

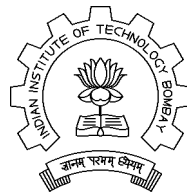
PARAMETRIC QUERY OPTIMIZATION

Submitted in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

by

ARVIND HULGERI



Department of Computer Science and Engineering
Indian Institute of Technology - Bombay
2004

Abstract

Database queries are typically specified declaratively, and the database system has to choose an appropriate execution plan for the query. A query optimizer in a database system is responsible for transforming an SQL query into an execution plan. Most modern optimizers are cost-based in that they decide between alternative execution plans by comparing their estimated execution costs.

The cost of a query plan depends on many parameters such as predicate selectivities, available memory, and presence of access paths, 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. At run time, when the actual parameter values are known, the candidate plan corresponding to the actual parameter values is picked and used. In this thesis we propose solutions for several cases of the PQO problem.

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 as a subroutine with minor modifications: the solution invokes the conventional query optimizer multiple times, with different parameter values. The solution works for an arbitrary number of parameters.

Second, we propose a solution for the PQO problem for the case when the cost functions are piecewise-linear in the given parameters. The solution requires modification to an existing query optimizer. This solution is general, since arbitrary cost functions can be approximated to piecewise-linear form.

We then 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. We have implemented the heuristic and the results of the tests on the TPCD benchmark indicate that the heuristic is very effective.

As the final contribution, we show how to make a query optimizer *memory cognizant*. Typical optimizers assume all the memory to be available to each operator in a plan. But while executing pipelines, memory has to be divided amongst all the operators running simultaneously in a pipeline. The cost of an operator generally depends on the available memory. Thus the choice of execution plan and memory division are interdependent and, if done separately may not give the optimal plan. We address the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator. We show how to make a query optimizer *memory cognizant*. We have implemented our techniques on a query optimizer based on the Volcano query optimization algorithm. Though parametric query optimization and memory cognizant query optimization seem to be dissimilar problems, there are similarities in the techniques we used to solve these problems.

Contents

1	Introduction	1
2	Background	5
2.1	Problem Definition: Parametric Query Optimization (PQO)	5
2.2	Prior Work on Parametric Query Optimization	5
2.3	Polytopes	7
2.3.1	Convex Polytopes	7
2.3.2	Polytope Representation	8
2.4	The Volcano Query Optimization Algorithm	8
2.4.1	The AND-OR DAG Representation of Queries	8
2.4.2	Physical AND-OR DAG	9
2.4.3	The Volcano Search Algorithm	9
3	PQO for Linear Cost Functions	11
3.1	Assumptions	12
3.2	Properties of Linear Cost Functions	12
3.3	The Recursive Decomposition Algorithm	13
3.4	The Cost Polytope Algorithm	15
3.4.1	Parametric Optimal Cost Function	16
3.4.2	Cost Hyperplane, Cost Halfspace and Cost Polytope	16
3.4.3	Algorithm for Cost Polytope Construction	17
3.4.4	Ordering of the vertices to be optimized	19
3.4.5	Complexity in terms of number of calls to the conventional optimizer	20
3.4.6	An example with single parameter	21
3.4.7	Initial cost polytope	22
3.4.8	Comparing the algorithms	22
3.4.9	Optimization given all optimal plans at a given point	23
3.5	Comparison with Ganguly's solution	24
3.6	Approximate Cost Polytope Algorithm	24
3.7	Discussion	25
3.8	Summary	26
4	PQO for Piecewise Linear Cost Functions	27
4.1	Piecewise Linear Cost Functions (<i>PLCF</i>)	27
4.1.1	Parameter Space Partitioning Scheme	28
4.1.2	Definition: Piecewise Linear Cost Functions (<i>PLCF</i>)	28
4.2	Approximating cost functions to piecewise linear form	29
4.3	Optimization for Piecewise Linear Cost Functions	30
4.3.1	Operations on Partitioning Scheme and <i>PLCF</i>	31
4.3.2	Cost based pruning	33

4.4	Extensions to System R Algorithm	33
4.5	Extensions to Volcano Query Optimizer	33
4.5.1	Extended Cost Function and Plan	33
4.5.2	Extensions to the search algorithm	34
4.6	Implementation	37
4.7	Summary	37
5	PQO for Nonlinear Cost Functions	38
5.1	An overview of AniPQO	39
5.2	Representation and manipulation of the decomposition	41
5.2.1	Facial lattice and edge skeleton of the decomposition	41
5.2.2	Updating the decomposition vertex set	42
5.2.3	Finding an equi-cost point	44
5.2.4	An example iteration of AniPQO	45
5.3	The DAG Representation of Plans	46
5.4	Experimental Evaluation	47
5.5	Summary	51
6	Memory Cognizant Query Optimization	52
6.1	Motivating Example	53
6.2	Related Work	54
6.3	Memory Cognizant Query Execution	55
6.4	Optimal division of Memory for a given Query Plan	59
6.4.1	Cost Functions with Arbitrary Shape	59
6.4.2	Piecewise Linear Approximation of Cost Functions	61
6.5	Memory Cognizant Optimization	62
6.5.1	Breaking Pipelined Edges	63
6.6	Memory Cognizant Volcano Optimizer	64
6.6.1	Extended Cost Function and Plan	65
6.6.2	Operations on Cost Functions	65
6.6.3	Detailed Algorithm	65
6.7	Experimental Evaluation	70
6.8	Summary	71
7	Conclusions	72
A	POSP: An Example	74
B	Polytope Construction	78
B.1	Polytope Construction	78
B.2	Adapting the algorithm to construct cost polytope	79
C	Proofs for Linear Cost Function Properties	80
D	Relaxing the assumption from Section 5.2.1	82
E	Approximating the parameter space decomposition	85

Chapter 1

Introduction

Database queries are typically specified declaratively and the DBMS has to choose an appropriate execution plan for the query. A query optimizer in a relational DBMS is responsible for transforming an SQL query into an execution plan. Most modern optimizers are cost-based in that they decide between execution plans by comparing their estimated execution costs.

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 some 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, in general, the memory available for query execution is not known until the query execution time.

Consider the following query with two parameterized range predicates, where $:1$ and $:2$ represent the parameters whose values are known at run time:

```
select *
from A, B
where A.x = B.y and A.z < :1 and B.w < :2
```

If the cost of the query execution is linear in the parameters, it can be represented by equation $a_1.s_1 + a_2.s_2 + a_3$, where variables s_1 and s_2 denote the selectivities of the parameterized range predicates ($A.z < :1$) and ($B.w < :2$) resp., and a_1 , a_2 and a_3 are constants. In general, the cost functions may be nonlinear.

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 as shown by Graefe and Ward [13]. 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 [20]. At query execution time, when the actual parameter values are known, an appropriate plan is chosen from the set of candidates, which can be much faster than re-optimizing the query.

In the above example, we can use either $:1$ and $:2$ or s_1 and s_2 as the parameters at compile time. In the latter case, at runtime, when the values of $:1$ and $:2$ are known, we can estimate the values of s_1 and s_2 and use them to pick the plan that is optimal at that point in the space of the parameter values.

Some modern commercial DBMS cache query plans and reuse cached plan when a “similar” query is executed again [11]. For example, the Oracle database system provides a mechanism called “stored outlines” to cache queries and plans. If the input query matches with any of the stored queries, the corresponding plan is used for evaluating the input query. The matching is done at syntactic level but bind variables are supported so that queries are treated to be the same if they are syntactically the same, modulo the constants appearing in the predicates.

Instead of caching a single plan for a given query with fixed parameter values, we can use PQO to

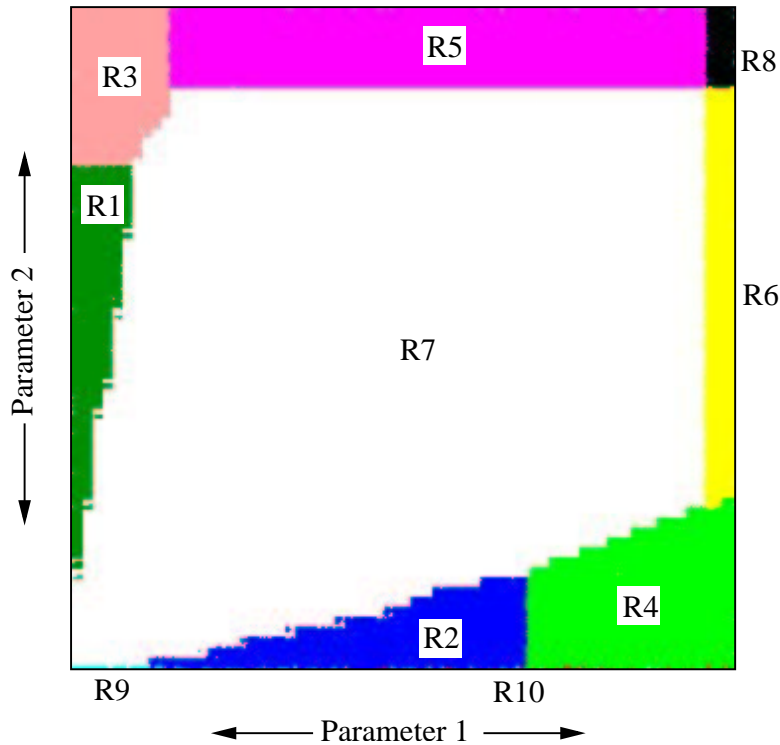


Figure 1.1: Parameter space decomposition: An example

generate and cache multiple plans for different parts of the space of the parameter values. 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.

A *parametric optimal set of plans (POSP)* is a minimal subset of the plan space that includes at least one optimal plan for each point in the parameter space. 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.

Figure 1.1 shows the parameter space decomposition for a query with two parameters. The *POSP* has ten plans and the corresponding ten regions are marked from $R1$ to $R10$ in the decomposition. The query is a simple two relation SPJ query with a parameterized selection predicate (`table.column < parameter_value`) on a column of each relation. The selectivities of the two parameterized selection predicates are the parameters. See Appendix A for details.

Parametric query optimization (*PQO*) involves two steps: finding the *POSP* and picking an appropriate plan from this set at run time, when the parameter values are known.

Summary of contributions

In this thesis we address the problem of parametric query optimization. We start with parametric query optimization for a simple linear cost model and then consider more practical cases of piecewise linear and nonlinear cost models. We then consider the closely related problem of memory cognizant query optimization. Our contributions are described in more detail below.

Parametric Query Optimization for Linear Cost Functions

We first consider the case of parametric query optimization with linear cost functions and, in Chapter 3, provide an algorithm, the *cost polytope algorithm*, to solve the problem.

The cost function of a plan can be represented by a linear equation in $n + 1$ variables (n parameters variables and one cost variable) and can be thought of as a hyperplane in \mathbb{R}^{n+1} ; we call such a hyperplane a *cost hyperplane*. We use these hyperplanes – one for each plan in the plan space – to define a convex polytope which we call the *cost polytope*. The cost polytope algorithm constructs the cost polytope. By projecting the cost polytope on the parameter space (which is \mathbb{R}^n) we can define the regions of optimality for the plans in *POSP*. 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). Ganguly [8] also presents a parametric query optimization algorithm for linear cost functions. Our algorithm is simple and has some advantages over the one proposed by Ganguly; we compare the algorithms in detail in Section 3.5.

We measure the complexity in terms of the number of calls to the conventional optimizer as done by Ganguly [8]. We prove a lower bound on the number of calls to the optimizer, and show that under certain assumptions, the number of calls is close to the lower bound.

Though the cost functions are seldom linear in real life and the results are theoretical, we use the results later in Chapter 5 where we develop AniPQO – a heuristic PQO solution for the case when the cost functions are nonlinear and discontinuous.

Parametric Query Optimization for Piecewise Linear Cost Functions

We next consider the case of parametric query optimization with piecewise linear cost functions. A cost 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.

In Chapter 4, we present extensions to a conventional optimization algorithm to turn it into a parametric query optimizer for piecewise linear cost functions. The solution conceptually works for an arbitrary number of parameters, and is general since nonlinear cost functions can be approximated to piecewise linear form. 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, and keep track of multiple plans, along with their regions of optimality, for each query/subexpression.

We show how to extend the System R and Volcano query optimization algorithms to handle piecewise linear cost functions in place of cost values. We have implemented the extensions by modifying an existing Volcano Query Optimizer prototype developed at IIT Bombay. We have tested the extensions for queries with two parameters. The solution is exact if the cost functions are piecewise linear but is intrusive and needs modifications to the conventional optimizer; our PQO heuristic for nonlinear cost functions described next overcomes this drawback.

Parametric Query Optimization for Nonlinear Cost Functions

In Chapter 5, we present 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 finds a subset of *POSP* and is based on the algorithms for the linear case described in Chapter 3. It differs from the earlier algorithms in the details of how it performs the parameter space decomposition, taking non-linearity of cost functions into account. AniPQO uses an AND-OR DAG representation of a set of plans found to boost the quality of the results and facilitate picking an optimal plan at run time. AniPQO works with arbitrary nonlinear and discontinuous cost functions¹. 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. AniPQO is minimally intrusive in the sense that it does not need to modify the conventional query optimizer, and

¹The cost functions of almost all relational operators are nonlinear and discontinuous; for example, the cost of nested-loops join, in its simplest form, is given by, $\lceil \frac{NumLeftRelationBlocks}{NumAvailableMemoryBlocks - 1} \rceil * NumRightRelationBlocks$, and is non-linear and discontinuous in $NumAvailableMemoryBlocks$.

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.

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 than the actual optimal plan; the maximum performance degradation observed on a sample set of queries is very small (3.5%). Although the optimization 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 practical for up to 4 parameters. If we consider either only query parameters or only system parameters, the limit of four seems reasonable; but if we consider both, the number of parameters may exceed four.

AniPQO, though heuristic, generates good solutions and is much faster than our solution for piecewise linear cost functions. And AniPQO, as already said, is minimally intrusive and applicable even if the cost functions are nonlinear and discontinuous.

Memory Cognizant Query Optimization

Typical optimizers assume all the memory to be available to each operator in a plan. But while executing a pipeline, memory has to be divided amongst all the operators running simultaneously in the pipeline. The cost of an operator generally depends on the available memory. If the memory allocated to an operator is less than what an optimizer assumes, the cost estimated by the optimizer would be wrong. Thus the query optimization and memory division are interdependent and if done separately may not give the optimal plan. The query optimizer should not only consider the total memory available but should also decide how to divide it optimally among the operators of the plan.

In Chapter 6, we address the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator. We build *memory cognizant query optimizer* to solve this problem. We have extended the Volcano optimizer to make it memory cognizant.

Though parametric query optimization and memory cognizant query optimization seem to be dissimilar problems, there are similarities in the techniques we used to solve these problems.

A part of the job of the optimizer is to decide which edges to pipeline and which edges to block. A pipelined edge can be converted into a blocking edge (*i.e.* the child operator writes the intermediate result to the disk and the parent operator reads it back). But the decision to convert a pipelined edge into a blocking edge depends upon whether the extra memory available to the individual pipelined trees thus formed can more than offset the extra disk IO of the intermediate result. This decision is integrated into our memory cognizant optimizer.

Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 formally defines the parametric query optimization problem and reviews previous work in parametric query optimization. It also provides some background material on polytopes and describes the Volcano query optimization algorithm. Chapter 3 describes our work on parametric query optimization for linear cost functions. Chapter 4 describes our work on parametric query optimization for piecewise linear cost functions. Chapter 5 describes our work on parametric query optimization for nonlinear cost functions. Chapter 6 describes our work on memory cognizant query optimization. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background

In this chapter we provide background material: Section 2.1 formally defines the parametric query optimization problem. Section 2.2 describes prior work in parametric query optimization. Section 2.3 provides some background material on polytopes. Section 2.4 describes the Volcano query optimization algorithm.

2.1 Problem Definition: Parametric Query Optimization (PQO)

The parametric query optimization (PQO) problem is defined by Ganguly [7] as follows: Let s_1, s_2, \dots, 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 $C_p(s_1, s_2, \dots, s_n)$. 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:

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

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

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

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 (a) to find a *POSP* and (b) to devise a mechanism to find the optimal plan at a given point in the parameter space at run-time.

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 Prior Work on Parametric Query Optimization

Graefe and Ward [13] make a case for parametric query optimization, and propose *dynamic query plans* that include a *choose-plan* operator, which chooses a plan, at runtime, from among multiple available plans depending upon the values of certain run-time parameters. They propose to introduce choose-plan operators at all the places in the plan where the choice amongst the alternative plans available is sensitive to the values of the parameters.

Cole and Graefe [5] present 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 costs 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; a partial order is defined over the plan space s.t. for plans p and q , $p \leq q$ if the cost interval of p lies to the left of the cost interval of q . Then the set of optimal plans w.r.t. this order is the set of parametric optimal plans. A prototype was built by extending the Volcano query optimizer by Graefe and McKenna [12]. The technique computes a superset

of the parametric optimal set and does not specify the regions of optimality for the plans. It is shown by Ganguly [7] that the expected number of plans generated by this algorithm could be much larger than the expected size of the parametric optimal set and Rao [30] verified this fact experimentally. Rao [30] devises a better partial order technique but is restricted to affine (*i.e.* linear) cost functions, and an extension called *affine extensible* cost functions. The affine extensible cost functions are defined later in this section.

Rao [30] studies the distribution of the parametric optimal plans in the parameter space for the 2-dimensional case and devises several sampling techniques. The sampling techniques involve 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. The sampling techniques include r - θ sampling, sampling with geometric approach and sampling with non-geometric approach. In r - θ the parameter space is divided radially and angularly and the center of each partition is sampled. The sampling with geometric approach considers cost functions of the form $a_0 + a_1 p_1 + a_2 p_2 + a_3 p_1 \cdot p_2$ where a_i 's are constants and p_i 's are parameter variables. It selects some values for either of the parameters. For each of the selected values, the parameter is fixed at the value and the resulting 1-parameter problem with linear cost functions is solved with the method proposed by Ganguly [7]. The sampling with non-geometric approach employs a partial order technique in two thin strips, each touching one parameter axis, and employs random uniform sampling in the rest of the parameter space. They tested the techniques on star and linear queries with two parameters. Rao [30] also presents a hybrid algorithm combining the sampling techniques with the partial order technique.

Ioannidis *et.al.* [20, 21] present a randomized approach – based on iterative improvement and simulated annealing techniques – for parametric query optimization with memory as a parameter. The technique presented is an extension of the transformation based randomized query optimization algorithms by Ioannidis and Kang [18, 19, 17, 36, 35]. In the transformation based randomized query optimization algorithms, the plan space is modeled as a graph with each vertex representing a plan and each edge representing a transformation from one plan to another. These algorithms start from some plan in the space, traverse the space in a controlled fashion and return the best plan found during the traversal. The technique proposed by Ioannidis *et.al.* [20, 21] assumes the parameter space to be discrete and runs the randomized query optimizer for each point in the parameter space. A *sideways information passing* mechanism facilitates information sharing across the runs of the randomized optimizer, and thus a run can make use of a plan found by other runs. The technique is unsuitable for continuous parameters, like selectivity.

Ganguly and Krishnamurthy [10] consider a PQO problem with one parameter 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. Ganguly [7] extends the work in [10] and proposes a solution for parametric query optimization with linear 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. Betawadkar [1] reports experimental results of this technique for the one parameter case for linear and star queries.

In a currently unpublished (independently done) work [8], Ganguly has extended the algorithm from [7] so as to work for more than two parameters. A detailed comparison of the algorithm with our parametric query optimization algorithm for linear cost functions appears in Section 3.5.

Ganguly and Krishnamurthy [10] and Ganguly [7, 8] 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 $Q = \{1, 2, 3, \dots, n\}$, a_S 's are constants, and p_i 's are parameter 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. Prasad [29] reports an experimental evaluation of the algorithm for the affine extensible cost functions.

Chu *et. al.* [4, 3] propose 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.

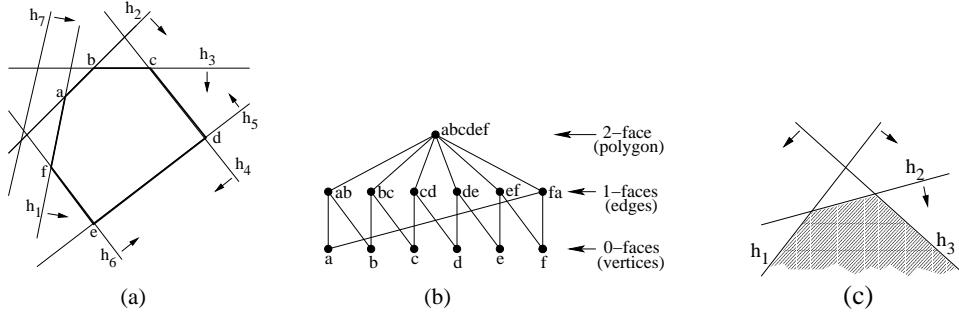


Figure 2.1: (a) a polytope, (b) its facial lattice and (c) a lower convex polytope in 2-dimensions

The *Plastic* system, proposed by Ghosh *et. al.* [11] 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. Unlike PQO, *Plastic* generates the set of candidate plans incrementally as different queries are executed. It does not define the regions of optimality for the candidate plans and matches a new query to the set of candidate queries to decide which candidate plan to use to execute the new query. If the new query does not match with any of the candidate queries, the new query is optimized to generate a new candidate plan and, the new query and the new plan are inserted into the candidate query set and the candidate plan set, respectively.

2.3 Polytopes

In the proposed solutions, we need to represent and manipulate parameter space partitions and we do it using polytopes. In Section 2.3.1, we define a convex polytope and a special type of polytope, lower convex polytope. Section 2.3.2 describes a method from Mulmuley [25] for representing convex polytopes. In Appendix B we outline a method for constructing polytopes, as described in [25].

2.3.1 Convex Polytopes

A *convex polytope* in \mathbb{R}^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 $a_1x_1 + a_2x_2 + \dots + a_dx_d \geq a_0$, where each a_j is a constant, the x_j 's denote the coordinates in \mathbb{R}^d , and a_1, a_2, \dots, a_n are not all zero. The boundary of this halfspace is the hyperplane defined by $a_1x_1 + a_2x_2 + \dots + a_dx_d = a_0$. We denote the bounding hyperplane of a halfspace M_i by ∂M_i .

Let $P = \cap_i M_i$ be any convex polytope in \mathbb{R}^d , where each M_i is a halfspace. It is possible that P lies completely within some hyperplane ∂M_i . This can happen, for example, when the set $\{M_i\}$ contains two half-spaces lying on the opposite sides of a common bounding hyperplane. In this case P can be thought of as a convex polytope of lower dimension within the containing hyperplane. Otherwise, P is a full d -dimensional convex polytope in \mathbb{R}^d . A d -dimensional convex polytope is also called a d -polytope. In what follows, we assume that P is a d -polytope in \mathbb{R}^d .

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 thus a $(d-1)$ -face. We can define the i -faces of P , where $i < d-1$, inductively: A $(d-2)$ -face of P is a $(d-2)$ -face of a facet of P , and so on. Each i -face

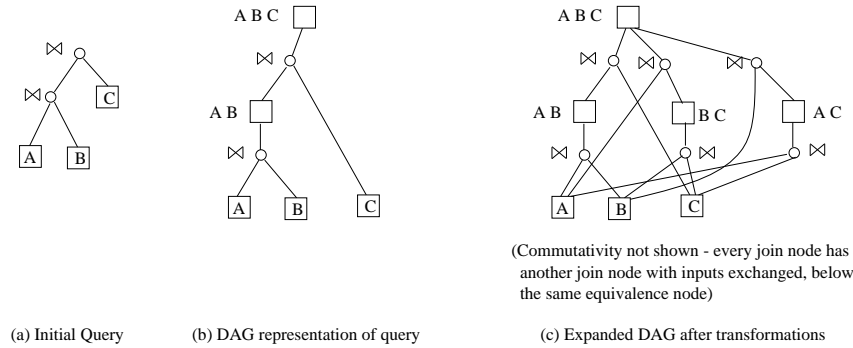


Figure 2.2: Initial Query and DAG Representations

of P is a i -dimensional convex polytope. A 0-face is also called a *vertex*. A 1-face is also called an *edge*. As a convention, the d -face of P is P itself. For example, in three dimensions, a polytope is a 3-face, 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. We say that a i -face f is adjacent to an j -face g , where $i < j$, if f is a sub-face of g .

Figure 2.1 (a) shows a polygon $abcdef$ in \mathbb{R}^2 (a polytope in \mathbb{R}^2 is a polygon.) It is defined by the halfspaces h_1, h_2, \dots, 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 2.1 (c) shows a lower convex polytope. (Part (b) of the figure is discussed below.)

2.3.2 Polytope Representation

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. We group the adjacency edges incident on a node in two classes: (1) the ones that correspond to the adjacencies with the faces of higher dimensions, and (2) the ones that correspond to the adjacencies with the faces of lower dimensions. Each node contains auxiliary information such as pointers to the half-spaces whose bounding hyperplanes contain the corresponding face. Figure 2.1 (b) shows the face lattice for the polytope in Figure 2.1 (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*.

2.4 The Volcano Query Optimization Algorithm

In this section we describe the Volcano query optimization by Graefe and McKenna [12]; our description is also based on Roy [31]. First we describe an AND-OR DAG representation for the search space of all possible query plans. Then we describe the Volcano search algorithm.

2.4.1 The AND-OR DAG Representation of Queries

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 the join operation (\bowtie) or a select operation (σ). It represents the expression defined by the operation and its inputs. Hereafter, we refer to the AND-nodes 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. We shall refer to the OR-nodes as *equivalence nodes* henceforth.

The given query tree is initially represented directly in the AND-OR DAG formulation. For example, the query tree of Figure 2.2 (a) is initially represented in the AND-OR DAG formulation, as shown in Figure 2.2 (b). Equivalence nodes (OR-nodes) are shown as boxes, while operation nodes (AND-nodes) are shown as circles.

The initial AND-OR DAG is then expanded by applying all possible transformations on every node of the initial query DAG representing the given set of queries. Suppose the only transformations possible are join associativity and commutativity. Then the plans $A \bowtie (B \bowtie C)$ and $(A \bowtie C) \bowtie B$, as well as several plans equivalent to these modulo commutativity can be obtained by transformations on the initial AND-OR-DAG of Figure 2.2 (b). These are represented in the DAG shown in Figure 2.2 (c). We shall refer to the DAG after all transformations have been applied as the *expanded DAG*. Note that the expanded DAG has exactly one equivalence node for every subset of $\{A, B, C\}$; the node represents all ways of computing the joins of the relations in that subset.

2.4.2 Physical AND-OR DAG

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*. Physical properties of intermediate results are important, since *e.g.* if an intermediate result is sorted on a join attribute, the join cost can potentially be reduced by using a merge join. It is straightforward to refine the above AND-OR DAG representation to represent physical properties and obtain a physical AND-OR DAG¹.

2.4.3 The Volcano Search Algorithm

The Volcano Search Engine follows a top-down, goal-driven approach. It generates the logical DAG and expands it into the physical DAG on the fly. We describe below a simplified version of the Volcano search algorithm.

Before the Volcano search algorithm is called on a query, the initial query DAG corresponding to the given query is created. Next, the initial DAG is fully expanded by applying the transformations as described earlier, to get an expanded logical AND-OR DAG. The search procedure is then called on the root of the expanded logical AND-OR DAG.²

Figure 2.3 presents the pseudo code for basic Volcano query optimization algorithm. The input to the Volcano search procedure is a logical equivalence node, an initial physical property specification and an optional cost limit.

The search procedure tries alternative physical property enforcers (such as sort) and algorithms (such as merge join) for the operation nodes below the equivalence node, recursively calling itself to find the best plan for the inputs of the operation nodes. A cost limit is passed as a parameter to the search algorithm, and if the cumulative cost of an operation node and the costs of the best plans for its inputs chosen so far exceeds the limit, the operation node can be abandoned from consideration.

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):

¹For example, an equivalence node is refined to multiple physical equivalence nodes, one per required physical property, in the physical AND-OR DAG. Enforcer operation nodes, such as sort, also get introduced.

²The description of Volcano by Graefe and McKenna [12] does not make this separation explicit, but the actual implementation does follow this two phase approach for completeness of transformations as communicated by McKenna [22].

```

Procedure FindBestPlan(LogExpr, PhysProp, Limit)
  if the pair LogExpr and PhysProp is in the lookup table
    if the cost in the lookup table < Limit
      if Plan exists then return Plan and Cost
      else return failure
    else return failure

  else /* Optimization required */
    create the set of possible "moves" from
      applicable transformations,
      algorithms that give the required PhysProp and
      enforcers for the required PhysProp

    for each move in the move set
      if the move uses a transformation
        apply the transformation creating NewLogExpr
        call FindBestPlan(NewLogExpr, PhysProp, Limit)
      else if the move uses an algorithm
        Limit = Limit - cost of the algorithm
        for each input I of the algorithm while Limit ≥ 0
          determine required physical properties PP for I
          Cost = FindBestPlan(I, PP, Limit)
          Limit = Limit - Cost
      else /* Move uses an enforcer */
        Limit = Limit - cost of enforcer
        modify PhysProp for enforced property
        call FindBestPlan for LogExpr with modified PhysProp

    /* Maintain the lookup table of explored facts */
    if LogExpr is not in the lookup table
      insert LogExpr into the lookup table
    insert PhysProp and best plan found into lookup table

```

Figure 2.3: Volcano Search Algorithm

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

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

Once the best plan for an (*equivalence node*, *physical property*) pair is found, it is stored in case it needs to be reused. Therefore, in fact, the first thing to check before performing the above optimization for a given node and a given physical property is to check if a plan for that (*equivalence node*, *physical property*) pair already exists. If a plan matching the property specification is found among the plans stored at the equivalence node, and the plan satisfies the cost limit, the plan is returned; if a plan is found but does not satisfy the cost limit a failure indication is returned. If there is no plan for the expression and the property specification, then actual optimization (as described above) starts.

The best plan for a logical equivalence node, physical property pair (thus, a physical equivalence node) is compactly specified by merely noting the corresponding physical operation node, and its input physical equivalence nodes. The overall best plan is reconstructed when required by recursively looking up the best plan for the inputs.

Chapter 3

Parametric Query Optimization for Linear Cost Functions

This chapter¹ provides two novel solutions for the parametric query optimization problem, for the case when the plan cost functions are linear in the parameters. The algorithms work for an arbitrary number of parameters and are non-intrusive in the sense that they sit on top of any conventional query optimizer and invoke it multiple times 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 by Ganguly [8], currently unpublished, that handles an arbitrary number of parameters; we describe the connection in Section 2.2. The solution proposed by Ganguly [7] works for up to two parameters. Our solutions are simple and efficient (with an asymptotic cost close to the lower bound in the case of the cost polytope algorithm), unlike earlier solutions to the PQO problem. We propose two algorithms:

- The first algorithm, *recursive decomposition algorithm*, is based on an 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 optimize the vertices of the polytope regions and decompose the regions based on the plans returned at the vertices. A polytope region is not decomposed further when all its vertices have the same optimal plan. This solution has two shortcomings: (a) It may form more than one region for a plan in the parameter space and may need to merge them in a post-pass. (b) The number of calls made to the conventional optimizer may be more than necessary. This solution forms the basis for the second solution which is more efficient.
- The second solution, *cost polytope algorithm*, seeks to remedy the problems with the first solution. It is based on an online polytope construction algorithm. It works in R^{n+1} space – where n is the number of parameters – with n dimensions for n parameters and one dimension representing cost. The cost function of each plan in the plan space can be represented by a hyperplane in this space. 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. We make use of an online polytope construction algorithm, by Mulmuley [25], which constructs a polytope by intersecting the constituent halfspaces in an online fashion. The hyperplanes corresponding to the plans in the *POSP* are the bounding hyperplanes of the polytope and the rest of the hyperplanes are redundant. And the projection of a facet on the parameter space gives the region of optimality for the corresponding plan. We measure the cost of the algorithm in terms of the number of calls made to the conventional optimizer with different parameter values and prove that, under certain assumptions, the number of calls made are very close to the lower bound.

¹Parts of this chapter appeared in VLDB 2002 [15].

The solutions proposed by Ganguly [7, 8] for linear cost functions need the conventional optimizer to return all optimal plans at a given point in the parameter space. No optimizer to our knowledge supports this and the number of such plans may be quite large. Neither of our algorithms needs the optimizer to return all the optimal plans at a given point and any one optimal plan is enough, as long as the cost function is available. This fits well with the existing optimizers.

Though the cost functions are seldom linear in real life and the results presented in this chapter are theoretical, we use the results later in Chapter 5 where we develop AniPQO – a heuristic PQO solution for the case when the cost functions are nonlinear and discontinuous.

The rest of the chapter is organized as follows. In Section 3.2 we reviews some basic properties of the linear cost functions and in Section 3.1 we state the assumptions that we make. Section 3.3 describes the recursive decomposition algorithm and Section 3.4 describes the cost polytope algorithm. We conclude in Section 3.8.

3.1 Assumptions

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 of the given plan at $n + 1$ (non-degenerate²) 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 \mathcal{R}^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*, and 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.

The solutions proposed will not work if the parameter space of interest is an open polytope as discussed in Section 3.7. But, in practice, we can use a huge symbolic box approaching infinity, as defined by Mulmuley [25], as the parameter space of interest.

3.2 Properties of Linear Cost Functions

We state the following properties regarding linear cost functions by Ganguly [7]. We prove them in Appendix C. (They appear in [7] without proofs.)

Property 3.2.1 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. □

Property 3.2.2 Each plan in a *POSP* has only one region of optimality and, the region is a convex polytope. □

Property 3.2.3 If all the vertices of a polytope in the parameter space have the same optimal plan then the plan is optimal within that polytope. □

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

We state the following properties (their proofs are presented in Section 3.4.2):

Lemma 3.2.1 *For linear cost functions, the decomposition of the parameter space induced by any *POSP* is the same.* □

²The points are not contained in a common hyperplane.

```

INITIALIZE PHASE:
  r = complete parameter space      /* Initial partition, assumed to be a convex polytope */
  for each vertex v of r do
    v.IsOptimized = FALSE          /* Not optimized */
  for any one vertex v of r do
    r.OptPlan = v.OptPlan = ConventionalOptimizer(v) /* One of the optimal plans */
    v.IsOptimized = TRUE           /* Optimized */

DECOMPOSE PHASE:
  Partition(r)

MERGE PHASE:
  If there are multiple regions having the same optimal plan /* They will be neighbors */
  Merge them all into a single region /* resulting region will be a convex polytope */

```

Figure 3.1: The recursive decomposition algorithm

Lemma 3.2.2 *For linear cost functions, the POSP is unique if no two plans have the same cost function.* \square

Without loss of generality, we assume that the POSP is unique.

3.3 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. In the solution, we recursively decompose the parameter space into convex polytopes. We optimize the vertices of the polytope regions and partition the regions based on the plans returned at the vertices. A polytope region is not partitioned further when all its vertices have the same optimal plan. We use the terms region, partition and polytope interchangeably.

Figure 3.1 shows the pseudo code of the algorithm. It is divided into three phases: initialize, decompose and merge. In the initialize phase, we define a partitioning scheme wherein the parameter space polytope itself is the only partition. One of the vertices of the polytope is optimized and an optimal plan returned at that point is set as a tentative optimal plan of the partition and the partition becomes a tentative region of optimality for that plan. In the decompose phase, we call function *Partition* on the initial partition. The function is shown in Figure 3.2. It recursively decomposes a given partition.

The function *Partition* either returns without further partitioning the input region, or partitions the input region into two partitions and calls itself recursively on each of them. We can divide the vertices of a polytope into two sets: optimized and non-optimized vertices. When we consider a region for partitioning, if all the vertices are optimized and the tentative optimal plan is optimal at all its vertices we stop further decomposition of the region. Else, if the tentative optimal plan of the region is optimal at all the optimized vertices we optimize a non-optimized vertex. Else we have a case where the tentative optimal plan of the region is not optimal at all the optimized vertices. Note that the conventional optimizer returns only one of the optimal plans at a given point in the parameter space. So, even if the plans returned by the optimizer are different, it is possible that the vertices share a common optimal plan. We evaluate the cost of the optimal plan of the region at each optimized vertex. If, at each optimized vertex, the tentative optimal plan of the region costs the same as the optimal plan at that vertex we replace the optimal plan for each optimized vertex by the tentative optimal plan of the region and proceed. If there is a optimized vertex, say v , at which the optimal plan of the region costs more than the optimal plan at that vertex, we need to either change the optimal plan of the region or partition the region. We evaluate the cost of the optimal plan of vertex v at all the optimized vertices. If the plan is as good as the optimal plan of the region at all the optimized vertices, we replace the tentative optimal plan of the region by the optimal plan at vertex v and proceed. Else, we need to partition the region.

```

function Partition(Region r)
  p' = r.OptPlan
  while there is an optimized vertex v of r with cost of v.OptPlan at v < cost of p' at v
    or there is a non-optimized vertex v of r
    if v is not optimized
      v.OptPlan = ConventionalOptimizer(v)
      v.IsOptimized = TRUE
    p = v.OptPlan
    if p == p' or costs of p and p' at v are the same
      continue
    else if p is optimal at all the optimized vertices of r          /* apart from p' */
      r.OptPlan = p = p'
    else /* need to decompose r */
      Divide r into two regions r1 and r2 with the dividing
      hyperplane defined by equating the cost functions of p' and p
      r1.OptPlan = p and r2.OptPlan = p'          /* v ∈ r1 */
      Mark all new vertices created as not optimized
      Partition(r1); Partition(r2)
    return

```

Figure 3.2: The *Partition* routine

Partitioning a region: Consider a region r with optimal plan p' and a vertex v with optimal plan p ($\neq p'$) where, at v , cost of p is less than that of p' . We need to decide the relative optimality of the plans p and p' in the region and partition the region accordingly. The dividing hyperplane is derived by equating the cost functions of the plans. In other words the dividing hyperplane is the region wherein, at each point, the costs of the plans are the same. Let the hyperplane divides region r into two regions r_1 and r_2 . In region r_1 , let p be better than p' . Then in region r_2 , p' will be better than p . We can see that p' can be optimal at no point in r_1 as p is better than it and likewise, p can be optimal at no point in r_2 as p' is better than it. Thus we have divided the region into two partitions, one is the probable region of optimality (within r) for p and the other one for p' . We recursively consider both of them for further decomposition.

At the end of the decompose phase, we get a partitioning scheme of the parameter space where each partition is a convex polytope with all its vertices having the same optimal plan; and this is the optimal plan for that entire partition (by the third property of the linear cost functions from Section 3.2). Note that the phase may generate more than one optimal region for a plan in *POSP*. But we know that there can be only one optimal region for each plan in *POSP* (by the second property of the linear cost functions from Section 3.2). Thus the multiple regions generated for a plan must be adjacent and when merged will form a convex polytope. The merge phase in Figure 3.1 carries out the job of finding and merging such regions.

The recursive decomposition algorithm has two shortcomings: It may form more than one region for a plan and may need to merge them in a post-pass³; 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 point⁴. 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, the cost polytope algorithm, is an outcome of this observation.

³If one or more of the optimal plans have multiple regions of optimality, picking the appropriate plan at run-time may prove expensive. A polytope is represented either (a) by the set of its bounding hyperplanes or, (b) by the set of its vertices. In the former case, merging polytopes involves deciding which hyperplanes of the constituent polytopes are the bounding hyperplanes of the resulting polytope and this is straightforward. But in the latter case merging involves deciding which vertices of the constituent polytopes are the vertices of the resulting polytope and this may prove expensive in higher dimensions.

⁴This may not be the case, though, if more than one plan is optimal at the point; and in that case, the point may lie on the boundary of the optimality regions of the plans.

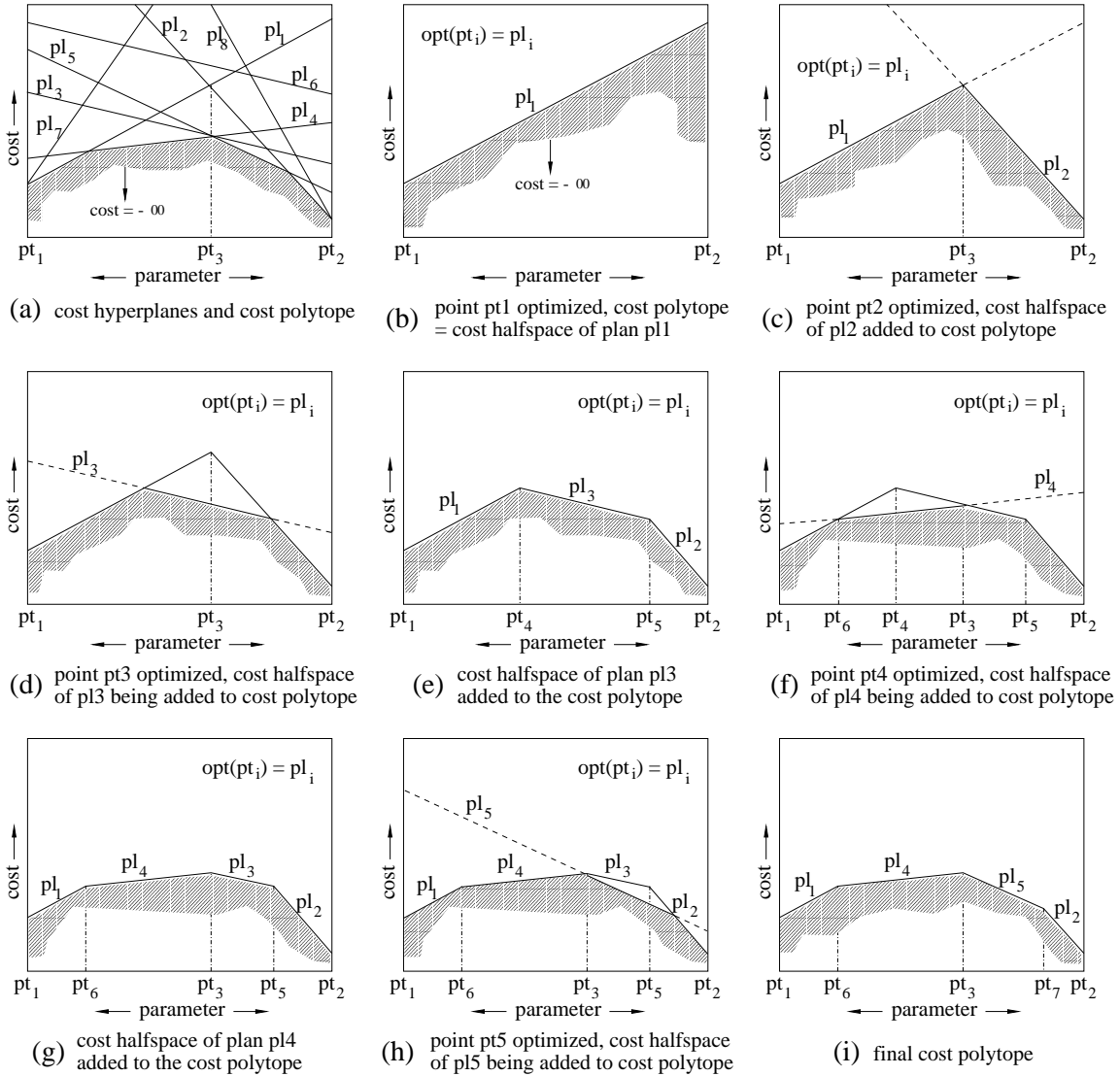


Figure 3.3: Cost Polytope Algorithm: One parameter example

Though we have not come up with upper bounds on the number of calls the recursive decomposition algorithm makes to the optimizer, and on the number of parameter space regions that it creates, as we discuss in Section 3.7, it makes at least as many calls as and creates at least as many regions as, the cost polytope algorithm.

3.4 The Cost Polytope Algorithm

The cost polytope algorithm works in the \mathbb{R}^{n+1} space with n dimensions representing n parameters and one dimension representing cost. The cost function of each plan in the plan space can be represented by a hyperplane in \mathbb{R}^{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 (\mathbb{R}^n).

We use a running example with one parameter, shown in Figure 3.3, throughout the section.

3.4.1 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, $POCF(v) = \text{cost of a plan } p \text{ that is optimal at } v$.

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 3.3 (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, \dots, pl_8 with cost functions as shown in Figure 3.3 (a). We have $POSP = \{pl_1, pl_2, pl_4, pl_5\}$. Figure 3.3 (i) shows the *POCF*, and the region of optimality of each plan in the *POSP*.

3.4.2 Cost Hyperplane, Cost Halfspace and Cost Polytope

Consider \mathfrak{R}^{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 $c_1s_1 + c_2s_2 + \dots + c_ns_n + c_{n+1}$. We can think of the cost function as a hyperplane in \mathfrak{R}^{n+1} whose equation is given by

$$s_{n+1} = c_1s_1 + c_2s_2 + \dots + c_ns_n + c_{n+1}$$

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 slope⁵. Figure 3.3 (a) shows cost hyperplanes for the plans pl_1, pl_2, \dots, pl_8 in the parameter range $pt_1 - pt_2$.

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

$$s_{n+1} \leq c_1s_1 + c_2s_2 + \dots + c_ns_n + c_{n+1}$$

We call such a halfspace a *cost halfspace*.

We represent each plan p in the plan space by its cost hyperplane in \mathfrak{R}^{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 3.3 (i) shows the cost polytope for the cost hyperplanes defined in Figure 3.3 (a). We can see that its boundary is the *POCF* for the plans pl_1, pl_2, \dots, pl_8 in the parameter range $pt_1 - pt_2$.

Theorem 3.4.1 *The boundary of the cost polytope defines the POCF.*

Proof: We will divide the proof into three parts:

Part I: The cost polytope boundary defines a cost function:

Let the cost polytope be P . We need to prove that for a point $v = (v_1, v_2, \dots, v_n)$ in the parameter space, there is one and only one point $v' = (v_1, v_2, \dots, v_n, v'_{n+1})$ on the boundary of the polytope.

\exists *one part:* As we assume that there is no degenerate cost hyperplane with infinite slope, each hyperplane spans over \mathfrak{R}^n and hence there is at least one point on the cost polytope boundary defining the cost for each point in the parameter space.

\exists *only one part:* Let us assume that there is one more point $v'' = (v_1, v_2, \dots, v_n, v''_{n+1})$ on the boundary. As v' and v'' are two different points, $v'_{n+1} \neq v''_{n+1}$. W.l.o.g. let $v'_{n+1} < v''_{n+1}$.

Let $v' \in hp'$ and $v'' \in hp''$ where hp' and hp'' are bounding hyperplanes of the polytope and are defined as:

$$hp' : s_{n+1} = c'_1s_1 + c'_2s_2 + \dots + c'_ns_n + c'_{n+1}$$

$$hp'' : s_{n+1} = c''_1s_1 + c''_2s_2 + \dots + c''_ns_n + c''_{n+1}$$

with the corresponding halfspaces defined as:

$$hs' : s_{n+1} \leq c'_1s_1 + c'_2s_2 + \dots + c'_ns_n + c'_{n+1}$$

$$hs'' : s_{n+1} \leq c''_1s_1 + c''_2s_2 + \dots + c''_ns_n + c''_{n+1}$$

And, we have $P \subseteq hs' \cap hs''$.

⁵Such 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.

$$v' \in hp' \Rightarrow v'_{n+1} = c'_1 v_1 + c'_2 v_2 + \dots + c'_n v_n$$

$$v'' \in hp'' \Rightarrow v''_{n+1} = c''_1 v_1 + c''_2 v_2 + \dots + c''_n v_n$$

$$v'_{n+1} < v''_{n+1} \Rightarrow c'_1 v_1 + c'_2 v_2 + \dots + c'_n v_n < v''_{n+1} \Rightarrow v''_{n+1} \notin hs' \Rightarrow v'' \notin P; \text{ a contradiction.}$$

Part II: Let $v' = (v_1, v_2, \dots, v_n, v'_{n+1})$ be a point on the boundary of the cost polytope P . Then, v'_{n+1} is the cost of the optimal plan at point $v = (v_1, v_2, \dots, v_n)$ in the parameter space:

Let us assume that the cost of the optimal plan at point $v = (v_1, v_2, \dots, v_n)$ is v''_{n+1} instead of v'_{n+1} . Thus, $v''_{n+1} < v'_{n+1}$. Then by the argument in part I, point v' can not be on the boundary of the polytope; A contradiction.

Part III: Let the cost of an optimal plan at a point $v = (v_1, v_2, \dots, v_n)$ from the parameter space be v'_{n+1} . Then the point $v' = (v_1, v_2, \dots, v_n, v'_{n+1})$ lies on the boundary of the cost polytope P :

From part I, we know that there is a unique point $v'' = (v_1, v_2, \dots, v_n, v''_{n+1})$ on the cost polytope. Let $v''_{n+1} \neq v'_{n+1}$. As v'_{n+1} is the optimal cost at v , we have $v'_{n+1} < v''_{n+1}$. Then by argument in part I, $v'' \notin P$; A contradiction. \square

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. Consider a cost hyperplane of a plan $p \notin POSP$. There is at least one point in the parameter space polytope at which we have a plan $p' \in POSP$ with its hyperplane having cost value lower than that of p . Thus, the hyperplane corresponding to p will not be bounding the polytope at any point and hence is redundant. Thus, the cost hyperplanes corresponding to the plans not in the $POSP$ can not form any facet of the cost polytope⁶. 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.

Recall Lemmas 3.2.1 and 3.2.2 stated in Section 3.2 without proofs; we now prove the lemmas.

Proof of Lemma 3.2.1: The lemma states that for linear cost functions, the decomposition of the parameter space induced by any $POSP$ is the same.

For a given plan space, the cost polytope is the intersection of the corresponding cost hyperplanes and hence is unique. The parameter space decomposition too is unique as it is obtained by projecting the facets of the cost polytope on the parameter space. \square

Proof of Lemma 3.2.2: The lemma states that for linear cost functions, the $POSP$ is unique if no two plans have the same cost function.

As no two plans have the same cost function, no two cost hyperplanes are the same and hence the set of bounding hyperplanes of the cost polytope is unique and, hence, so is the $POSP$. \square

3.4.3 Algorithm for 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 $POSP$, as communicated by Ganguly [9]. 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 on the parameter space is v . We use this property to avoid enumerating all the cost hyperplanes.

Our algorithm uses an online polytope construction algorithm such as that by Mulmuley [25], 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 3.4 shows pseudo code for the Cost Polytope algorithm. We transfer the equations of all bounding hyperplanes of the parameter space polytope to \mathbb{R}^{n+1} space and get corresponding hyperplanes

⁶The 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.*, a 1-face, *i.e.*, an edge, for $n \geq 2$), but can not form a facet (*i.e.*, a n -face) in \mathbb{R}^{n+1} .

```

PSpacePTope = parameter space polytope /* See Section 3.2; the polytope is in  $\mathfrak{R}^n$  */
PSPTHalfspaces = Halfspaces defining PSpacePTope in  $\mathfrak{R}^n$ 
Let  $v = (v_1, v_2, \dots, v_n)$  be any vertex of PSpacePTope
 $p = \text{ConventionalOptimizer}(v)$  /*  $p$  is one of the optimal plans at  $v$  */
VerticesOptimized = { $v$ }
 $v_{n+1} = \text{cost of } p \text{ at } v$ 
Let  $v' = (v_1, v_2, \dots, v_n, v_{n+1})$ 
Let  $h_{s_p}$  be the cost halfspace of plan  $p$  in  $\mathfrak{R}^{n+1}$ 
CostPolytope = intersection of all PSPTHalfspaces and  $h_{s_p}$  in  $\mathfrak{R}^{n+1}$  /* Initial cost polytope */
Queue = {vertices of CostPolytope} \  $v'$ 
While Queue  $\neq \emptyset$  do
     $v' = \text{Queue.RemoveFirstEntry}()$ 
    Let the coordinates of  $v'$  be  $(v_1, v_2, \dots, v_n, v_{n+1})$ 
    Let  $v = (v_1, v_2, \dots, v_n)$  /* projection of  $v'$  on parameter space */
     $p = \text{ConventionalOptimizer}(v)$  /*  $p$  is one of the optimal plans at  $v$  */
    VerticesOptimized = VerticesOptimized  $\cup$  { $v$ }
    Let  $h_{s_p}$  be the cost halfspace of plan  $p$ 
    If (cost of  $p$  at  $v$ ) <  $v_{n+1}$  /*  $v'$  is in conflict with  $h_{s_p}$  */
        CostPolytope = CostPolytope  $\cap$   $h_{s_p}$ 
        Remove from Queue vertices no longer in CostPolytope
        For each new vertex  $w' = (w_1, \dots, w_n, w_{n+1})$  added to CostPolytope
            if  $w = (w_1, \dots, w_n) \notin \text{VerticesOptimized}$ 
                add  $w'$  to Queue

```

Figure 3.4: The Cost Polytope Algorithm

in \mathfrak{R}^{n+1} space. The polytope defined by these hyperplanes in \mathfrak{R}^{n+1} space is open in directions $v_{n+1} \rightarrow \infty$ and $v_{n+1} \rightarrow -\infty$. We optimize any one vertex, say v , of the parameter space polytope to get an optimal plan at it and the corresponding cost halfspace in \mathfrak{R}^{n+1} . We intersect the polytope defined above with this halfspace to get a lower convex polytope (extending to ∞ in negative x_{n+1} direction) and this is our initial cost polytope.

We then put all vertices, except v , of the initial cost polytope in a queue. In Section 3.4.4 we elaborate below on the need of maintaining a queue and not a set. 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 latter 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, all the vertices of the polytope are optimized and we terminate the algorithm. When this condition is reached, the cost hyperplane of an optimal plan at each vertex is either a facet of the cost polytope or is touching the cost polytope. The resultant cost polytope is the final cost polytope. The plans corresponding to the facets of the final cost polytope form the *POSP* and the cost hyperplanes of the rest of the plans are redundant.

An online algorithm for polytope construction is presented by Mulmuley [25]. The algorithm allows a sequence of half spaces to be intersected, resulting in a (possibly) new polytope after each intersection. The algorithm in [25] uses *conflict* and *history* structures to identify a *conflicting vertex*⁷ 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

⁷Consider 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.* $v \in \overline{S}$.

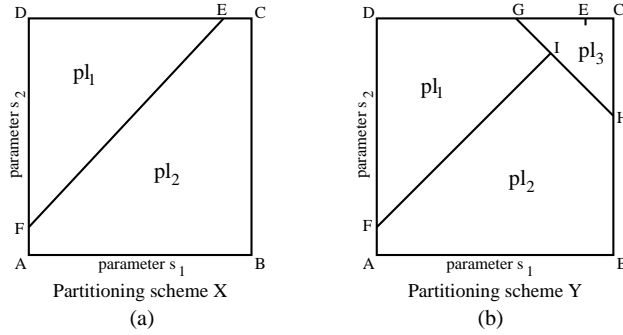


Figure 3.5: Ordering of vertices to be optimized: An example

construction algorithm for our application by eliminating the conflict and history structures. Details as to how to adapt the original algorithm to compute the cost polytope appear in Appendix B.2.

Theorem 3.4.2 *The cost polytope algorithm correctly computes the cost polytope.*

Proof: We divide the proof into two parts:

Part I: *The algorithm, if terminates, computes the cost polytope:* A subset of the cost hyperplanes is the set of bounding hyperplanes of the polytope built by the algorithm. When the algorithm terminates, no vertex of the polytope is in conflict with any of the cost hyperplanes. Thus, the polytope is exactly the intersection of all the cost hyperplanes [25] and hence is the cost polytope.

Part II: *The algorithm terminates:* As can be seen from Theorem 3.4.4 (presented later), the algorithm makes a finite number of calls to the optimizer and hence it terminates. \square

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 3.3, point pt_3 in the parameter space appears twice: in Figure 3.3 (d) it is optimized and vanishes from the decomposition (*i.e.*, is no longer a vertex of the decomposition), and in Figure 3.3 (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.

3.4.4 Ordering of the vertices to be optimized

To avoid redundant calls to the conventional optimizer, the cost polytope algorithm needs to optimize all the vertices of the parameter space polytope before optimizing any other point in the parameter space.

An example in Figure 3.5 illustrates this. The figure shows a rectangle $ABCD$, the parameter space polytope, and its intermediate decompositions during an execution of the algorithm.

Figure 3.5 (a) shows a decomposition (partitioning scheme X) wherein the parameter space is divided into two regions with plans pl_1 and pl_2 being (relatively) optimal in the resp. regions. Points A, B, D and F are optimized and points C and E are to be optimized. Plan pl_1 is optimal at points D and F whereas plan pl_2 is optimal at points A, B and F .

Figure 3.5 (b) shows a new decomposition (partitioning scheme Y) of the parameter space when either point C or point E is optimized. The optimizer returns plan pl_3 and a region for it is created in the parameter space. Note that we arrive at the same decomposition – that in Figure 3.5 (b) – by optimizing either point C or point E . Which point to optimize first?

Point C is a vertex of the rectangle $ABCD$ – the parameter space polytope in this example. And, hence, point C is also one of the vertices defining the final parameter space decomposition. By Lemma 3.4.1 we need to optimize point C anyway. If we choose to optimize point C then, as seen from Figure 3.5 (b), point E disappears and we need not optimize it. Instead, if we choose to optimize point E , we reach the decomposition in Figure 3.5 (b); but subsequently we need to optimize point C also.

Thus the algorithm should optimize all the vertices of the parameter space polytope before optimizing any other point in the parameter space. (Note that vertex E in the example need not lie on the boundary of the parameter space polytope and may lie in the interior.) Hence the algorithm maintains a queue of the points to be optimized and all the vertices of the parameter space polytope are loaded into the queue before the first iteration of the algorithm.

Note that there is no need of any order within the vertices of the parameter space polytope and within the rest of the vertices; and, as proved in Theorem 3.4.4, the upper bound on the number of calls to the optimizer is independent of the order in which the vertices within each set are optimized. We may do away with the queue by maintaining two separate sets of vertices to be optimized – one for the former and one for the latter and by optimizing all the vertices in the former before any in the latter. But the queue simplifies the pseudo-code of the algorithm.

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

We measure the complexity in terms of the number of calls to the optimizer as done by Ganguly [8]. 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.

Lemma 3.4.1 *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: Consider a decomposition of the parameter space with each region having an optimal plan defined for it. Consider one of the vertices, say p , defining the decomposition. Let the set of plans optimal in the regions adjacent to vertex p be $POSP'$. We divide the proof into two parts:

Sufficiency: If a plan is optimal at all the vertices of a polytope in the parameter space then the plan is optimal within that polytope. If each plan in $POSP'$ is optimal at vertex p and this is true for all vertices defining the decomposition then the parameter space decomposition is optimal. To ascertain this, it is sufficient to make v calls, one for each vertex, to the optimizer.

Necessity: Let all v vertices, except vertex p be optimized. If the parameter space decomposition is optimal then each plan in $POSP'$ is optimal at vertex p . But as we have not optimized p we do not know if $POSP'$ satisfies this condition. In fact, we can come up with a hypothetical plan, cost of which at p is less than the cost of each plan in $POSP'$. (And such a plan will then have its region of optimality carved around point p .) \square

The theorem below follows directly from Lemma 3.4.1.

Theorem 3.4.3 *The lower bound on the number of calls to be made to the optimizer to solve the PQO problem for linear cost functions in a non-intrusive manner is v .* \square

Lemma 3.4.2 *No vertex of an intermediate cost polytope is in the interior of any i -face of the final cost polytope, with $i > 0$.*

Proof: Any intermediate cost polytope is a superset of the final cost polytope and, as the algorithm progresses, the polytope is intersected with new hyperplanes and it shrinks. 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 progresses⁸. Hence the proof. \square

⁸A face of the polytope may reduce in dimension but it never disappears; it may become a 0-face, *i.e.* it may shrink to a vertex of the polytope.

Each point that is optimized is a vertex of either an intermediate or the final cost polytope and, thus, no point that is optimized lies in the interior of any i -face of the final cost polytope, with $i > 0$.

Theorem 3.4.4 *The cost polytope algorithm makes at most F calls to the optimizer.*

Proof: 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 cost polytope, or has to be in the interior of exactly one of the i -faces of the final cost polytope, for some $i > 0$. In the former case, we say that v' maps to v , a vertex of the final cost polytope. In the latter case, let the face be h and we say that v' maps to face h .

We claim that no two optimized vertices map to the same i -face, with $i > 0$, of the final cost polytope. If not, since vertices are optimized in some order, *w.l.o.g.*, let w' be an intermediate cost 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 the polytope with the hyperplane corresponding to face h , we have $w = w'$. But then w' is a vertex of an intermediate cost polytope, and, by Lemma 3.4.2, it can not be in the interior of face h .

Thus each vertex optimized maps either to a different i -face, with $i > 0$ or to a vertex of the final cost polytope, and no point is optimized more than once, leading to the upper bound of F calls. \square

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 $O(n^{\lfloor d/2 \rfloor})$, 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 when:

- For each face of the final cost polytope, we have a cost hyperplane such that its intersection with the cost polytope is the face itself. That is, an i -face is contained in a cost hyperplane but none of the $(i + 1)$ -faces that this i -face is on the boundary of is contained in the cost hyperplane. Let us call such a cost hyperplane a *touching cost hyperplane* of the face.
- And, the algorithm detects one (and only one, by Theorem 3.4.4) touching cost hyperplane for each face of the final cost polytope.

By the argument used in proving Theorem 3.4.4, once we detect a touching cost hyperplane of a face of the final cost polytope, touching hyperplanes of its sub-faces may not be detected. Thus, when the worst case occurs, a touching cost hyperplane of an i -face is detected only after touching cost hyperplanes, one each, of all the $(i - 1)$ -faces on the boundary of the i -face are detected.

If the coefficient vectors⁹ 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; Thus, the expected number of calls to the conventional optimizer is only f . In high dimensional non-degenerate cases, the number of vertices of a polytope are generally exponential in the number of facets (Ziegler [40]), 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.

Closing the gap between the lower bound on the number of calls and the number of calls made by the cost polytope algorithm is an open problem.

3.4.6 An example with single parameter

Figures 3.3 (b)-(h) show iterations of the algorithm for the example with one parameter from Figure 3.3 (a). The algorithm optimizes the points pt_1, pt_2, pt_3, pt_4 and pt_5 in that order and creates the final cost polytope as shown in Figure 3.3 (i).

We make two observations here: (a) A vertex pt_3 of the intermediate cost polytope created in Figure 3.3 (c) disappeared in 3.3 (e) and resurfaced in Figure 3.3 (g). If we remember that this point was

⁹The coefficient vector of a cost function $c_1 s_1 + c_2 s_2 + \dots + c_n s_n + c_{n+1}$ is $(c_1, c_2, \dots, c_{n+1})$.

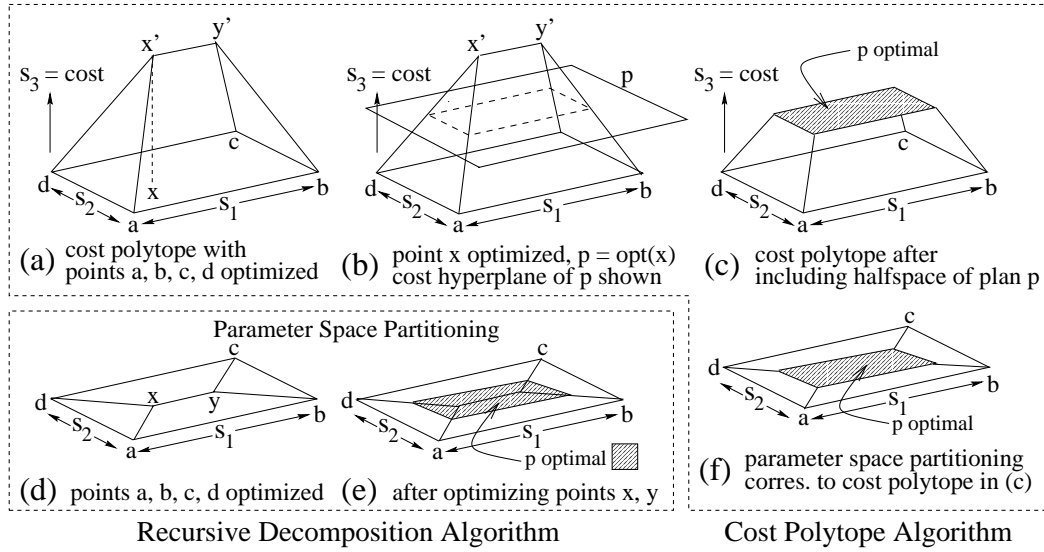


Figure 3.6: Comparison of two algorithms for linear cost functions

encountered before and hence optimized, we can avoid re-optimizing the point. To do so, we need to maintain a list of all the points in the parameter space that are optimized. Note that we need not remember the cost hyperplane of an optimal plan at a point, as it has already been intersected with the cost polytope. (b) We assume that the optimizer returns plan pl_1 at pt_1 and pl_2 at pt_2 . Instead, if the optimizer returns, resp., plans pl_7 and pl_8 , we will incur two extra calls to the optimizer to detect plans pl_1 and pl_2 which are in the $POSP$. We incur maximum calls to the optimizer – 9 calls – if plan pl_7 is returned at point pt_1 , plan pl_8 is returned at point pt_2 and plan pl_3 is returned at point pt_3 .

3.4.7 Initial cost polytope

We may use a huge symbolic box approaching infinity, as defined by Mulmuley [25], as an initial cost polytope. In general we are not interested in the whole parameter space \mathbb{R}^n as only a part of it would constitute legal combinations of the parameter values. We assume that we are interested only in a convex polytope in the parameter space and we need to find the $POCF$ only in this polytope. When we transfer the equations of all bounding hyperplanes of this polytope to \mathbb{R}^{n+1} space, we get corresponding hyperplanes in \mathbb{R}^{n+1} space. The object defined by these hyperplanes in \mathbb{R}^{n+1} space is open in directions $v_{n+1} \rightarrow \infty$ and $v_{n+1} \rightarrow -\infty$. We optimize any one vertex of the parameter space polytope to get a optimal plan at it and the corresponding cost halfspace in \mathbb{R}^{n+1} . We intersect the object defined above with this halfspace to get a lower convex polytope (extending to ∞ in negative x_{n+1} direction) and this is our initial cost polytope.

Generally each parameter will have a range of legal values. Thus the parameter space polytope would be a hypercube defined by the lower and higher legal value for each parameter.

3.4.8 Comparing the algorithms

We compare the recursive decomposition algorithm and the cost polytope algorithm using an example with two parameters (s_1 and s_2) shown in Figure 3.6. Let an intermediate cost polytope be as shown in Figure 3.6 (a). The corresponding partitioning of the parameter space is shown in Figure 3.6 (d). At this stage, let points a, b, c and d be optimized and points x and y to be optimized. Thus points x and y are in the queue of to be optimized vertices in the cost polytope algorithm. The recursive decomposition algorithm will optimize both x and y . Let the optimal plan at both of them be p with the cost function as shown in Figure 3.6 (b). The recursive decomposition algorithm will further partition the parameter space as shown in Figure 3.6 (e). Note that, all the hatched regions in Figure 3.6 (e) have the same optimal plan p . The cost

polytope algorithm will optimize one of the vertices (x or y) and will intersect the cost polytope with the cost halfspace of plan p . Both the vertices will be removed from the polytope. The resulting cost polytope and corresponding parameter space partitioning are shown in Figures 3.6 (c) and (f) respectively. Thus the cost polytope algorithm calls the optimizer at only one of the vertices (x or y) and it creates only one region of optimality for plan p , whereas the recursive decomposition algorithm calls the optimizer at both the vertices and creates more than one optimal regions for plan p .

3.4.9 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, Ganguly [7, 8] makes 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 3.3 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 us call the region of optimality of a plan an *exterior region of optimality* if it shares a vertex with the parameter space polytope and an *interior region of optimality* if it shares no vertex with the parameter space polytope. Analogously, let us call a facet of the final cost polytope an *exterior facet* if the corresponding optimality region is an exterior region of optimality and an *interior facet* if the corresponding optimality region is an interior region of optimality. There is a one-to-one correspondence between the exterior regions of optimality and the exterior facets of the cost polytope and, there is a one-to-one correspondence between the interior regions of optimality and the interior facets of the cost polytope.

The reduction in the number of calls is based on the following observations: (a) When we optimize the vertices of the parameter space polytope, we detect all the exterior facets of the cost polytope and, thus, no separate calls are needed to detect the exterior facets. (b) 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.

In the following theorem, f' denotes the number of plans in the *POSP* whose regions of optimality are the interior regions.

Theorem 3.4.5 *If the optimizer returns all optimal plans at a given point in the parameter space, the cost polytope algorithm makes at most $v + f'$ calls to the optimizer.*

Proof: 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) a vertex of the final cost polytope, or (b) in the interior or on the boundary of a facet, but not a vertex, of the final cost polytope; *i.e.* v is in the interior of the facet or in the interior of one of its i -faces, with $i > 0$. In the former case, we say that v' maps to v , a vertex of the final cost polytope. In the latter case, let v lie in the interior or on the boundary¹⁰ of facet h of the final cost polytope; we say that vertex v' maps to facet h .

We claim that no optimized vertex maps to any exterior facet. Since the vertices of the parameter space polytope are optimized first, we have already intersected the polytope with all the hyperplanes corresponding the exterior facets¹¹. Thus, if h is an exterior facet then, $v = v'$. As v is not a vertex of facet h , it lies in the interior of either facet h or one of its i -faces, with $i > 0$. But then v' is a vertex of an intermediate cost polytope, and, by Lemma 3.4.2, it can not be in the interior of any face of the final cost polytope.

We claim that no two optimized vertices map to the same facet of the final cost polytope. If not, since vertices are optimized in some order, *w.l.o.g.*, let w' be an intermediate cost polytope vertex that is optimized

¹⁰If v lies on the boundary of a facet then it may lie on the boundary of more than one facet; h is any one of them.

¹¹This is the first of the two accounts on which Theorem 3.4.4 differs from this theorem; in Theorem 3.4.4, optimizing the vertices of the parameter space polytope may not detect all the exterior facets of the final cost polytope.

after v' , and w be the point in the interior or on the boundary of facet h whose projection on the parameter space is the same as that of w' . But since we have already intersected¹² the polytope with the hyperplane corresponding to facet h , we have $w = w'$. As w is not a vertex of facet h , it lies in the interior of either facet h or one of its i -faces, with $i > 0$. But then w' is a vertex of an intermediate cost polytope, and, by Lemma 3.4.2, it can not be in the interior of any face of the final cost polytope.

Thus each vertex optimized maps either to a different interior facet or to a vertex of the final cost polytope, and no point is optimized more than once, leading to the upper bound of $v + f'$ calls. \square

Since $f' < f < v$, 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 of the parameter space polytope before optimizing any other point, as is done in the algorithm (see Section 3.4.4).

3.5 Comparison with Ganguly's solution

Ganguly's algorithm for parametric query optimization with linear cost functions [8] calls itself recursively on the faces of the parameter space polytope, moving from lower to higher dimensional faces. When the algorithm works on a face, it has already worked on all its sub-faces and the set of optimal plans along the boundary of the face is the union of the sets of optimal plans of all the sub-faces. The algorithm constructs the decomposition of the face induced by the set of optimal plans along the boundary. The decomposition may create some vertices in the interior of the face; these vertices are optimized one by one. If optimization of a vertex returns a new plan, the optimal region for the plan is carved out within 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 \mathfrak{R}^n , whereas our algorithm works in \mathfrak{R}^{n+1} , which results in a considerable simplification. 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. Also, given the plans optimal along the boundary, finding the optimal decomposition of the parameter space polytope induced by these plans involves the same chipping procedure (which starts with the parameter space polytope) for each plan in the set and may prove to be expensive. In contrast since we work in higher dimensional space, we only require a single hyperplane intersection. We also avoid the complexity of the recursive decomposition of the faces of the parameter space polytope.

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.

3.6 Approximate Cost Polytope Algorithm

If the cost of a plan at a point is within a small factor of the optimal plan at the point then we may treat the plan to be optimal at the point. In this section we propose to modify the cost polytope algorithm¹³ to bring down the number of calls to the optimizer. Before we outline the modification, we state the following results.

Theorem 3.6.1 *If the cost of a plan at any two points in the parameter space is within a factor t (with, $1 \leq t$) of the resp. optimal costs at the points then, at each point along the line segment connecting the two points, the cost of the plan is within factor t of the optimal cost at that point.*

Proof: Consider a line segment $x - y$; let z be a point on this segment. We have $x - z - y$ and,

$$x - z - y \Rightarrow z = ax + by \text{ where } a, b \in \mathfrak{R}, 0 \leq a, b \text{ and } a + b = 1$$

¹²This is the second of the two accounts on which Theorem 3.4.4 differs from this theorem; in Theorem 3.4.4, if v is on the boundary of and not in the interior of facet h , optimizing v' may not detect h .

¹³The recursive decomposition algorithm too can be similarly modified.

Let plan p_x , p_y and p_z be optimal resp. at points x , y and z . Let the cost of a plan p be within factor t of the resp. optimal costs at point x and y .

$$\text{cost}_{p_x}(x) \leq \text{cost}_{p_z}(x) \quad (p_x \text{ is optimal at } x) \quad (3.1)$$

$$\text{cost}_{p_y}(y) \leq \text{cost}_{p_z}(y) \quad (p_y \text{ is optimal at } y) \quad (3.2)$$

$$\text{cost}_{p_z}(ax + by) = \text{cost}_{p_z}(z) \quad (z = ax + by)$$

$$a.\text{cost}_{p_z}(x) + b.\text{cost}_{p_z}(y) = \text{cost}_{p_z}(z) \quad (\text{cost functions are linear})$$

$$a.\text{cost}_{p_x}(x) + b.\text{cost}_{p_y}(y) \leq \text{cost}_{p_z}(z) \quad (\text{from 3.1 and 3.2, as } 0 \leq a, b) \quad (3.3)$$

$$\text{cost}_p(x) \leq t.\text{cost}_{p_x}(x) \quad (\text{assumption}) \quad (3.4)$$

$$\text{cost}_p(y) \leq t.\text{cost}_{p_y}(y) \quad (\text{assumption}) \quad (3.5)$$

$$\text{cost}_p(z) = \text{cost}_p(ax + by) \quad (z = ax + by)$$

$$= a.\text{cost}_p(x) + b.\text{cost}_p(y) \quad (\text{cost functions are linear})$$

$$\leq a.t.\text{cost}_{p_x}(x) + b.t.\text{cost}_{p_y}(y) \quad (\text{from 3.4 and 3.5, as } 0 \leq a, b)$$

$$\leq t(a.\text{cost}_{p_x}(x) + b.\text{cost}_{p_y}(y))$$

$$\text{cost}_p(z) \leq t.\text{cost}_{p_z}(z) \quad (\text{from 3.3})$$

Hence the proof. \square

Corollary 3.6.1 *If the cost of a plan at all the vertices of a polytope in the parameter space is within a factor t of the resp. optimal costs at the vertices then, at each point in the polytope, the cost of the plan is within factor t of the optimal cost at that point.*

Proof: We prove this by induction on dimension d of the polytope. The corollary is proved for $d = 1$ in the above theorem. Assume it to be true for dimensions $d < n$. Consider a n -dimensional polytope P . Let the cost of a plan p be within a factor t of the optimal cost at each of its vertices. All the facets of polytope P are polytopes in lower dimensions with the cost of plan p within factor t of the optimal cost at each of their vertices. Hence by inductive hypothesis the cost of plan p is within factor t of the optimal cost at each point within each facet of polytope P . Consider a point x in polytope P . Any line passing through point x will intersect polytope P into two of its facets. Let the intersection points be y and z . We have $y - x - z$. By inductive hypotheses the cost plan p is within factor t of resp. optimal costs at points y and z . By the above theorem the cost plan p is within factor t of the optimal cost at point x also. Hence the proof. \square

We define the approximate cost polytope algorithm as follows: At each iteration, when we optimize one of the un-optimized vertices, say $v^i = (v_1, v_2, \dots, v_n, v_{n+1})$, of the intermediate cost polytope, if v_{n+1} is within a small factor t of the cost of the optimal plan then we ignore the cost hyperplane of the optimal plan and the intermediate cost polytope continues to be the same.

From the above corollary, we can see that for the case with linear cost functions, the approximate cost polytope algorithm guarantees that the cost function defined by the boundary of the final cost polytope is within factor t of the *POCF* in the parameter space polytope.

3.7 Discussion

Our solutions assume that the parameter space of interest is a closed convex polytope and will not work if it is an open polytope. For example, consider a one dimensional parameter space, with the parameter, say x , spanning from 0 to $+\infty$ – this space of interest is an open polytope in one dimension. It has only one vertex, namely $x = 0$. We optimize the vertex and get an optimal plan at it but we can not proceed further, as there are no more vertices to be optimized. But, in practice, we can define a huge symbolic box approaching infinity, as defined by Mulmuley [25], as the parameter space of interest.

The properties of the linear cost functions enumerated in Section 3.2 do not hold for nonlinear cost functions and the solutions proposed are not applicable if the cost functions are nonlinear.

3.8 Summary

In this chapter, we proposed a parametric query optimization algorithm for the case when the cost functions are linear in the given parameters. The solution works for arbitrary number of parameters and is minimally intrusive in the sense that an existing query optimizer can be used as a subroutine with minor modifications. The solution invokes a conventional query optimizer multiple times, with different parameter values. 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.

Though the cost functions are seldom linear in real life and the results presented in this chapter are theoretical, we use the results later in Chapter 5 where we develop AniPQL – a heuristic PQL solution for the case when the cost functions are nonlinear and discontinuous.

Chapter 4

Parametric Query Optimization for Piecewise Linear Cost Functions

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 be approximated by piecewise linear cost functions.

This chapter¹ proposes 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 by Selinger *et.al.* [33] and the Volcano query optimization algorithm by Graefe and McKenna [12] to perform parametric query optimization with piecewise linear cost functions. The solution works for an arbitrary number of parameters. We have implemented the extensions for the Volcano query optimization algorithm.

The rest of the chapter is organized as follows. Section 4.1 defines piecewise linear cost functions and Section 4.2 discusses how to approximate non-linear cost functions by piecewise linear cost functions. Section 4.3 presents an overview of the extensions to a conventional optimization algorithm to turn it into a parametric query optimizer for piecewise linear cost functions. Section 4.4 outlines how to extend the System R query optimization algorithm by Selinger *et.al.* [33] and Section 4.5 outlines how to extend the Volcano query optimization algorithm by Graefe and McKenna [12] to perform parametric query optimization with piecewise linear cost functions. Section 4.6 discusses implementation details. Finally, Section 4.7 summarizes the chapter.

Assumptions

As said in the previous chapter (Section 3.1), in general, we are not interested in the whole parameter space \mathcal{R}^n (where n is the number of parameters) as only a part of it would constitute legal combinations of the parameter values and, typically, the parameter space of interest is a hyper-rectangle defined by a range of legal values specified for each parameter. So, as done in the previous chapter (Section 3.1), we assume that the parameter space of interest is a closed convex polytope, which we call the *parameter space polytope*, denoted by P , and is provided to the optimizer.

4.1 Piecewise Linear Cost Functions (PLCF)

In conventional optimizers, the cost of an operator or a plan is a single quantity. But for parametric query optimization, we define the cost of an operator to be a function of the parameters it depends upon. We

¹Parts of this chapter appeared in VLDB 2002 [15].

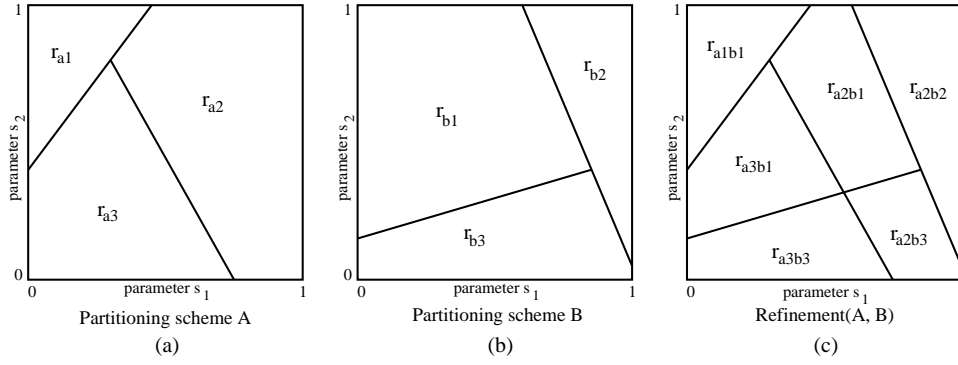


Figure 4.1: Refinement of parameter space partitioning: An example with two parameters

propose to approximate the cost functions to piecewise linear form in each parameter. The idea behind handling piecewise linear cost functions is to partition the parameter space into convex polytopes such that within each polytope, the cost is a linear function in the parameters. In this section we give some definitions related to parameter space partitioning and piecewise linear cost functions.

4.1.1 Parameter Space Partitioning Scheme

A *partitioning scheme* partitions the parameter space polytope (P) into disjoint partitions. We assume that each partition is a convex region and hence a convex polytope. Consider a partitioning scheme PS defined on the parameter space polytope (P). Let PS contain m partitions: $\{r_i : 1 \leq i \leq m\}$. Then, $P = \cup_{i=1}^m r_i$ and $\forall i, j, 1 \leq i < j \leq m : r_i \cap r_j = \emptyset$.

Figures 4.1 (a) and (b) show partitioning schemes defined on a two dimensional parameter space. Both parameters range over $[0, 1]$. The partitioning scheme in Figure 4.1 (a) creates three partitions of the parameter space, namely r_{a1}, r_{a2} and r_{a3} and, the partitioning scheme in Figure 4.1 (b) creates three partitions of the parameter space, namely r_{b1}, r_{b2} and r_{b3} . (Part (c) of Figure 4.1 is discussed later.)

4.1.2 Definition: Piecewise Linear Cost Functions (PLCF)

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.

Let the number of parameters be n ; thus the parameter space polytope (P) is a n -dimensional closed convex polytope in \mathbb{R}^n (see Section 4). Consider a piecewise linear cost function C which involves n parameters. We can create a partitioning scheme R_c for C that partitions the parameter space polytope (P) into convex partitions r_1, r_2, \dots, r_m and within each partition r_i the cost function is linear in the parameters. Let the cost function within a partition r_i be denoted by c_i and be given by:

$$c_i = c_{i,1}s_1 + c_{i,2}s_2 + \dots + c_{i,n}s_n + c_{i,n+1}$$

where $c_{i,j}$ is a constant and s_j is the value of j^{th} parameter; c_i represents a hyperplane in \mathbb{R}^{n+1} , but represents the cost function C only in partition r_i .

Let the value of the cost function C at a point s_1, s_2, \dots, s_n in the parameter space S be denoted by $C(s_1, s_2, \dots, s_n)$. It is given by:

$$C(s_1, s_2, \dots, s_n) = c_i(s_1, s_2, \dots, s_n),$$

where $(s_1, s_2, \dots, s_n) \in r_i$

It is easy to see that if we fix all the parameters except one, say s_j , the cost function will be piecewise linear in s_j . By fixing all the parameters except one, we define a line in the parameter space. In fact, if we take any line in the parameter space, the cost function will be piecewise linear along the line.

Figures 4.2 (a) and (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

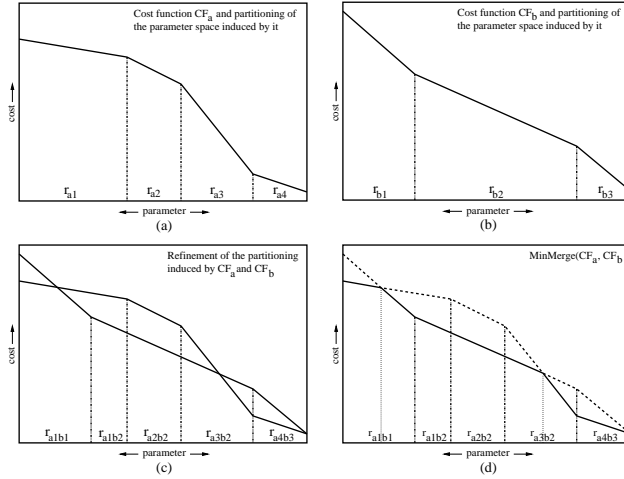


Figure 4.2: Parameter Space Partitioning, and *Refine* and *MinMerge* operations: a one-parameter example

cost function in Figure 4.2 (a) creates four partitions of the parameter space, namely r_{a1} , r_{a2} , r_{a3} and r_{a4} , whereas the cost function in Figure 4.2 (b) creates three 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 4.2 are discussed later.)

4.2 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 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 4.3 (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 4.3 (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.

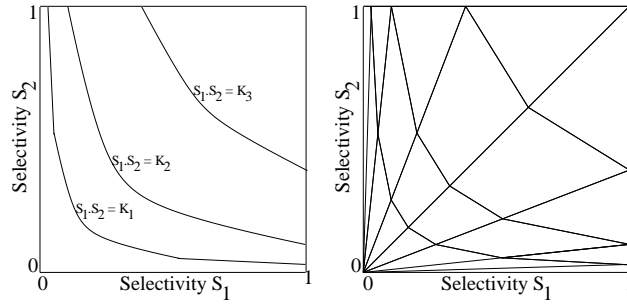


Figure 4.3: Cost function approximation: (a) Equi-cost lines (b) Corresponding decomposition

4.3 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 have a cost function associated with an operation or a plan and each point in the parameter space has a corresponding cost value; thus we need to handle cost functions in place of costs.
- For a query/sub-query expression, no single plan may be optimal in the entire parameter space and we need to keep track of multiple plans; thus we need to partition the parameter space, with a different plan optimal in each region.

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 4.2 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 procedures to *refine* partitioning schemes and to *add* piecewise linear cost functions are outlined in Section 4.3.1.

- Comparing alternative operations 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 4.3.1.

Linear vs. piecewise linear techniques: The cost polytope algorithm for the linear case described in Chapter 3 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, our algorithm for the piecewise linear case refines the space for each operation and plan separately: thus the space is refined only as much as is needed for that operation or plan.

```

Refine( $PS_a, PS_b$ )
  PartitioningScheme  $PS_o = \emptyset$ 
  for each partition  $r_{a_i}$  in  $PS_a$ 
    for each partition  $r_{b_j}$  in  $PS_b$ 
       $r_{a_i b_j} = r_{a_i} \cap r_{b_j}$ 
      if  $r_{a_i b_j} \neq \emptyset$  then  $PS_o = PS_o \cup \{r_{a_i b_j}\}$ 
  return  $PS_o$ 

```

Figure 4.4: Pseudo code for the *Refine* operation

We consider 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.

4.3.1 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.

Operation *Refine* on partitioning schemes

Figure 4.4 gives pseudo-code for the *Refine* operation. It 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.

The refined partitioning scheme has the following properties:

- A *PLCF* defined on either of the input partitioning schemes will be linear in each partitions of the refined partitioning scheme, as $r_{a_i b_j} \subseteq r_{a_i}$ and $r_{a_i b_j} \subseteq r_{b_j}$.
- If two points are in the same partition in each input partitioning scheme then they will fall in the same partition in the resulting partitioning scheme. Consider points p and q in the parameter space. Let $p, q \in r_{a_i}$ in PS_a and $p, q \in r_{b_j}$ in PS_b , then $p, q \in r_{a_i b_j}$ in PS_o as $r_{a_i b_j} = r_{a_i} \cap r_{b_j}$.
- If two points are in different partitions in either of the input partitioning schemes then they will fall in different partitions in the resulting partitioning scheme. Consider point p and q in the parameter space. Let $p \in r_{a_i}$ and $q \in r_{a_k}$ in PS_a and $p \in r_{b_j}$ and $q \in r_{b_l}$ in PS_b . Then $p \in r_{a_i b_j}$ and $q \in r_{a_k b_l}$ in PS_o . And $r_{a_i b_j} \neq r_{a_k b_l}$ if $i \neq k$ and/or $j \neq l$ as $r_{a_i} \cap r_{a_k} = \phi$ and $r_{b_j} \cap r_{b_l} = \phi$.

Figure 4.2 (c) shows the result of refinement of partitioning schemes of a one dimensional parameter space defined in Figures 4.2 (a) and (b). A partition $r_{a_i b_j}$ in Figure 4.2 (c) results from an intersection of a partition r_{a_i} from Figure 4.2 (a) and a partition r_{b_j} from Figure 4.2 (b). As an example, we see that the partition $r_{a_1 b_1}$ is created by intersecting partition r_{a_1} and r_{b_1} . The partition $r_{a_2 b_3}$ does not exist as the partitions r_{a_2} and r_{b_3} do not intersect. Figures 4.1 (a)-(c) show partitioning schemes in 2 dimensions. The partitioning scheme in Figure 4.1 (c) is the result of refinement of the partitioning schemes in Figures 4.1 (a) and (b). The output partitioning scheme can be thought of as the superposition of the two input partitioning schemes.

Operations *Add* and *Subtract* on *PLCF*

We describe only the *Add* operation here; the *Subtract* operation is similar. 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.

The *Subtract* operation is similar to the *Add* operation, except that the value of the output cost function at each point in the parameter space is obtained by subtracting the value of the second cost function from the first one at that point.

Operations *MinMerge* and *MaxMerge* on *PLCF*

We describe only *MinMerge* here; *MaxMerge* operation is similar. 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.²

Figures 4.2 (a)-(d) show an example with one parameter. Figure 4.2 (c) shows the result of refining the two partitioning schemes defined in Figures 4.2 (a) and (b). Figure 4.2 (d) shows the result of the *MinMerge* operation on the cost functions defined in Figures 4.2 (a) and (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 than 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.

The *MaxMerge* operation is similar to the *MinMerge* operation, except that it picks the maximum of the two input cost values at each point in the parameter space.

²This partitioning technique can be thought of as a specialization of the recursive decomposition algorithm (Section 3.3) to the case where only two cost functions need to be combined.

4.3.2 Cost based pruning

Cost based pruning permits pruning of plans from consideration if while enumerating a sub-plan we find that the cost is already too high (higher than that of the best plan so far, or over a pre-specified limit). In the original Volcano optimizer, if while optimizing an expression, we find that an earlier attempt failed with some cost limit, and the old cost limit is less than the new cost limit, we re-optimize the expression and replace the old cost limit either by a plan (if the attempt succeeds) or the new cost limit (if the attempt fails).

But in case of parametric query optimization, (a) cost limit is a function and (b) we may not wish to replace the old optimization information with the new one throughout the parameter space as we may lose the old information in some regions. For example, consider a point in the parameter space where no optimal plan is available and the old cost limit, say $c_{limitOld}$, is greater than the new one. If we replace the old cost limit by the new one we lose the information that “no plan with cost $\leq c_{limitOld}$ is possible”. As another example, consider a point in the parameter space where a plan p is optimal with cost, say $cost_p$, which is greater than the new cost limit. The optimizer returns failure at this point with the new cost limit and if we replace the optimal plan by the cost limit, we lose the information that “plan p is optimal at the point”

We can avoid getting into this problem by disabling cost based pruning; An optimization call will not specify any cost limit and will return successful plans throughout the parameter space. So re-optimization is never sought. This approach is simple but is expensive as it explores the whole plan space. We propose two alternatives. One option is to re-optimize only the regions where re-optimization is required and merge the new results with the old one. But the regions may be scattered across the parameter space and we need to invoke optimization for each of them individually which may prove to be expensive. A better option is to save the old optimization information, re-optimize the whole parameter space with the input cost limit and then merge the old optimization result with the new one such that no information from either of them is lost. Though the new result is the combination of the old and the new optimization invocation we need to return the optimal plans or failure indication in different regions w.r.t. the input cost limit. This alternative is considered in the parametric query optimization algorithm presented in Section 4.5.

4.4 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 4.5 shows the pseudo-code of (a recursive formulation of) the System R optimization algorithm.

Figure 4.6 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*.

4.5 Extensions to Volcano Query Optimizer

In this section we outline the details of the extensions to the Volcano optimizer.

4.5.1 Extended Cost Function and Plan

We present here some definitions and extensions that are used in parametric query optimization (PQO) algorithm.

A conventional optimizer has a single value as cost for an operator or a query plan and its corresponding (*LogExpr*, *PhysProp*) pair. Here we have a cost associated with each point in the parameter space, *i.e.* we define cost as a function of parameter values and we have one such cost function for each operator and query plan with a (*LogExpr*, *PhysProp*) pair. *Plan.CostFunction* denotes cost function of a plan,

```

Input: SPJ query  $q$  on a set of relations  $Q = \{R_1, \dots, R_n\}$ 
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 4.5: System R Algorithm

```

Input: SPJ query  $q$  on a set of relations  $Q = \{R_1, \dots, R_n\}$ 
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 4.6: Extended System R Algorithm

AlgorithmCostFunction denotes cost function of an algorithm and *EnforcerCostFunction* denotes cost function of an enforcer.

Further, for a $(LogExpr, PhysProp)$ pair, as we are considering the parameter space, more than one physical plan may be optimal, each being optimal in a particular parameter partition.³ The optimal *Plan* for a $(LogExpr, PhysProp)$ pair will contain a list of pairs. Each pair will contain a region of the parameter space and an optimal physical plan in that region.

A conventional optimizer has a single value as a cost limit. Here we have a cost limit for each parameter space point. We use *CostLimitFunction* to denote a cost limit function. We divide the parameter space into convex regions and the cost limit would be linear in each of these regions.

Additionally, in a cost function for a $(LogExpr, PhysProp)$ pair, an algorithm or an enforcer, we may have a parameter space partition where we have no optimal plan but a failure indication. In this partition, the cost function actually indicates a cost limit on the plan. The cost of the plan will be more than the cost function at each point in the partition.

4.5.2 Extensions to the search algorithm

Figure 4.7 shows the Volcano Search Algorithm *FindBestPlan* extended for parametric query optimization. To optimize a given $(LogExpr, PhysProp)$ pair within a given *costLimitFunction*, if there has been no previous attempt to optimize the $(LogExpr, PhysProp)$ pair, the search algorithm proceeds as

³This may increase the search space

```

FindBestPlan(LogExpr, PhysProp, CostLimitFunction)
  if the pair LogExpr and PhysProp is in the lookup table with Plan as the optimal plan
    /* optimized already, attempting reuse */
    if  $\exists i$ , in parameter space, at which Plan is failed
      and its cost is  $< CostLimitFunction(i)$ 
      goto Label X /* re-optimization required */
    else /* no re-optimization required */
      if  $\exists i$ , at which Plan is successful and its cost  $\leq CostLimitFunction(i)$ 
        return Plan
      else return FAILURE /*  $\forall i$ , cost of Plan at  $i > CostLimitFunction(i)$  */

  else /* Optimization required */
  Label X:
    optPlanOld = PlanGroup.OptPlan

    Plan = ApplyTransformations(LogExpr, PhysProp, CostLimitFunction)
    /* if plan returned, merge it with the planGroup optPlan */
    if Plan  $\neq$  FAILURE then
      optPlan = MinMerge(optPlan, Plan)
      CostLimitFunction = optPlan.CostFunction()

    Plan = ApplyAlgorithms(LogExpr, PhysProp, CostLimitFunction)
    /* if plan returned, merge it with the planGroup optPlan */
    if Plan  $\neq$  FAILURE then
      optPlan = MinMerge(optPlan, Plan)
      CostLimitFunction = optPlan.CostFunction()

    Plan = ApplyEnforcers(LogExpr, PhysProp, CostLimitFunction)
    /* if plan returned, merge it with the planGroup optPlan */
    if Plan  $\neq$  FAILURE then
      optPlan = MinMerge(optPlan, Plan)
      CostLimitFunction = optPlan.CostFunction()

    optPlan = MaxMerge(optPlanOld, optPlan)

    /* Maintain the lookup table of explored (expression, physical property) pairs */
    if LogExpr is not in the lookup table
      insert LogExpr into the lookup table
    insert (LogExpr, PhysProp, optPlan) into lookup table

```

Figure 4.7: Parametric Query Optimizer: *FindBestPlan* Routine

```

ApplyTransformations(LogExpr, PhysProp, CostLimitFunction)
  for each applicable transformation
    create NewLogExpr by applying the transformation
    Plan = FindBestPlan(NewLogExpr, PhysProp, CostLimitFunction)
    /* if plan returned, merge it with the planGroup optPlan */
    if Plan  $\neq$  FAILURE then
      optPlan = Plan
      CostLimitFunction = Plan.CostFunction()

  if optPlan is successful for some point in the parameter space
    return optPlan
  else return FAILURE

```

Figure 4.8: Parametric Query Optimizer: *ApplyTransformations* Algorithm

```

ApplyAlgorithms(LogExpr, PhysProp, CostLimitFunction)
  for each applicable Algorithm do
    if AlgorithmCostFunction is greater than CostLimitFunction at all points
      continue /* cost of the operator is more than the cost limit */
    AlgoPlan.CostFunction = AlgorithmCostFunction
    CostLimitFunction = CostLimitFunction - AlgorithmCostFunction

    for each input I of the algorithm while CostLimitFunction  $\geq$  0 at all points
      determine required physical properties PP for I
      Plan = FindBestPlan(I, PP, CostLimitFunction)
      if Plan = FAILURE then break /* no plan within the given cost limit */
      AlgoPlan.CostFunction+ = Plan.CostFunction /* at each point i,
        add Plan.CostFunction(i) to AlgoPlan.CostFunction(i) */
      CostLimitFunction = CostLimitFunction - Plan.CostFunction
        /* at each point i, subtract Plan.CostFunction(i)
          from CostLimitFunction(i) */

    /* Merge the operator plan with the planGroup optPlan */
    optPlan = MinMerge(optPlan, AlgoPlan)
    CostLimitFunction = optPlan.CostFunction()

  if optPlan is successful for some point in the parameter space
    return optPlan
  else return FAILURE

```

Figure 4.9: Parametric Query Optimizer: *ApplyAlgorithms* Routine

follows: It first applies transformation on the given logical expression to generate all equivalent logical expressions. Figure 4.8 shows pseudo code for application of transformations. Then it recursively optimizes transformed (*LogExpr*, *PhysProp*) pairs by applying each applicable operator (algorithm or enforcer) with the specified cost limit.

If there has been a previous attempt to optimize the (*LogExpr*, *PhysProp*) pair then we have an optimal plan and cost function available. The (*LogExpr*, *PhysProp*) pair may have successful plan in some parameter space partitions and failures w.r.t the previous cost limit in some other parameter space partitions.

If the optimal plan returned by the previous attempt has at least one point with failure indication and cost less than the cost limit then we need to re-optimize the (*LogExpr*, *PhysProp*) pair. Else, if the optimal plan has cost less than or equal to the cost limit at some point we return the optimal plan. Else we return FAILURE as at no point in the parameter space we can have a plan within the specified cost limit.

When we re-optimize a (*LogExpr*, *PhysProp*) pair, we save the old plan and optimize the pair for the new cost limit *CostLimitFunction*. After optimization, we need to merge the old plan with the new plan so that no optimization information from either of them is lost (see Section 4.3.2). We merge the old and the new plan using *MaxMerge* operation. The *MaxMerge* operation is similar to the *MinMerge* operation, except that it picks the maximum of the two input cost values at each point in the parameter space.

MaxMerge'ing of the old and the new plan prevents information loss as follows: At a point in the parameter space, if both plans have a successful plan then their costs are the same. If both plans fail at a point then *MaxMerge* picks the higher value cost failure indication and no information is lost. If one plan returns success and the other returns failure indication then failure cost will be less than the success cost and *MaxMerge* picks success cost and no information is lost.

Figure 4.9 shows application of an algorithm. Application of an enforcer is similar. Application of an operator is done as follows: First we evaluate cost function of the operator. Then we optimize the children. For the first child the cost limit will be the input cost limit minus the cost of the operator. For next children,

the cost limit is found by subtracting the operator cost and the cost functions of the children optimized so far from the input cost function. If addition of the operator cost and the cost functions of the children optimized so far exceeds the input cost function at each point in the parameter space then we can not have a plan within the specified cost limit at any point in the parameter space and hence the optimizer returns *FAILURE*.

If the child optimization returns *FAILURE* (i.e. within the given cost limit, the optimizer could not find a plan even for a single point in the parameter space) then for the given $(LogExpr, PhysProp)$ pair there exists no plan within the given cost limit, and the optimizer returns *FAILURE*. Instead, if the child optimization returns a plan for even a point in the parameter point with the given cost limit, the optimization continues.

Each applicable operator is optimized and the optimal plans returned are merged to find the best one at each point in the parameter space using the function *MinMerge*. Finally when the optimization is over, if *optPlan* contains a plan within the given cost limit for at least one point, the optimizer returns *optPlan*; Else it returns *FAILURE*.

4.6 Implementation

We have implemented the extensions by modifying an existing query optimizer based on the Volcano optimization algorithm, developed at IIT Bombay. We used *Polylib*, a public domain polytope manipulation library to implement polytope operations [28]. 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. The current implementation is too slow to be competitive with the conventional query optimizer. Exploring alternative polytope manipulation techniques that do not require arbitrary precision arithmetic, to reduce the overhead, is a topic of further work.

4.7 Summary

In this chapter, we proposed a solution for the parametric query optimization problem for the case when the cost functions are piecewise linear in the given parameters. The solution requires extensions to the conventional optimizer. We outline how to extend the System R and Volcano query optimization algorithms to handle piecewise linear cost functions in place of cost values. We have implemented the extensions by modifying an existing query optimizer based on the Volcano optimization algorithm, developed at IIT Bombay and tested the extensions for queries with two parameters. Although the solution conceptually works for arbitrary number of parameters, the cost of optimization increases exponentially with the number of parameters. The solution is intrusive and needs modifications to the conventional optimizer; our PQO solution for nonlinear cost functions described next overcomes this drawback.

The solution presented in this chapter is exact whereas, the solution presented in the next chapter is heuristic. If the cost functions are piecewise linear, the former will generate an exact solution whereas the latter will generate only a heuristic solution. If the cost functions are not piecewise linear, but can be approximated to piecewise linear form within some error tolerance then the solution generated by the former will be such that, at each point in the parameter space, the cost of the plan returned by the solution will be within some threshold of the cost of the optimal plan. The same can not be said about the solution generated by the latter.

Chapter 5

Parametric Query Optimization for Nonlinear Cost Functions

In the previous chapter 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 chapter¹, 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 optimization 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 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 5.3 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 than the actual optimal plan; the maximum performance degradation observed on a sample set of queries 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.

¹Parts of this chapter appeared in VLDB 2003 [16].

```

Algorithm: AniPQO
Input:  $n$  (the number of parameters),
       parameter space polytope /* A polytope in  $\mathbb{R}^n$  */
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 UpdateDecompositionVertices
        from Figure 5.4 in Section 5.2.2 */
return CSOP

```

Figure 5.1: Algorithm to find *POSP*

The rest of the chapter is organized as follows. Section 5.1 describes AniPQO, while Section 5.2 describes the representation and manipulation of the parameter space decomposition. Section 5.3 describes the DAG representation of the plans. Section 5.4 presents the results of the experimental evaluation of AniPQO. Section 5.5 summarizes the chapter.

5.1 An overview of AniPQO

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

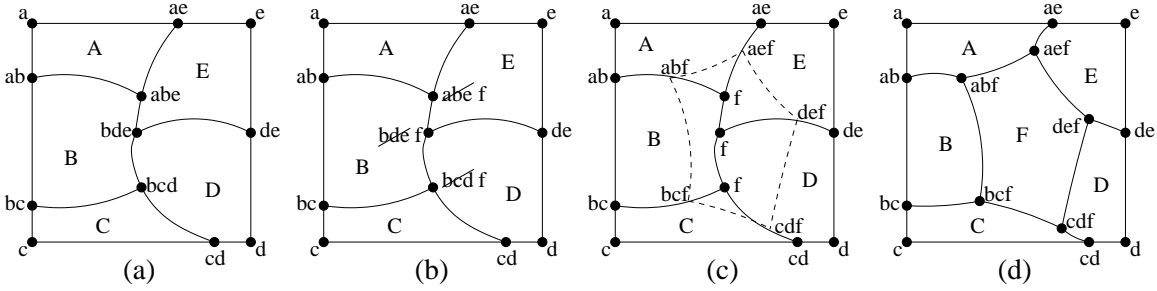
As said in the previous chapters (see Section 3.1), in general, we are not interested in the whole parameter space \mathbb{R}^n (where n is the number of parameters) as only a part of it would constitute legal combinations of the parameter values and, typically, the parameter space of interest is a hyper-rectangle defined by a range of legal values specified for each parameter. So, as done in the previous chapters (see Section 3.1), we assume that the parameter space of interest is a closed convex polytope, which we call the *parameter space polytope*.

As said in Section 3.1 of Chapter 3, conventional query optimizers return an optimal plan along with its cost (at a given point in the parameter space) and, 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:

- 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 5.1. 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.

Figure 5.2: Carving out the region for a new plan in *CSOP*

At an abstract level this algorithm is the same as our cost polytope algorithm (Section 3.4) and Ganguly’s algorithm [8]² for linear cost functions. For the case with linear cost functions, the algorithm is exact and finds the complete *POSP*.

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 of cost functions into account; the details are described in Section 5.2. We use the term *AniPOSP* to refer to the set of plans returned by AniPQO.

Example 5.1.1 Consider an example presented in Figure 5.2. The cost functions are nonlinear and the parameter space polytope is a 2-dimensional rectangle. Let *CSOP* be $\{A, B, C, D, E\}$. Figure 5.2 (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 5.2 (d). We get the new decomposition from the old one by carving out the region of optimality for the new plan *F*. Figures 5.2 (b)-(d) show the steps in this operation; the steps are explained in detail later. \square

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 un-optimized 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, as proved in Section 3.6, 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. 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* (or $AniPOSP \subseteq POSP$); we propose a method for approximate indexing in Appendix E. 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 5.3 provides details of this optimization.

²Ganguly’s algorithm and the cost polytope algorithm mainly differ in that the former operates in \mathbb{R}^n and the latter operates in \mathbb{R}^{n+1} , where *n* is the number of dimensions; the abstraction we present here works in \mathbb{R}^n and, hence, is closer to Ganguly’s algorithm.

5.2 Representation and manipulation of the decomposition

Figure 5.2 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.

5.2.1 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 5.2.

Example 5.2.1 The edges defining the decomposition in Figure 5.2 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. \square

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,

$$\mathcal{O}_{\mathcal{S}}^P = \bigcap_{v \in \mathcal{S}} \mathcal{O}_v^P$$

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.

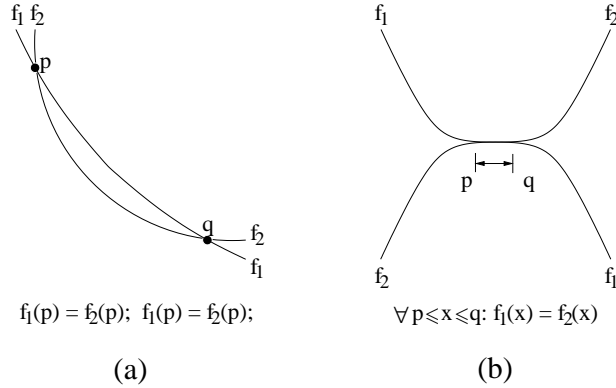


Figure 5.3: Counter-examples to the assumption

In general the assumption is not true; two cases that contradict the assumption are shown in Figures 5.3 (a) and (b). In Figure 5.3 (a) the cost functions intersect at two points and the points are separated. In Figure 5.3 (b) the cost functions intersect at infinite number of points and the intersection points are continuous. In Figure 5.3 (b), we can pick any one of the points where the functions are intersecting as the equi-cost point without creating much problem but, it is not, in general, easy to get around the problem in Figure 5.3 (a) without introducing large errors in the functions. Note that the functions in this example are continuous and non-increasing but this does not prevent them from intersecting at more than one point. But we can relax the assumption if the cost functions are continuous and intersect transversally; see Appendix D for details. The effect of this assumption is that, AniPQO may miss some plans from *POSP*; however, experiments in Section 5.4 suggest that this does not affect the quality of the solution much.

Lemma 5.2.1 $v \in \mathcal{V}_P \Leftrightarrow \mathcal{F}_{\{v\}} < |\mathcal{O}_{\{v\}}^P|$ □

Lemma 5.2.2 *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 5.2.1 *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 5.2.2.

Figure 5.2 (a) shows the decomposition of parameter space rectangle induced by plan set $\{A, B, C, D, E\}$ for non-linear cost functions and Figure 5.7 (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.

Note that the solution works for linear cost functions (special case of nonlinear cost functions). But unlike our solution from Chapter 3 and Ganguly’s solution [8] for linear cost functions, AniPQO need not maintain the complete description of the intermediate polytopes and this simplifies implementation³.

5.2.2 Updating the decomposition vertex set

Figure 5.4 gives pseudo code for an algorithm `UpdateDecompositionVertices` which updates the set of decomposition vertices when a new plan is detected. This is an exact algorithm for the linear case, but

³We may need to build the complete descriptions of the final polytope regions, if we choose to use them to pick the appropriate plan at run time – a given set of parameter values defines a point in the parameter space and the optimal plan is the one in whose region of optimality the point falls in.

Algorithm: UpdateDecompositionVertices

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  $u$ 
     $P = \mathcal{O}_{\{u,v\}}^{CSOP} \cup \{p\}$  /*  $\mathcal{O}_{\{u,v\}}^{CSOP}$  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
     $w = \text{FindEquiCostPoint}(R, P)$  /* Find vertex  $w$  in  $R$  s.t. at  $w$  plans in  $P$  are equi-cost;
        this may fail; see Figure 5.5 in Section 5.2.3. */

If previous step fails
     $R$  = parameter space hyper-rectangle
     $w = \text{FindEquiCostPoint}(R, P)$  /* 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 5.4: Algorithm to update decomposition vertices

may 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 5.4 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 suboptimal 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 5.2.2 Consider the example in Figure 5.2. 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. \square

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 5.2.2 we have,

$$\mathcal{F}_{\{u,v\}} \leq |\mathcal{O}_{\{u,v\}}^{CSOP}| \quad (5.1)$$

There is a contour⁴ between vertices u and v along which the plans in $\mathcal{O}_{\{u,v\}}^{CSOP}$ are equi-cost. Plan p is optimal at v , and its cost is less than the cost of the plans in $\mathcal{O}_{\{u,v\}}^{CSOP}$ at v ; plan p is not optimal at u , and its cost is more than the cost of the plans in $\mathcal{O}_{\{u,v\}}^{CSOP}$ at u .

⁴A straight line segment in the linear case.

```

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 atleast 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 5.5: Algorithm to find approximate equi-cost point for a set of plans

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

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

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

and, from Equation 5.1, we get,

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

By Lemma 5.2.1, 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 5.2.3. If this fails, we search the smallest hyper-rectangle which contains the parameter space polytope⁵ 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 5.2.4). The AniPQO algorithm works with such ill-defined decompositions but may not detect some plans that it would have detected otherwise.

5.2.3 Finding an equi-cost point

Figure 5.5 gives pseudo code for a heuristic algorithm `FindEquiCostPoint` to find an equi-cost point within 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

⁵In our implementation, the parameter space polytope itself is a hyper-rectangle and we search it.

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. If there are multiple such equi-cost points, we pick any one of them, as described in Section 5.2.1.

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 atleast 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 5.6 with the regions of optimality defined for three plans. Each plan is optimal at atleast 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,

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

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

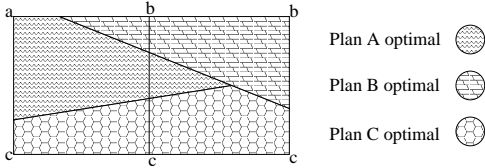


Figure 5.6: Example of equi-cost point approximation

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-cost⁶

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 5.6 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 5.2.4); 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 5.6 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.

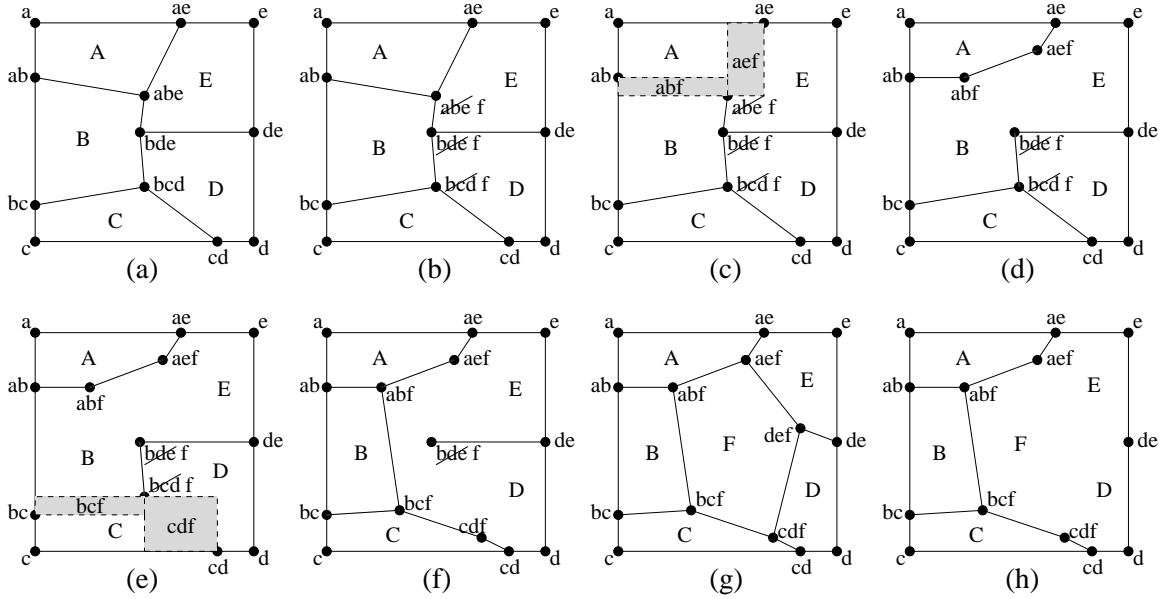
5.2.4 An example iteration of AniPQO

Let us consider the example from Figure 5.2 and step through the algorithm.

Figure 5.7 (a) shows the edge skeleton of the decomposition in Figure 5.2 (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 5.7 (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 5.7 (c)) and assume that the equi-cost 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.

⁶This 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 5.7: An iteration of *AniPQO* algorithm

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 5.7 (d).

Next, we consider two more conflicting edges (bcd, bc) and (bcd, cd) and repeat the above mentioned procedure. See Figures 5.7 (e) and (f).

Now, we consider the remaining conflicting edge (bde, de) . As we can see from Figures 5.2 (d) and (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 (Figure 5.7 (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 5.7 (g). Figure 5.7 (g) is the edge skeleton of the nonlinear decomposition in Figure 5.2 (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 5.7 (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.

5.3 The DAG Representation of Plans

In this section we describe how we use the AND-OR DAG representation of a set of plans (see Section 2.4.1) to boost the quality of the results and facilitate picking an optimal plan at run time.

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 run-time:** In the DAG framework, if two plans share a opera-

tor/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 sub-plans 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, $s_{e_1}^{p_1}$ is cheaper than $s_{e_1}^{p_2}$ and $s_{e_2}^{p_2}$ is cheaper than $s_{e_2}^{p_1}$; then a hybrid of p_1 and p_2 (containing $s_{e_1}^{p_1}$ and $s_{e_2}^{p_2}$) 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*.

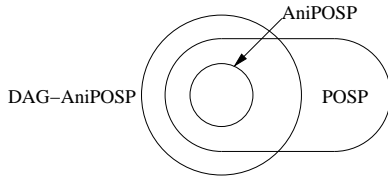


Figure 5.8: Venn diagram of the plan sets

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 5.8.

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

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 by Graefe and McKenna [12].

5.4 Experimental Evaluation

We implemented our algorithm on top of a query optimizer based on the Volcano optimization algorithm, 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.

We evaluated two types of SPJ queries, RiPj queries and PlasticQuery from *Plastic* project by Ghosh *et.al.* [11]. The queries are listed in Figure 5.9. We used selectivities of the non-join parameterized selection predicates as parameters, with selectivities varying from 0 to 1, and have tested our algorithm for up to four parameters. In RiPj queries, i indicates the number of relations, j indicates the number of

```
R2P2 query:  select partsupp.ps_suppkey
              from partsupp, supplier
              where partsupp.ps_suppkey = supplier.s_suppkey
              and ps_partkey < :1
              and s_suppkey < :2

R3P2 query:  select partsupp.ps_suppkey
              from partsupp, supplier, nation
              where partsupp.ps_suppkey = supplier.s_suppkey
              and supplier.s_nationkey = nation.n_nationkey
              and ps_partkey < :1
              and s_suppkey < :2

R3P3 query:  select partsupp.ps_suppkey
              from partsupp, supplier, nation
              where partsupp.ps_suppkey = supplier.s_suppkey
              and supplier.s_nationkey = nation.n_nationkey
              and ps_partkey < :1
              and s_suppkey < :2
              and n_nationkey < :3

R4P4 query:  select partsupp.ps_suppkey
              from partsupp, supplier, nation, region
              where partsupp.ps_suppkey = supplier.s_suppkey
              and supplier.s_nationkey = nation.n_nationkey
              and nation.n_regionkey = region.r_regionkey
              and ps_partkey < :1
              and s_suppkey < :2
              and n_nationkey < :3
              and region.r_regionkey < :4;

PlasticQuery: select partsupp.ps_suppkey
               from partsupp, supplier, nation, region, part
               where partsupp.ps_suppkey = supplier.s_suppkey
               and supplier.s_nationkey = nation.n_nationkey
               and nation.n_regionkey = region.r_regionkey
               and part.p_partkey = partsupp.ps_partkey
               and part.p_size < :1
               and partsupp.ps_supplycost < :2;
```

Figure 5.9: Test queries

Query	# Plans					DAG Size			AniPQO Max. degradation (%)			
	POSP	AniPOSP		DAG-AniPOSP \cap POSP		POSP	AniPOSP		w/o DAG		with DAG	
		t (%)		t (%)			t (%)		t (%)		t (%)	
		1	10	1	10		1	10	1	10	1	10
R2P2	10	10	9	10	9	21	21	20	0.00	0.61	0.00	0.61
R3P2	15	14	6	15	8	30	30	23	0.27	2.86	0.00	2.86
R3P3	36	24	15	31	17	39	37	30	10.62	4.14	2.40	3.52
R4P4	134	49	29	87	53	85	61	51	12.12	8.28	2.59	3.69
PlasticQuery	30	13	8	20	16	41	35	33	1.98	7.88	0.13	1.90

Figure 5.10: Quality of the results for queries on the TPCD catalog with no indices on the relations

Query	# Plans					DAG Size			AniPQO Max. degradation (%)			
	POSP	AniPOSP		DAG-AniPOSP \cap POSP		POSP	AniPOSP		w/o DAG		with DAG	
		t (%)		t (%)			t (%)		t (%)		t (%)	
		1	10	1	10		1	10	1	10	1	10
R2P2	6	5	5	5	5	16	14	14	3.50	3.50	3.50	3.50
R3P2	9	6	5	6	5	24	18	16	3.50	3.50	3.50	3.50
R3P3	11	8	6	8	6	27	22	18	3.50	3.50	3.50	3.50
R4P4	29	20	12	25	15	51	44	36	0.11	3.48	0.11	3.48
PlasticQuery	11	5	5	7	6	29	26	24	0.03	0.68	0.02	0.02

Figure 5.11: Quality of the results for queries on the TPCD catalog with indices on the relations

parameters and :1, :2, :3 and :4 are the parameters whose values will be known only at run-time. We tested queries R2P2, R3P2, R3P3, R4P4 and PlasticQuery.

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 by Rao [30]); 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. As an example, we present the *POSP* and parameter space decomposition for query R2P2 in Appendix A.

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 5.1), 1% and 10%.

The quality of the results is tabulated in Figure 5.10 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 run-time 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

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 (%)			
1	10	1	10	1	10	1	10	1	10	1	10	
R2P2	21	11	22	10	764	140	12	7	13	9	209	141
R3P2	23	6	22	6	688	49	12	7	14	9	215	141
R3P3	41	20	42	18	1339	532	31	13	31	13	1425	288
R4P4	101	46	95	41	4587	1708	61	33	62	31	4009	1227
PlasticQuery	20	10	18	8	897	140	7	7	6	6	34	18

Figure 5.12: Optimization overhead for queries on the TPCD catalog

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 5.11 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 5.12 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 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 found⁷. 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 5.12 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; see Chapter 3.

⁷The 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 (see Chapter 3). Since the decomposition may be incomplete in our case, this number is approximate.

5.5 Summary

In this chapter, we presented AniPQO – a heuristic solution for the PQO problem for the general case when the cost functions may be nonlinear in the given parameters. 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. 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 a set of plans found to boost the quality of the results and facilitate picking an optimal plan at run time.

We have implemented the AniPQO algorithm and presented 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 than the actual optimal plan; the maximum performance degradation observed on a sample set of queries is very small (3.5%). Although the optimization 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 practical for up to 4 parameters. We also showed how AniPQO can degrade gracefully and provide a slightly inferior solution with significant reduction in the cost of optimization. If we consider either only query parameters (as we did in the performance evaluation) or only system parameters, the limit of four on the number of parameters seems reasonable; but if we consider both, the number of parameters may exceed four. Evaluating AniPQO with system parameters is left as a future work.

Chapter 6

Memory Cognizant Query Optimization

Complex queries make heavy use of join, aggregation and sorting operations which are memory intensive. Conventional query optimizers have the drawback that they assume that each of these operators can use the entire memory available to the query execution engine or that a fixed fraction of the memory is allocated to each operator. This assumption is clearly not valid when executing a pipeline, where the available memory has to be divided among several concurrently executing operators in the pipeline and, each of which may have a cost function defining the operation cost as a function of memory made available to it. This may lead to a suboptimal plan.

In this chapter¹ we address the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator.

We propose two approaches to solve this problem: a *2-phase* approach and a *1-phase* approach.

In the 2-phase approach, we first optimize the query using a conventional optimizer to get a traditional optimal plan, and in the second phase we divide memory among operators in each pipeline of the plan so that each pipeline runs optimally in the available memory.

In the 1-phase approach we modify the traditional query optimizer to make it memory cognizant. The modified optimizer takes into account division of the memory amongst operators while choosing between equivalent plans.

Although the 2-phase approach is able to optimally divide the available memory amongst the operators of a given pipeline, it may not necessarily give the optimal execution time for a query since the plan being considered may itself be suboptimal for the available memory. Therefore, the 1-phase approach seems to be a better alternative.

We consider the problem of building a 1-phase *memory cognizant query optimizer*; our contributions are as follows:

- We design efficient techniques to divide the available memory optimally among operators in a pipeline. If done naively, this process can take time quadratic in the available memory size, and is impractical. We show how to reduce the computation time by using piecewise linear approximations of the cost-versus-memory functions of various operators.
- We show how some of the assumptions made while evaluating the cost of various operators are not valid when cost is a function of the available memory and we are dividing memory amongst operators. Based on these observations, we define various memory cognizant execution algorithms/schemes for each operator.
- We have implemented our techniques by modifying an existing query optimizer by Roy [31], based on the Volcano optimization algorithm by Graefe and McKenna [12].

¹Parts of the chapter appeared in COMAD 2000 [14].

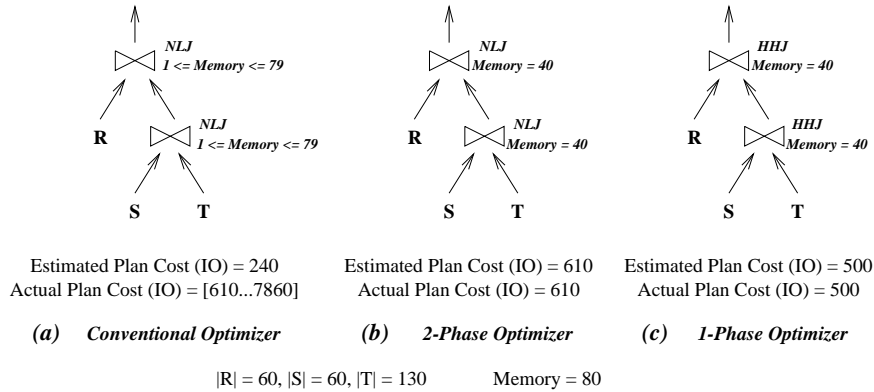


Figure 6.1: Motivating Example

- We show how to make a cost-based decision of converting a pipelined edge into a blocking edge (*i.e.* the child operator writes the intermediate result to the disk and the parent operator reads it back). The decision to convert a pipeline edge into a blocking edge depends upon whether the extra memory available to individual pipeline trees thus formed can more than offset the extra disk IO of the intermediate result. This decision is integrated into memory cognizant optimizer.
- It has been conjectured by Nag and DeWitt [26] that 1PO will perform no better than 2PO. But the paper gives no conclusive evidence of this claim. We evaluate 1PO against 2PO and study the results obtained. Performance results show that 2PO performs reasonably well as compared to 1PO for the tests conducted.

Our discussion is in the context of the Volcano query optimization algorithm by Graefe and McKenna [12] but the techniques can also be used with a System R style optimization algorithm by Selinger [33].

Unlike PQO and like conventional query optimization, in the memory cognizant query optimization all parameters (including memory) take fixed values and the optimizer generates a single optimal plan wherein, for each pipeline, the total available memory is divided optimally amongst all the operators in the pipeline. However, there are similarities in the techniques used for the two apparently dissimilar problems, namely parametric query optimization and memory cognizant query optimization. Inside the optimizer, plan costs are not single cost values but are cost functions with memory as the parameter. Like PQO for piecewise-linear cost functions (Chapter 4), we approximate these cost functions to piecewise-linear form for efficient processing.

The rest of the chapter is organized as follows. We present a motivating example in Section 6.1 We discuss the related work in Section 6.2. Section 6.3 defines memory cognizant execution schemes for various algorithms. In Section 6.4 we present our technique for choosing the optimal division of memory among a set of concurrently running operators. Section 6.5 describes how the Volcano optimizer is extended to include memory-cognizance and also describes how the decision to convert a pipeline edge into a blocking edge is taken in cost-based manner. We present some experimental results in Section 6.7 and summarize the chapter in Section 6.8.

6.1 Motivating Example

Consider a query $R \bowtie T \bowtie S$ with two join predicates: one between the relations R and T and the other between the relations S and T . Cross products are not allowed. The relation sizes are $|R| = 60$, $|S| = 60$ and $|T| = 130$ disk blocks, the memory available is 80 blocks. Cost is measured in terms of number of disk block accesses.

The traditional optimizer generates the plans shown in Figures 6.1 (a) and (b). Since the available memory is sufficient to execute either of the two nested loops join operators *in memory*, the two nested loops join operators in the plan are assumed to run in pipelined fashion. The estimated cost of this plan is the cost of reading each relation once from the disk and equals $|R| + |S| + |T| = 240$. However, both the nested loops join operators can not be executed simultaneously in the given memory. Thus the cost predicted by the optimizer is inaccurate; the actual cost is more than the estimated cost and depends upon the scheduling of the operators in the given memory. For example, as shown in Figure 6.1 (a) the worst-case memory scheduling will allocate 79 blocks to one of the nested loops join operators and only one unit to the other, thus making the cost 7860 units which is significantly higher than the estimated cost.

For the plan generated by the traditional optimizer, the least cost would be incurred when the memory is divided equally between the operators as shown in Figure 6.1 (b), Here the total cost incurred would be 610 units and is the optimal for the given query plan and the given memory. This corresponds to our 2-phase approach.

However, if we employ hybrid hash join² instead of nested loops join for the two join operations as shown in Figure 6.1 (c) and equally divide the memory between the two operators, the cost incurred would be 500 and the plan is optimal for the given memory. This corresponds to our 1-phase approach.

The example illustrates two key issues: First, the cost of a plan may change when the division of memory is changed. Second, the choice of plan itself needs to involve consideration of memory division.

6.2 Related Work

Previous work on resource management mainly deals with scheduling resources efficiently for executing a given query plan. These resource scheduling decisions do not interact with the query optimizer and the two problems (query optimization and scheduling) are solved in separate phases: the query is optimized first to come up with a plan, and then the plan is scheduled.

Query schedulers can be broadly classified into two categories: *static* schedulers and *dynamic* schedulers. Static scheduling is applied after query optimization and before execution. The dynamic scheduling strategy is integrated with the query execution engine and makes the engine adaptive to fluctuations in resource availability.

To the best of our knowledge no previous work has tried to integrate the memory allocation and pipelining decisions with the query optimizer (*i.e.* implemented IPO) and our work is the first one to do this. Several papers have considered memory allocation for a given query plan (2PO). Again, to the best of our knowledge no previous work has tried cost-based scheduling and all the previous scheduling strategies are heuristic based, the only exception being a strategy proposed by Nag and DeWitt [26] which assumes operator cost functions to be linear w.r.t to memory. They [26] conjecture that 1PO will perform no better than 2PO but give no conclusive evidence of this claim.

Yu and Cornell [38] consider an environment of concurrently running queries and study the problem of memory allocation to individual queries. They define a concept of *return-on-consumption (ROC)* to study overall reduction in response time due to additional memory allocation to a query. Each query is assumed to be a single-join query. A memory scheduling policy is proposed wherein more memory is allocated to queries which have high value of *ROC*.

Mehta and DeWitt [23] also consider the problem of memory scheduling in multi-query environment. They divide queries in different categories depending upon their memory requirement and provide several heuristics for memory allocation depending upon the classification. Again, only single-join queries are considered and only hash join operator is considered.

Bouganim *et. al.* [2] propose various static and dynamic scheduling schemes for a query tree. They split a given query tree into a set of maximal pipeline chains, each called *pc-task* and scheduled separately. Under static scheduling they propose several heuristic-based strategies to divide memory among operators depending upon minimum (M_{min}^{op}) and maximum (M_{max}^{op}) memory requirements of each operator. The

²assume the left input to be the build input and the right input to be the probe input

heuristics proposed include: divide memory equally amongst the operators, allocate maximum memory to the operator which has the smallest M_{max}^{op} , give each operator memory in proportion to its M_{max}^{op} , and give maximum amount of memory to an operator having the largest M_{max}^{op} . Under dynamic scheduling strategy they propose a memory adaptive execution engine, which dynamically allocates memory and changes query scheduling as and when required. If the memory requirement of a pc_task can not be satisfied, they split the pc_task into segments to be scheduled separately. The scheduling strategies defined consider only maximum and minimum memory requirement of the operators and the memory allocation decision is not cost-based.

Nag and DeWitt [26] propose several heuristic based static scheduling strategies which are more or less similar to those proposed by Bouganim *et. al.* [2]. They divide a query tree into concurrently schedulable units called *shelves* and divide memory amongst the operators in a shelf. They also propose a cost based strategy which assumes operator cost model to be linear. We consider a more general cost model.

Davison and Graefe [6] and Zeller and Gray [39] show how to make a single hash join adaptive to memory fluctuations but do not consider scheduling of an entire query plan. Pang *et.al.* [27] examine the same problem in the context of real-time databases where the priority of a query needs to be considered and a query may need to be scheduled in absence of sufficient memory.

6.3 Memory Cognizant Query Execution

Generally the following assumptions are made while evaluating the cost of an operator, but are not valid when the cost is a function of the available memory and the memory is divided amongst operators:

- It is assumed that each operator in a pipeline utilizes all the available memory and the input is streamed into it in a pipelined fashion. This assumption is not valid as already described and, the cost of an operator is decided by the memory available to the operator and the size of its input.
- It is assumed that for a join operator, the smaller of the two inputs is the outer or left input and the larger one is the inner or right input and this yields the optimal cost. This assumption is not valid as described further.
- It is assumed that in multiphase sort or hash operation each merging or partitioning phase utilizes all the available memory but, in reality, the first and the last phase need to share the available memory with the child operator and the parent operator respectively, and only the intermediate phases utilize all the available memory.

Based on these observations we define various execution schemes for each operator which primarily dictate (a) scheduling of the various phases of an operator, (b) memory utilization of these phases, and (c) which one of the two inputs acts as the left input for join operation.

We use the following notations:

- L = the size of the left input
- R = the size of the right input
- M_{tot} = the total available memory units
- M = the number of memory units available for the plan tree rooted at the operator in question

For simplicity, we assume the memory unit size to be one disk block.

A scheme for an operator may divide the execution of the scheme into several phases. For a given memory M , the cost of a scheme is the sum of the costs of various phases of the scheme and the cost of an operator is the minimum of the costs of all its schemes.

Sorting

The input to this operator is an unsorted stream of tuples and output is a single sorted stream of tuples. Of the three phases defined below, the second one is optional. The operator performs disk based sorting if the second phase is present and performs in-memory sorting if it is absent.

- **Phase I:** This phase runs in pipelined fashion with the input tree. It collects the tuples from the input tree and creates sorted runs of them. The total available memory M_{tot} is divided into two parts: say, M units to the operator to hold and sort the tuples and $(M_{tot} - M)$ units to the input tree. Each output partition will be of size M units and the number of partitions will be $\lceil \frac{X}{M} \rceil$, where X is the size of the input.
- **Phase II (optional):** This phase merges the sorted partitions created in the first phase to reduce the number partitions and may take more than one merging phase. This phase is allocated with all the available memory M_{tot} .
- **Phase III:** This phase runs in pipelined fashion with the parent operator and performs final merging of the sorted partitions created in the previous phase. It creates a single sorted stream and this stream is fed to the parent operator. The total available memory M_{tot} is divided among the operators, including the parent operator, running in pipeline, with this phase. The memory allocated to this phase must be at least as much as the number of partitions produced by the previous phase.

The difference between the regular sort operator and our sorting scheme is that the former assumes that all the memory is available for sorting whereas the later is aware that its first and last phase need to share the memory with the input tree and the parent operator respectively.

Hashing

Of the three phases defined below, the second one is optional. The operator performs disk based hashing if the second phase is present and performs in-memory hashing if it is absent.

- **Phase I:** This phase runs along with the input tree in a pipeline. It collects the tuples from the input tree and creates hash partitions which are written to the disk as they are generated. The total available memory M_{tot} is divided into two parts. Say, $(M_{tot} - M)$ units are allocated for the input tree and M units are used for holding the output partitions. So each output partition will be of size $\lceil \frac{X}{M} \rceil$ units and the number partitions would be M , where X is the size of the input.
- **Phase II (optional):** The partitions created in the first phase are further partitioned to reduce the partition sizes. This may take more than one partitioning phases. This phase too is carried out for both inputs. This phase is allocated with all the available memory M_{tot} .
- **Phase III:** This phase runs along with the parent operator in pipeline and performs in-memory hashing of each partition created in the previous phase. The total available memory M_{tot} is divided among the operators, including the parent operator, running in pipeline, with this phase. The memory allocated to this phase must be at least as much as the size of the biggest partition created by the previous phase.

Nested Loops Join (NLJ)

We define the following three schemes for executing a nested loops join operator.

- **Scheme 1:** This scheme uses the smaller input as the right input. Initially the right input tree is completely processed to generate all the right input tuples and these tuples are stored in-memory. This requires an allocation of at least R units of memory. Then, the left input tree is processed and the left input tuples, as they are generated, are matched with the right input tuples, which are all

memory resident. Thus the right input is blocking, though it is not written to the disk, and the left input is pipelined.

Out of the memory available, say, M units are allocated to the tree rooted at NLJ . Of the M units, R units are used by NLJ for storing the entire right input. So, the remaining $(M - R)$ units are used for processing the right input tree. Once the right input tree is processed completely, we will have all the right input tuples available in memory (in R unit space). Then the left input is executed and its tuples, as produced, are matched with in-memory right input. So, the left input tree also gets $(M - R)$ units of memory for its execution.

- **Scheme 2:** Here again the right input tree is blocking and the left input is pipelined but the right input is written to the disk. The right input is completely processed before NLJ starts and all the right input tuples are stored on the disk. Then left input is executed in pipelined fashion with NLJ . As the right input is written to the disk, NLJ can use memory less than R units (unlike *Scheme 1*), and read the right input multiple times from the disk.

As the right input is processed independently, it uses $(M_{tot} - 1)$ units of memory for its execution; one unit is used for buffering the tuples before writing them to the disk. Then the left input is executed in pipelined fashion with NLJ , and possibly its ancestors. So if the memory allocated to the tree rooted at NLJ is M , it will be divided optimally between NLJ and the left input tree.

- **Scheme 3:** Same as *Scheme 2* but with the roles of the left and right inputs reversed.

In *Scheme 2*, the cost of executing the tree rooted at NLJ in M units of memory is the summation of the cost of executing NLJ in say $M' (< M)$ units of memory, the cost of the right input running in $M_{tot} - 1$ units of memory and the cost of the left input running in $M - M'$ units of memory. It is easy to see that we can not decide which one of the two inputs should be the left input based only on their sizes and we need evaluate the cost for both the alternatives, hence the need for *Scheme 3*.

Sort Merge Join (*SMJ*)

We define the following five schemes for executing a sort merge join operator.

- **Scheme 1:** In this scheme, both the inputs are sorted and merged in-memory. The operation is divided into three phases:
 - **Phase I:** The left input tree is executed completely and the left input tuples are stored and sorted in-memory.
 - **Phase II:** The right input tree is executed completely and the right input tuples are stored and sorted in-memory.
 - **Phase III:** The two sorted inputs are merged and joined in-memory.

In the first phase, of the M units of memory allocated to the tree rooted at the SMJ , L units are allocated to the SMJ for storing and sorting the left input tuples and the left input tree is executed with $(M - L)$ memory.

In the second phase, of the M units of memory allocated to the tree rooted at the SMJ , an additional memory of R units is allocated to the SMJ , for storing and sorting the right input tuples. Hence, the right input tree is executed with memory $(M - L - R)$ units.

- **Scheme 2:** In this scheme, the left input is stored and sorted in-memory while the right input employs disk-based sorting. The disk-based sorting of the right input is done by a separate sort operator and the SMJ assumes that the right input is partially sorted and is available in the form of a set of sorted runs on the disk. The operation is carried out in two phases:

- **Phase I:** The left input tree is executed completely and the left input tuples are stored and sorted in-memory.
- **Phase II:** Sorted runs of the right input are merged and joined with the in-memory sorted left input.

In the first phase, of the M units of memory allocated to the tree rooted at the SMJ , L units are allocated to the SMJ for storing and sorting the left input tuples and the left input tree is executed with $(M - L)$ memory.

After the processing of the left input tree (in phase I) is finished, the memory $(M - L)$ is freed. In the second phase, this memory is used to read the sorted partitions of the right input and to merge and join them with sorted in-memory left input. This implies that, we can have a maximum of $(M - L)$ partitions of the right input. Thus the right input should be processed beforehand to have a maximum of $(M - L)$ sorted partitions.

- **Scheme 3:** In this scheme, none of the inputs is sorted in-memory. Each input is assumed to be partially sorted and is available in the form of a set of sorted runs on the disk. The operator reads the sorted runs of the two inputs from the disk, merges and joins them.

All the available memory, M units, allocated to the tree rooted at the SMJ is used by the SMJ operator itself and is used for reading the sorted runs of the two inputs. Thus, this memory needs to be divided between the sorted runs of the two inputs in optimal fashion.

- **Scheme 4:** Same as *Scheme 1* but with the roles of the left and right inputs reversed.
- **Scheme 5:** Same as *Scheme 2* but with the roles of the left and right inputs reversed.

The reason for defining *Scheme 4* and *Scheme 5* for the SMJ operator is similar for defining *Scheme 3* for the NLJ operator.

Hash Join (HJ)

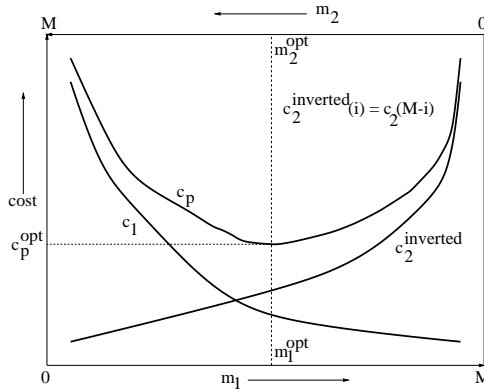
We define the following two schemes for executing a hash join.

- **Scheme 1:** This scheme uses the smaller input as the left input. In this scheme, both the inputs are hashed and matched in-memory. The operation is divided into three phases:
 - **Phase I:** The left input tree is executed completely and the left input tuples are stored and hashed in-memory.
 - **Phase II:** The right input tree is executed and the right input tuples are probed with the in-memory left input tuples as they are generated.

In the first phase, of the M units of memory allocated to the tree rooted at the HJ , L units are allocated to the HJ for storing and hashing the left input tuples and the left input tree is executed with $(M - L)$ memory.

After the processing of the left input tree (in the first phase) is finished, the memory $(M - L)$ is freed. Thus, in the second phase, the memory of $(M - L)$ units is used to execute the right input tree.

The hybrid hash join is a variant of this scheme and in the hybrid hash join, L' units of memory is used for the in-memory hash partitions of the left input and $L - L'$ buckets of the left input are written to the disk. In the second phase, the tuples of the right input that match the in-memory tuples of the left input are matched as they are generated. But the tuples of the right input that do not match the in-memory tuples of the left input are written to the disk and this takes $L - L'$ units of memory to hold the tuples as they are written to the disk. Once, the right input matches all the in-memory

Figure 6.2: Optimal division of memory between o_1 and o_2

partitions of the left input, the hash buckets of the left input written to the disk are read one-by-one and hash partitioned in-memory. For each left input bucket, the corresponding right input bucket is read from the disk and matched with the left input.

- **Scheme 2:** This scheme uses the smaller input as the left input. In this scheme, each input is assumed to be partially hashed and is available in the form of a set of hash partitions on the disk. The operator reads each partition of the left input and hashes it in memory; it then reads tuples from the corresponding right partition and probes them with the in-memory left input tuples.

All the available memory, M units, allocated to the tree rooted at the HJ is used by the HJ operator itself and is used for in-memory hashing of the partitions of the left input. We do not need any memory the right input partitions as each tuple from the respectively partition is matched with the corresponding left input partition as soon as it is read.

6.4 Optimal division of Memory for a given Query Plan

We present here a crucial building-block of our memory-cognizant optimizer: a technique to optimally divide memory among the operators running in a pipeline.

When a pipeline is executed, all operators in the pipeline run simultaneously in the given memory. The cost of running such a pipeline depends upon how much memory each operator in the pipeline gets, and hence it is important to choose the optimal division of memory among the operators.

6.4.1 Cost Functions with Arbitrary Shape

In this section, we consider an optimal memory allocation for the operators in a pipeline, where operator cost functions are of an arbitrary shape. We describe our technique by first considering the problem of sharing available memory between two operators. We then consider three operators, and finally the general case of n operators.

Dividing memory between two operators: Consider a pipeline P composed of two operators o_1 and o_2 with cost functions c_1 and c_2 respectively. Let the available memory be M units. Here, unit refers to the unit of allocation. We can divide this memory between the two operators by giving m_1 units to operator o_1 and m_2 units to operator o_2 ($m_1 + m_2 = M$). Clearly, the cost of executing the pipeline is a function of this division. Let c_p be the cost of the pipeline as a function of m_1 , with $m_2 = M - m_1$. Thus $c_p(m_1) = c_1(m_1) + c_2(M - m_1)$.

This function can be computed by inverting cost function c_2 along the memory axis and adding it to cost function c_1 , as shown in Figure 6.2. We need to find a value of m_1 , say m_1^{opt} at which c_p is

minimal, say c_p^{opt} . This is an optimal allocation to operator o_1 . The corresponding optimal allocation to operator o_2 is $m_2^{opt} = M - m_1^{opt}$.

$$\forall i, 1 \leq i \leq M, \forall j, 1 \leq j \leq M : i + j = M \Rightarrow c_1(i) + c_2(j) \geq c_1(m_1^{opt}) + c_2(m_2^{opt})$$

For cost functions with an arbitrary shape we need to examine all possible division points and calculate c_p for each value of m_1 from 0 to M . This takes $O(M)$ time. Note that there is a trade-off between the time and the granularity of the unit of memory.

Dividing memory amongst three operators: Consider a pipeline P composed of three operators o_1 , o_2 and o_3 with cost functions c_1 , c_2 and c_3 resp. The goal is to find an optimal memory allocation, say m_1^{opt} , m_2^{opt} and m_3^{opt} units to operators o_1 , and o_2 and o_3 resp. such that the execution cost of pipeline P is minimized and $m_1^{opt} + m_2^{opt} + m_3^{opt} = M$. Such a division is given by:

$$\begin{aligned} \forall i, 1 \leq i \leq M, \forall j, 1 \leq j \leq M, \forall k, 1 \leq k \leq M : \\ i + j + k = M \Rightarrow c_1(i) + c_2(j) + c_3(k) \geq c_1(m_1^{opt}) + c_2(m_2^{opt}) + c_3(m_3^{opt}) \end{aligned}$$

A naive method would check all possible combinations of memory allocations and take $O(M^3)$ time. But we can do better by *merging* the cost functions (and thus the operators) incrementally, as follows:

First, we consider two operators o_1 and o_2 and find cost of a pipeline P' consisting these two operators as a function of memory used by the pipeline. Thus we need to calculate the optimal cost of running the pipeline P' with memory i , for each $i : 0 \leq i \leq M$. The cost function of P' stores the optimal division of memory along with the cost value for each memory point i (*i.e.* amount of memory j and k to be given to operators o_1 and o_2 resp., where $j + k = i$). With this the operators o_1 and o_2 are virtually *merged* into plan P' with a cost function defined for it. For further memory division operations, say between plan P' and some other operator or plan, the plan P' will be treated as a *super-operator*.

We can calculate optimal division for a given memory size i and hence cost of running the pipeline P' with memory size i in $O(M)$ time. To calculate this for all $i : 0 \leq i \leq M$ we need $O(M^2)$ time. Thus the cost function of the pipeline P' is calculated in $O(M^2)$ time.

Next, we find the optimal memory division between the plan P' and the operator o_3 . Once we get this division, say m_3^{opt} and $m_{p'}^{opt}$, we can trace back optimal division of memory $m_{p'}^{opt}$ between operators o_1 and o_2 .

Dividing memory amongst n operators: Consider a general case: a pipeline P composed of n operators o_1, o_2, \dots , and o_n with cost functions c_1, c_2, \dots, c_n respectively. To find the optimal memory division amongst these operators we extend the strategy used for the pipeline with 3 operators, and merge the cost functions/operators incrementally. We first merge the cost functions of two operators to get their combined cost function. We then merge this combined cost function with the cost function of the third operator to get the combined cost function of three operators. We continue in this manner and merge all operator cost functions to get the cost function of the pipeline P . Now, we can trace back each step and find, from the cost functions of the intermediate plans, the optimal memory allocation within each of these intermediate plans. This takes $O(n.M^2)$ time for cost functions with an arbitrary shape.

OptMerge Procedure: This procedure optimally merges two input cost functions running in a pipeline and generates the combined optimal cost function along with the optimal memory division for all memory points. Time complexity of this procedure depends upon the shape of the input cost functions. If the input cost functions are of an arbitrary shape then the procedure examines all possible memory division alternatives for each memory point and hence the complexity is $O(M^2)$.

In the next section, we consider how to reduce the cost associated with the memory division operation using piecewise-linear approximations of the cost functions.

```

OptMerge( $c_1, c_2$ )
   $mergeCost = \infty$ 
  for each change-over point  $(m, c)$  in  $c_1$  do
     $c'_2 = c_2$  shifted_by  $(m, c)$ 
     $mergeCost = MinMerge(mergeCost, c'_2)$ 
  for each change-over point  $(m, c)$  in  $c_2$  do
     $c'_1 = c_1$  shifted_by  $(m, c)$ 
     $mergeCost = MinMerge(mergeCost, c'_1)$ 
  return  $mergeCost$ 

```

Figure 6.3: Pseudo Code for *OptMerge* Procedure for Piecewise Linear cost Functions

6.4.2 Piecewise Linear Approximation of Cost Functions

Consider a pipeline P with two operators o_1 and o_2 with cost functions c_1 and c_2 resp. Let the cost function of the pipeline be c_p . If the cost functions c_1 and c_2 are linear, the procedure *OptMerge* defined in the previous subsection takes constant time. Assume that c_1 has slope $slope_1$ in range 0 to m_1 ($m_1 \leq M$) and slope 0 in range m_1 to M , i.e. allocating memory beyond m_1 units yields no benefit. And c_2 has slope $slope_2$ in the range 0 to m_2 ($m_2 \leq M$) and slope 0 in range m_2 to M . Assuming that providing more memory will not increase the cost of an operator/plan, the cost functions are non-increasing and the slopes are non-positive, i.e. $slope_1 \leq 0$ and $slope_2 \leq 0$.

The *OptMerge* procedure simply allocates the maximum possible memory to the operator which gains more per unit memory allocation (i.e. the cost function of which has less slope³) and allocates the remaining memory to the other operator. Let $slope_1 \leq slope_2$. The procedure would allocate m_1 units of memory to operator o_1 and c_p follows the slope $slope_1$ in the range 0 to m_1 . Next, it allocates m_2 units ($m_2 \leq M - m_1$) of the memory to operator o_2 , and c_p follows the slope $slope_2$ in the range m_1 to $m_1 + m_2$. For memory in the range $m_1 + m_2$ to M , c_p has slope 0 and the cost value the same as that for $m_1 + m_2$.

To use this mechanism, we need to approximate cost functions of various operators to linear form. However, even after this approximation, cost functions derived by the procedure *OptMerge* may not be linear; in fact, the derived functions may be piecewise linear. Thus the cost function of a plan can take a piecewise linear form. Since the input to *OptMerge* procedure can be cost function of a plan, the procedure needs to handle piecewise linear cost functions.

Moreover, the ability to handle piecewise linear cost functions means we can use piecewise-linear approximations for single-operator cost functions. This provides a better approximation than the linear approximation. It is easy to approximate cost functions of various database operators⁴ to a piecewise linear form.

The *OptMerge* procedure dealing with piecewise linear cost functions is shown in Figure 6.3. The operation *shifted_by* used in the procedure shifts the cost function along the memory and the cost axes by resp amount. The routine *MinMerge* used in the procedure compares input cost functions for the entire memory range and at each memory point picks up the lower cost value.

If the input cost functions to this procedure have x and y segments (i.e. number of straight line segments in a piece-wise linear function) resp. and $z = \max(x, y)$ then the number of segments in the output cost function will be $\leq z^2$ and the time complexity of the procedure would be $O(z^2 \log z)$.

For each given point i , the algorithm essentially checks each possible memory division, say j units to the first cost function and k units to the second cost function ($i = j + k$), where at least one of j and k is at a change-over point (a point where the cost function changes slope) in the resp. cost functions. And the following theorem establishes that the procedure correctly calculates the optimal cost function of the pipeline.

³considering sign. If we consider absolute values then it would pick the one with higher slope.

⁴including multiphase sort and hash operators which typically have discontinuous cost functions w.r.t. memory

Theorem 6.4.1 *If a pipeline tree P composed of two operators o_1 and o_2 with cost functions c_1 and c_2 respectively is executed in memory i , then at least one of the (possibly many) optimal memory divisions, say j units to o_1 and k units to o_2 ($i = j + k$), is such that at least one of j and k is at a change-over point.*

Proof: Consider one of the optimal divisions of the available memory i , say j units to o_1 and k units to o_2 ($i = j + k$). Let neither of j and k be at a changeover point. We come up with an alternative memory division, say j' units to o_1 and k' units to o_2 ($i = j' + k'$) such that at least one of j' and k' is at a change-over point and $c_1(j') + c_2(k') \leq c_1(j) + c_2(k)$.

Assume that the point j lies on segment s_1 in c_1 and k on segment s_2 on c_2 . Assume further that the segment s_1 begins at memory point b_{s_1} and ends at memory point e_{s_1} and the segment s_2 begins at memory point b_{s_2} and ends at memory point e_{s_2} .

Let the slopes of the segments s_1 and s_2 be $slope_{s_1}$ and $slope_{s_2}$ resp. Assuming that providing more memory will not increase the cost of an operator/plan, the cost functions are non-increasing and slopes are non-positive ($slope_{s_1} \leq 0$ and $slope_{s_2} \leq 0$), though the algorithm does not depend on this assumption.

We consider three cases:

Case I: $slope_{s_1} = slope_{s_2}$

Consider two sub-cases:

Cast I.A: $j - b_{s_1} \leq e_{s_2} - k$

Let $j' = b_{s_1}$ and $k' = k + j - b_{s_1}$. It is easy to see that, $c_1(j') + c_2(k') = c_1(j) + c_2(k)$ and $j' + k' = i$.

Cast I.B: $j - b_{s_1} > e_{s_2} - k$

Let $j' = j - e_{s_2} + k$ and $k' = e_{s_2}$. It is easy to see that, $c_1(j') + c_2(k') = c_1(j) + c_2(k)$ and $j' + k' = i$.

Case II: $slope_{s_1} > slope_{s_2}$

Consider two sub-cases:

Cast II.A: $j - b_{s_1} \leq e_{s_2} - k$

Let $j' = b_{s_1}$ and $k' = k + j - b_{s_1}$. It is easy to see that, $c_1(j') + c_2(k') < c_1(j) + c_2(k)$ and $j' + k' = i$.

Cast II.B: $j - b_{s_1} > e_{s_2} - k$

Let $j' = j - e_{s_2} + k$ and $k' = e_{s_2}$. It is easy to see that, $c_1(j') + c_2(k') < c_1(j) + c_2(k)$ and $j' + k' = i$.

Case III: $slope_{s_1} < slope_{s_2}$

Symmetric to case II.

We see that in all the cases, either j' or k' is at a changeover point and the cost of this alternative division is at most that of the original division. Hence the proof. \square

If the piecewise linear approximation introduces maximum error of $\pm\delta$ at any memory point in each operator cost function in a plan with n operators then cost of the plan calculated using piecewise linear approximation will be within $\pm n\delta$ of the actual cost.

6.5 Memory Cognizant Optimization

In this section, we present an overview of the extensions to make an optimizer algorithm memory cognizant. We have implemented our techniques on a query optimizer based on the Volcano optimization algorithm and the details are presented in Section 6.6.

Our execution algorithms (described in Section 6.3) for query operators include memory-awareness and division of memory among concurrent operators. In this framework, we propose following extensions to make the optimization process memory-cognizant:

- While evaluating cost of an operator (*AND* node in Volcano *DAG* framework), evaluate cost functions for all the execution schemes (as defined in Section 6.3) and for each memory size pick the one with the minimum cost. Note that the inputs to the operator are already optimized and we know their cost functions.
- While comparing alternative operators or plans, for evaluating an expression (*OR* node in Volcano *DAG* framework), compare their cost functions for each memory size. For simplicity assume that two plans/operators evaluate a given expression. If one operator/plan is consistently better than the other in the entire memory range, retain the operator/plan with less cost and discard the other one. If one operator/plan is better at some memory range and the other is better at some other memory range, maintain both of them indicating which one is better in which range. The cost of the expression at a memory size m is the cost of the operator/plan which incurs minimum cost at that memory size.
- While evaluating the cost of a plan P (*AND* node in Volcano *DAG* framework), given the cost function of the root operator O and that of the sub-plan $P' = P \setminus O$, we consider the following possibilities:

- **Edge between O and P' is blocked:** P' runs fully before O starts executing. Thus full memory can be allocated to P' . This will result in minimum execution cost for P' and it will be $P'.CostFunction(MaxAvailMem)$. The memory to be allocated to O can not be decided independently of what type of edge it will be connected to its parent by and what is the execution cost of its parent/ascendants. Thus cost function of the plan P will be the cost function of O with $P'.CostFunction(MaxAvailMem)$, a constant, added for each memory point.

$\forall i, 1 \leq i \leq MaxAvailMem :$

$$P.CostFunction(i) = O.CostFunction(i) + P'.CostFunction(MaxAvailMem)$$

- **Edge between O and P' is pipelined:** O and P' run simultaneously in the memory allocated to P . The cost function of P is obtained by *OptMerge*-ing the cost functions of O and P' . Recall the definition of the procedure *OptMerge* from Section 6.4.

$$P.CostFunction = OptMerge(O.CostFunction, P'.CostFunction)$$

6.5.1 Breaking Pipelined Edges

Consider a pipeline plan P and a pipeline edge E in it. If we break the plan P at edge E we get two independent sub-plans P_1 and P_2 and these plans can be scheduled separately. Let us assume that output of plan P_1 is fed to plan P_2 through edge E .

We have two options for evaluating plan P :

- Schedule the entire plan P in given memory with the edge E behaving as a pipeline edge. Here all operators in the plan P execute simultaneously sharing the available memory. There is no I/O incurred at edge E , as it is a pipeline edge.
- Schedule P_1 first, store its output on the disk. Then schedule P_2 with its input read from the disk. Here, as P is divided into two parts and each part is scheduled separately, the operators will have more memory for execution. However, we incur materialization I/O at edge E which now behaves as a blocking edge.

Clearly, there is a trade-off between letting the edge E behave as a pipeline edge and converting it into a blocking edge. If it is a pipelined edge, no materialization IO is incurred but operators in P will get less memory for execution as all the operators in the plan execute simultaneously in the available memory. If the edge E is a blocking edge, materialization IO is incurred but as the operators in the plan are divided into two independent plans and scheduled separately, the operators will get more memory for execution.

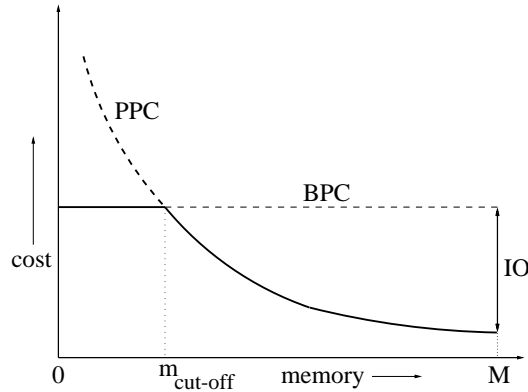


Figure 6.4: Breaking a pipelined edge

We incorporate, into our memory cognizant optimizer, a cost-based technique for deciding when to break a pipelined edge as described below.

Consider a plan P feeding its output to its parent operator C in pipelined fashion. Let the pipelined cost function of P be PPC (with its output edge pipelined and no IO incurred at it). And let read/write cost be IO at its output edge if it is blocked. The problem is to decide at each memory point, say i :

- Let P and C execute in pipelined fashion. The cost of P is $PPC(i)$.
- Let P execute independent of C utilizing all available memory, say $MaxAvailMem$ ⁵ and then write intermediate result to the disk which will be read by the parent C . The cost of the plan P is $PPC(MaxAvailMem - 1) + IO$. Note that it is independent of the available memory i .⁶

Let blocking cost function of plan P be BPC (with its output edge blocked and IO incurred at it). It is given by:

$$\forall i, 1 \leq i \leq MaxAvailMem : BPC(i) = P(MaxAvailMem - 1) + IO$$

The optimal cost function for plan P with the blocking decision incorporated within is given by:

$$MinMerge(PPC, BPC)$$

The routine *MinMerge* compares two input cost functions for the entire memory range and at each memory point picks up the lower cost value. Thus *MinMerge* chooses better of the options: blocking and pipelining the edge. Figure 6.4 shows the operation graphically. If an operator or a plan is made to execute in memory less than certain threshold $m_{cut-off}$, it will, instead, choose to utilize the full memory ($MaxAvailMem$) and write its output to disk. For a cost function with an arbitrary shape, the time complexity of this decision is $O(M)$, whereas for a piecewise linear cost function with x linear segments it is $O(x)$.

6.6 Memory Cognizant Volcano Optimizer

In this section we present the details of the extensions to Volcano query optimizer to make it memory cognizant. Section 6.6.1 presents some definitions and extensions to the data structures; Section 6.6.2 presents various operations on the cost functions and, Section 6.6.3 presents the extended algorithm.

⁵Actually, the child will get $(MaxAvailMem - 1)$ for its execution as one unit of memory will be used for holding the intermediate tuples as they are written to the disk.

⁶Actually, we need $1 \leq i$ since at least one buffer is needed to read back the intermediate result from disk and feed the parent.

6.6.1 Extended Cost Function and Plan

A conventional optimizer has a single value as cost for an operator or a query plan and its corresponding $(LogExpr, PhysProp)$ pair. Here we have a cost associated with each memory size, *i.e.* we define cost as a function of memory size. We refer to a function of cost versus memory size as $CostFunction(m)$, and we have one such cost function for each operator and query plan with a $(LogExpr, PhysProp)$ pair. $Plan.CostFunction(m)$ denotes cost function of a plan, $AlgorithmCostFunction(m)$ denotes cost function of an algorithm and $EnforcerCostFunction(m)$ denotes cost function of an enforcer.

For each operator, as we are considering a range of memory, more than one memory cognizant execution schemes (described in Section 6.3) may be optimal, each being optimal in a particular memory range.

Further, for a $(LogExpr, PhysProp)$ pair, as we are considering the range of memory, more than one physical plan may be optimal, each being optimal in a particular memory range⁷. The optimal plan, $Plan$, for a $(LogExpr, PhysProp)$ pair will contain a list of pairs. Each pair will contain a range of memory size and an optimal physical plan in that range. A physical plan P specified with a memory range (x, y) implies that the optimal way of evaluating the $(LogExpr, PhysProp)$ pair, given that the memory size is in the range (x, y) , is by using plan P .

A conventional optimizer has a single value as a cost limit. Here we have a cost limit for each memory point. We use $CostLimitFunction(m)$ to denote a function giving the value of the cost limit for memory m .

Additionally, in a cost function for a $(LogExpr, PhysProp)$ pair of an algorithm or an enforcer, we may have a segment where we have only a failure indication and no optimal plan or execution scheme. In this segment, the cost function actually indicates a cost limit on the plan. The cost of the plan will be more than the cost function at each memory point in this range.

6.6.2 Operations on Cost Functions

Let $MaxAvailMem$ be the available memory. We define following operators on $CostFunction$:

- \leq_{all} : $costFunction_x \leq_{all} costFunction_y$ means,
 $\forall i, 1 \leq i \leq MaxAvailMem : costFunction_x(i) \leq costFunction_y(i)$
- $>_{all}$: $costFunction_x >_{all} costFunction_y$ means,
 $\forall i, 1 \leq i \leq MaxAvailMem : costFunction_x(i) > costFunction_y(i)$
- $AddCostFunction$: It takes two $CostFunctions$ as arguments, and creates a new $CostFunction$ by adding the input $CostFunctions$ at each memory point.
- $SubtractCostFunction$: it takes two $CostFunctions$ as arguments, and creates a new $CostFunction$ by subtracting second input $CostFunction$ from the first one.
- $MinMerge$: It compares two cost functions for the entire memory range and at each memory point picks up the lower cost value.
- $OptMerge$: It optimally combines the $CostFunctions$ of the two operators/plans which run simultaneously. Given two $CostFunctions$ corresponding to two plans it divides the memory available between the two plans such that the combined execution cost is minimized and does this for all memory points from 0 and $MaxAvailMem$. This procedure has been described in Section 6.4.

6.6.3 Detailed Algorithm

Figure 6.5 shows the Memory Cognizant Volcano Search Algorithm $FindBestPlan$.

The function $FindBestPlan$ returns:

⁷This may increase the search space

```

FindBestPlan(LogExpr, PhysProp, CostLimitFunction)
  if the pair LogExpr and PhysProp is in the lookup table with
    Plan as the optimal plan /* optimized already, attempting reuse */
  if there exists a memory point i at which Plan is failed and
    its cost is < CostLimitFunction(i)
    goto Label X /* re-optimization required */
  else /* no re-optimization required */
    if for all memory points i, Plan is successful and
      its cost ≤ CostLimitFunction(i)
      return (SUCCESS, Plan)
    else if there exists a memory point i where Plan is successful and
      its cost ≤ CostLimitFunction(i)
      return (PARTIAL_SUCCESS, Plan)
    else /* for all memory points i, Plan cost > CostLimitFunction(i) */
      return FAILURE

else /* Optimization required */
Label X:
  (Result, Plan) =
    ApplyTransformations(LogExpr, PhysProp, CostLimitFunction)
  /* if plan returned, merge it with the planGroup optPlan */
  if Result ≠ FAILURE then
    optPlan = MinMerge(optPlan, Plan)
    CostLimitFunction = optPlan.CostFunction()

  (Result, Plan) = ApplyAlgorithms(LogExpr, PhysProp, CostLimitFunction)
  /* if plan returned, merge it with the planGroup optPlan */
  if Result ≠ FAILURE then
    optPlan = MinMerge(optPlan, Plan)
    CostLimitFunction = optPlan.CostFunction()

  (Result, Plan) = ApplyEnforcers(LogExpr, PhysProp, CostLimitFunction)
  /* if plan returned, merge it with the planGroup optPlan */
  if Result ≠ FAILURE then
    optPlan = MinMerge(optPlan, Plan)
    CostLimitFunction = optPlan.CostFunction()

  /* Maintain lookup table of explored (expression, physical property) pairs */
  if LogExpr is not in the lookup table
    insert LogExpr into the lookup table
  insert (LogExpr, PhysProp, optPlan) into lookup table

```

Figure 6.5: Memory Cognizant Volcano Search: *FindBestPlan* Algorithm

```

ApplyTransformations(LogExpr, PhysProp, CostLimitFunction)
  for each applicable transformation
    create NewLogExpr by applying the transformation
    (Result, Plan) =
      FindBestPlan(NewLogExpr, PhysProp, CostLimitFunction)
    /* if plan returned, merge it with the planGroup optPlan */
    if Result ≠ FAILURE then
      optPlan = Plan
      CostLimitFunction = Plan.CostFunction()

  if optPlan is successful for all memory points
    return (SUCCESS, optPlan)
  if optPlan is successful for no memory point
    return (FAILURE)
  return (PARTIAL_SUCCESS, optPlan)

```

Figure 6.6: Memory Cognizant Volcano Search: *ApplyTransformations* Algorithm

- *SUCCESS*: when the optimized plan *optPlan* for the $(LogExpr, PhysProp)$ pair to be optimized is s.t. $optPlan.CostFunction() \leq_{all} CostLimitFunction$.
- *FAILURE*: when the optimized plan *optPlan* for $(LogExpr, PhysProp)$ pair to be optimized is s.t. $optPlan.CostFunction() >_{all} CostLimitFunction$.
- *PARTIAL_SUCCESS*: when the optimized plan *optPlan* for the $(LogExpr, PhysProp)$ pair to be optimized is s.t. $\exists m, 1 \leq i \leq MaxAvailMem : optPlan.CostFunction()(m) \leq CostLimitFunction(m)$.

To optimize a given $(LogExpr, PhysProp)$ pair within a given *costLimitFunction*, if there has been no previous attempt to optimize this $(LogExpr, PhysProp)$ pair, the search algorithm proceeds as follows: It first applies transformation on the given logical expression to generate all equivalent logical expressions. Figure 6.6 shows pseudo code for the application of transformations. Then it recursively optimizes transformed $(LogExpr, PhysProp)$ pairs by applying each applicable operator (algorithm or enforcer) with the specified cost limit.

If there has been a previous attempt to optimize this $(LogExpr, PhysProp)$ pair then we have a plan and cost function available. The $(LogExpr, PhysProp)$ pair may have successful plan in some memory ranges and failures w.r.t the previous cost limit in some other memory ranges.

If the plan returned by the previous attempt has at least one point with failure indication and cost less than the cost limit then we need to re-optimize the $(LogExpr, PhysProp)$ pair.

Else, if for each memory point we have a successful plan and the cost limit is more than or equal to the cost of the plan at each point we return *SUCCESS* along with the plan. Else, if the plan has cost less than or equal to the cost limit at some point we return *PARTIAL_FAILURE* along with the plan. Else the plan has, at each memory point, cost greater than the cost limit and hence we return *FAILURE*.

Figure 6.7 and Figure 6.8 show the application of an algorithm and an enforcer resp. The application of an operator is done as follows: First we evaluate cost function of the operator. We need to optimize its children and for this we need to evaluate the child cost limit. As of now, we do not know exactly how much memory the tree rooted at this operator is going to take. For calculating the child cost limit, we assume that the tree will execute in *MaxAvailMem* memory units.

If an edge between the operator and the child is pipelined and the operator takes $MaxAvailMem - i$ units of memory, the child will take i units of memory. Thus the child cost limit for memory i , with pipelined edge, is calculated by subtracting the cost of the operator at memory point $MaxAvailMem - i$ from the cost limit of the plan at the memory point $MaxAvailMem$.

```

ApplyAlgorithms(LogExpr, PhysProp, CostLimitFunction)
  for each applicable Algorithm do
    if AlgorithmCostFunction >all CostLimitFunction
      continue /* cost of the operator is more than the cost limit */
    AlgoPlan.CostFunction = AlgorithmCostFunction
    for each input I of the algorithm
      for memory i = 1 to MaxAvailMem
        ChildCostLimit(i) = CostLimitFunction(MaxAvailMem) -
          Min(AlgorithmCostFunction(MaxAvailMem - i),
            AlgorithmCostFunction(MaxAvailMem) +
              CostOfResultIO(I))
        if  $\forall i : \text{ChildCostLimit}(i) < 0$ 
          break /* no plan within the given cost limit */

        determine required physical properties PP for I
        (Result, Plan) = FindBestPlan(I, PP, ChildCostLimit)
        if result = FAILURE
          break /* no plan within the given cost limit */

        for memory i = 1 to MaxAvailMem
          CostWithChildBlocked(i) = AlgoPlan.CostFunction(i)
            + Plan.CostFunction(MaxAvailMem)
            + CostOfResultIO(I)
          if an edge between the algorithm and the child I is pipeline edge
            /* divide memory optimally between the operator and its child */
            CostWithChildPipelined =
              OptMerge(AlgoPlan.CostFunction, Plan.CostFunction)
            /* consider blocking the edge between the operator and its child */
            AlgoPlan.CostFunction =
              MinMerge(CostWithChildPipelined, CostWithChildBlocked)
          else /* blocking edge, full memory is available to the input */
            AlgoPlan.CostFunction = CostWithChildBlocked

        /* Merge the operator plan with the planGroup optPlan */
        optPlan = MinMerge(optPlan, AlgoPlan)
        CostLimitFunction = optPlan.CostFunction()

    if optPlan is successful for all memory points
      return (SUCCESS, optPlan)
    if optPlan is successful for no memory point
      return (FAILURE)
    return (PARTIAL_SUCCESS, optPlan)

```

Figure 6.7: Memory Cognizant Volcano Search: *ApplyAlgorithms* Algorithm

```

ApplyEnforcers(LogExpr, PhysProp, CostLimitFunction)
  for each applicable Enforcer do
    if EnforcerCostFunction >all CostLimitFunction
      /* cost of the enforcer is more than the cost limit */
      continue

    EnforcerPlan.CostFunction = EnforcerCostFunction
    modify PhysProp to ModifiedPhysProp MPP for enforced property
    (Result, Plan) = FindBestPlan(LogExpr, MPP
      CostLimitFunction - EnforcerPlan.CostFunction)
    if result = FAILURE
      /* no plan within the given cost limit */
      continue

    if an edge between the enforcer and its child is pipeline edge
      /* divide memory optimally between the enforcer and its child */
      EnforcerPlan.CostFunction =
        OptMerge(EnforcerCostFunction, Plan.CostFunction)

      /* consider blocking the edge between the enforcer and its child */
      for memory i = 1 to MaxAvailMem
        ChildBlockingCost = EnforcerCostFunction(i)
          + Plan.CostFunction(MaxAvailMem)
          + CostOfResultIO(LogExpr)
        if EnforcerPlan.CostFunction(i) > ChildBlockingCost
          /* convert the pipeline edge to blocking edge */
          EnforcerPlan.CostFunction(i) = ChildBlockingCost
        else /* blocking edge; full memory is available to the input */
          for memory i = 1 to MaxAvailMem
            EnforcerPlan.CostFunction(i) = EnforcerCostFunction(i)
              + Plan.CostFunction(MaxAvailMem)
              + CostOfResultIO(LogExpr)

      /* merge the enforcer plan with the planGroup optPlan */
      optPlan = MinMerge(optPlan, EnforcerPlan)
      CostLimitFunction = optPlan.CostFunction()

  if optPlan is successful for all memory points
    return (SUCCESS, optPlan)
  if optPlan is successful for no memory point
    return (FAILURE)
  return (PARTIAL_SUCCESS, optPlan)

```

Figure 6.8: Memory Cognizant Volcano Search: *ApplyEnforcers* Algorithm

If an edge between the operator and the child is blocked the child cost limit is calculated by subtracting the cost of the operator running in memory *MaxAvailMem* and the materialization cost at the edge.

The cost limit passed to the child is the maximum of the two child cost limits described above at each memory point. After optimizing each child, we merge the cost of the child with that of the plan cost and this is used as the operator cost to calculate the cost limit of the next child to be optimized.

If the child optimization returns *FAILURE* (i.e., within the given cost limit, the optimizer could not find a plan even for a single memory point) then for the given (*LogExpr*, *PhysProp*) pair there exists no plan within the given cost limit for any memory points, and the optimizer returns *FAILURE*. Instead, if the child optimization returns a plan for even a single memory point with the given cost limit, the optimization continues.

Finally when the optimization is over, if the optimizer could find plans within the given cost limit for all memory points, it returns *SUCCESS*. If it could find plans within the given cost limit for some memory points but not for all of them, then it returns *PARTIAL_SUCCESS*. The cost function will have multiple segments. If for a memory range we get a successful plan then within that range the cost function will indicate success along with the plan. If no successful plan is found in a memory range failure will be indicated in that range along with the cost limit. If no plan is found within the given cost limit at any of the points in the given memory range then it returns *FAILURE*.

6.7 Experimental Evaluation

In this section we describe our experimental setup and the results obtained.

The 1PO and 2PO algorithms are based on the Volcano query optimization algorithm. The first phase of 2PO (which uses a query optimizer based on the Volcano optimization algorithm to optimize the query in the conventional manner) assumes that each operator in the plan uses all available memory. We have implemented all the schemes of each operator defined in Section 6.3, except the hybrid hash join variant of the first scheme for the hash join operator.

The memory block size is taken as 4K. Standard techniques are used for estimating costs, using statistics about relations. The cost estimates contain an I/O component and a CPU component. The metric used to compare the goodness of the optimization algorithms is the estimated cost of the optimal plan produced by the optimizer; all our cost numbers are estimates from the optimizer.

The tests are performed on a Sun workstation with UltraSparc 10 333Mhz processor, 256MB main memory, running Solaris 5.7.

Test Queries

We tested our algorithms with around 20,000 randomly generated queries on a TPCD-based star schema similar to the one proposed by Scheuermann [32]. The schema has a central *orders* fact table, and four dimension tables *part*, *supplier*, *customer* and *time*. The size of each of these tables is the same as that in the TPCD-1 database. This corresponds to base data size of approximately 1 GB. Each generated query is of the form:

```
select sum(quantity)
from orders, supplier, part, customer, time
where join-list and select-list
group by groupby-list;
```

The *join-list* enforces equality between attributes of the order fact table and primary keys of the dimension tables. The *select-list*, i.e., the predicates for the selects are generated by selecting some attributes at random from the join result, and creating random equality or inequality predicates on the attributes. The *groupby-list* is generated by picking a subset of {*custkey*, *suppkey*, *partkey*, *custkey*, *month*, *year*} at random.

We randomly choose, between 10 blocks to 10,000 blocks, the total memory available to the execution engine and this forms a part of the input to the optimizer.

Experimental Results

We tested total 23,603 randomly generated queries and the performance benefit of 1PO over 2PO is reported below⁸:

Cost Reduction of 1PO over 2PO	#Queries	%Queries
00-10 %	22682	96.097
10-20 %	57	0.241
20-30 %	527	2.232
30-40 %	238	1.001
40-50 %	99	0.419

The maximum cost reduction reported by 1PO over 2PO in our experiments is 50%. For around 96% of the queries reduction is between 0% to 10%, and for only 4% of the queries reduction is between 10% to 50%. Thus, for the class of queries we considered, 1PO gives benefits, but generally 2PO performs about as well as 1PO.

The average optimization time taken by 2PO and 1PO is shown in the table below:

Algorithm	Optimization Time (msec)
2PO	150
1PO	1110

The cost based pruning feature of Volcano is not implemented in 1PO algorithm and 1PO explores full search space. Whereas, 2PO uses standard volcano implementation in its first phase and hence includes cost based pruning.

6.8 Summary

In this chapter we presented our memory cognizant query optimization which addresses the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator. We have extended the Volcano optimizer to make it memory cognizant. We have designed efficient techniques to divide available memory optimally among operators in a pipeline. If done naively, this process is impractical. We showed how to improve optimization time by using piecewise linear approximations of the cost-versus-memory functions of various operators and this made evaluation of 1PO feasible. Though parametric query optimization and memory cognizant query optimization seem to be dissimilar problems, there are similarities in the techniques we used to solve these problems.

It has been conjectured that 1PO will perform no better than 2PO, but there has been no published evidence of this claim. We designed a practical cost-based algorithm for 1PO and compared it against 2PO. For the class of queries we considered, 1PO gives benefits, but generally 2PO performs about as well as 1PO. Thus, the preliminary results indicate that using 1PO for query optimization may not be beneficial. This is a good news in general as the optimizer remains simpler and faster. The techniques developed here are of independent interest and can very well be applied to other problems.

⁸Since we are using strictly cost-based exhaustive exploration of the search space, 1PO will never miss a 2PO plan, and hence is at least as cheap as 2PO.

Chapter 7

Conclusions

In this thesis, we presented various solutions for parametric query optimization problem depending upon the type of the plan cost functions, namely linear, piecewise linear and nonlinear. In memory cognizant query optimization, we address the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator.

In Chapter 3, we proposed a parametric query optimization algorithm for the case when the cost functions are linear in the given parameters. The solution works for arbitrary number of parameters and is minimally intrusive in the sense that an existing query optimizer can be used as a subroutine with minor modifications. The solution invokes a conventional query optimizer multiple times, with different parameter values. 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. Though the cost functions are seldom linear in real life and the results presented in this chapter are theoretical, we use the results later in Chapter 5 where we develop AniPQO – a heuristic PQO solution for the case when the cost functions are nonlinear and discontinuous.

In Chapter 4, we proposed a solution for the parametric query optimization problem for the case when the cost functions are piecewise linear in the given parameters. The solution is exact if the cost functions are piecewise linear but intrusive and needs extensions to the conventional optimizer. We outline how to extend the System R and Volcano query optimization algorithms to handle piecewise linear cost functions in place of cost values. We have implemented the extensions by modifying an existing query optimizer based on the Volcano optimization algorithm, developed at IIT Bombay and tested the extensions for queries with two parameters. Although the solution conceptually works for arbitrary number of parameters, the cost of optimization increases exponentially with the number of parameters.

In Chapter 5, we presented AniPQO – a heuristic solution for the PQO problem for the general case when the cost functions may be nonlinear in the given parameters. 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. 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 a set of plans found to boost the quality of the results and facilitate picking an optimal plan at run time.

We have implemented the AniPQO algorithm and presented 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 than the actual optimal plan; the maximum performance degradation observed on a sample set of queries is very small (3.5%). Although the optimization 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 practical for up to 4 parameters. We also showed how AniPQO can degrade gracefully and provide a slightly inferior solution with significant reduction in the cost of optimization. If we consider either only query parameters (as we did in the performance evaluation) or only system parameters, the limit of four on the number of parameters seems reasonable; but if we consider both, the number of parameters may exceed four. Evaluating AniPQO with system parameters is left as a future work.

In Chapter 6, we presented our memory cognizant query optimization technique which addresses the problem of choosing an optimal, memory-division-aware execution plan for a query, given the *cost versus memory allocation* function for each operator. We extended the Volcano optimizer to make it memory cognizant. We designed efficient techniques to divide available memory optimally among operators in a pipeline. If done naively, this process is impractical. We showed how to improve optimization time by using piecewise linear approximations of the cost-versus-memory functions of various operators and this made evaluation of 1PO feasible. Though parametric query optimization and memory cognizant query optimization seem to be dissimilar problems, there are similarities in the techniques we used to solve these problems.

It has been conjectured that 1PO will perform no better than 2PO, but there has been no published evidence of this claim. We designed a practical cost-based algorithm for 1PO and compared it against 2PO. For the class of queries we considered, 1PO gives benefits, but generally 2PO performs about as well as 1PO. Thus, the preliminary results indicate that using 1PO for query optimization may not be beneficial. This is a good news in general as the optimizer remains simpler and faster. The techniques developed here are of independent interest and can very well be applied to other problems.

Appendix A

Parametric Optimal Set of Plans (*POSP*): An Example

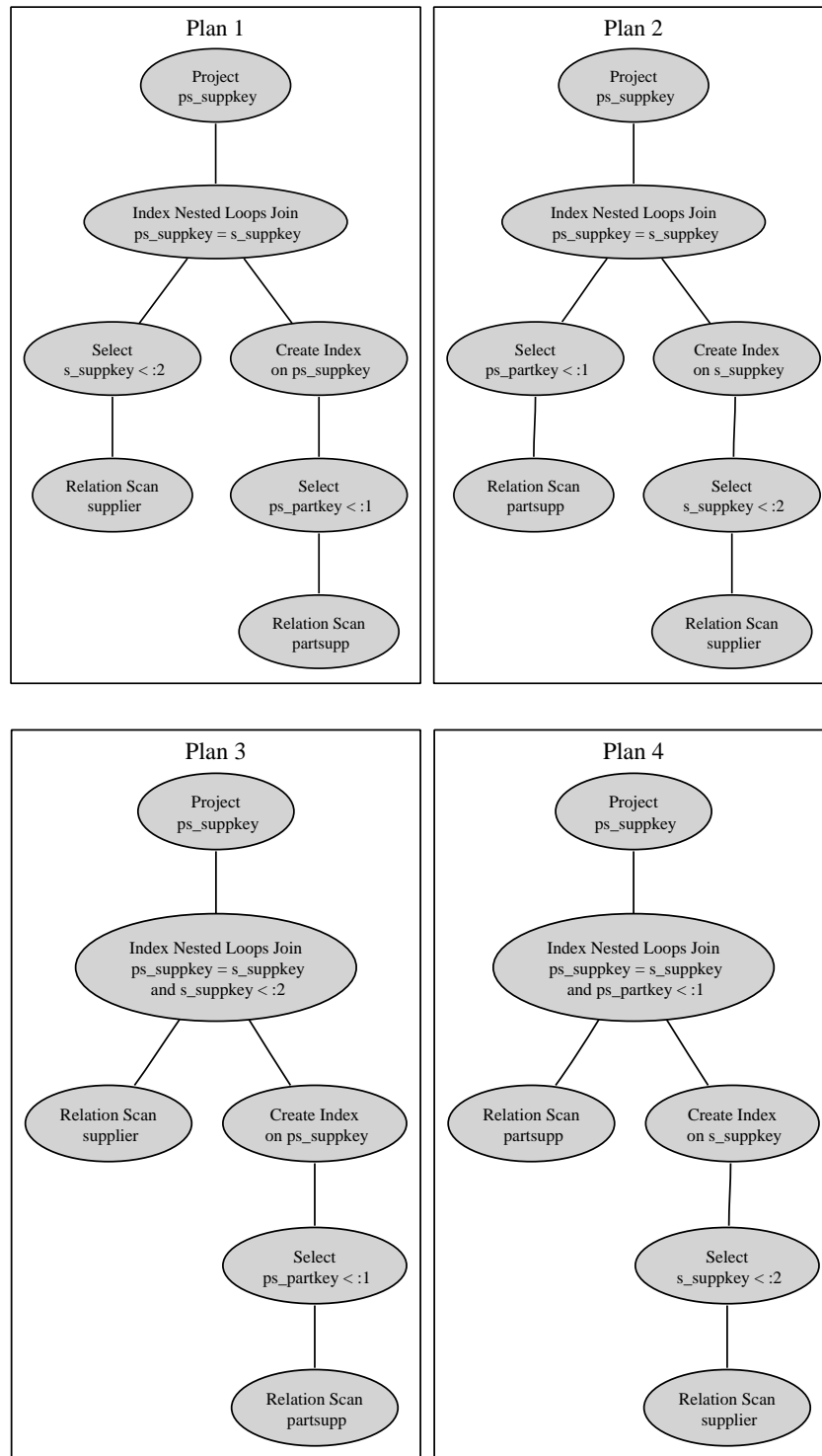
In this appendix we present the *POSP* for query R2P2 from Section 5.4. We have already presented the corresponding parameter space decomposition, as an example, in Figure 1.1, and present it here again. The *POSP* has ten plans and the corresponding ten regions are marked from *R1* to *R10* in the decomposition.

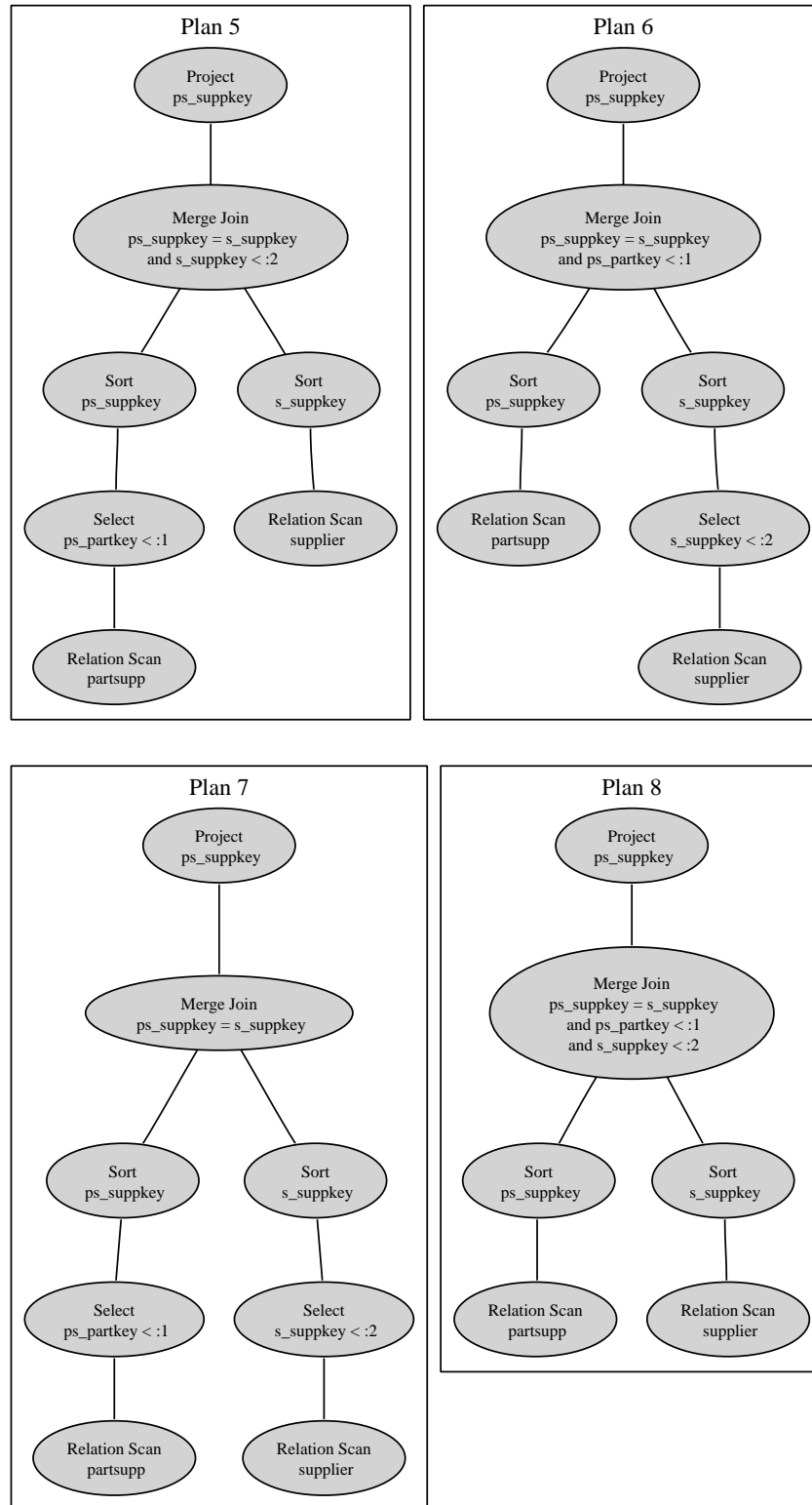
The physical operators that the optimizer considers for evaluating a join operation are nested loops join, indexed nested loops join and merge join; whereas, the physical operators that it considers for evaluating a select operation are select scan and indexed select. See Section 5.4 for the details of the optimizer and the mechanism we used to generate *POSP* and parameter space decomposition for a given query.

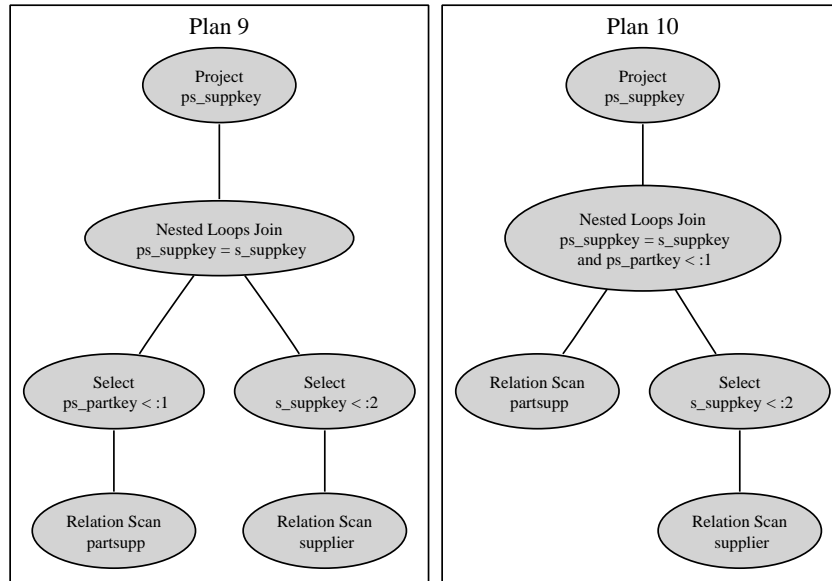
The query R2P2 is:

```
select partsupp.ps_suppkey
from partsupp, supplier
where partsupp.ps_suppkey = supplier.s_suppkey
and ps_partkey < :1
and s_suppkey < :2
where, :1 and :2 are the parameters.
```

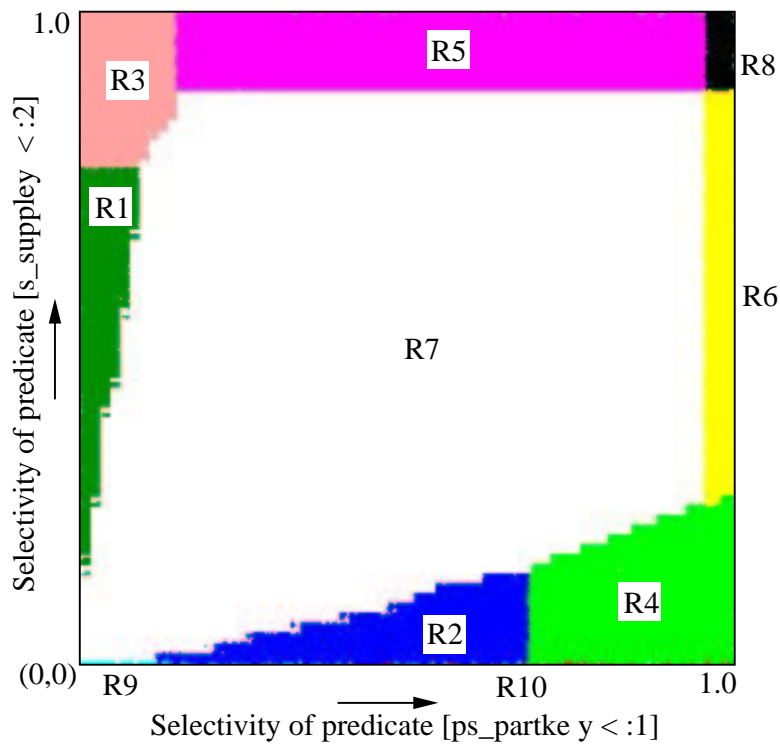
The *POSP* is listed below:







The parameter space decomposition is shown below:



Appendix B

Polytope Construction

In this appendix, we describe an efficient on-line algorithm for constructing high-dimensional polytopes from Mulmuley [25]. We then show how to adapt the algorithm to construct the cost polytope (defined in Chapter 3 to solve the PQO problem with linear cost functions).

B.1 Polytope Construction

In this section we describe a randomized incremental algorithm from Mulmuley [25] for constructing the facial lattice of a d -polytope in $O(n^{\lfloor d/2 \rfloor})$ expected time. Let N be a set of n half-spaces in \mathfrak{R}^d . Let $H(N)$ be the convex polytope formed by intersecting these half-spaces. Our goal is to construct the facial lattice of this polytope. It suffices to construct the 2-skeleton, because it can be easily extended to the full facial lattice (Mulmuley [25]). We shall add the half-spaces in N , one at a time, in random order. Let N^i be the set of first i half-spaces. We assume that $H(N^i)$ is bounded. We can ensure this using a *bounding box* trick: We add to N^0 , and hence to every N^i , half-spaces enclosing a huge symbolic box approaching infinity.

At the i -stage of the algorithm, we maintain the 2-skeleton of the polytope $H(N^i)$. Consider the addition of the $(i + 1)$ th half space $S = S_{i+1}$. We obtain $H(N^{i+1})$ from $H(N^i)$ by splitting off the *cap* which is defined as $\text{cap}(S_{i+1}) = H(N^i) \cap \overline{S}$. Here \overline{S} denotes the complement of S . Details of the operation are as follows: Let us say that an edge of $H(N^i)$ conflicts with S if it intersects the complement half-space \overline{S} . A conflicting vertex or 2-face of $H(N^i)$ is defined similarly. Assume for a moment that we are given some vertex $p \in H(N^i)$ in conflict with S (if any). If there is no such vertex, then S must be redundant, because $H(N^i)$ is bounded. In that case, S can be thrown away. Otherwise, we visit all conflicting edges and vertices of $H(N^i)$ by a search on the edge skeleton of $H(N^i)$ starting at p . During this search, we take care not to enter the half-space S at any time. This search works because the conflicting edges and vertices of $H(N^i)$ form a connected subgraph of the edge skeleton of $H(N^i)$. The latter fact holds because the convex polytope $H(N^i) \cap \overline{S}$ is bounded.

At the end of the above search, we know all the conflicting edges and vertices of $H(N^i)$. We also know its conflicting 2-faces because they are adjacent to the conflicting edges. We remove the conflicting vertices, edges and 2-faces lying completely within \overline{S} . The remaining edges and 2-faces intersect S partially. They are split and only their intersections with S are retained. We also introduce new vertices and edges which correspond to their intersections with the bounding hyperplane ∂S .

It remains to determine the 2-faces in $H(N^i) \cap \partial S$. In the trivial three-dimensional case there is a unique such 2-face. In general, there can be many. But we already know the edge skeleton of $H(N^i) \cap \partial S$ at this stage. So we can extend it to the 2-skeleton in linear time (Mulmuley [25]).

Figure B.1 (a) shows a polygon $abcdef$ in \mathfrak{R}^2 and Figure B.1 (b) shows its face lattice. (Figures B.1 (a) and (b) are the same as Figures 2.1 (a) and (b) respectively.) Figure B.1 (c) shows introduction a new halfspace h_8 to the polygon in Figure B.1 (a). Let a vertex in conflict with the halfspace be a . We traverse

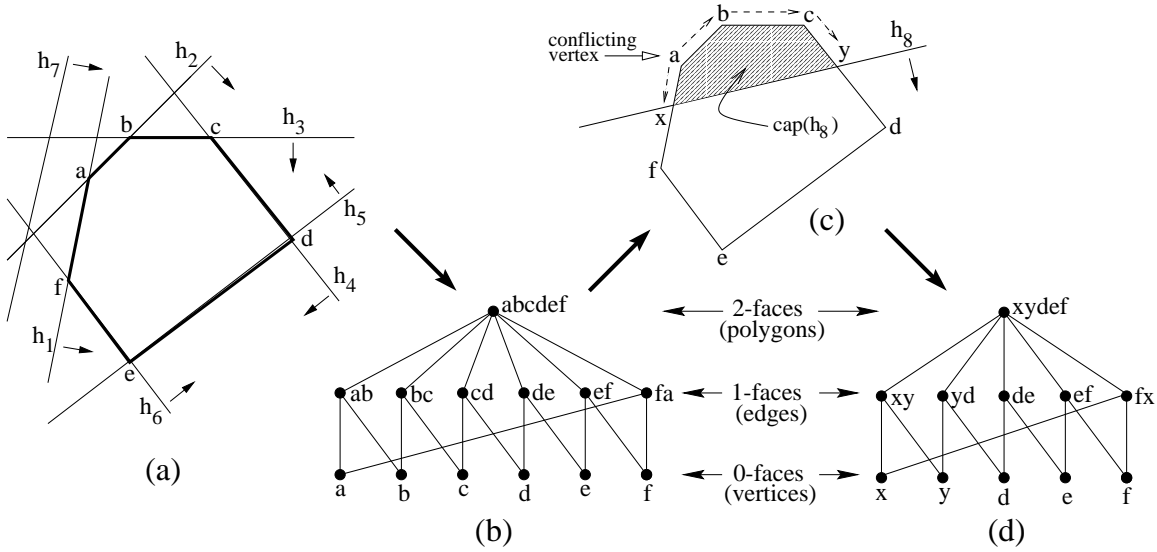


Figure B.1: Polytope: Definition, Representation and Construction

the facial lattice (Figure B.1 (b)) starting from a to identify all the vertices in conflict with the halfspace. They are a, b and c and we remove them from the lattice. Edges ab, af, bc and cd are in conflict with the halfspace. Of which, edges ab and bc are removed as they lie completely within $\overline{h_8}$. Remaining two edges intersect $\overline{h_8}$ partially. So they are split by creating two new vertices x and y . New edges xf and yd replace old edges af and cd resp. Finally we introduce a new edge xy . Figure B.1 (d) shows the new face lattice.

Conflict maintenance and history

To locate a vertex of $H(N)$ in conflict with S_{i+1} , at every time i , one maintains, for every half-space $I \in N \setminus N^i$, a pointer to one vertex in $H(N^i)$ in conflict with it, if there is one. During the addition of $S = S_{i+1}$, the conflict information is updated.

The algorithm presented is incremental but not on-line. This is because it maintains, at every stage, conflicts of the half-spaces not yet added. One can maintain an on-line structure that will let us do this job efficiently. The search structure at time i will be the history of previous i additions. We omit the details for brevity (details may be found in Mulmuley [25]).

B.2 Adapting the algorithm to construct cost polytope

We can use the above mechanism for the cost polytope algorithm proposed in Section 3.4 for parametric query optimization problem for linear cost functions but can make a few simplifications. We need not maintain the conflicts and the history of the hyperplane additions and this makes the algorithm simpler. In the original algorithm we need to maintain the conflict and the history structures to be able to identify a conflicting vertex for a hyperplane to be added. But in our algorithm we use a different approach: we find if a given vertex on the cost polytope is in conflict with any cost hyperplane using a conventional optimizer. We call the conventional optimizer at the vertex to know an optimal plan with its cost hyperplane. If the cost of the plan at the vertex is lower than that represented by the polytope, we have a hyperplane with which the vertex is conflicting and we intersect the hyperplane with the polytope. If the cost of the plan at the vertex is the same as that represented by the polytope then we ignore the hyperplane. In that case no hyperplane can intersect the cost polytope at the vertex and the vertex will be a vertex of the final cost polytope. When all the vertices of the polytope satisfy this property we get the final cost polytope.

Appendix C

Proofs for the Properties of Linear Cost Functions from Section 3.2

In this appendix we provide proofs for the properties of linear cost functions taken from Ganguly [7] and stated in Section 3.2. (They appear in [7] without proofs.)

Proof for Property 3.2.1: The property states that 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.

Consider a line segment xy in the parameter space with plan p optimal at points x and y . Suppose, there exists a point z on the line segment at which plan p is not optimal. Let plan p' be optimal at point z .

$$x - z - y \Rightarrow z = ax + by \text{ where }^1 a, b \in \mathbb{R}, 0 < a, b \text{ and } a + b = 1$$

$$cost_p(x) \leq cost_{p'}(x) \quad (p \text{ is optimal at } x) \quad (C.1)$$

$$cost_p(y) \leq cost_{p'}(y) \quad (p \text{ is optimal at } y) \quad (C.2)$$

$$\begin{aligned} cost_{p'}(z) &< cost_p(z) && (p \text{ is not optimal at } z) \\ &< cost_p(ax + by) && (z = ax + by) \\ &< a \cdot cost_p(x) + b \cdot cost_p(y) && (\text{cost functions are linear and } a + b = 1) \\ &< a \cdot cost_{p'}(x) + b \cdot cost_{p'}(y) && (\text{from C.1 and C.2, as } 0 < a, b) \\ &< cost_{p'}(ax + by) && (\text{cost functions are linear and } a + b = 1) \\ cost_{p'}(z) &< cost_{p'}(z) && (z = ax + by) \end{aligned}$$

A contradiction. □

Proof for Property 3.2.2: The property states that each plan in a *POSP* has only one region of optimality and, the region is a convex polytope.

We will divide the proof into two parts:

Part I: Each plan in *POSP* has only one region of optimality:

If a plan in *POSP* has two regions of optimality, we can come up with a line segment whose end points lie in different regions of optimality of the plan and there exists at least one point on the line segment which lies in the region of optimality of some other plan in *POSP*. This contradicts the above property.

Part II: The region of optimality of a plan in *POSP* is a convex polytope:

From Property 3.2.1, we know that the region of optimality of each plan in *POSP* is convex. Otherwise, if the region of optimality of a plan is non-convex then we can have a line segment whose end points lie in the region of optimality of the plan and there exists at least one point on the line segment which lies in the region of optimality of some other plan. We have assumed that the parameter space of interest is a convex

¹In a n dimensional parameter space, $x = (x_1, x_2, \dots, x_n)$ and $ax = (ax_1, ax_2, \dots, ax_n)$.

polytope and convex partitions of a convex polytope are convex polytopes. \square

Proof for Property 3.2.3: The property states 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 prove this by induction on dimension d of the polytope. The result is true for $d = 1$ by the first property. Assume that the result is true for dimensions $d < n$. Consider a n -dimensional polytope P . Let plan p be optimal at each of its vertices. All the facets of polytope P are polytopes in lower dimensions with plan p optimal at each of their vertices. Hence by inductive hypothesis plan p is optimal at each point within each facet of polytope P . Consider a point x in polytope P . Any line passing through point x will intersect polytope P in two of its facets. Let the intersection points be y and z . We have $y - x - z$. By inductive hypotheses plan p is optimal at points y and z . By the first property it is optimal at point x also. Hence the proof. \square

Appendix D

Relaxing the assumption from Section 5.2.1

In this appendix we define a set of conditions under which the assumption on page 41 from Section 5.2.1 in Chapter 5 is true. This appendix is written with inputs from Sohoni [34]. We referred a standard text on Differential Topology by Milnor [24].

Original 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.

In general the assumption is not true and even simple cases depicted in Figures D.1 (a) and (b) contradict the assumption. (Figures D.1 (a) and (b) are the same as Figures 5.3 (a) and (b) respectively.) In Figure D.1 (a) the cost functions intersect at two points and the points are separated. In Figure D.1 (b) the cost functions intersect at infinite number of points and the intersection points are continuous.

Figure D.1 (c) shows how a slight vertical shift in one of the functions in Figure D.1 (b) results in reducing the number of intersection points to one. We can perturb cost functions slightly so that the intersection points are not continuous. But bringing down the number of intersection points to one without introducing large errors in the functions is not easy in general, as is the case in Figure D.1 (a). Note that both the functions in this example are continuous and non-increasing but this does not prevent them from intersecting at more than one point.

We show that under the following conditions, we can perturb the functions slightly so that the number of intersection points for each subset of size $n + 1$ of cost functions in n parameters is finite:

- **Condition 1:** The parameter space polytope is closed and bounded.
- **Condition 2:** The cost functions are continuous.

We explain the rationale behind the conditions in the rest of the appendix. Before that, we define a few terms.

Discrete set: A set \mathcal{Z} of points in \mathbb{R}^n is discrete if for each point $x \in \mathcal{Z}$, we can define an open ball B_x of non-zero radius *s.t.* $B_x \cap \{\mathcal{Z} - x\} = \emptyset$; *i.e.* no other point from the set is in the ball.

Tangent space: The tangent space to a function (or surface) at a point on it is the space orthogonal to the normal to the function at that point. That is, the tangent space contains all the vectors that are perpendicular to the normal. The tangent space at a point is the hyperplane that best approximates the function at the point. We denote the tangent space to function f at point p as \mathcal{T}_f^p .

Transversal intersection: A set of functions $\mathbb{R}^n \rightarrow \mathbb{R}$ intersect transversally over a given domain, if at each intersection point in the domain, their tangent spaces together span \mathbb{R}^{n+1} . For example, functions $\{f_1, f_2, f_3, \dots\}$, where $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, intersect transversally, if at each of their intersection points $p \in \mathbb{R}^{n+1}$, $\sum_i \mathcal{T}_{f_i}^p = \{\sum v_i : v_i \in \mathcal{T}_{f_i}^p\} = \mathbb{R}^{n+1}$, *i.e.* $\mathcal{T}_{f_i}^p$'s together span \mathbb{R}^{n+1} . The functions in Figures D.1 (a) and (c) intersect transversally whereas the function in Figure D.1 (b) does not. Figure D.2

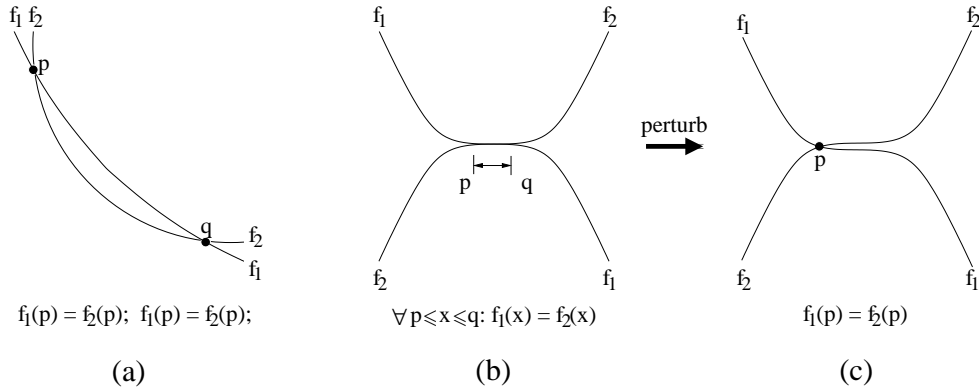


Figure D.1: (a), (b) Counter-examples to the assumption; (c) Perturbing functions in (b)

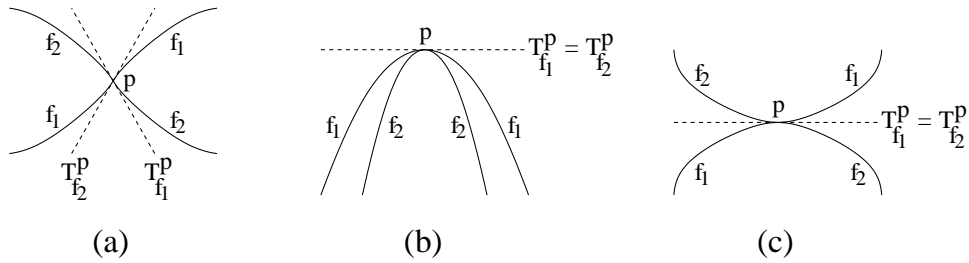


Figure D.2: Examples of (a) transversal and (b), (c) non-transversal intersections

shows more examples of transversal and non-transversal intersections in \mathbb{R}^2 with the tangent spaces of the functions at the respective intersection points.

As each cost function is a function $\mathbb{R}^n \rightarrow \mathbb{R}$ for n parameters, we can rephrase the assumption as follows.

Rephrased assumption: If a set, P , where $n < |P|$, of functions $\mathbb{R}^n \rightarrow \mathbb{R}$ is intersecting at a point in \mathbb{R}^{n+1} then no $P' \subseteq P$, where $n < |P'|$, is intersecting at any other point in \mathbb{R}^{n+1} .

Consider a set of functions $F = \{f_1, f_2, f_3, \dots, f_m\}$ where $\forall 1 \leq i \leq m, f_i : \mathcal{D} \rightarrow \mathbb{R}, \mathcal{D} \subseteq \mathbb{R}^n$ and $n < m$.

$$\text{Let } \forall P_F \subseteq F : \mathcal{Z}_{P_F} = \{x \in \mathcal{D} \mid \forall f_i, f_j \in P_F : f_i(x) = f_j(x)\}$$

\mathcal{Z}_{P_F} is a set of point at which the functions in set P_F are intersecting. In general, P_F is non-transversal and hence \mathcal{Z}_{P_F} with $|P_F| = n + 1$ may be non-discrete. We can make the functions transversal by slightly perturbing them as follows if they are continuous:

Given $0 < \epsilon$, we define an ϵ -**envelope** of F as a set of functions $G = \{g_1, g_2, g_3, \dots, g_m\}$ s.t. $\forall 1 \leq i \leq m, \forall x \in \mathcal{D} : |f_i(x) - g_i(x)| < \epsilon$

$$\text{Let } \forall P_G \subseteq G : \mathcal{Z}_{P_G} = \{x \in \mathcal{D} \mid \forall g_i, g_j \in P_G : g_i(x) = g_j(x)\}$$

For a given set F of continuous functions, we can choose a transversal G s.t., (a) if $|P_G| = n + 1$ then \mathcal{Z}_{P_G} is discrete and (b) if $n + 1 < |P_G|$ then $\mathcal{Z}_{P_G} = \emptyset$. (The cost functions don't actually need to obey part (b).)

If \mathcal{D} is a closed and bounded domain then for P_G with $|P_G| = n + 1$, \mathcal{Z}_{P_G} is finite also.

Thus, if the functions are continuous over the domain, a slight perturbation will make the functions transversal and then (a) the set of intersection points for each $n + 1$ size subset of the functions is discrete and (b) no subset of size greater than $n + 1$ intersect. Furthermore, if the domain is closed and bounded then the number of intersection points of each $n + 1$ size subset of the functions is finite.

The parameter space polytope is a closed convex polytope and hence is closed and bounded.

The cost functions are generally not continuous and typically involve discontinuities¹. We can separate the continuous parts of a discontinuous function and treat them as individual functions. We can extend each part suitably such that it is defined over the entire parameter space polytope and is continuous over it. Let us call the new cost functions as extended cost functions. The extended cost functions are thus continuous.

The extended cost functions, defined over a closed and bounded parameter space polytope, satisfy the above two conditions and thus the number of equi-cost points of $n + 1$ extended cost functions is finite. Note that an intersection point of a given set of extended cost functions may be on extended parts of some of the cost functions and hence need to be excluded while counting the number of cost functions intersecting at that point.

Further, one can decompose \mathcal{D} *s.t.* within each decomposed domain, for each $P_G \subseteq G$ with $|P_G| = n + 1$, \mathcal{Z}_{P_G} is either empty or singleton.

¹A typical example is the IO cost function of a sort operator with memory as a parameter. The IO cost is zero if the memory is big enough to hold the entire relation; otherwise the operation involves IO and the IO cost is non-zero. Thus, the IO cost function has a discontinuity at a point where the memory size is equal to the relation size.

Appendix E

Approximating the parameter space decomposition

In this appendix we devise a scheme to find the decomposition of parameter space induced by the *POSP* or its subset. The decomposition is based on an assumption we state below. We essentially divide the parameter space hyper-rectangle into smaller hyper-rectangles and construct a hierarchical index on the hyper-rectangles. We recursively divide a rectangle until a single plan is optimal throughout the rectangle or the cost of a plan is within some threshold of the optimal cost at any point in the rectangle.

Notations: Let the cost of plan p at vertex v be denoted by C_v^p . a - b - c means three points a , b and c are co-linear and point b lies in between points a and c on the line segment joining points a and c . Let p_a be an optimal plan at point a in the parameter space.

Assumption: If a - b - c for three points a , b and c in the parameter space and $C_a^p \leq C_c^p$ for plan p , then $C_a^p \leq C_b^p \leq C_c^p$.

Thus the cost of a plan is either non-decreasing or non-increasing in a given direction on a line in the parameter space. That is, when we move from one end to the other end of a line segment in the parameter space, cost of a plan either decreases, increases or remains constant; and does not increase first and then decrease or vice-versa.

We state following obvious lemma which follows directly from the above assumption.

Lemma E.1.1 *If a - b - c for three points a , b and c in the parameter space and the above assumption holds then for any plan p in the plan space*

$$\min(C_a^p, C_c^p) \leq C_b^p \leq \max(C_a^p, C_c^p)$$

□

A line segment is a polytope in one dimension and the above lemma can be generalized to be applicable to polytopes in any dimension.

Theorem E.1.1 *For a polytope R in the parameter space, any point $b \in R$ and any plan p in the plan space, if the above assumption holds,*

$$\min_{v \in \mathcal{V}_R} C_v^p \leq C_b^p \leq \max_{v \in \mathcal{V}_R} C_v^p$$

Proof: We prove this by induction on dimension d of the polytope. The theorem is proved for $d = 1$ in the above lemma. Assume that the theorem is true for dimensions $d < n$. Consider a n -dimensional polytope R .

W.l.o.g., let $a \in \mathcal{V}_R$ be a point s.t. $\forall v \in \mathcal{V}_R, C_a^p \leq C_v^p$; thus,

$$C_a^p = \min_{v \in \mathcal{V}_R} C_v^p$$

Let the line passing through a and b intersect the boundary of polytope R into a point c ; thus a - b - c .

Let R' be the facet¹ of polytope R s.t. $c \in R'$. The dimension of R' is less than n (the dimension of R).

By inductive hypothesis,

$$\min_{v \in \mathcal{V}_{R'}} C_v^p \leq C_b^p \leq \max_{v \in \mathcal{V}_{R'}} C_v^p$$

But $\mathcal{V}_{R'} \subseteq \mathcal{V}_R$ and hence,

$$C_a^p = \min_{v \in \mathcal{V}_R} C_v^p \leq C_c^p \leq \max_{v \in \mathcal{V}_R} C_v^p$$

As a - b - c , we have $C_a^p \leq C_b^p \leq C_c^p$; and hence,

$$C_a^p = \min_{v \in \mathcal{V}_R} C_v^p \leq C_b^p \leq C_c^p \leq \max_{v \in \mathcal{V}_R} C_v^p$$

□

The following lemma defines a lower and an upper bound on the cost of an optimal plan at any point on the line segment in terms of the costs of the optimal plans at its end-points.

Lemma E.1.2 *If a - b - c for three points a , b and c in the parameter space and the above assumption holds then,*

$$\min(C_a^{p_a}, C_c^{p_c}) \leq C_b^{p_b} \leq \min(\max(C_a^{p_a}, C_c^{p_c}), \max(C_a^{p_c}, C_c^{p_a}))$$

Proof: We will divide the proof in two parts.

Part I: $\min(C_a^{p_a}, C_c^{p_c}) \leq C_b^{p_b}$

We will prove this by contradiction. Assume $C_b^{p_b} < \min(C_a^{p_a}, C_c^{p_c})$.

$$C_b^{p_b} < C_a^{p_a} \quad (\text{assumption}) \quad (\text{E.1})$$

$$C_a^{p_a} \leq C_a^{p_b} \quad (\text{as } p_a \text{ is optimal at } a) \quad (\text{E.2})$$

$$C_b^{p_b} < C_a^{p_b} \quad (\text{from E.1 and E.2}) \quad (\text{E.3})$$

$$C_b^{p_b} < C_c^{p_c} \quad (\text{assumption}) \quad (\text{E.4})$$

$$C_c^{p_c} \leq C_c^{p_b} \quad (\text{as } p_c \text{ is optimal at } c) \quad (\text{E.5})$$

$$C_b^{p_b} < C_c^{p_b} \quad (\text{from E.4 and E.5}) \quad (\text{E.6})$$

Equations E.3 and E.6 together contradict the assumption made in the beginning of the section. .

Part II: $C_b^{p_b} \leq \min(\max(C_a^{p_a}, C_c^{p_a}), \max(C_a^{p_c}, C_c^{p_c}))$

$$C_b^{p_a} \leq \max(C_a^{p_a}, C_c^{p_a}) \quad (\text{by Lemma E.1.1}) \quad (\text{E.7})$$

$$C_b^{p_c} \leq \max(C_a^{p_c}, C_c^{p_c}) \quad (\text{by Lemma E.1.1}) \quad (\text{E.8})$$

$$C_b^{p_b} \leq C_b^{p_a} \quad (\text{as } p_b \text{ is optimal at } b) \quad (\text{E.9})$$

$$C_b^{p_b} \leq C_b^{p_c} \quad (\text{as } p_b \text{ is optimal at } b) \quad (\text{E.10})$$

The proof follows. □

A line segment is a hyper-rectangle in one dimension and the above lemma can be generalized to be applicable to hyper-rectangles in any dimension.

Theorem E.1.2 *For a polytope R in the parameter space and any point $b \in R$, if the above assumption holds,*

$$\min_{v \in \mathcal{V}_R} C_v^p \leq C_b^{p_b} \leq \min_{p \in \mathcal{U}_R^p} \max_{v \in \mathcal{V}_R} C_v^p$$

¹If there are more than one such facets, we pick any one of them.

Proof: We prove this by induction on dimension d of the polytope. The theorem is proved for $d = 1$ in the above lemma. Assume that the theorem is true for dimensions $d < n$. Consider a n -dimensional polytope R .

Part I: $\min_{v \in \mathcal{V}_R} C_v^{p_v} \leq C_b^{p_b}$

We prove this by contradiction.

Assume $C_b^{p_b} < \min_{v \in \mathcal{V}_R} C_v^{p_v}$; i.e., $\forall v \in \mathcal{V}_R : C_b^{p_b} < C_v^{p_v}$.

W.l.o.g., let $a \in \mathcal{V}_R$ be a point s.t.

$$\forall v \in \mathcal{V}_R : C_a^{p_a} \leq C_v^{p_v} \quad \text{i.e.} \quad C_a^{p_a} = \min_{v \in \mathcal{V}_R} C_v^{p_v} \quad (\text{E.11})$$

$$C_b^{p_b} < C_a^{p_a} \quad (\text{assumption}) \quad (\text{E.12})$$

$$C_a^{p_a} \leq C_a^{p_b} \quad (\text{as } p_a \text{ is optimal at } a) \quad (\text{E.13})$$

$$C_b^{p_b} < C_a^{p_b} \quad (\text{from E.12 and E.13}) \quad (\text{E.14})$$

Any line passing through a and b will intersect the boundary of the polytope R into a point, say c , and a - b - c . Let R' be the facet of the polytope R s.t. $c \in R'$. The dimension of R' is less than n (the dimension of R).

$$\min_{v \in \mathcal{V}_{R'}} C_v^{p_v} \leq C_c^{p_c} \quad (\text{by inductive hypothesis}) \quad (\text{E.15})$$

$$\min_{v \in \mathcal{V}_R} C_v^{p_v} \leq C_c^{p_c} \quad (\text{as } \mathcal{V}_{R'} \subseteq \mathcal{V}_R) \quad (\text{E.16})$$

$$C_a^{p_a} \leq C_c^{p_c} \quad (\text{from E.11}) \quad (\text{E.17})$$

$$C_c^{p_c} \leq C_c^{p_b} \quad (\text{As } p_c \text{ is optimal at } c) \quad (\text{E.18})$$

$$C_a^{p_a} \leq C_c^{p_b} \quad (\text{from E.17 and E.18}) \quad (\text{E.19})$$

$$C_b^{p_b} < C_a^{p_a} \quad (\text{assumption}) \quad (\text{E.20})$$

$$C_b^{p_b} < C_c^{p_b} \quad (\text{from E.19 and E.20}) \quad (\text{E.21})$$

Equations E.14 and E.21 together contradict the assumption made in the beginning of the section. .

Part II: $C_b^{p_b} \leq \min_{p \in \mathcal{U}_R^P} \max_{v \in \mathcal{V}_R} C_v^p$

By Theorem E.1.1, we have,

$$\forall p \in \mathcal{U}_R^P : C_b^p \leq \max_{v \in \mathcal{V}_R} C_v^p$$

As p_b is optimal at b , we have $\forall p \in \mathcal{U}_R^P : C_b^{p_b} \leq C_b^p$; and hence,

$$\forall p \in \mathcal{U}_R^P : C_b^{p_b} \leq \max_{v \in \mathcal{V}_R} C_v^p$$

Thus,

$$C_b^{p_b} \leq \min_{p \in \mathcal{U}_R^P} \max_{v \in \mathcal{V}_R} C_v^p$$

□

The result defines a lower and an upper bound on the cost of an optimal plan at any point in a hyper-rectangle in terms of the costs of the optimal plans at its vertices. If the difference between the two is small (within a threshold), we choose not to decompose the hyper-rectangle further. The following corollary follows.

Corollary E.1.1 *A plan p is not optimal at any point in a hyper-rectangle R if the above assumption holds and*

$$\min_{p' \in \mathcal{U}_R^P} \max_{v \in \mathcal{V}_R} C_v^{p'} < \min_{v \in \mathcal{V}_R} C_v^p$$

□

```

/* Finds decomposition of the parameter space hyper-rectangle
   induced by plan set  $P = POSP$  */
Let  $\mathcal{V}_R$  be the set of vertices of hyper-rectangle  $R$  and let  $\mathcal{H}_R$  be the dimension of  $R$ 
 $RSet = \{\text{parameter space hyper-rectangle}\}$ 
 $Hierarchical-Index = \emptyset$ 

while  $RSet \neq \emptyset$ 
   $R = RSet.RemoveEntry()$ 

  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 atleast one vertex of  $R$ . */

  Let  $minOptCost = \min_{v \in \mathcal{V}_R} C_v^{p^*}$ 
  Let  $maxOptCost = \max_{p \in \mathcal{U}_R^P} \max_{v \in \mathcal{V}_R} C_v^p$ 
  /* Optimal cost at any point in  $R$  is within  $minOptCost$  and  $maxOptCost$ ;
     see Theorem E.1.2 */
  Let  $diff = maxOptCost - minOptCost$ 

  If  $|\mathcal{U}_R^P| = 1$  with  $\mathcal{U}_R^P = \{p\}$  and  $maxOptCost \leq \min_{p' \in POSP \setminus p} \min_{v \in \mathcal{V}_R} C_v^{p'}$ 
    /* Plan  $p$  is optimal at all the points in  $R$ ; see Corollary E.1.1 */
    Insert  $R$  in  $Hierarchical-Index$  with  $p$  as optimal plan in it
  Else If  $diff \leq diffThreshold$ 
    Let  $p \in \mathcal{U}_R^P$  with  $\max_{v \in \mathcal{V}_R} C_v^p = maxOptCost$ 
    /* Worst case,  $p$  is costlier by  $diff$  at any point in  $R$ ; see Theorem E.1.2 */
    Insert  $R$  in  $Hierarchical-Index$  with  $p$  as optimal plan
  Else
    Partition  $R$  into smaller  $2^{\mathcal{H}_R}$  rectangles and put them in  $RSet$ 

```

Figure E.1: Algorithm for approximate parameter space decomposition

The pseudo code for the algorithm is shown in Figure E.1; it assumes that the parameter space polytope is a hyper-rectangle. The algorithm decomposes the parameter space hyper-rectangle recursively and at each recursion level it examines the plans that are optimal at the vertices of the hyper-rectangle in question. Based on Theorem E.1.2, it calculates $minOptCost$ and $maxOptCost$, the maximum and minimum optimal costs, resp., at any point in the rectangle.

If plan $p \in POSP$ is the only optimal plan at all the vertices of the rectangle then, by Corollary E.1.1, no plan in $POSP \setminus p$ is cheaper than p at any point in R ; hence R is not decomposed further and is inserted in the hierarchical index with p as its optimal plan. Else, if difference between the minimum and maximum optimal cost at the vertices of R is less than some threshold then, by Theorem E.1.2, we can pick a plan, say p , from \mathcal{U}_R^P that is within the threshold of the optimal cost at each point in R ; hence R is not decomposed further and is inserted in the hierarchical index with p as its optimal plan. Else, R is partitioned further.

A similar idea is used by Vleugels and Overmars [37] for approximating generalized Voronoi diagrams.

Bibliography

- [1] BETAWADKAR, A. V. Query optimization with one parameter. Tech. rep., IIT, Kanpur, Feb 1997. <http://www.cse.iitk.ac.in/research/mtech1997>.
- [2] BOUGANIM, L., KAPITSKAIA, O., AND VALDURIEZ, P. Memory-adaptive scheduling for large query execution. In *CIKM* (1998).
- [3] CHU, F., HALPERN, J., AND GEHRKE, J. Least expected cost query optimization: what can we expect? In *PODS* (2002).
- [4] CHU, F., HALPERN, J. Y., AND SESHADRI, P. Least expected cost query optimization: An exercise in utility. In *SIGMOD* (1999).
- [5] COLE, R. L., AND GRAEFE, G. Optimization of dynamic query evaluation plans. In *SIGMOD* (1994).
- [6] DAVISON, D. L., AND GRAEFE, G. Memory-contention responsive hash joins. In *VLDB* (1994).
- [7] GANGULY, S. Design and analysis of parametric query optimization algorithms. In *VLDB* (1998).
- [8] GANGULY, S. A framework for parametric query optimization (unpublished manuscript; personal communication). 2001.
- [9] GANGULY, S. Personal communication. Dec, 2000.
- [10] GANGULY, S., AND KRISHNAMURTHY, R. Parametric query optimization for distributed databases based on load conditions. In *COMAD* (1994).
- [11] GHOSH, A., PARIKH, J., SENGAR, V., AND HARITSA, J. Plan selection based on query clustering. In *VLDB* (2002).
- [12] GRAEFE, G., AND MCKENNA, W. J. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE* (1993).
- [13] GRAEFE, G., AND WARD, K. Dynamic query evaluation plans. In *SIGMOD* (1989).
- [14] HULGERI, A., SESHADRI, S., AND SUDARSHAN, S. Memory cognizant query optimization. In *COMAD* (2000).
- [15] HULGERI, A., AND SUDARSHAN, S. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB* (2002).
- [16] HULGERI, A., AND SUDARSHAN, S. *AniPQO*: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *VLDB* (2003).
- [17] IOANNIDIS, Y., AND KANG, Y. Left deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD* (1991).

- [18] IOANNIDIS, Y. E., AND KANG, Y. Query optimization by simulated annealing. In *SIGMOD* (1987).
- [19] IOANNIDIS, Y. E., AND KANG, Y. Randomized algorithms for optimizing large join queries. In *SIGMOD* (1990).
- [20] IOANNIDIS, Y. E., NG, R. T., SHIM, K., AND SELLIS, T. K. Parametric query optimization. In *VLDB* (1992).
- [21] IOANNIDIS, Y. E., NG, R. T., SHIM, K., AND SELLIS, T. K. Parametric query optimization. *VLDB Journal* 6, 2 (1997).
- [22] MCKENNA, B. Personal communication.
- [23] MEHTA, M., AND DEWITT, D. J. Dynamic memory allocation for multiple query workload. In *VLDB* (1993).
- [24] MILNOR, J. W. *Topology from the differentiable viewpoint*. Univ Press Virginia, 1965.
- [25] MULMULEY, K. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, 1994.
- [26] NAG, B., AND DEWITT, D. J. Memory allocation strategies for complex decision support queries. In *CIKM* (1998).
- [27] PANG, H., CAREY, M. J., AND LIVNY, M. Managing memory for real-time queries. In *SIGMOD* (1994).
- [28] PolyLib: A library of polyhedral functions. Available at <http://icps.u-strasbg.fr/PolyLib>.
- [29] PRASAD, V. G. V. Parametric query optimization: A geometric approach. Tech. rep., IIT, Kanpur, Feb 1997. <http://www.cse.iitk.ac.in/research/mtech1997>.
- [30] RAO, S. V. U. M. Parametric query optimization: A non-geometric approach. Tech. rep., IIT, Kanpur, Feb 1997. <http://www.cse.iitk.ac.in/research/mtech1997>.
- [31] ROY, P. *Multi-Query Optimization and Applications*. PhD thesis, Indian Institute of Technology - Bombay, 2001.
- [32] SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. Dynamic caching of query results for decision support systems. In *Intl. Conf. on Scientific and Statistical Database Management* (1999).
- [33] SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. Access path selection in a relational database management system. In *SIGMOD* (1979).
- [34] SOHONI, M. Personal communication. Dec, 2003.
- [35] SWAMI, A. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *SIGMOD* (1989).
- [36] SWAMI, A., AND GUPTA, A. Optimization of large join queries. In *SIGMOD* (1988).
- [37] VLEUGELS, J., AND OVERMARS, M. Approximating generalized voronoi diagrams in any dimension. Tech. rep., Utrecht University, 1995. Available at <http://www.cs.uu.nl/research/techreps/UU-CS-1995-14.html>.
- [38] YU, P. S., AND CORNELL, D. W. Buffer management based on return on consumption in a multi-query environment. *VLDB Journal* 2, 1 (1993).
- [39] ZELLER, H., AND GRAY, J. An adaptive hash join algorithm for multiuser environment. In *VLDB* (1990).
- [40] ZIEGLER, G. M. *Lectures on Polytopes*. Springer Verlag, 1994.

Acknowledgements

I am deeply indebted to my advisor S. Sudarshan for his constant guidance, support and encouragement, without which the work in this dissertation, and the dissertation itself, would not have been possible. I am grateful to S. Seshadri who, along with Sudarshan, introduced me to the world of databases during my Masters program.

I thank the members of my research progress committee, Krithi Ramamritham, Sunita Sarawagi and N. L. Sarada, for their guidance. I am thankful to my other teachers, Soumen Chakrabarti, Milind Sohoni and Sundar Vishwanathan.

I am grateful to Sumit Ganguly for introducing me to the problem of parametric query optimization. I appreciate the efforts of my external examiners, Jayant Haritsa and Paul Larson, in evaluating the contents of this dissertation. Jayant's constructive comments and invaluable suggestions have significantly improved the contents of this dissertation.

The pleasant atmosphere in the Informatics Lab at IIT-Bombay made my work enjoyable, and I am grateful to all the members of the lab, past and present, for the same. Special thanks to fellow Ph.D. students Bharat Adsul and Prasan Roy for their company and many technical discussions. Thanks to Nandprasad Joshi for all his administrative help.

I am grateful to Vinit Kapoor for his help with the polytope handling code, to Vincent Loechner for his help with the Polylib library, and to Prasan Roy for providing the code for a Volcano-based conventional query optimizer prototype.

I thank Johann-Christoph Freytag, Anandi Herlekar, Shetal Shah and numerous anonymous referees of our conference submissions for their comments which helped improve the presentation significantly.

I am grateful to Infosys for supporting this work in part by an Infosys Ph.D. fellowship.

Arvind Hulgeri