Parametric Query Optimization for Linear and Piecewise Linear 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 first propose a solution for the PQO problem for the case when the cost functions are linear in the given parameters. This solution is minimally intrusive in the sense that an existing query optimizer can be used with minor modifications: the solution invokes the conventional query optimizer multiple times, with different parameter values.

We then propose a solution for the PQO problem for the case when the cost functions are piecewise-linear in the given parameters. The solution is based on modification of an existing query optimizer. This solution is quite general, since arbitrary cost functions can be approximated to piecewise linear form. Both the solutions work for an arbitrary number of parameters.

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 unbound variables, the values of the variables are known only at run time. In general, the available memory is not known until runtime. Optimizing a query into a single plan may result in a substantially sub-optimal plan if the actual 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, INSS97, INSS92, GK94, Gan98]. At run time, when the actual parameter values are known, the appropriate plan can be chosen.

The contributions of this paper lie in providing two novel solutions for the parametric query optimization problem:

- We provide a novel parametric query optimization algorithm for the case when the plan cost functions are linear in the parameters. The algorithm works for an arbitrary number of parameters and is minimally-intrusive in the sense that it does not modify the conventional query optimizer, and merely uses it as a subroutine (invoking it with different parameter values). To the best of our knowledge, no exact solution published so far works for an arbitrary number of parameters; however, there is a related work [Gan01], currently unpublished, that handles an arbitrary number of parameters; we describe the connections in Section 6. Our solution is simple and efficient, unlike earlier solutions to the PQO problem.

- In general, the cost function of an operation may be non-linear and discontinuous in the parameters involved. The cost function of a plan, which is the sum total of the cost functions of the operations involved, will then also be non-linear and discontinuous. It is, in general, difficult and costly to deal with such nonlinear functions and this is particularly true when the functions involve many parameters. However, nonlinear cost functions can

* Work supported by an Infosys Fellowship

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

be approximated by piecewise linear cost functions.

We propose an approach for parametric query optimization with piecewise linear cost functions, based on extending existing optimization algorithms to use cost functions in place of costs. We show how to extend the System-R query optimization algorithm [SAC+79] to perform parametric query optimization with piecewise linear cost functions. We have also extended the Volcano query optimization algorithm [GM93] in a similar fashion. The solution works for an arbitrary number of parameters.

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 non-intrusive algorithms for PQO with linear cost functions. Section 4 presents definitions related to piecewise linear cost functions. Section 5 describes (intrusive) algorithms for PQO with piecewise linear cost functions. Related work is described in Section 6. We conclude the paper in Section 7.

2 Definitions

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 $POSP$ and the region of optimality for each plan in the $POSP$.

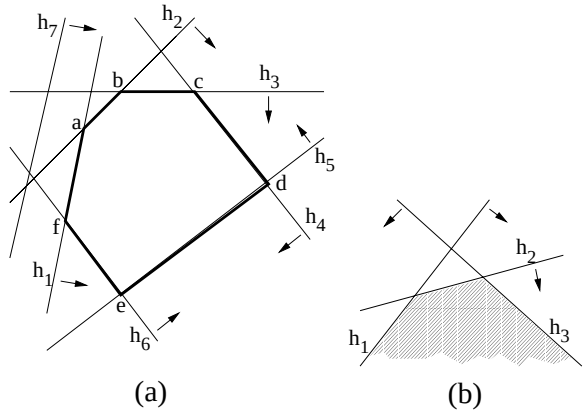

Figure 1: (a) a polytope and (b) a lower convex polytope in 2-dimensions

2.2 Polytopes

In the proposed solutions, we need to represent and manipulate parameter space partitions. For parametric query optimization with linear cost functions, the regions of optimality are convex; if the parameter space of interest is a convex polytope, the regions of optimality are also convex polytopes. In this section we define what are polytopes and describe a special type of polytope, lower convex polytope.

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 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.

Let the set of halfspaces defining P be M. *Lower convex polytopes* are a special class of convex polytopes where all halfspaces in M extend to infinity in the negative x_d direction. Then each element in M can be viewed as a hyperplane that implicitly stands for the halfspace bounded by it and extending in negative x_d direction. We say that P is the lower convex polytope formed by such hyperplanes in M. Figure 1(b) shows a lower convex polygon.

3 Parametric Query Optimization for Linear Cost Functions

In this section we propose minimally-intrusive solutions for linear cost functions. First we review some basic properties of the linear cost functions; we give a brief outline of a naive *recursive decomposition algorithm* and then we present our main algorithm, the *cost polytope algorithm*.

Conventional query optimizers return an optimal plan along with its cost. For parametric query optimization, the cost of a plan is a function of the parameters, and the cost function of a plan is required to compare it with other plans. We can extend the statistics/cost-estimation component of the optimizer to make it return the cost function of a given plan; one way to do so is to do conventional cost estimation on the given plan at $n+1$ (non-degenerate[1]) points in the parameter space, where n is the number of parameters, and thereby infer its cost function. The optimizer itself is not modified in any way, and continues to use the original statistics/cost-estimation code.

In general we are not interested in the whole parameter space $\Re^n$ 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*, and it is provided to the optimizer. Typically, the parameter space polytope is a hyper-rectangle defined by a range of legal values specified for each parameter.

3.1 Properties of Linear Cost Functions

We state the following properties regarding linear cost functions from [Gan98]:

- If two points in the parameter space have the same optimal plan then the plan is optimal along the line segment connecting the two points.
- Each plan in a $POSP$ has only one region of optimality and the region is a convex polytope.
- If all the vertices of a polytope in the parameter space have the same optimal plan then the plan is optimal within that polytope.

[1]The points are not contained in a common hyperplane.

Thus the partitioning of the parameter space is convex and the solution will divide the parameter space into convex polytopes.

Note that for linear cost functions, the decomposition of the parameter space induced by any $POSP$ is the same and the $POSP$ is unique if no two plans have the same cost function. Details may be found in the full version of the paper, [HS02]. Without loss of generality, we assume that the $POSP$ is unique.

3.2 The Recursive Decomposition Algorithm

This solution is based on the observation that if all the vertices of a polytope in the parameter space have the same optimal plan, then the plan is optimal within that polytope. We recursively decompose the parameter space into convex polytopes.

We find the optimal plans at the vertices of each polytope, starting with the parameter space polytope, using a conventional query optimizer. If two of the vertices of a polytope have two different optimal plans (or more precisely, optimal plans with different cost functions), then we partition the polytope into two polytopes: the dividing hyperplane is derived by equating the cost functions of the plans. As a result, one plan is better in one of the polytopes, and the other plan is better in the other. We then recursively apply the above test to each of the two polytopes. A polytope region is not decomposed further when all its vertices have the same optimal plan.[2] The detailed algorithm may be found in [HS02].

This solution has two shortcomings: It may form more that one region for a plan and may need to merge them in a post-pass; And the number of calls made to the conventional optimizer may be more than necessary.

In fact, we can combine the decompose and merge phases by noticing that the optimality region for an optimal plan at a point may surround the point[3]. So instead of partitioning each polytope adjacent to the point independently, we can partition all of them simultaneously by *carving* out a single polytope around the point and subtracting it from each adjacent partition. Our next algorithm is an outcome of this observation.

3.3 The Cost Polytope Algorithm

The cost polytope algorithm works in the $\Re^{n+1}$ space with n dimensions representing n parameters and one dimension representing cost. The cost function of each

[2]We can devise an approximate version of the algorithm, which does not partition the polytope if the cost of the optimal plan at one vertex is within a small percentage of the cost of the optimal plan at each of the remaining vertices.

[3]This may not be the case, though, if more than one plan is optimal at the point; in that case, the point lies on the boundary of the optimality regions of the plans.

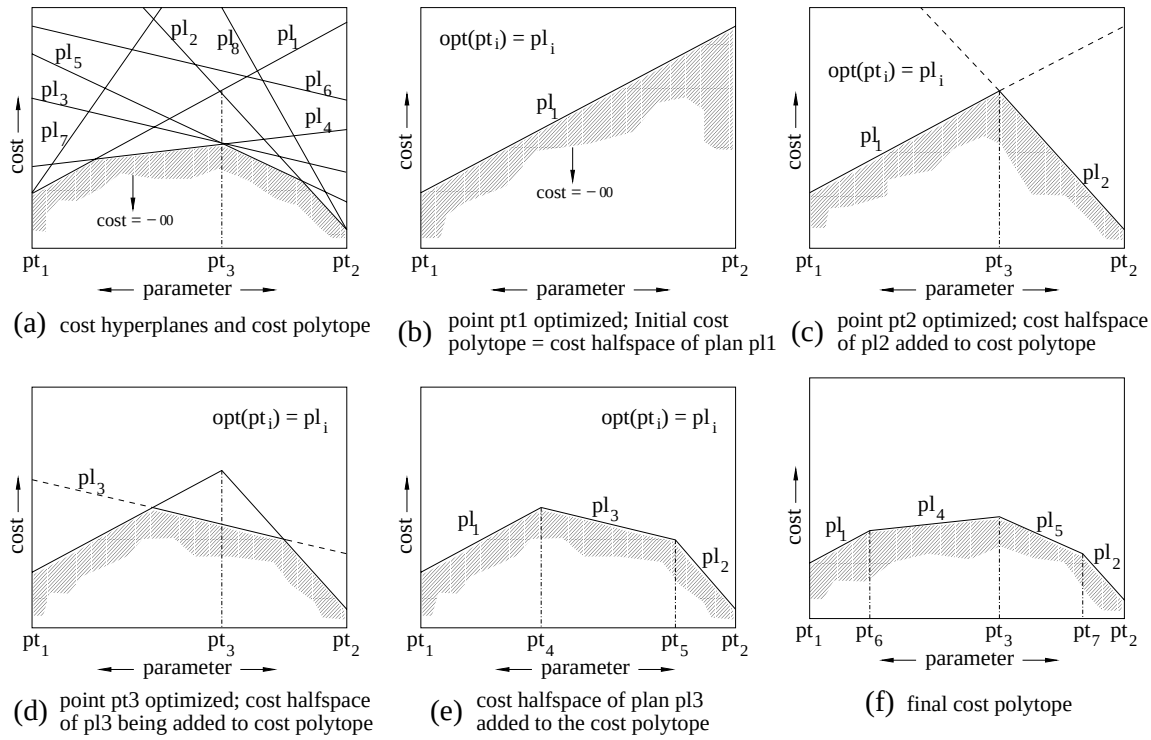

Figure 2: Cost Polytope Algorithm: One parameter example

plan in the plan space can be represented by a hyperplane in $\Re^{n+1}$. We work on these hyperplanes to construct a lower convex polytope that represents the optimal cost among all plans at each point in the parameter space. Each facet of this polytope corresponds to a plan in the parametric optimal set of plans ($POSP$) and one can obtain its optimality region by projecting the facet on the parameter space ($\Re^n$).

We use a running example with one parameter, shown in Figure 2, throughout the section.

3.3.1 Parametric Optimal Cost Function

The *parametric optimal cost function (POCF)* over the parameter space is defined as follows. For a point v in the parameter space:

$POCF(v)$ = cost of a plan p that is optimal at v.

It follows that, for any plan p in the $POSP$, at any point v in its region of optimality, the value of $POCF(v) = C_p(v)$, the cost of p at v. Thus within the region of optimality of a plan, the $POCF$ follows the cost function of the plan.

Consider an example with one parameter shown in Figure 2(a). The horizontal axis represents the parameter space and the vertical axis represents the cost. The line segment $pt_1 - pt_2$ is the parameter space polytope. Let the plan space contain eight plans $pl_1, pl_2, \ldots, pl_8$ with cost functions as shown in Figure 2(a). We have $POSP = \{pl_1, pl_2, pl_4, pl_5\}$. Figure 2(f) shows the $POCF$, and the region of optimality of each plan in the $POSP$.

3.3.2 Cost Hyperplane, Cost Halfspace and Cost Polytope

Consider $\Re^{n+1}$ space with n dimensions representing n parameters and the $n+1^{th}$ dimension representing the cost. Let the cost of a plan p be $c_1 s_1 + c_2 s_2 + \cdots + c_n s_n + c_{n+1}$. We can think of the cost function as a hyperplane in $\Re^{n+1}$ whose equation is given by

$$s_{n+1} = c_1 s_1 + c_2 s_2 + \cdots + c_n s_n + c_{n+1}$$

where s_{n+1} denotes the cost of the plan. We call such a hyperplane a *cost hyperplane*. We assume that no plan has a degenerate cost hyperplane with infinite slope[4]. Figure 2(a) shows cost hyperplanes for the plans $pl_1, pl_2, \ldots, pl_8$ in the parameter range $pt_1 - pt_2$.

We define the lower halfspace (extending to $cost = s_{n+1} = -\infty$) of the cost hyperplane as:

$$s_{n+1} \leq c_1 s_1 + c_2 s_2 + \cdots + c_n s_n + c_{n+1}$$

We call such a halfspace a *cost halfspace*.

We represent each plan p in the plan space by its cost hyperplane in $\Re^{n+1}$ space. The *Cost Polytope* is defined as the lower convex polytope obtained by intersection of the cost halfspaces of all the plans in the plan space.

Figure 2(f) shows the cost polytope for the cost hyperplanes defined in Figure 2(a). We can see that its boundary is the $POCF$ for the plans $pl_1, pl_2, \ldots, pl_8$ in the parameter range $pt_1 - pt_2$.

[4]Such a cost function would be completely unrealistic since it would divide the parameter space into two halves with each point in one half having cost positive infinity and each point in the other half having cost negative infinity.

Theorem 3.1 *The boundary of the cost polytope defines the POCF.* □

For the proof see [HS02]. Note that the cost hyperplanes corresponding to the plans in the $POSP$ are the bounding hyperplanes of the cost polytope and the rest of the hyperplanes are redundant. Thus, the cost hyperplanes corresponding to the plans not in the $POSP$ cannot form any facet of the cost polytope.[5] Whereas, the cost hyperplane corresponding to each plan in the $POSP$ forms one facet of the cost polytope, and the projection of the facet on the parameter space (the hyperplane $s_{n+1} = 0$; i.e. cost $= 0$) gives the region of optimality for the plan.

3.3.3 Cost Polytope Construction

We discuss the cost polytope construction algorithm in this section. A naive algorithm would be to intersect all the halfspaces that are in the input set. In the case of cost polytopes, enumerating all the halfspaces amounts to enumerating all the plans in the plan space and getting their cost functions. The plan space can be very large; and only a handful of plans constitute the $POSP$ [Gan00]. Such a naive algorithm would be prohibitively expensive. But we have an additional tool: Given a point v in the parameter space, we can use the conventional optimizer to obtain a cost hyperplane that bounds or touches the cost polytope at the point whose projection is v. We use this property to avoid enumerating all the cost hyperplanes.

Our algorithm uses an online polytope construction algorithm such as that in [Mul94], as a subroutine. A polytope construction algorithm is given a set of halfspaces and the algorithm intersects the halfspaces to construct the desired polytope. In the case of an online algorithm, the halfspaces are given one at a time, and at each stage the algorithm maintains an intermediate polytope.

Figure 3 shows pseudo code for the Cost Polytope algorithm. We first optimize any one vertex, say v, of the parameter space polytope to get a optimal plan at it, from which we derive the corresponding cost halfspace in $\Re^{n+1}$. We transfer the equations of all bounding hyperplanes of the parameter space polytope to $\Re^{n+1}$ space and intersect them with the cost halfspace obtained above, to get the initial cost polytope.

We then put all vertices, except v, of the initial cost polytope in a queue. We pick one vertex at a time from this queue and optimize it, i.e. invoke the conventional optimizer on the parameter coordinate values of the vertex. Consider an intermediate cost polytope and one of its vertices, say v. We optimize vertex v to get an optimal plan p (at vertex v) and its cost hyperplane. Note that, as plan p is optimal at vertex v, the cost hyperplane of plan p must either touch or intersect the cost polytope at vertex v. In the later case, we intersect the cost hyperplane with the current cost polytope to get a new cost polytope. The intersection operation may delete some of the vertices from the polytope and may add some new vertices. If a vertex which is deleted from the polytope is present in the queue, the vertex is removed from the queue. All the new vertices of the polytope are added to the queue.

When the queue become empty, we terminate the algorithm. When this condition is reached, the cost hyperplane of an optimal plan of each vertex is either a facet of the cost polytope or is touching the cost polytope. The plans corresponding to the facets of the cost polytope form the $POSP$. The cost hyperplanes for all other plans are redundant.

An online algorithm for polytope construction is

[5]The cost hyperplane of a plan not in the $POSP$ may touch the polytope at a vertex or along an i-face, for $0 < i < n$ (e.g., an edge, for $n > 1$), but cannot form a facet (i.e., a n-face) in $\Re^{n+1}$.

PSpacePTope = parameter space polytope
    /* See Section 3.1; the polytope is in $\Re^n$ */
PSPTHalfspaces =
    Halfspaces defining *PSpacePTope* in $\Re^n$
Let $v = (v_1, v_2, \ldots v_n)$ be any vertex of *PSpacePTope*
p = *ConventionalOptimizer*(v)
    /* p is one of the optimal plans at v */
VerticesOptimized = $\{v\}$
v_{n+1} = cost of p at v
Let $v' = (v_1, v_2, \ldots v_n, v_{n+1})$
Let hs_p be the cost halfspace of plan p in $\Re^{n+1}$
CostPolytope = intersection of all *PSPTHalfspaces*
    and hs_p in $\Re^{n+1}$ /* Initial cost polytope */
Queue = {vertices of *CostPolytope* } $\setminus$ v'
While *Queue* $\neq \emptyset$ do
    v' = *Queue.RemoveFirstEntry*()
    Let the coordinates of v' be $(v_1, v_2, \ldots v_n, v_{n+1})$
    Let $v = (v_1, v_2, \ldots v_n)$
        /* projection of v' on parameter space */
    p = *ConventionalOptimizer*(v)
        /* p is one of the optimal plans at v */
    VerticesOptimized = *VerticesOptimized* $\cup$ $\{v\}$
    Let hs_p be the cost halfspace of plan p
    If (cost of p at v) $< v_{n+1}$
            /* v' is in conflict with hs_p */
        CostPolytope = *CostPolytope* $\cap$ hs_p
        Remove from *Queue* vertices no longer
                in *CostPolytope*
        For each new vertex $w' = (w_1, \ldots w_n, w_{n+1})$
                added to *CostPolytope*
            if $w = (w_1, \ldots w_n) \notin$ *VerticesOptimized*
                add w' to *Queue*

Figure 3: The Cost Polytope Algorithm

presented in [Mul94]. The algorithm allows a sequence of half spaces to be intersected, resulting in a (possibly) new polytope after each intersection. The algorithm in [Mul94] uses *conflict* and *history* structures to identify a *conflicting vertex*[6] for a hyperplane to be added. In our algorithm, each hyperplane added is guaranteed to conflict with at least one vertex: the vertex whose optimization resulted in the hyperplane. Thus, we can optimize the online polytope construction algorithm for our application by eliminating the conflict and history structures; see [HS02] for details.

Theorem 3.2 *The cost polytope algorithm correctly computes the cost polytope.*

Proof: Theorem 3.5 (presented later) shows that the algorithm terminates. When the algorithm terminates, it returns a polytope with the property that none of its vertices is in conflict with any cost hyperplane. Thus, the polytope is exactly the intersection of all the cost hyperplanes [Mul94]. □

The proof can be understood in another way in terms of $POSP$ and regions of optimality. There is a one-to-one correspondence between the vertices of the cost polytope and the vertices that define the partitioning of the parameter space polytope; there is also a one-to-one correspondence between the facets of the cost polytope and the plans in the $POSP$. Since an optimal plan is found for each vertex and its halfspace is intersected with the polytope, the plan corresponding to a facet is optimal at each of its vertices. Since the cost function is linear, the plan is optimal at all parameter space points in the projection of the facet.

Avoiding re-optimizing points: In Figure 2, point pt_3 in the parameter space appears twice: in Figure 2(d) it is optimized and vanishes from the decomposition (i.e., is no longer a vertex of the decomposition), and in Figure 2(f) it is one of the vertices defining the final decomposition. Thus it has reappeared after vanishing. The algorithm needs to remember which points are optimized to avoid making redundant calls to the conventional optimizer; the set *VerticesOptimized* is used for this purpose.

3.3.4 Complexity in terms of number of calls to the conventional optimizer

We measure the complexity in terms of number of calls to the optimizer as in [Gan98]. In the following theorems, f denotes the number of plans in the $POSP$, which is also the number of facets of the cost polytope; F denotes the total number of i-faces, across all i, of the final cost polytope; and v denotes the number of vertices that define the decomposition of the parameter space, including the vertices of the parameter space polytope.

[6]Consider intersecting a halfspace S with a polytope P. A vertex v of the polytope is said to be conflicting with the halfspace if it lies in the complement of the halfspace; *i.e.* $v \in \overline{S}$.

Lemma 3.3 *A total of v calls to the optimizer are necessary and sufficient to check if a given set of plans, with a parameter space decomposition defining a region of optimality for each plan, is the $POSP$.*

Proof Sketch: If a plan is optimal at all the vertices of a polytope in the parameter space decomposition then the plan is optimal within that polytope and hence the parameter space decomposition is optimal. The number of calls to the optimizer sufficient to ascertain this is v – one per vertex. But if we do not optimize a vertex, we do not know if the above condition is satisfied by the regions surrounding the vertex. Thus, the necessary number of calls to the optimizer is v. □

For the detailed proof see [HS02]. The theorem below follows directly from Lemma 3.3.

Theorem 3.4 *The lower bound on the number of calls to be made to the optimizer to solve the parametric query optimization problem for linear cost functions in a non-intrusive manner is v.* □

Theorem 3.5 *The cost polytope algorithm makes at most F calls to the optimizer.*

Proof: Any intermediate cost polytope is a superset of the final cost polytope and, as the algorithm progresses, the polytope shrinks as it is intersected with new hyperplanes. Hence, any intermediate face is a superset of a face of the final cost polytope and a face of the polytope can only shrink as the algorithm progresses[7]. Thus, no vertex of an intermediate cost polytope is in the interior of any face of the final cost polytope. Each point that is optimized is a vertex of an intermediate or the final cost polytope.

Consider a vertex of an intermediate cost polytope v' that is optimized. Let point v be the point on the final cost polytope whose projection on the parameter space is the same as that of v'. Now, v has to be either a vertex of the final polytope, or has to be in the interior of exactly one of the i-faces of the final polytope for some $i(> 0)$. In the latter case, let this face be h and we say that vertex v' maps to face h. We claim that no two optimized points map to the same face of the final cost polytope. If not, since points are optimized in some order, let w' be an intermediate polytope vertex that is optimized after v', and w be the point in the interior of face h whose projection on the parameter space is the same as that of w'. But since we have already intersected with the hyperplane corresponding to face h, $w = w'$, But then w' was a vertex of an intermediate polytope, which means it cannot be in the interior of face h. Thus each point optimized either maps to a different i-face or is a vertex

[7]A face of the polytope may reduce in dimension but it never disappears; it may become a 0-face i.e. it shrinks to a vertex of the polytope.

of the final polytope, and no point is optimized more than once, leading to the upper bound of F calls. □

In the worst case, the total number of faces of a polytope is $O(n^{\lfloor d/2 \rfloor})$, and the number of vertices is $O({}^nC_{\lfloor d/2 \rfloor})$, where n is the number of halfspaces defining the polytope and d is the number of dimensions. Thus, the optimizer can make significantly more calls than the lower bound. The worst case occurs only when for each i-face of the final cost polytope we have a cost hyperplane such that its intersection with the cost polytope is the i-face itself; see [HS02] for details. If the coefficient vectors[8] of the cost functions are distributed uniformly in a unit sphere then the probability that a cost hyperplane not defining the final cost polytope touches it is zero; see[HS02] for details. Thus, the expected value of F is f. In high dimensional non-degenerate cases, the number of vertices of a polytope are generally exponential in the number of facets [Zie94], and we have $f \ll v$. Thus, the number of calls made to the conventional optimizer is expected to be close to the lower bound under the above assumption.

3.3.5 Optimization given all optimal plans at a given point

We assumed that the conventional optimizer returns one of the optimal plans at a given point in the parameter space, along with its cost hyperplane. In contrast, [Gan98] and [Gan01] make the stronger assumption that the optimizer returns all optimal plans at a given point, which is harder to implement, since the optimizer code would have to be modified, and a large number of plans may be returned in degenerate cases.

Given an optimizer that returns all the plans, we can pick any one of the plans returned and algorithm remains the same. However, we can then save on the number of calls to the optimizer by intersecting the intermediate cost polytope with all the returned hyperplanes. For example, in the example in Figure 2 optimization of point pt_3 returns three plans and hence three cost functions. If we intersect all of them with the intermediate cost polygon we need not call the optimizer at points pt_4 and pt_5.

Let V' be the set of vertices of the parameter space polytope. If the optimizer returns all the optimal plans at a given point, the number of calls made to the optimizer would be at most $(v + f')$ where $f'(< f)$ is the number of plans (in the $POSP$) whose regions of optimality are adjacent to none of the vertices in set V'. Since $f < v$, the number of calls is at most two times v, and is thus very close to the lower bound. This upper bound is valid only if we optimize all the vertices in set V' before optimizing any other point, as is done in the Cost Polytope algorithm. (An example illustrating this point can be found in [HS02].) The reduction in the upper bound on the number of calls is based on two observations: (a) when we optimize the vertices of the parameter space polytope, we detect all the facets of the cost polytope that are touching the boundary vertices and, thus, no separate calls are needed to detect such facets. (b) we detect all the faces and facets of the final cost polytope that touch the vertex of the final cost polytope corresponding to the point optimized and, thus, a call at a vertex that is not a vertex of the final cost polytope detects at least one facet of the final cost polytope. For details, see [HS02].

[8]The coefficient vector of a cost function $c_1 s_1 + c_2 s_2 + \cdots + c_n s_n + c_{n+1}$ is $(c_1, c_2, + \cdots c_{n+1})$.

4 Piecewise Linear Cost Functions (*PLCF*)

Real world cost functions are often not linear. The properties of the linear cost functions enumerated in Section 3.1 do not hold for non-linear cost functions, and hence the solution proposed for linear cost functions are not applicable. We now describe how to handle piecewise linear cost functions (*PLCF*s): a function is piecewise linear if the parameter space can be partitioned into convex polytopes such that within each partition, the function is linear in the parameters. Piecewise linear cost functions can be used to approximate any non-linear cost function, and thus a solution for the piecewise linear case is very general and of practical importance.

In the rest of this section we give some definitions related to parameter space partitioning and piecewise linear cost functions. In Section 5 we outline how to perform parametric query optimization with piecewise linear cost functions.

Partitioning Scheme: A *partitioning scheme* partitions the parameter space into disjoint partitions. We assume that for each operation, the cost function is piecewise linear, that is, the parameter space can be divided into partitions such that the cost function is linear in each partition. We also assume that the parameter space polytope and the partitions are closed convex polytopes.

Figures 4(a) and 4(b) show piecewise linear cost functions with one parameter and partitioning of the parameter space induced by them. In one dimensional space polytopes are just line segments. The cost function in 4(a) creates 4 partitions of the parameter space, namely r_{a1}, r_{a2}, r_{a3} and r_{a4} whereas the cost function in 4(b) creates 3 partitions of the parameter space, namely r_{b1}, r_{b2} and r_{b3}. Both cost functions are linear within each of their partitions. (Parts (c) and (d) of Figure 4 are discussed later.)

Figures 5(a) and 5(b) show partitioning schemes defined on a two dimensional parameter space. Both parameters range over $[0, 1]$. The partitioning scheme in 5(a) creates 3 partitions of the parameter space, namely r_{a1}, r_{a2} and r_{a3}. The partitioning scheme in 5(b) creates 3 partitions of the parameter space,

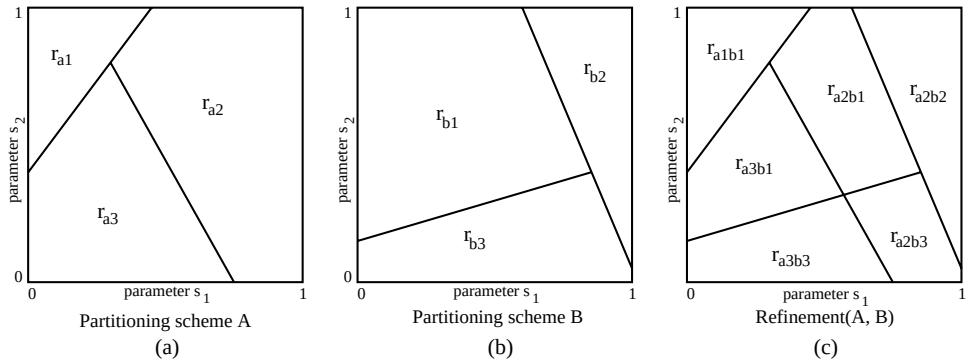

Figure 5: Refinement of parameter space partitioning: An example with two parameters

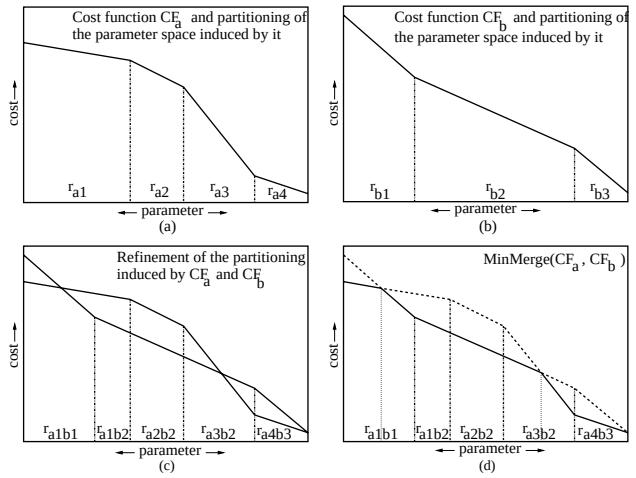

Figure 4: Parameter Space Partitioning, Refinement and $MinMerge$ operation: a one-parameter example

namely r_{b1}, r_{b2} and r_{b3}.

5 Optimization for Piecewise Linear Cost Functions

In this section we present an overview of the extensions to a conventional optimization algorithm to turn it into a parametric query optimizer for piecewise linear cost functions.

In a conventional optimizer we have a single value as the cost for an operation or a plan and a single optimal plan for a query/sub-query expression. But in parametric query optimization, we need to handle cost functions in place of costs, and keep track of multiple plans, along with their regions of optimality, for each query/subexpression. To handle these differences, a conventional query optimizer can be extended in the following ways:

- The cost of an operation: The cost of an operation is now a piecewise linear function of the parameters. The parameter space can be divided into partitions and within each partition the cost of the operation is a linear function of the parameters. See Section 5.3 for details.
- Evaluating the cost of a plan P: Given the cost function of the root operation O, say C_o, and that of the sub-plan $P' = P \setminus O$, say $C_{p'}$, consider a point in the parameter space. The cost of the plan at this point will be an addition of the cost of P' and that of O at this point. The cost depends upon the polytopes in the cost functions C_o and $C_{p'}$ in which the point fall.

  To get the cost function C_p of the plan P, we need to refine the two partitioning schemes of the parameter space (one from the cost function C_o and the other from the cost function $C_{p'}$) such that both functions are linear in each of the partitions of the refined partitioning scheme. Each partition in the refined partitioning scheme is an intersection of partitions from the two input partitioning schemes, and the cost function C_p in the partition is the addition of the linear cost functions from the two intersecting partitions. The algorithm for to *refine* partitioning schemes is outlined in Section 5.1.1, while Section 5.1.2 outlines how to *add* piecewise linear cost functions.
- Comparing alternative operation or plans, say P_1 and P_2, for evaluating a particular expression: Here, again, we refine the two partitioning schemes of parameter space (one from P_1 and one from P_2), and compare the linear cost functions in each refined partition. Details of a function *MinMerge* which does this are presented in Section 5.1.3.

5.1 Operations on Partitioning Scheme and *PLCF*

In this section we define some operations on the partitioning schemes of the parameter space and piecewise linear cost functions.

5.1.1 Operation *Refine* on partitioning schemes

The function *Refine*(PS_a, PS_b) takes as input two partitioning schemes PS_a and PS_b, computes the pairwise intersection of the partitions (polytopes) in the

input schemes, and discards the null intersections. The non-null intersections define a new partitioning scheme of the parameter space, which the function returns. A *PLCF* defined on either of the input partitioning schemes will be linear in each partition of the refined partitioning scheme.

Figure 4(c) shows the result of refinement of partitioning schemes of a one dimensional parameter space defined in Figures 4(a) and 4(b). A partition $r_{a_ib_j}$ in Figure 4(c) results from an intersection of a partition r_{a_i} from Figure 4(a) and a partition r_{b_j} from Figure 4(b). As an example, we see that the partition $r_{a_1b_1}$ is created by intersecting partition r_{a_1} and r_{b_1}. The partition $r_{a_2b_3}$ does not exist as the partitions r_{a_2} and r_{b_3} do not intersect. Figures 5(a)-(c) show partitioning schemes in 2 dimensions. The partitioning scheme in Figure 5(c) is the result of refinement of the partitioning schemes in Figure 5 (a) and (b). The output partitioning scheme can be thought of as the superimposition of the two input partitioning schemes.

5.1.2 Operation *Add* on *PLCF*

Logically, the value of the output cost function at each point in the parameter space is obtained by adding two input cost functions at that point.

Computationally, we take the refinement of the partitioning schemes defined by the input cost functions to create the partitioning scheme for the output cost function. In each output partition each input cost function is linear and in each such partition we add the two input cost functions to define the output cost function in that partition.

5.1.3 Operation *MinMerge* on *PLCF*

Logically, the value of the output cost function at a point in the parameter space is the minimum of the values of the two input cost functions at the point. So, we compare the input cost functions at each point in the parameter space and pick the minimum of them to create the output cost function.

Computationally, the function *MinMerge* does the following:

1. Take the refinement of the partitioning schemes defined by the input cost functions to create the partitioning scheme for the output cost function.
2. Within each partition of the refined partitioning scheme, the input cost functions are linear. Do the following for each partition:
   (a) Consider the hyperplane defined by equating the two (linear) input cost functions, say f_1 and f_2; this hyperplane divides the parameter space into two halves; f_1 is less than or equal to f_2 in one half, and f_2 is less than or equal to f_1 in the other.
   (b) If the hyperplane does not intersect the output partition then the partition lies on one side of the hyperplane and hence one of the two input cost function, say f_1, is better than the other throughout the partition; the partition is not refined further, and f_1 is its output cost function.
   (c) Else the hyperplane is used to split the partition into two parts, thus refining the partition. Function f_1 is better in one of the parts, for which it is the output cost function, and f_2 is better in the other part, for which it is the output cost function.[9]

Figures 4(a) - (d) show an example with one parameter. Figure 4(c) shows the result of refining the two partitioning schemes defined in Figure 4(a) and Figure 4(b). Figure 4(d) shows the result of the $MinMerge$ operation on the cost functions defined in Figure 4(a) and Figure 4(b). In output partitions $r_{a_1b_2}$ and $r_{a_2b_2}$ the input cost function CF_b is less than the input cost function CF_a and the output cost function is equal to CF_b in these partitions. In partition $r_{a_4b_3}$ the input cost function CF_a is less that the input cost function CF_b and the output cost function is equal to CF_a in this partition. However in partitions $r_{a_1b_1}$ and $r_{a_3b_2}$ neither of the input cost functions is less than the other throughout and hence we need to split these partitions further. The separating point (a hyperplane in 1-dimension is a point) in each of the partitions is given by equating the two input cost functions in the partition.

5.2 Extensions to System R Algorithm

We now outline how to extend the System R optimization algorithm to handle cost functions in place of cost values. Figure 6 shows the pseudocode of (a recursive formulation of) the System R optimization algorithm.

Figure 7 shows the pseudo code for the extended algorithm. Note the key differences: cost function addition replaces addition of cost values, and the *MinMerge* operation replaces the selection of the minimum cost plan. In the cost value case, only one of the alternative plans for a subset of relations is chosen. In the cost function case, different plans may be optimal at different points; all of these are retained by *MinMerge*.

5.3 Approximating cost functions to piecewise linear form

In general, the cost of an operation in a query plan is a function of statistics such as its input sizes. In the case of PQO, the statistics may be estimated as a function of the optimization parameters. Thus, we can express the cost function of the operation as a function of the

[9]This partitioning technique can be thought of as a specialization of the recursive decomposition algorithm (Section 3.2) to the case where only two cost functions need to be combined.

Input: SPJ query q on a set of
relations $Q = \{R_1, \ldots, R_n\}$
Output: Optimal plan P_Q for the query q
/* For $S \subseteq Q$, $Cost_S$ denotes cost of P_S */
for $i = 1$ to n do
  P_{R_i} = Access plan of R_i
for $i = 2$ to n do
  for all $S \subseteq Q$ s.t. $\|S\| = i$ do
    P_S = dummy plan with infinite cost
    for all R_j, S_j s.t. $S = \{R_j\} \cup S_j$
      op = join operation on S_j and R_j
      $p = S_j \bowtie R_j$
      $Cost_p = Cost_{op} + Cost_{R_j} + Cost_{S_j}$
      if $Cost_p < Cost_{P_S}$ then $P_S = p$

Figure 6: System R Algorithm

parameters, and then approximate it in the fashion outlined below.

For non-linear functions, the general approximation approach is as follows: We find the equi-cost contours in the parameter space and divide the parameter space into bands such that an equi-cost contour separates two adjacent bands. The cost function within a band is the linear interpolation of the cost along the two boundary contours and the width of a band is such that within the band the linear cost function can approximate the actual cost function to a desired degree. Given the non-linear nature of the cost functions, such bands would typically be non-convex. We further divide each band into regions, and approximate each region by a convex polytope such that the polytope approximates the corresponding region closely.

Figure 8(a) shows some equi-cost lines for a two parameter case where the cost is proportional to the product of the parameter values and Figure 8(b) shows a partitioning scheme for this case. Consider a nested-loops join operation on two relations, each with a parametrized selection; the cost of the join is proportional to the product of the sizes of the inputs, and is thus a function of the product of the parameters, which would be handled by this approximation.

We can create specific approximation procedures, based on the above approach, for simple non-linear functions such as product of parameters. Cost functions obtained by adding such non-linear functions can be approximated by approximating each part, and then combining the approximations using refinement of the partitions.

Operator cost functions may not only be non-linear, but may also be discontinuous in the given parameters. For typical cost functions (e.g. the merge-sort operation when its input becomes bigger than memory) the contours of the discontinuities involved are similar to the equi-cost contours and the approach outlined above can be applied for approximating the cost functions involving discontinuities as well.

Input: SPJ query q on a set of
relations $Q = \{R_1, \ldots, R_n\}$
Output: Optimal $CostFn_Q$ for the query q
/* The optimal cost function contains a partitioning
scheme on the parameter space with each partition
having an optimal plan attached to it */
for $i = 1$ to n do
  $CostFn_{R_i}$ = Access cost function of R_i
for $i = 2$ to n do
  for all $S \subseteq Q$ s.t. $\|S\| = i$ do
    $CostFn_S$ = dummy cost function with
    infinite cost
    for all R_j, S_j s.t. $S = \{R_j\} \cup S_j$
      op = join operation on S_j and R_j
      $p = S_j \bowtie R_j$
      $CostFn_p = CostFnAdd(CostFn_{op}, CostFn_{R_j}, CostFn_{S_j})$
      $CostFn_S = MinMerge(CostFn_S, CostFn_p)$

Figure 7: Extended System R Algorithm

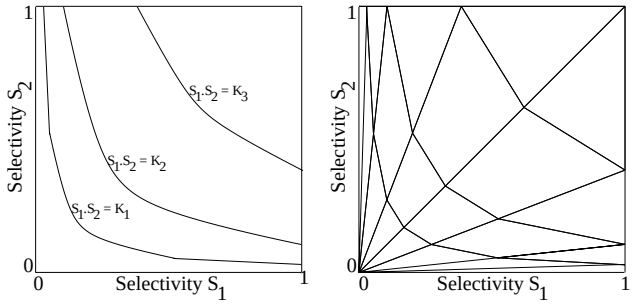

Figure 8: Cost function approximation: (a) Equi-cost lines (b) Corresponding decomposition

5.4 Discussion

We have extended the Volcano query optimization algorithm to handle parametric query optimization. The extensions are similar to the extensions of the System R algorithm, except for some extra care to be taken when using cost-based pruning. Details are available in [HS02].

We have implemented the extensions by modifying an existing Volcano-based query optimizer developed at IIT Bombay. We are using *Polylib*, a public domain polytope manipulation library to implement polyhedron operations [Pol]. We have tested the extensions for queries with two parameters. One of the problems we faced is that the library requires calculation using exact rationals, leading to unbounded integer sizes, which it handles using the GNU MP arbitrary precision integer package, and this results in a significant overhead. We are exploring alternative polytope

manipulation techniques that do not require arbitrary precision arithmetic, to reduce the overhead. We plan to conduct a performance evaluation subsequently.

Linear vs. piecewise linear techniques

The cost polytope algorithm for the linear case can be applied in the piecewise linear case by pre-partitioning the parameter space in a way that every cost function is linear in every partition. However, doing so would result in a grossly over-refined space. Instead, what our algorithm for the piecewise linear case have achieved is to refine the space for each operation and plan separately: thus the space is refined only as much as is needed for that operation or plan.

The algorithm considers only two cost functions at a time and while adding or comparing the cost functions, the partitioning scheme is refined only as much as is needed – within each partition of the refinement both the cost functions are linear. We could apply the recursive decomposition technique for merging linear cost functions within each refined partition. However, we need to consider only two cost functions at a time, unlike the recursive decomposition (and the cost polytope) algorithm which considers (in effect) all possible plans; merging two cost functions at a time is considerably simpler. Although real world cost functions are often not linear, our work on linear cost functions is still of theoretical interest, and may have applications in restricted domains. Our work on piecewise linear cost functions is of practical significance.

6 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. [Gan98] and [Rao99] show that the expected number of plans generated by this algorithm could be much larger than the expected size of the parametric optimal set. [Rao99] presents sampling techniques based on selecting points from the parameter space at random and optimizing them using a conventional optimizer. The set of plans returned will be a subset of the parametric optimal set of plans. [Rao99] also presents a hybrid algorithm combining the sampling techniques with the partial order technique.

[INSS92, INSS97] present a randomized approach for parametric query optimization with memory as a parameter. The technique proposed is heuristic based and does not guarantee generation of all parametric optimal plans or any bound on the sub-optimality of the plans.

[GK94] considers a one parameter parametric query optimization problem involving a two site distributed database system with relative load factor as the parameter. This is a specific instance of 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 (or "affine") cost functions in two parameters. The solution involves a complicated mechanism of traversing the parameter space along the boundary of the polyhedral decomposition and finding neighboring regions. The complexity involved restricts it to 2 parameters. [Bet99] reports experimental results of this technique for the one parameter case for linear and star queries. [Pra99] reports an experimental evaluation of the algorithm for "affine extensible" cost functions proposed in [Gan98].

In a currently unpublished followup work [Gan01], Ganguly has extended the algorithm to more than two parameters. Ganguly's algorithm calls itself recursively on the faces of the parameter space polytope, moving from lower dimensional faces to higher dimensional faces, and finds optimal plans and the decomposition of the faces induced by these plans. At the end of the recursive phase, it takes the plans optimal along the boundary of the parameter space polytope, and constructs the decomposition induced by these plans. This decomposition creates some vertices in the interior of the parameter space polytope; these vertices are optimized one by one. If optimization of a vertex returns a new plan, the optimal region for it is carved out from the existing decomposition. This procedure is continued till all the vertices are optimized.

The basic difference in the two algorithms is that Ganguly's algorithm works in $\Re^n$, whereas our algorithm works in $\Re^{n+1}$, which results in considerable simplification of the procedures. Specifically, in Ganguly's algorithm, the procedure for carving out the optimal region for a new plan (given the existing decomposition) begins with the parameter space polytope and chips out the optimal region of each plan in the existing decomposition. This procedure is invoked for each new plan added and may prove to be expensive. In contrast since we work in higher dimensional space, we only require a single hyperplane intersection. We also completely avoid the complexity of the initial decomposition phase of Ganguly's algorithm.

If the conventional optimizer returns all plans, the number of calls to the optimizer is roughly the same for both algorithms. However, our algorithm can be used even if the optimizer returns only one plan, unlike Ganguly's algorithm. [Gan01] discusses several extensions, including a special case of nonlinear cost functions, which we do not consider.

Our algorithm for the piecewise linear case is novel

to the best of our knowledge. The memory cognizant optimization algorithm which we developed earlier [HSS00] can be viewed as a special case of our current algorithm, for the case of a single parameter, namely memory. [HSS00] shows how to divide the available memory optimally amongst the operations running in a pipeline, given the *cost versus memory allocation* function for each operation and extends the conventional optimizer to build a memory cognizant optimizer; the extended optimizer takes into account the division of memory amongst operations and generates an optimal, memory-aware execution plan.

7 Conclusion

In this paper, we first proposed a parametric query optimization algorithm for linear cost functions which is non-intrusive in that it uses a conventional query optimizer without modifying it. Unlike approaches published earlier, it is simple, yet general enough to handle an arbitrary number of parameters. We proved a lower bound on the number of invocations of the conventional optimizer, and showed that under certain assumptions, the number of invocations made is close to the lower bound.

We then proposed a solution for the PQO problem for the more general case when the cost functions are piecewise linear. The solution is based on modification of an existing query optimizer. The solution works for an arbitrary number of parameters, and is very general since nonlinear cost functions can be approximated to piecewise linear form.

We have implemented our PQO algorithm for piecewise linear cost functions, and tested it for queries with two parameters. Future work includes implementing or using polyhedron handling code that minimizes overheads, and characterizing the performance of our algorithms.

Acknowledgments

We wish to thank Bharat Adsul, Milind Sohoni, Sudeep K. S. and Sundar Vishwanathan for their feedback, Vinit Kapoor for help with the polytope handling code, and Anandi Herlekar and the referees for their comments which helped improve the presentation significantly. We are grateful to Prasan Roy for providing code of basic Volcano Query Optimizer and to Vincent Loechner for his help with the Polylib library.

References

[Bet99] A. V. Betawadkar. Query optimization with one parameter. Technical report, Indian Institute of Technology, Kanpur, Feb 1999. Available at http://www.cse.iitk.ac.in/research/mtech1997.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the SIGMOD*, pages 150–160, 1994.

[Gan98] Sumit Ganguly. Design and analysis of parametric query optimization algorithms. In *Proc. of the VLDB*, pages 228–238, 1998.

[Gan00] Sumit Ganguly. Personal communication. Dec, 2000.

[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 *Proc. of the COMAD, Pune, India*, pages 20–41, 1994.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. of the ICDE*, pages 209–218, 1993.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proc. of the SIGMOD*, pages 358–366, 1989.

[HS02] Arvind Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. Technical report, Indian Institute of Technology, Bombay, June 2002. Available at http://www.cse.iitb.ac.in/aru.

[HSS00] Arvind Hulgeri, S. Seshadri, and S. Sudarshan. Memory cognizant query optimization. In *Proc. of COMAD*, pages 181–193, 2000.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *Proc. of the VLDB*, pages 103–114, 1992.

[INSS97] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *VLDB Journal*, 6(2):132–151, 1997.

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

[Pol] PolyLib: A library of polyhedral functions. Available at http://icps.u-strasbg.fr/PolyLib.

[Pra99] V. G. V. Prasad. Parametric query optimization: A geometric approach. Technical report, Indian Institute of Technology, Kanpur, Feb 1999. Available at http://www.cse.iitk.ac.in/research/mtech1997.

[Rao99] S. V. U. M. Rao. Parametric query optimization: A non-geometric approach. Technical report, Indian Institute of Technology, Kanpur, Feb 1999. Available at http://www.cse.iitk.ac.in/research/mtech1997.

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proc. of the SIGMOD*, pages 23–34, 1979.

[Zie94] Gunter M. Ziegler. *Lectures on Polytopes*. Springer Verlag, 1994.