implemented an optimization of this approach, using an AND-OR DAG framework, which also helps improve the quality of plans, when a heuristic algorithm to find $POSP$ returns only a subset of $POSP$. Section 5 provides details of this optimization.

4 Representation and manipulation of the decomposition

Figure 3 illustrates the operation of carving out the region of optimality for a new plan added to $CSOP$; the regions of optimality are nonlinear and defining and manipulating them exactly even for a 2 parameter case is not easy. We approximate them with convex polytopes which can be represented and manipulated efficiently.

4.1 Facial lattice of a polytope

In this section we describe a method (from [Mul94]) to represent convex polytopes in high-dimension.

A polytope P can be represented by its *facial lattice*. The facial lattice of P contains a node for each face of P. Two nodes are joined by an adjacency edge iff the corresponding faces are adjacent and their dimensions differ by one. Figure 1(b) shows the face lattice for the polygon in Figure 1(a).

Let d be the dimension of polytope P. For $j \leq d$, we define the j-skeleton of P to be the collection of its i-faces, for all $i \leq j$, together with the adjacencies among them. Thus, the j-skeleton can be thought of as a sublattice of the facial lattice of P. An important special case is the 1-skeleton of P. It is also called the *edge skeleton*. If we remove the node *abcdef* – the 2-face – and the edges incident on it from Figure 1(b), we get the edge skeleton of the polytope in Figure 1(a).

4.2 Facial lattice and edge skeleton of the decomposition

We define the facial lattice of the parameter space decomposition as the combined facial lattice of the polytopes defining the decomposition. We club all the polytope lattices by merging duplicate (shared) faces, to get the combined facial lattice.

The algorithm for finding $POSP$ from the previous section needs only the set of decomposition vertices, and not the complete decomposition. When a new plan is added to $CSOP$ we need to update the set of decomposition vertices; to do this, we need to know only the edge skeleton of the decomposition, as explained in the next section.

For each decomposition vertex, we store the set of plans from $CSOP$ that are optimal (within $CSOP$) at the vertex. This information is enough to find the edge skeleton of the decomposition and hence to update the set of decomposition vertices when a new plan is added to $CSOP$, as explained below.

Before we formalize the condition for an edge to exist between two vertices of the decomposition, we informally explain it using the example in Figure 3.

Example 4.1 The edges defining the decomposition in Figure 3 are of two types: those on the boundary of the rectangle and those inside it. Each edge in the former set (e.g. *ab*-*bc*) is on the boundary of the region of optimality of a single plan and hence its endpoints share one common label. Each edge in the latter set (e.g. *abe*-*bde*) is on the common boundary of the regions of optimality of two plans and hence its endpoints share two common labels.

Consider vertices with labels *ab* and *bc*; the line segment between them lies on the boundary of the rectangle and they have one common label and hence there is an edge between them in the edge skeleton. Consider vertices with labels *ab* and *c*; they do not share common label and hence there is no edge between them in the edge skeleton.

Consider vertices with labels *abe* and *bde*; the line segment between them lies inside the rectangle and they have two common labels and hence there is an edge between them in the edge skeleton. Consider vertices with labels *abe* and *bcd*, the line segment between them lies inside the rectangle and they have only one label in common and hence there is no edge between them in the edge skeleton. □

We now formally define the vertices and the edges of the decomposition. Let P be a set of plans and $\mathcal{V}_P$ be the vertices of the parameter space decomposition induced by P.

For each $v \in \mathcal{V}_P$, we define:

Set of optimal plans, $\mathcal{O}_v^P$: as the set of plans optimal (within P) at v; formally,

$$\mathcal{O}_v^P = \{p | p \in P \wedge \forall p' \in P, \text{cost of } p \text{ at } v \leq \text{cost of } p' \text{ at } v\}$$

For each $\mathcal{S} \subseteq \mathcal{V}_P$ we define:

Set of optimal plans, $\mathcal{O}_\mathcal{S}^P$: the set of plans optimal (within P) at each vertex in $\mathcal{S}$; formally,
$\mathcal{O}_\mathcal{S}^P = \cap_{v\in\mathcal{S}}\mathcal{O}_v^P$

Face dimension, $\mathcal{F}_\mathcal{S}$: the minimum i s.t. $\mathcal{S}$ is contained in an i-face of the parameter space polytope.

Assumption: If a set, P, where $n < |P|$, of plans is equi-cost at a point in a n dimensional face of the parameter space polytope then no $P' \subseteq P$, where $n < |P'|$, is equi-cost at any other point in the parameter space polytope.

Such a strong assumption is made for the sake of ease of description[2] and it may not hold true in practice. In the implementation of the algorithm we assume that if more than n cost functions meet at a point in a n dimensional face of the parameter space polytope then the point is a vertex of the parameter space decomposition; if there are multiple such points, we pick any one of them.

Lemma 4.1 $v \in \mathcal{V}_P \Leftrightarrow \mathcal{F}_{\{v\}} < |\mathcal{O}_{\{v\}}^P|$ □

Lemma 4.2 *The edge skeleton of the decomposition of the parameter space polytope induced by P contains edge (u, v), where $u, v \in \mathcal{V}_P$, iff $\mathcal{F}_{\{u,v\}} \leq |\mathcal{O}_{\{u,v\}}^P|$.* □

Theorem 4.3 *The edge skeleton of the parameter space decomposition induced by P can be constructed given the set of decomposition vertices $\mathcal{V}_P$, with each vertex v annotated by $\mathcal{O}_v^P$.* □

For a given set of decomposition vertices, the edge skeleton can be easily computed by testing each pair of vertices with a time complexity of $O(|\mathcal{V}_P|^2|P|)$. But we need to maintain the edge skeleton incrementally; each time a plan is added to the $CSOP$, we need to update the edge skeleton and this is done incrementally as described in Section 4.3.

Figure 3(a) shows the decomposition of parameter space rectangle induced by plan set $\{A, B, C, D, E\}$ for non-linear cost functions and Figure 7(a) shows the edge skeleton of the decomposition; for a 2-dimensional parameter space the facial lattice of the decomposition is the same as its edge skeleton. Each vertex of the decomposition is tagged by the set of plans that are optimal (within CSOP) at the vertex.

4.3 Updating the decomposition vertex set

Figure 4 gives pseudo code for an algorithm `UpdateDecompostionVertices` which updates the set of decomposition vertices when a new plan is detected. This is an exact algorithm for the linear case, but may

[2]We can relax the assumption if the cost functions intersect transversally; see [HS03] for details.

```
Algorithm: UpdateDecompostionVertices
Input: $CSOP$ (the current set of optimal plans),
        $\mathcal{V}$ (the current set of decomposition vertices),
        $p$ (a new plan)
Output: $\mathcal{V}$ (the updated set of decomposition vertices)

/* Update the set of decomposition vertices $\mathcal{V}$
   when a new plan $p$ is added to $CSOP$ */
For each edge $(u, v)$ in edge skeleton s.t.
        $p$ is optimal (w.r.t. $CSOP \cup \{p\}$) at $v$ and
        $p$ is not optimal (w.r.t. $CSOP \cup \{p\}$) at $u$
    $P = \mathcal{O}^{CSOP}_{\{u,v\}} \cup \{p\}$ /* $\mathcal{O}^{CSOP}_{\{u,v\}}$ is the set of plans
            in $CSOP$ that are optimal (within $CSOP$)
            at both $u$ and $v$ */
    $R =$ hyper-rectangle with $u$ and $v$
            as diagonal vertices
    $w =$ FindEquiCostPoint($R$, $P$) /* Find vertex $w$ in $R$
        s.t. at $w$ plans in $P$ are equi-cost; this may fail;
        See Figure 5 in Section 4.4. */
    If previous step fails
        $R =$ parameter space hyper-rectangle
        $w =$ FindEquiCostPoint($R$, $P$)
        /* This step too may fail; */
    If vertex $w$ is found, insert $w$ in $\mathcal{V}$
    Remove from $\mathcal{V}$ the vertices at which only $p$ is optimal
        and no plan from $CSOP$ is optimal
```

Figure 4: Algorithm to update decomposition vertices

miss some vertices in the non-linear case. As a result, AniPQO may miss some plans that it would have found if all the decomposition vertices had been detected. However, experiments in Section 6 suggest that this does not affect the quality of the solution much.

The algorithm finds the vertices of the existing decomposition at which the new plan is optimal and finds "conflicting" edges. A **conflicting edge** is defined as an edge in the decomposition s.t. the new plan (to be added to $CSOP$) is optimal at one end and sub-optimal at the other end. Each conflicting edge gives rise to a new decomposition vertex; before formalizing this, we informally explain it using an example.

Example 4.2 Consider the example in Figure 3. We optimize the vertex with label *bcd* and find a new plan F. We wish to update the decomposition by carving out the region of optimality for plan F. Along the contour *ab*-*abe*, plans A and B are equi-cost; the set of equi-cost plans along a contour is the intersection of the labels of the endpoints. Plan F is optimal at vertex *abe* and suboptimal at vertex *ab*; plans A and B are optimal at vertex *ab* and suboptimal at vertex *abe*. So plans A, B and F are optimal at a point on the contour and we wish to locate the point. □

Now we formalize the claim that each conflicting edge gives rise to a new decomposition vertex. Let the set of plans found so far be $CSOP$; the set of decomposition vertices would be $\mathcal{V}_{CSOP}$. We optimize one of the unoptimized vertices and let this return a new plan $p \notin CSOP$. Let $CSOP' = CSOP \cup \{p\}$ be the new set of optimal plans.

Consider a conflicting edge (u, v). Let at vertex v plan p be optimal (w.r.t $CSOP'$) and at vertex u plan p be sub-optimal (w.r.t $CSOP'$); thus vertex v would lie in the region of optimality of plan p, and vertex u would lie outside the region of optimality of plan p in the decomposition induced by $CSOP'$.

Using Lemma 4.2 we have,

$$\mathcal{F}_{\{u,v\}} \leq |\mathcal{O}^{CSOP}_{\{u,v\}}| \qquad (1)$$

There is a contour[3] between vertices u and v along which the plans in $\mathcal{O}^{CSOP}_{\{u,v\}}$ are equi-cost. Plan p is optimal at v, and its cost is less than the cost of the plans in $\mathcal{O}^{CSOP}_{\{u,v\}}$ at v; plan p is not optimal at u, and its cost is more than the cost of the plans in $\mathcal{O}^{CSOP}_{\{u,v\}}$ at u.

Thus, for the plans in $\mathcal{O}^{CSOP}_{\{u,v\}} \cup \{p\} \subseteq CSOP'$, the equi-cost point lies on the equi-cost contour of the plans in $\mathcal{O}^{CSOP}_{\{u,v\}}$ between vertices u and v; let the point be w. We have,

$$\mathcal{F}_{\{w\}} \leq \mathcal{F}_{\{u,v\}}$$

$$\mathcal{O}^{CSOP'}_{\{w\}} = \mathcal{O}^{CSOP}_{\{u,v\}} \cup \{p\}$$

and, from Equation 1, we get,

$$\mathcal{F}_{\{w\}} < |\mathcal{O}^{CSOP'}_{\{w\}}|$$

By Lemma 4.1, w is a vertex of the new parameter space decomposition (induced by $CSOP'$).

In the case of linear cost functions, the equi-cost contour is a straight line and finding the new vertex is straightforward. But in the case of nonlinear cost functions the equi-cost contour may not be a straight line; hence finding the new vertex is not easy. We assume that the new vertex lies in the hyper-rectangle with the line segment (u, v) as a diagonal and try to find the vertex in this hyper-rectangle as described in Section 4.4. If this fails, we search the smallest hyper-rectangle which contains the parameter space polytope[4] but this may also fail.

If we fail to find the new vertex, the resulting decomposition may not be well-defined (for an example see Section 4.5). The AniPQO algorithm works with such ill-defined decompositions but may not detect some plans that it would have detected otherwise.

4.4 Finding an equi-cost point

Figure 5 gives pseudo code for a heuristic algorithm `FindEquiCostPoint` to find an equi-cost point within

[3]A straight line segment in the linear case.

[4]In our implementation, the parameter space polytope itself is a hyper-rectangle and we search it.

```
Algorithm: FindEquiCostPoint($R$, $P$)
Input: $R$ (a hyper-rectangle), $P$ (a set of plans)
Output: point $c \in R$ at which plans in $P$ are equi-cost

/* Find a point in hyper-rectangle $R$
    at which plans in $P$ are equi-cost */
Let $\mathcal{V}_R$ be the set of vertices of $R$
Let $\mathcal{H}_R$ be the dimension of $R$
Label each vertex $v \in \mathcal{V}_R$ by $\mathcal{O}_v^P$
/* $\mathcal{O}_v^P$ is the set of plans from $P$ that are
    optimal (within $P$) at $v$ */
Let $\mathcal{U}_R^P = \cup_{v \in \mathcal{V}_R} \mathcal{O}_v^P$
/* Each plan in $\mathcal{U}_R^P$ is optimal (within $P$) at
    at least one vertex of $R$. */
If $\mathcal{U}_R^P \neq P$
    return NULL /* the desired point is not found in $R$ */
Let $c$ be the centre of $R$
If {all the plans in $P$ are equi-cost (within a threshold)
    at $c$} OR {$R$ can not be partitioned further}
    return $c$
Partition $R$ into smaller $2^{\mathcal{H}_R}$ rectangles and
apply the same procedure till we find the desired point
```

Figure 5: Algorithm to find approximate equi-cost point for a set of plans

a given hyper-rectangle, for a given set of plans with nonlinear cost functions. The algorithm tries to find a point at which the given plans are approximately equi-cost (i.e. their costs are within some threshold of each other) and, if such a point is found, takes that point as an approximation of the actual equi-cost point. The algorithm uses a heuristic test, explained later in this section, to determine if the equi-cost point is contained in a given hyper-rectangle with each of its vertices tagged with the plans optimal at it. We start with a hyper-rectangle for which the test evaluates positive; partition it and pick a partition on which the test evaluates positive. We keep doing this recursively till the plans are approximately equi-cost at the centre of the hyper-rectangle and take the centre as an approximation of the actual equi-cost point.

For n parameters, we need at least $n+1$ plans to define an equi-cost point. We heuristically claim that iff the size of a plan set is more than the dimension of a hyper-rectangle in the parameter space and each plan in the set is optimal at at least one vertex of the hyper-rectangle then the equi-cost point of the plans in the set lies in the hyper-rectangle. Consider the rectangle shown in Figure 6 with the regions of optimality defined for three plans. Each plan is optimal at at least one vertex of the rectangle and the equi-cost point lies in the rectangle.

Consider a hyper-rectangle R and a set of plans P. Let $\mathcal{V}_R$ be the set of vertices of R and $\mathcal{H}_R$ be the dimension of R. We define $\mathcal{U}_R^P \subseteq P$ as follows,

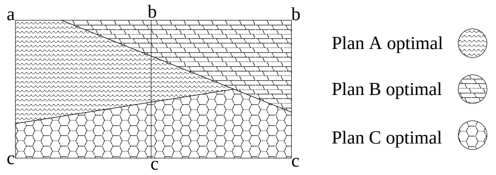

Figure 6: Example of equi-cost point approximation

$$\mathcal{U}_R^P = \cup_{v \in \mathcal{V}_R} \mathcal{O}_v^P$$

Thus, each plan in $\mathcal{U}_R^P$ is optimal (within P) at at least one vertex of R.

We make following heuristic assumption.
Iff $\mathcal{H}_R < |\mathcal{U}_R^P|$ then R contains a point at which the plans in P are equi-cost[5]

The above assumption may fail in one of two ways:

- It fails if R contains a equi-cost point but $|\mathcal{U}_R^P| \leq \mathcal{H}_R$. The square on the right in Figure 6 is an example. The square contains a decomposition vertex though $\mathcal{U}_R^P = \{b, c\}$ and $|\mathcal{U}_R^P| = \mathcal{H}_R = 2$. We miss the equi-cost point in this case and this results in an incomplete edge skeleton (see Section 4.5); but our experiments suggest that this does not affect the quality of the solution much.

- It fails if R does not contain an equi-cost point though $\mathcal{H}_R < |\mathcal{U}_R^P|$. The square on the left in Figure 6 is an example. The square does not contain a decomposition vertex though $\mathcal{U}_R^P = \{a, b, c\}$ and $2 = \mathcal{H}_R < |\mathcal{U}_R^P| = 3$. In this case, we unnecessarily explore the region which we need not.

If $\mathcal{H}_R < |\mathcal{U}_R^P|$, we evaluate the costs of the plans in P at the centre point of R. If, either the plans are equi-cost (within a threshold) or R can not be further partitioned, we take the centre as an approximation of the equi-cost point. Else, we partition R into $2^{\mathcal{H}_R}$ equal sized hyper-rectangles and recursively examine them.

4.5 An example iteration of AniPQO

Let us consider the example from Figure 3 and step through the algorithm.

Figure 7(a) shows the edge skeleton of the decomposition in Figure 3(a). We optimize the vertex with tag *abe* and generate a new plan F. We evaluate its cost at all the vertices and, say, it is optimal at vertices with tags *abe*, *bde* and *bcd* and suboptimal at the rest of the vertices (Figure 7(b)). The conflicting edges are (abe, ab), (abe, ae), (bcd, bc), (bcd, cd) and (bde, de).

Consider conflicting edges (abe, ab) and (abe, ae). We create two rectangles, one with each edge as a diagonal (Figure 7(c)) and assume that the equi-cost

[5]This may not be true even for the linear case except when $\mathcal{H}_R = 1$ i.e. R is a line segment; it is true for $\mathcal{H}_R = 1$ with continuous cost functions (by the intermediate value theorem).

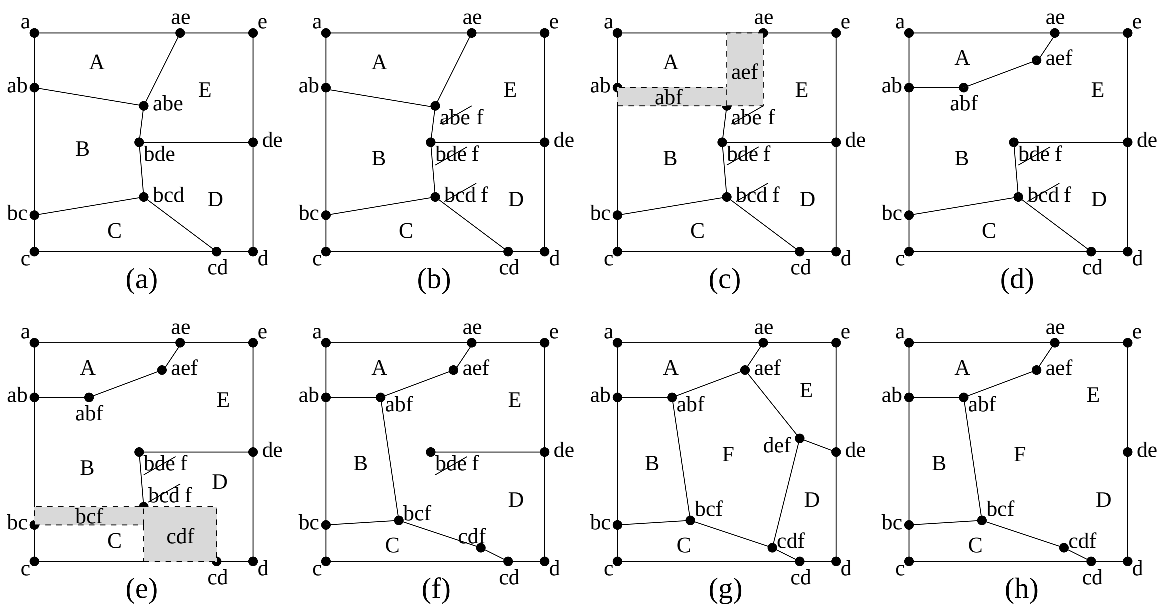

Figure 7: An iteration of *AniPQO* algorithm

vertex corresponding to a conflicting edge lies in the rectangle thus created. Each rectangle is labeled by the set of plans that would be equi-cost at a point we search for in the rectangle.

Assume that we find both the vertices and insert them, with proper labels, into the set of decomposition vertices. The vertex with label *abe* ceases to be a decomposition vertex and we remove it from the set of decomposition vertices. The resulting intermediate edge skeleton is shown in Figure 7(d).

Next, we consider two more conflicting edges (bcd, bc) and (bcd, cd) and repeat the above mentioned procedure. See Figures 7(e) and (f).

Now, we consider the remaining conflicting edge (bde, de). As we can see from Figures 3(d) and 7(f), the desired new vertex is not contained in the rectangle formed by the edge – in fact, it is not a rectangle but a straight-line (Figures 7(f)). So we search the entire parameter space for the vertex; we may or may not find the vertex.

If we find the new vertex, it is inserted in the set of decomposition vertices with proper tagging and the resulting edge skeleton is shown in Figure 7(g). Figure 7(g) is the edge skeleton of the nonlinear decomposition in Figure 3(d).

If we fail to find the new vertex, we end up in a situation where we do not have all the decomposition vertices and hence the edge skeleton is incomplete; see Figure 7(h). Missing vertices in the decomposition lead to an incomplete edge skeleton. Whenever a new plan is added, some conflicting edges may be missed and hence so may some vertices in the new decomposition; we may miss some plans because of this. As explained earlier, our experiments show that the quality of the solution is not greatly affected.

5 The DAG Representation of Plans

In this section we describe an AND-OR DAG representation of a set of plans and how we use it to boost the quality of the results and facilitate picking an optimal plan at run time. This representation is used, for example, in the Volcano optimizer generator [GM93], and provides a very compact and efficient way of representing alternative plans, without redundancy.

An AND–OR DAG is a directed acyclic graph whose nodes can be divided into AND-nodes and OR-nodes; the AND-nodes have only OR-nodes as children and OR-nodes have only AND-nodes as children.

An AND-node in the AND-OR DAG corresponds to an algebraic operation, such as a join operation ($\bowtie$) or a select operation (σ). It represents the expression defined by the operation and its inputs. The AND-nodes are referred to as *operation nodes*. An OR-node in the AND-OR DAG represents a set of logical expressions that generate the same result set; the set of such expressions is defined by the children AND nodes of the OR node, and their inputs. The OR-nodes are referred to as *equivalence nodes*.

Properties of the results of an expression, such as sort order, that do not form part of the logical data model are called *physical properties*. It is straightforward to refine the above AND-OR DAG representation to represent physical properties and obtain a physical AND-OR DAG.

Let $children(e)$ and $children(o)$ be the set of children of equivalence node e and operation node o resp.; let p_o be the optimal plan with o as the root operation; let $cost(e)$, $cost(o)$ and $cost(p_o)$ be the cost of the optimal plan for equivalence node e, the cost of operation node o and the cost of the optimal plan rooted at operation node o. Then the costs of the equivalence nodes and the operation nodes are given by the following recursive equations (relation scan and index scan form the base case):

$cost(p_o) = cost(o) + \sum_{e_i \in children(o)} cost(e_i)$ [AND]

$cost(e) = \min_{o_i \in children(e)} cost(p_{o_i})$ [OR]

Storing *AniPOSP* in a DAG

AniPQO builds an AND-OR DAG of the plans in *AniPOSP* at compile time. Common/equivalent subexpressions (across plans) are represented by a single equivalence node. At run time, we choose the best plan (amongst the plans in the DAG) at the given point in the parameter space. The cost of finding the best plan in an AND-OR DAG is linear in the size of the DAG. We can re-use parts of the optimizer code to build and manipulate the DAG.

The DAG framework provides two benefits:

- **Reduced effort in picking a plan at runtime:** In the DAG framework, if two plans share a operator/subplan, we need to cost the operator/subplan only once. The benefit is clearly illustrated by one of the queries we tested, where the number of plans in $POSP$ is 134 and the sum of the number of operators across these plans is 1816, but the number of operators in the DAG built using these plans is just 85.

- **Choosing a plan not in *AniPOSP*:** When we merge a number of plans in a DAG, an equivalence node may have more than one subplan under it, each coming from a different original plan. When we find an optimal plan for the equivalence node for given parameter values, we evaluate the cost of all the subplans of the equivalence node and pick the one with the least cost.

  This may result, for the given parameters, in finding an optimal plan which is not amongst the plans used to build the DAG. For example, consider two plans p_1 and p_2 used to build a DAG. Let e_1 and e_2 be the equivalence nodes present in both the plans. In plan p_1, let subplan $s_{e_1}^{p_1}$ evaluate e_1 and subplan $s_{e_2}^{p_1}$ evaluate e_2. In plan p_2, let subplan $s_{e_1}^{p_2}$ evaluate e_1 and subplan $s_{e_2}^{p_2}$ evaluate e_2. For the given parameter values, say, s_{e1}^{p1} is cheaper than $s_{e_1}^{p_2}$ and s_{e2}^{p2} is cheaper than $s_{e_2}^{p_1}$; then a hybrid of p_1 and p_2 (containing s_{e1}^{p1} and s_{e2}^{p2}) is better than either for the given parameter values. In fact, some of the hybrid plans may actually be in $POSP$ although absent in *AniPOSP*.

Let the DAG built from the plans in *AniPOSP* be *DAG-AniPQO* and the set of plans in the DAG be *DAG-AniPOSP*. The Venn diagram of the sets $POSP$, *AniPOSP* and *DAG-AniPOSP* is shown in Figure 8.

The experiments conducted confirm the importance of generating hybrid plans. For example, for one of the queries we tested, $|POSP| = 134$, $|AniPOSP| = 49$ and $|DAG\text{-}AniPOSP \cap POSP| = 87$. This helps improve the quality of the solution.

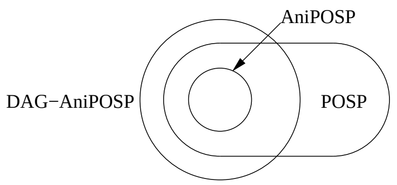

Figure 8: Venn diagram of the plan sets

Although *DAG-AniPOSP* may not contain some plans in $POSP$, it may contain some plans that are not in the $POSP$. Consider a point in the parameter space s.t. the optimal plan (in $POSP$) at the point is neither in *AniPOSP* nor in the other plans covered by *DAG-AniPOSP*. At such a point, the best plan in *DAG-AniPOSP* may not be in $POSP$ but may be better than any plan in *AniPOSP*. This could also help improve the quality of the solution.

Note that there is *no extra cost* for considering hybrid plans. The algorithm for finding the best plan in the DAG finds the cost of each node in the DAG only once. All the operators in the DAG are from the individual plans and we need to find their costs even if we decide to find the costs of the plans individually. It is also possible to use branch-and-bound pruning while searching for the best plan in the DAG, as described in [GM93].

6 Experimental Evaluation

We implemented our algorithm on top of a Volcano based query optimizer developed earlier at IIT-Bombay. The optimizer generates a bushy plan space and uses standard techniques for estimating costs, using statistics about relations. The cost estimates contain an I/O component and a CPU component and are nonlinear in general. We have extended the algorithm to return the cost of a given plan at a given point in the parameter space. (The exact cost functions need not be exposed to AniPQO.)

We tested our algorithm on five queries on a TPCD-based benchmark with and without indices on the primary keys of the relations involved. We use the TPCD database at scale factor 1; this corresponds to base data size of 1 GB. The `RiPj` queries compute the join of the first i of the relations `partsupp, supplier, nation` and `region`; they also have parametrized selections (attribute < parameter_value), with selectivities varying from 0 to 1, on the first j of the attributes `ps_partkey, s_suppkey, n_nationkey` and `r_regionkey`. We tested queries `R2P2, R3P2, R3P3` and `R4P4`. The `PlasticQuery` query (from [GPSH02]) computes the join of the above tables and the table `region`, and has parametrized selections (as mentioned above) on `p_size` and `ps_supplycost`.

	# Plans					DAG Size			AniPQO			
		AniPOSP		DAG-AniPOSP ∩ POSP			AniPOSP		Max. degradation (%)			
									w/o DAG		with DAG	
Query	POSP	t (%)		t (%)		POSP	t (%)		t (%)		t (%)	
		1	10	1	10		1	10	1	10	1	10
2R2P	10	10	9	10	9	21	21	20	0.00	0.61	0.00	0.61
3R2P	15	14	6	15	8	30	30	23	0.27	2.86	0.00	2.86
3R3P	36	24	15	31	17	39	37	30	10.62	4.14	2.40	3.52
4R4P	134	49	29	87	53	85	61	51	12.12	8.28	2.59	3.69
PlasticQuery(5R2P)	30	13	8	20	16	41	35	33	1.98	7.88	0.13	1.90

Figure 9: Quality of the results for queries on the TPCD catalog with no indices on the relations

	# Plans					DAG Size			AniPQO			
		AniPOSP		DAG-AniPOSP ∩ POSP			AniPOSP		Max. degradation (%)			
									w/o DAG		with DAG	
Query	POSP	t (%)		t (%)		POSP	t (%)		t (%)		t (%)	
		1	10	1	10		1	10	1	10	1	10
2R2P	6	5	5	5	5	16	14	14	3.50	3.50	3.50	3.50
3R2P	9	6	5	6	5	24	18	16	3.50	3.50	3.50	3.50
3R3P	11	8	6	8	6	27	22	18	3.50	3.50	3.50	3.50
4R4P	29	20	12	25	15	51	44	36	0.11	3.48	0.11	3.48
PlasticQuery(5R2P)	11	5	5	7	6	29	26	24	0.03	0.68	0.02	0.68

Figure 10: Quality of the results for queries on the TPCD catalog with indices on the relations

For each query we generated a very close approximation of the $POSP$ by optimizing the query at a large number of randomly selected points in the parameter space. We observed that the regions of optimality are concentrated along the parameter axis (i.e. with small parameter values) and close to the origin (as noted in [Rao97]); hence the coordinates of the points are generated with exponential distribution skewed towards lower values. We sampled enough points so as to be reasonably confident that all the plans in the $POSP$ are detected. (If the last new plan is found at sample number x, we sampled at least 10x points, except for query `R4P4`. For that query x was 551,963 with no indices and 1,603,186 with indices; so we sampled 2x points.) The method is expensive and not practical but we expect it to generate the $POSP$ with high probability. We assume its result to be the $POSP$ for the rest of the performance study.

To judge the quality of the results generated by AniPQO, we compared the plans in *AniPOSP* with those in the $POSP$ at a large number of randomly chosen points (Except for query `R4P4`, the number of samples is at least 5x, where x is as mentioned above.) At each point, we found the cost of the optimal plan and that of the best plan from *AniPOSP* and *DAG-AniPOSP*, and calculated the percentage difference. We find out the maximum degradation at any point in the set of sampled points.

We experimented with two values of the optimality threshold t (defined in Section 3) 1% and 10%.

The quality of the results is tabulated in Figure 9 for the case when the queries are optimized with no indices on the relations involved. The numbers in the columns with top heading "# Plans" show that the DAG optimization significantly increases the coverage of the plans in $POSP$, while increasing the optimality threshold from 1% to 10% decreases the coverage slightly.

The numbers in the columns with top heading "DAG Size" indicate that the size of the DAG is quite small, implying that the cost of plan selection at runtime would be correspondingly small.

The last set of columns with top heading "AniPQO Max. degradation" indicate the maximum degradation in the quality of the output plan compared to the optimal plan. The numbers indicate that the quality of the output plans is good with plain *AniPOSP*, and improves further with the DAG optimization. The quality of plans is in general better with a smaller optimality threshold (t value); but surprisingly, for queries `R3P3` and `R4P4` without the DAG optimization, the quality of results is better for t=10% than for $t = 1\%$, although the number of plans found is more for the optimality threshold of 1%. This may be attributed to the fact that there are enough "good" plans and, with the optimality threshold of 10%, the vertices that were optimized happened to be such that the plans optimal at the vertices do well globally.

Figure 10 shows results for the case with indices on the primary keys of the relations involved. The AniPQO algorithm continues to perform very well in this case. However, in this case, except for query `R4P4`, neither the DAG option nor the optimality threshold has a significant effect on the quality of the result.

The table in Figure 11 reports the number of calls made to the conventional optimizer, and the number of plan-cost evaluation probes; a plan-cost evaluation probe involves finding the cost of a given plan at a given point in the parameter space.

The columns labeled "AniPQO" list the number of optimizer calls made by AniPQO. The columns labeled "Linear case (approx.)" list the approximate

Query	Without Indices						With Indices					
	# Optimizer calls				Plan-cost evaluation calls		# Optimizer calls				Plan-cost evaluation calls	
	Linear case (approx.)		AniPQO				Linear case (approx.)		AniPQO			
	t (%)		t (%)		t (%)		t (%)		t (%)		t (%)	
	1	10	1	10	1	10	1	10	1	10	1	10
2R2P	21	11	22	10	764	140	12	7	13	9	209	141
3R2P	23	6	22	6	688	49	12	7	14	9	215	141
3R3P	41	20	42	18	1339	532	31	13	31	13	1425	288
4R4P	101	46	95	41	4587	1708	61	33	62	31	4009	1227
PlasticQuery(5R2P)	20	10	18	8	897	140	7	7	6	6	34	18

Figure 11: Optimization overhead for queries on the TPCD catalog

number of optimizer calls required if we were to get the same parameter space decomposition with linear cost functions. This is derived from the number of decomposition vertices and the number of plans found[6]. The comparison indicates that the number of optimizer calls made by AniPQO is comparable with that made in the linear case. The table in Figure 11 also lists the number of plan-cost evaluation calls made by AniPQO. The cost of a plan-cost evaluation call is very small compared to the cost of a call to the optimizer.

In addition to the above calls, AniPQO has to maintain the vertices and edges of the decomposition of the parameter space. This cost can be exponential in the number of parameters, but with a small number of parameters (say up to 4) this is not a major cost.

We have implemented two versions of AniPQO: one with a loose integration of AniPQO with the conventional optimizer, where AniPQO makes separate invocations for each parameter value, and the other with a tight integration, where the optimizer equivalence rules are applied only once, and the resultant DAG of equivalent plans is used repeatedly, to find the optimal plan with different parameter values.

For a representative query (`R4P4` with no indices), a single invocation of the underlying optimizer takes about 16 ms. With the optimality threshold $t = 1\%$, the loosely integrated AniPQO takes about 1900 ms (with 95 calls to the optimizer) and the tightly integrated AniPQO takes about 850 ms, a saving of a factor of over two. With the optimality threshold $t = 10\%$, the loosely integrated AniPQO takes about 910 ms (with 41 calls to the optimizer) and the tightly integrated AniPQO takes only about 350ms.

Comparison of the results for the two optimality thresholds (t) shows that AniPQO degrades gracefully; changing t from 1% to 10% decreases the cost of optimization significantly, while only marginally reducing the quality of the plans.

Scaling with the number of parameters: Our experiments indicate that the number of calls to the conventional optimizer appears to grow exponentially with the number of parameters, but remains practical for up to 4 parameters. The exponential growth is not unexpected, since even for the special case of linear cost functions, the worst case number of calls to the conventional optimizer has an exponential lower bound if we seek the exact solution [Gan01, HS02].

7 Related work

[GW89] makes a case for parametric query optimization, and proposes *dynamic query plans* that include a *choose-plan* operator, which chooses a plan, at run-time, from among multiple available plans depending upon the values of certain run-time parameters.

[CG94] presents a technique wherein the cost of a plan p is modeled as an interval $[l, u]$, where l and u are the highest and the lowest cost of the plan p over the parameter space, and plans whose lower bound is greater than the upper bound of some plan are pruned out. The technique computes a superset of the parametric optimal set and [Gan98] shows that the expected number of plans generated by this algorithm could be much larger than the expected size of the parametric optimal set; and [Rao97] confirms this empirically.

[INSS92] presents a randomized approach – based on iterative improvement and simulated annealing techniques – for parametric query optimization with memory as a parameter. The technique proposed assumes the parameter space to be discrete and runs the randomized query optimizer for each point in the parameter space. The technique is unsuitable for continuous parameters, like selectivity.

[GK94] provides a solution for the parametric query optimization with linear cost functions in one parameter. [Gan98] extends the work of [GK94] and proposes a solution for parametric query optimization with linear cost functions in two parameters.

In [HS02] we developed an efficient algorithm for solving parametric query optimization problem with arbitrary number of parameters and linear cost functions. In [HS02] we also presented a solution to parametric query optimization problem when the cost functions are piecewise linear. The solution is intrusive and the conventional optimizer needs to be extended. The memory cognizant optimization algorithm in [HSS00]

[6]The number is $v + f'$ where v is the number of final decomposition vertices and f' is the number of regions of the parameter space which are adjacent to none of vertices of the parameter space polytope. This is a lower bound for the linear case [HS02]. Since the decomposition may be incomplete in our case, this number is approximate.

can be viewed as a special case of the algorithm for piecewise linear cost functions in [HS02], although it does an extra task of optimal memory allocation to operators.

In a currently unpublished (independently done) work [Gan01], Ganguly has also extended the algorithm from [Gan98] so as to work for more than two parameters. The algorithm for finding the $POSP$ from Section 3 is an abstraction of the algorithms from [Gan01, HS02].

[GK94, Gan98, Gan01] extend the algorithm proposed for linear cost functions so as to support a special case of nonlinear cost functions – namely *affine extensible* cost functions. An affine extensible cost function, in its general form, can be defined as $\sum_{S \subseteq Q} a_S \prod_{i \in S} p_i$, where $Q = \{1, 2, 3, \ldots, n\}$, a_S's are constants, and p_i's are parameters variables. The solution proposed embeds the affine extensible cost functions into linear cost functions with a larger number of parameters and then uses the techniques developed for linear cost functions.

[Rao97] studies the distribution of the parametric optimal plans in the parameter space for the 2-dimensional case and devises several sampling techniques. [Bet97] reports experimental results of the technique proposed in [Gan98] with linear cost functions in one parameter for linear and star queries. [Pra97] reports an experimental evaluation of the algorithm for affine extensible cost functions proposed in [Gan98].

[CHS99] proposes least expected cost query optimization which takes distribution of the parameter values as its input and generates a plan that is expected to perform well when each parameter takes a value from its distribution at run-time.

The *Plastic* system, proposed in [GPSH02], amortizes the cost of query optimization by reusing the plans generated by the optimizer. It groups similar queries into clusters and uses the optimizer generated plan for the cluster representative to execute all future queries assigned to the cluster. Query similarity is evaluated by a classifier which is based on query structure, the associated table schema and statistics.

8 Conclusion

In this paper we proposed a heuristic solution for parametric query optimization when the cost functions are nonlinear. The algorithm works for arbitrary nonlinear cost functions, and is minimally intrusive. Initial experiments indicate that AniPQO finds a plan set that is a good approximation of the $POSP$ and the quality of the plans generated is close to that of the $POSP$. The algorithm conceptually works for an arbitrary number of parameters and although optimization time increases sharply with the number of parameters, our implementation indicates that AniPQO is quite practical for up to 4 parameters.

Acknowledgments: Arvind Hulgeri's work was supported by an Infosys Fellowship. We wish to thank Bharat Adsul, Prasan Roy and Milind Sohoni for their feedback. We are grateful to Prasan Roy for providing the code for the Volcano-based conventional optimizer.

References

[Bet97] A. V. Betawadkar. Query optimization with one parameter. Technical report, IIT, Kanpur, Feb 1997. http://www.cse.iitk.ac.in/research/mtech1997.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD*, 1994.

[CHS99] F. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *SIGMOD*, 1999.

[Gan98] Sumit Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, 1998.

[Gan01] Sumit Ganguly. A framework for parametric query optimization (unpublished manuscript; personal communication). 2001.

[GK94] Sumit Ganguly and Ravi Krishnamurthy. Parametric query optimization for distributed databases basd on load conditions. In *COMAD*, 1994.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[GPSH02] A. Ghosh, J. Parikh, V. Sengar, and J. Haritsa. Plan selection based on query clustering. In *VLDB*, 2002.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *SIGMOD*, 1989.

[HS02] Arvind Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB*, 2002.

[HS03] Arvind Hulgeri and S. Sudarshan. *AniPQO*: Almost non-intrusive parametric query optimization for nonlinear cost functions. Technical report, IIT, Bombay, June 2003. Available at http://www.cse.iitb.ac.in/aru.

[HSS00] Arvind Hulgeri, S. Seshadri, and S. Sudarshan. Memory cognizant query optimization. In *COMAD*, 2000.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *VLDB*, 1992.

[Mul94] Ketan Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, 1994.

[Pra97] V. G. V. Prasad. Parametric query optimization: A geometric approach. Technical report, IIT, Kanpur, Feb 1997. http://www.cse.iitk.ac.in/research/mtech1997.

[Rao97] S. V. U. M. Rao. Parametric query optimization: A non-geometric approach. Technical report, IIT, Kanpur, Feb 1997. http://www.cse.iitk.ac.in/research/mtech1997.