Parametric Query Optimization

Yannis E. Ioannidis*
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
yannis@cs.wisc.edu

Raymond T. Ng†
Kyuseok Shim‡
Timos K. Sellis§
Computer Science Department
University of Maryland
College Park, MD 20742
{rng,shim,timos}@cs.umd.edu

Abstract

In most database systems, the values of many important run-time parameters of the system, the data, or the query are unknown at query optimization time. Parametric query optimization attempts to identify several execution plans, each one of which is optimal for a subset of all possible values of the run-time parameters. We present a general formulation of this problem and study it primarily for the buffer size parameter. We adopt randomized algorithms as the main approach to this style of optimization and enhance them with a *sideways information passing* feature that increases their effectiveness in the new task. Experimental results of these enhanced algorithms show that they optimize queries for large numbers of buffer sizes in the same time needed by their conventional versions for a single buffer size, without much sacrifice in the output quality.

*Partially supported by NSF under PYI Grant IRI-9157368 and by grants from DEC, HP, and AT&T.

†Supported by a University of Maryland Dissertation Fellowship and by NSF under Grants IRI-8719458 and IRI-9109755.

‡Supported by a Korean Government Overseas Scholarship and by NSF under Grant IRI-9057573.

§Partially supported by NSF under Grants IRI-8719458 and IRI-9057573 (PYI Award), by AFOSR under Grant AFOSR-89-0303, by grants from DEC, IBM and Bellcore, and by the University of Maryland Institute for Advanced Computer Studies.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1 Introduction

Relational query optimization is an expensive process, primarily because the number of alternative access plans for a query grows at least exponentially with the number of relations participating in the query. The application of several useful heuristics eliminates some alternatives that are likely to be suboptimal [SAC+79], but it does not change the combinatorial nature of the problem. In the future, database systems will need to optimize queries over much larger sets of alternative plans. The traditional, heuristically pruning, almost exhaustive query optimization algorithms are inadequate to fulfill the increased requirements, and new algorithms need to be developed.

One of the primary reasons for the increase in the number of alternative plans is that optimization will be required for many different values of important run-time parameters whose actual values are unknown at optimization time. To avoid the above, current database systems make certain assumptions about the database contents (e.g., value distribution in relation attributes), the physical schema (e.g., index types), the values of the system parameters (e.g., number of available buffers), and the values of the query constants. Some of these assumptions, however, may be violated at run time: the database contents and the physical schema change incessantly [ML86], the multiprogramming level of the system and the resource needs of concurrently running queries cannot be predicted, and queries may be executed with different bindings for their constants, e.g., a selection within a for-loop in a query embedded in a C program or calls to recursive rules in deductive databases. When these optimization-time assumptions are violated at execu-

tion time, re-optimization is needed or performance suffers.

Motivated by the above, we have studied the problem of optimizing queries for all possible values of run-time parameters that are unknown at optimization time (a task that we call *Parametric Query Optimization*), so that the need for re-optimization is reduced. This study has also been motivated by recent results on flexible buffer allocation [NFS91, FNS91]. It has been shown that in deciding how many buffers to allocate to a query, taking run-time conditions into account leads to improvement in system performance (e.g., throughput). The reported improvement has been obtained based on fixed plans that assume a specific number of allocated buffers. Further improvement in performance is expected if a plan is not fixed and can be chosen to match the actual number of allocated buffers.

In principle, the optimal plan generated by parametric query optimization may be different for each distinct value combination of all the possible run-time parameters. In practice, however, the total cost of generating all these plans would be prohibitive. A different approach would seek to produce distinct plans for values of a selected subset of run-time parameters in less time. It is this approach that we study in this paper, where we focus on the number of buffer pages allocated to a query (the *buffer size*) as the unknown parameter. We propose the use of randomized algorithms to address the tremendous increase in the number of alternative plans. Such algorithms have been successfully applied to various combinatorial optimization problems in the past, including the optimization of queries with many joins. We adapt three such algorithms (*Simulated Annealing* (SA) [KGV83, IW87], *Iterative Improvement* (II) [NSS86, SG88], and *Two Phase Optimization* (2PO) [IK90, IK91]) for parametric query optimization of select-project-join queries, and present experimental results that show the effectiveness of the devised adaptations.

Several projects have considered supporting multiple plans for a query. The earliest significant work in this area is by Graefe and Ward [GW89]. They discuss the implementation of *dynamic query plans* in the Volcano optimizer generator [GM91]. These are plans that include a *choose-plan* operator, which chooses among multiple available conventional plans given the values of certain run-time parameters. The proposal is for choose-plan operators to be introduced in all places of a plan where the choice of subplans underneath is sensitive to the values of these parameters. This work introduces many important concepts related to parametric query optimization but does not include a complete search strategy to identify the dynamic plans and the places where the choose-plan operators should be placed.

The XPRS project proposes to select at run-time a parallel plan from a set of plans based on buffer allocations [SKPO88]. Two different optimization algorithms have been proposed for this task. In an earlier reference [SKPO88], a 'binary-search' approach is advocated, where a query is first optimized for the smallest (m) and the largest (M) possible buffer size; if the two obtained plans are far from optimal for the buffer size for which they were not chosen, the query is optimized again for the midpoint between m and M and the process is repeated. The disadvantage of this approach is that the amount of time spent in query optimization grows linearly with the number of buffer sizes for which the query is optimized, which may be prohibitive. Also, as it has been pointed out elsewhere as well [GW89], this approach may work for one or two parameters, but would not scale up. In a more recent reference [HS91], the assumption is made that the buffer size is greater than the minimum required for efficient execution of hash-join. Based on that assumption, experimental evidence is provided that the optimal plan is in general insensitive to buffer size. Hence, an enhanced version of a conventional query optimizer for a fixed buffer size is proposed. The enhancements deal with some special cases where the insensitivity claim does not hold, and consist of essentially introducing choose-plan operators [GW89].

The Starburst project has also considered incorporating a second optimization phase that chooses plans at run-time [HP88]. To the best of our knowledge, however, no technique has been developed to find those plans. Also, Cornell and Yu [CY89] use an integer programming model to optimize queries in a transaction environment and their buffer allocations simultaneously. Nonetheless, as their concern is different from ours, still only one plan is produced per query, and that plan is susceptible to changes in the environment.

Our work differs from all the proposals mentioned above in several aspects. First, we present a general framework for parametric query optimization that is applicable to arbitrary parameters and not only buffer size. (In that respect, the work of Graefe and Ward is general as well [GW89].) Second, we develop complete parametric query optimization algorithms that produce multiple plans as output. These algorithms are not based on any assumptions like those made in the XPRS project [HS91], so they are much more generally applicable. Third, the experimental results of these algorithms on the buffer size parameter show that generality is not achieved at the expense of efficiency or output quality. Hence, we expect that these algorithms can easily be incorporated in the systems mentioned above, without jeopardizing their perfor-

mance goals.

This paper is organized as follows. Section 2 gives preliminary descriptions on SA, II, and 2PO. Section 3 introduces a general framework for parametric query optimization and provides experimental evidence for the need of obtaining multiple plans for different run-time values of the buffer size parameter. Section 4 presents the family of algorithms that we have developed and discusses several of their characteristics. Section 5 contains the results of several experiments with these algorithms, showing their effectiveness with respect to both running time and output quality. Finally, Section 6 summarizes our overall approach and presents some directions for future work.

2 Randomized Algorithms for Conventional Query Optimization

In this section, we briefly describe randomized algorithms as they have been applied to conventional, non-parametric query optimization. This is a necessary basis for the description of the parametric query optimization algorithms in the following sections.

Each solution to a combinatorial optimization problem can be thought of as a *state* in a space, i.e., a node in a graph, that includes all such solutions. Each state has a cost defined by some problem-specific cost function. The goal of an optimization algorithm is to find a state with the globally minimum cost. Randomized algorithms perform *random walks* in the state space via a series of *moves*. The states that can be reached in one move from a state S are called the *neighbors* of S. A move is called *uphill* (*downhill*) if the cost of the source state is lower (higher) than the cost of the destination state. A state is a *local minimum* if, in all paths starting at that state, every downhill move comes *after* at least one uphill move. It is a *global minimum* if it has the lowest cost among all states. It is on a *plateau* if it has no lower cost neighbor and yet it can reach lower cost states without uphill moves. Using the above terminology, we briefly outline three randomized optimization algorithms that have been used for query optimization [IW87, SG88, IK90, IK91].

First, Iterative Improvement (II) performs a large number of *local optimizations*. A local optimization starts at a random state and improves the solution by repeatedly accepting random downhill moves until it reaches a local minimum. Its output at the end is the least cost local minimum that has been visited.

Second, Simulated Annealing (SA) starts at a random state and proceeds by random moves, which if uphill, are only accepted with certain probability. As time progresses this probability gradually decreases until it becomes zero, which signifies the termination of the algorithm. The output of the algorithm as used in practice is again the least cost state that has been visited.

Third, Two Phase Optimization (2PO) is divided into two phases. In the first phase, II is run for a small period of time, i.e., a few local optimizations. The output of that phase is the initial state of the next phase, where SA is run with very low initial probability for uphill moves.

When the above generic optimization algorithms are applied to query optimization, three parameters need to be specified: the state space, the neighbors of each state, and the cost function. Each state in query optimization corresponds to an *access plan* (or simply *plan*) of the query to be optimized. By performing selections and projections as early as possible and excluding unnecessary cross products [SAC+79], a plan can be represented as a *join processing tree*, i.e., a tree whose leaves are base relations, internal nodes are join operators, and edges indicate the flow of data. If all internal nodes of such a tree have at least one leaf as a child, then the tree is called *deep*. Otherwise, it is called *bushy*. In this study, we deal with the plan space that includes both deep and bushy trees.

The neighbors of a state, which is a join processing tree (i.e., a plan), are determined by a set of transformation rules. Each neighbor is the result of applying one of these rules to some internal nodes of the original plan once, replacing them by some new nodes, and usually leaving the rest of the nodes of the plan unchanged. There are several sets of transformation rules from which one could choose. With A, B, and C being arbitrary join processing formulas, the ones adopted in this study are described below [IK90, IK91]:

(1) *Method choice:* $A \bowtie_{method_i} B \rightarrow A \bowtie_{method_j} B$
(2) *Commutativity:* $A \bowtie B \rightarrow B \bowtie A$
(3) *Associativity:* $(A \bowtie B) \bowtie C \leftrightarrow A \bowtie (B \bowtie C)$
(4) *Left exchange:* $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$
(5) *Right exchange:* $A \bowtie (B \bowtie C) \rightarrow B \bowtie (A \bowtie C)$

Rule (1) changes the join method of a join, e.g., from nested-loops to merge-scan.

Finally, the cost of every plan is usually a combination of the I/O and cpu cost of the plan. The above algorithms have been successfully applied to conventional, non-parametric query optimization [SG88, IK90, IK91], which assumes a certain number of buffers b_0 for a given query, and produces a single plan that is optimal for b_0. 2PO has been shown to be the dominant algorithm for a wide range of values of b_0.

The main reason for this is that, in all cases, the shape of the cost function of the plan space forms a *'well'*. That is, some small percentage of local minima in the space have high cost but most of them have low cost, and the connection cost[1] between local minima is still relatively low compared to the cost range in the whole space. 2PO takes advantage of the first fact in its II-phase to reach the 'well'-bottom quickly and then takes advantage of the second fact in its SA-phase to explore the 'well'-bottom without climbing over very high hills.

3 Problem Formulation and Justification

3.1 Problem Formulation

Throughout this paper, we use S to denote the set of all plans that can be used to answer a given query. We also use $\overline{c}$ to denote the vector of all those parameters whose values are assumed to remain unchanged between optimization and run time. Each plan s in S has an associated cost $c(s, \overline{c})$. The goal of any conventional optimization algorithm is to find the plan s_0 in S that satisfies the condition $c(s_0, \overline{c}) = \min\{c(s, \overline{c}) \mid s \in S\}$. In reality, many parameters that are part of $\overline{c}$ in the above formulation do not remain constant between optimization and execution time. Hence, if we use $\overline{p}$ to denote the parameters that can change, the cost of a plan s is more appropriately written as $c(s, \overline{p}, \overline{c})$. The task of parametric query optimization is to optimize the cost for all possible values of the $\overline{p}$ vector. More formally, a *plan function* $s()$ is of the form $s() : \overline{P} \rightarrow S$, where $\overline{P}$ denotes the domain of $\overline{p}$. In the sequel, we use the notation $\mathcal{S}_{()}$ to denote the set of all such plan functions. Parametric query optimization finds the optimal plan function in $\mathcal{S}_{()}$, i.e. the one that generates as output the optimal plan for any vector of values of $\overline{p}$ that may be given as input; given the vector of actual values of $\overline{p}$ at run-time, the plan function returns the plan that should be used by the query processor. In general, for every plan function $s()$, $\overline{P}$ can be partitioned so that, for all $\overline{p}_1$, $\overline{p}_2$ in the same partition, the plans $s(\overline{p}_1)$ and $s(\overline{p}_2)$ are identical. These partitions are called *image partitions*. Having defined these notations, we introduce below two equivalent formulations of parametric query optimization.

Formulation A There are $|\overline{P}|$ separate optimization problems, each one identical to the traditional, non-parametric case with a different $\overline{p}$ vector:

$$\forall \overline{p} \in \overline{P} \text{ find } s_0 \in S \text{ s.t.}$$
$$c(s_0, \overline{p}, \overline{c}) = \min\{c(s, \overline{p}, \overline{c}) \mid s \in S\}.$$

Formulation B There is a single optimization problem over plan functions:

$$\text{Find } s_0() \in \mathcal{S}_{()} \text{ s.t. } \forall \overline{p} \in \overline{P}$$
$$c(s_0(\overline{p}), \overline{p}, \overline{c}) = \min\{c(s(\overline{p}), \overline{p}, \overline{c}) \mid s() \in \mathcal{S}_{()}\}.$$

Example 1 Suppose parametric query optimization is applied to two parameters: buffer size and the kind of index available for a certain relation. Let the buffer size values of interest be in the range B=[2,151] and the set of possible indices be $I = \{no\_index, clustered\_Btree, non\_clustered\_Btree\}$. The domain $\overline{P}$ is the cross product $B \times I$ and $\overline{p} = \langle 15, no\_index \rangle$ is one of the 450 possible vectors of values defined in the domain. Under Formulation A, there are 450 different, non-parametric query optimization problems that must be solved. The optimal plan function can be obtained by integrating all the plans found in those optimizations. Under Formulation B, there is a single optimization problem, whose solution is the optimal plan function. □

In principle, the two formulations are equivalent. In practice, while Formulation A is simpler to conceptualize, Formulation B is more efficient to process.

3.2 Justification for Using Parametric Query Optimization

One may argue that the conventional approach of optimizing for a single vector $\overline{p}_0$ produces a plan that is (close to) optimal for all vectors $\overline{p} \in \overline{P}$. We present experimental results to show that, at least for the buffer size parameter, the above is not the case, therefore justifying the use of parametric query optimization.

Throughout this paper, we use $s_0(b)$ to denote the (approximately optimal) plan produced by 2PO for b buffers. Furthermore, for notational simplification, we drop the vector $\overline{c}$ of parameters that remain constant between optimization time and run time, and use $c(s_0(b_0), b)$ to denote the cost of the plan that is optimal for b_0 buffers when executed in the presence of b

[1] Roughly, the connection cost is the height (cost) of the hills that need to be climbed up to reach one local minimum from another.

buffers. If the difference between the costs $c(s_0(b_0), b)$ and $c(s_0(b), b)$ were generally small, parametric query optimization for the buffer size parameter would not be needed. Figure 1 shows that this difference can be quite high as buffer size changes. The x-axis is the

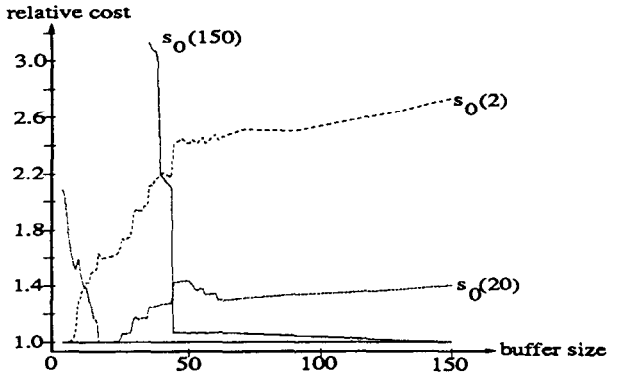

Figure 1: Relative costs of plans $s_0(2), s_0(20)$, and $s_0(150)$.

buffer size b which varies from 2 to 150 pages. The y-axis is the ratio $c(s_0(b_0), b)/c(s_0(b), b)$, which we call *relative cost* of $s_0(b_0)$ with b buffers. Since by definition the cost $c(s_0(b), b)$ is very close to the actual minimum for buffer size b, the closer the relative cost is to 1, the higher the quality of $s_0(b_0)$ is. Throughout this paper, the notion of relative cost is used to judge the quality of plans and plan functions.

Figure 1 includes three typical curves for plans $s_0(b_0)$ with $b_0 = 2, 20$, and 150. These curves are obtained by running ten 20-join queries five times each and show the average relative cost over all queries of the average over the five runs. The specifics of how the queries and corresponding data sets are generated are given in Section 5.1. In each case, the same general behavior is observed. For buffer sizes close to b_0, the relative cost is close to 1. As the buffer size moves away from b_0, however, the relative cost increases significantly. Part of the reason why this pattern is formed is that when there is a sufficient number of buffers, the costs of hash-joins are lower than those for merge-scans and nested-loops, but when buffers are scarce, the converse is true [Sha86]. Thus, as the buffer size grows, the optimal plan for that size tends to include more and more hash-joins and fewer and fewer merge-scans and nested-loops. The optimal ordering of the joins is affected by the value of b as well. Consequently, based on the results of Figure 1, parametric query optimization appears to be necessary for efficient processing of queries at all buffer sizes.

4 Randomized Algorithms for Parametric Query Optimization

4.1 Basic Algorithm

Consider a range $[b_{min}, b_{max}]$ of buffer sizes. Applying Formulation B of parametric query optimization for the buffer size parameter (and ignoring the vector $\bar{c}$ of constants) results in the following problem:

$$\text{Find } s_0() \in \mathcal{S}_() \text{ s.t. } \forall b_{min} \leq b \leq b_{max}$$
$$c(s_0(b), b) = \min\{c(s(b), b) \mid s() \in \mathcal{S}_()\}.$$

Let R be any randomized algorithm of the type described in Section 2 (II, SA, and 2PO are simply three examples). Instead of using R to optimize a given query separately for each buffer size $b_{min} \leq b \leq b_{max}$, which would be the case under Formulation A, we proceed concurrently for all buffer sizes. Abstractly, for each buffer size b, there is one co-routine R[b] that runs R on the conventional plan space (denoted by $G[b]$[2]) to identify the optimal plan for the given query when b buffers are available. These co-routines have synchronization points. When the running co-routine reaches one of these points, it releases control to another co-routine that is randomly chosen among those still running. In our study, the synchronization points of R[b] have been chosen to be right in-between attempted moves[3] (from the current plan to one of its neighbors) in R. After the active co-routine R[b] attempts a move to a neighbor of its current plan (successfully or not), another co-routine gains control to attempt a move to a neighbor of its own current plan.

4.2 Sideways Information Passing

The above concurrent version of the optimization does not offer many advantages compared to a serial optimization for each buffer size separately, because essentially there is no communication among the co-routines

[2]The graph structure of $G[b]$ is the one described in Section 2 and is identical for all values of b, but the node costs may differ. That is why we distinguish each graph by the index b.

[3]For every algorithm R, the success of an attempt to move from plan s to plan t depends on the cost difference between the plans and the specific characteristics of R.

(as in Formulation A). We enhance the above co-routines with the ability to share information. Specifically, let $s()$ be the current plan function defined by the current plans of the individual co-routines. When the active co-routine R[b] attempts to move from plan $s(b)$ to a neighbor t of $s(b)$ in $G[b]$, it communicates and sends t to a preselected subset of the remaining co-routines. The co-routines in this preselected set are called *friends* of R[b]. Each recipient R[b'] of t compares $c(t, b')$ with $c(s(b'), b')$ (which is the cost of its current plan), and then decides on whether to move to t or not in exactly the same way as if t and $s(b')$ were neighbors in $G[b']$. We use the term *sideways information passing* to refer to this exchange of plans between co-routine friends.

Consider the image partition of the current plan function $s()$ to which b belongs. Let b^- and b^+ be the minimum and maximum buffer size of that image partition, respectively. Given the natural total order that exists on buffer sizes, we have chosen the friends of R[b] to be all the co-routines R[b'] where $b^- - k \leq b' \leq b^+ + k$, for some $k \geq 0$. Thus, there is sideways information passing from the co-routine R[b] to the co-routines associated with buffer sizes that are similar to b. The value of k determines the *depth* of the sideways information passing. If $k = 0$, no information is shared among the co-routines that have different current plans. In that case, the algorithm can be thought of as a smart implementation of Formulation A (separate optimizations for each buffer size), because this allows co-routines of buffer sizes in the same image partition to exchange information, as they are always friends. This information, however, is in some sense trivial since it is always a plan that is a neighbor of the current plan of the recipient co-routine. Thus, in terms of graph traversal, this algorithm is identical to the non-parametric case. In that sense, in the rest of the paper, we refer to the case of $k = 0$ as featuring no sideways information passing. If $k = \infty$, the active co-routine sends its candidate new plan to all other co-routines, so there is complete information passing. Other values of k represent intermediate situations. This concurrent version of the optimization algorithm R that employs sideways information passing at depth k is denoted by sipR(k).

To be more concrete on how sideways information passing works, we present in Figure 2 pseudo-code for sipR(k), as it traverses a single random path. The code fully captures SA (and the second phase of 2PO),

```
procedure sipR(k)
  begin
  B := {b | b_min ≤ b ≤ b_max };
  s := random plan in S;
  foreach b ∈ B do s(b) := s;
  while B ≠ ∅ do
    begin
    b := random buffer size in B;
    t := neighborR[b](s(b));
    foreach b⁻ − k ≤ b' ≤ b⁺ + k do
      begin
      compare&moveR[b'](s(b'), t);
      if movedR[b'] then B := B ∪ {b'};
      end
    if finishedR[b] then B := B − {b};
    end
  end
```

Figure 2: Algorithm sipR(k).

whereas it captures a single local optimization of II (and the first phase of 2PO). For II, the code shown is executed as many times as it is necessary to perform local optimizations, and then some postprocessing integrates the results of these local optimizations. The code in Figure 2 captures the concurrent execution of all co-routines together by showing at all times which one is active. For that, it uses the following notation for parts of these co-routines: neighborR[b] is the part of R[b] that accepts a plan as input and returns one of its neighbors as output based on the R algorithm; finishedR[b] is a predicate indicating whether R[b] has finished or not; compare&moveR[b] takes two plans $s(b)$ and t as input, calculates their costs for buffer size b, and then decides whether or not to move from $s(b)$ to t based on the R algorithm; and movedR[b] is a predicate indicating whether the comparison in compare&moveR resulted in a successful move. Note that the foreach-loop within the while-loop captures the sideways information passing. Having $k = 0$ in this line essentially eliminates this feature, as only co-routines for buffer sizes within the same image partition are allowed to share information.

Example 2 Suppose at the beginning of the current iteration of the while-loop, there are 3 image partitions for the buffer sizes from 13 to 16. As illustrated in Figure 3, S_1 is the plan in the image partition for 13, S_2 the one for 14 and 15, and S_3 the one for 16. Suppose the random buffer size chosen in this iteration is $b = 14$. Given S_2 as input, suppose that the routine neighborR[14] returns the plan S_4 which uses

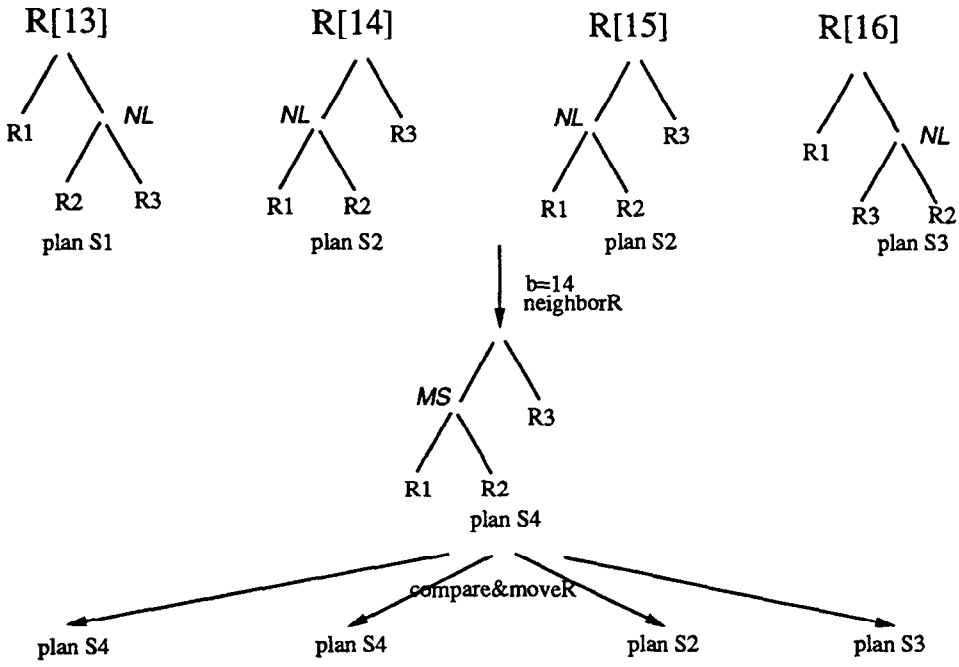

Figure 3: Illustration for Example 2.

a merge-scan, instead of a nested-loop, for the join between relations R_1 and R_2. Furthermore, let k be 1, for the scenario described in Figure 3. Then the routines compare&moveR[b'] are invoked for $b' = 13, 14, 15, 16$. Let us assume for this example that S_4 is a cheaper plan than S_1 and S_2 for buffer sizes 13 and 14 respectively, but that it is not as good as S_2 and S_3 for buffer sizes 15 and 16 respectively. Consequently, at the end of this iteration, there are 3 image partitions: S_4 for buffer sizes 13 and 14, S_2 for buffer size 15, and S_3 for 16.

If there is no sideways information passing, i.e., $k = 0$, then compare&moveR[b'] will only be invoked for $b' = 14, 15$. Consequently, S_1, S_4, S_2 and S_3 will be the plans for buffer sizes 13, 14, 15 and 16 respectively. □

As shown in Figure 2, the depth k of sideways information passing is measured in terms of buffer sizes. We have also experimented with a different algorithm, where the depth k is measured in terms of the image partitions of the current plan function. Let these partitions be identified by their distance (measured in number of partitions) from the lowest buffer size and let $r[b]$ be the image partition where b belongs. This algorithm can be seen as a modification of the original sipR algorithm where the foreach-loop that implements the sideways information passing becomes

foreach b' s.t. $r[b] - k \leq r[b'] \leq r[b] + k$ **do** .

To distinguish between the two versions of the algorithms, we use sipRs (for 's'ize) to denote the original one and sipRr (for 'r'ange) to denote the modified one.

4.3 Plan Space Abstraction

Any analysis of the performance and behavior of randomized algorithms requires that the three problem specific parameters mentioned in Section 2 be specified. When sipR does not incorporate any nontrivial sideways information passing ($k = 0$), it is equivalent to running R separately for each buffer size b in exactly the same way as in conventional query optimization. With sideways information passing, however, the notion of neighbors becomes more complicated (although the set of plans and the cost function remain the same for each co-routine). This is due to the communication of plans occurring among friends. The current plan $s(b)$ of R[b] may be replaced by an arbitrary plan t when a friend attempts to move to t, even if $s(b)$ and t are not neighbors in $G[b]$.

In order to model R[b] with sideways information passing as a regular randomized algorithm always moving between neighbors, we construct below a new

graph $G^*[b]$ that can be used as the abstract space on which R[b] is executed, following the conventional steps of R and without any communication with any other co-routines. For every node s in $G[b]$, there is a set of nodes $\{(s, s') \mid s' \in S\}$ in $G^*[b]$, i.e., s generates as many nodes as there are plans in S. Plans s and s' are called the *primary* and *secondary* plan of a node (s, s'), respectively. The intuition behind the above is that s signifies the current plan of R[b] while s' signifies the current plan of a friend of R[b]. The edges of $G^*[b]$ are defined as follows. First, any pair of nodes with the same primary plan are directly connected by a type-1 edge, i.e., all nodes with the same primary plan form a clique. Second, if t and u are neighbors in $G[b]$, then for all s there is a type-2 edge between (s, t) and (u, u) in $G^*[b]$. Figure 4 shows a simple example of how the above graph abstraction is constructed from a conventional plan space[4]. Note that, because of the cliques formed by type-1 edges, starting from any node in $G^*[b]$, it is possible to move to some node with an arbitrary primary plan in at most two moves. Finally, the cost of a node (s, s') in $G^*[b]$ is equal to $c(s, b)$ for all s', which implies that each aforementioned clique forms a plateau.

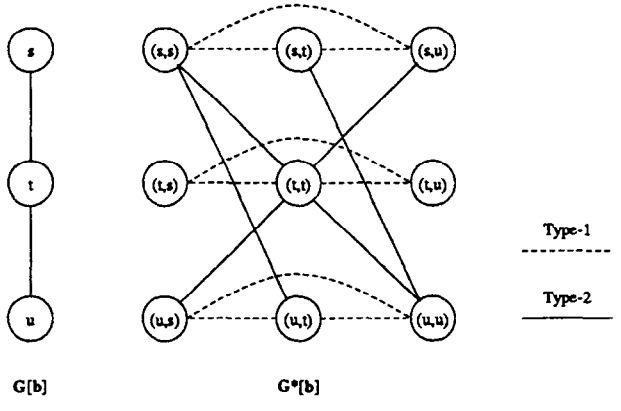

Figure 4: Constructing the graph abstraction $G^*[b]$ from the conventional plan space $G[b]$.

We claim that running R[b] on the conventional plan space $G[b]$ under control of sipR(k) with sideways information passing ($k > 0$) is equivalent to running R[b] on $G^*[b]$ with no communication to any other co-routines. To see why this is the case, first note that the random choice of buffer-size/co-routine and the sideways information passing (first statement and foreach-loop in the while-loop of Figure 2, respectively) are the only parts that need attention. Let s be the current plan of R[b] and (s, s) be the current node of R[b] in $G^*[b]$ (cf. Figure 4). Choosing a new buffer size b' in the first statement of the while-loop of sipR(k), such that R[b'] is a friend of R[b] based on the value of k, is equivalent to moving from (s, s) to (s, t) in $G^*[b]$, where t is the current plan of R[b']. Clearly the two nodes are connected in $G^*[b]$ via a type-1 edge and have the same cost, so the move is always legal and is always successful. Choosing a neighbor u of t in R[b'] under sipR(k) and sending it to its friend R[b] for a possible move is equivalent to attempting to follow a type-2 edge from (s, t) to (u, u) in $G^*[b]$. Therefore, since $c(s, b) = c((s, t), b)$ and $c(u, b) = c((u, u), b)$, the two algorithmic abstractions are equivalent.

Based on the above, in the next subsection, we use all the results derived for each conventional algorithm R to understand sipR better and draw conclusions about its behavior. We should note, however, that running R on $G^*[b]$ represents only an abstraction which if implemented directly, would be extremely expensive due to the size of $G^*[b]$.

As mentioned in Section 2, one of the key factors that determine the success or failure of randomized algorithms is whether or not the cost function c forms a 'well' over the plan space. We claim that the $G^*[b]$ graph constructed above forms a very definitive 'well', and therefore, randomized algorithms are expected to be effective for parametric query optimization. Specifically, let g be a global minimum plan in the conventional plan space $G[b]$. As mentioned in the previous subsection, the distance between any node (s, s') of the graph and node (g, g) is at most 2. Moreover, the intermediate node that connects them is of the form (s, t), where t is a conventional neighbor of g in $G[b]$. Comparing the costs of the three nodes yields

$c((s, s'), b) = c((s, t), b)$, by construction of the clique,
$c((s, t), b) \geq c((g, g), b)$, as g a global minimum in $G[b]$.

Hence, the only local minima in the new graph $G^*[b]$ are also global minima, and they are all mutually connected. The above implies that a 'perfect well' is formed. Although in practice randomized algorithms identify approximations to local minima, experiments

[4]Strictly speaking, some nodes, such as (t, s) and (t, t), in Figure 4 are connected by both type-1 and type-2 edges. However, for the purpose of randomized algorithms, it makes no difference whether two nodes are connected directly once or twice. The introduction of these two types of edges is merely for the purpose of presentation.

that we have conducted have verified that they can effectively exploit the 'well'-shape [INSS92].

5 Algorithm Behavior

5.1 Experiment Testbed

We implemented all four algorithms sipIIs/r(k) and sip2POs/r(k) for several values of the depth k and run several experiments to test their effectiveness for query optimization. Details about the implementations are given in the full version of the paper [INSS92]. The machine used for the experiments was a DECstation 3100. The algorithms were run on tree queries [Ull82] consisting of equality joins only. The size of these queries, which were generated randomly, ranged from 1 to 20 joins. They were tested with a randomly generated relation catalog where relation cardinalities ranged from 1000 to 100000 tuples, and the numbers of unique values in join columns varied from 10% to 100% of the corresponding relation cardinality[5]. Each page of a relation was assumed to contain 16 tuples. Each relation had four attributes, and was clustered on one of them. If a relation was not physically sorted on the clustered attribute, there was a B^+-tree or hashing primary index on that attribute. These three alternatives were equally likely. For each of the other attributes, the probability that it had a secondary index was 1/2, and the choice between a B^+-tree and hashing secondary index were again uniformly random. As for join methods, we used block nested-loops, merge-scan, and simple and hybrid hash-join [Sha86]. The query cost was a weighted sum of the cpu time and the number of I/O accesses. The weight was chosen so that the cost of one disk read or write corresponded to 30msec. The specific cost formulas are given elsewhere [INSS92]. In what follows, unless otherwise stated, the results presented for each algorithm are averages of five runs of the algorithm on each of 10 queries with similar characteristics. The values of the buffer size parameter were those in the range $[2, 70]$.

For every specific query instance, we first ran 2PO separately for each buffer size (which resulted in the (approximately) optimal plan function $s_0()$ as described in Section 3.2) and obtained its average running time over all buffer sizes. We then allowed sipII to run for exactly that amount of time on the query. Thus, sipII used the same amount of time to optimize a query over a range of buffer sizes, as 2PO used to optimize that query for a single buffer size.

[5]This was the most varied catalog (catalog 'relcat3') that we used in previous experiments [IK90].

5.2 2PO versus II

We first present results that compare the effectiveness and performance of sip2PO and sipII. These results have consistently indicated that unlike the situation for conventional query optimization, sipII is very competitive with sip2PO. As a representative, Figure 5 shows the relative costs of the output plan functions found by sip2POr(1) and sipIIr(1) for five 20-join queries. The figure includes the results of two

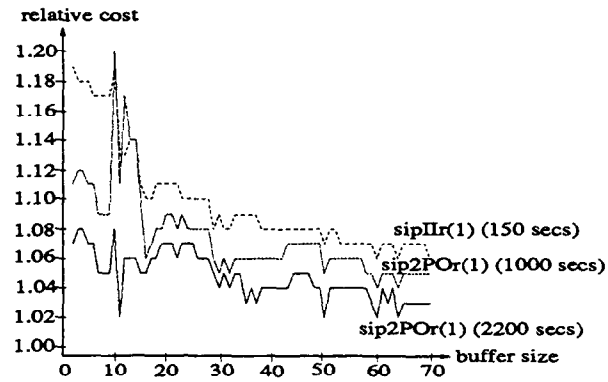

Figure 5: Output quality of sip2POr(1) and sipIIr(1).

different versions of sip2POr(1) that took around 1000 sec and 2200 sec, respectively. When compared with sipIIr(1), which only takes around 150 seconds, the relative output costs of both versions of sip2POr(1) are lower by a mere 1–4% on the average.

The fact that sipIIr compares favorably with sip2POr is actually not surprising. Recall from the previous section that $G^*[b]$ forms a 'perfect well'. Hence, the second phase of sip2PO is not really necessary. Based on the above, in the remainder of this paper we concentrate on sipII only.

5.3 Optimal Depth for Sideways Information Passing

To evaluate the effectiveness of sideways information passing, we compare the performance of sipIIs(k) and sipIIr(k) for various values of k. Table 1 shows the average relative costs over the buffer size range [2,70], of the plan functions found by sipIIs(k) and sipIIr(k), for $k = 0, 1, 2, 5, 10$, and ∞ (given the same amount

of time). The specific results are for 20-join queries,

k	0	1	2	5	10	∞
sipIIr(k)	1.67	1.10	1.11	1.17	1.20	1.17
sipIIs(k)	1.67	1.08	1.08	1.08	1.11	1.17

Table 1: Output quality of sipIIr(k) and sipIIs(k).

but similar results were obtained for other queries as well. As Table 1 shows, the improvement from depth $k = 0$ to $k = 1$ is significant. This demonstrates the usefulness of sideways information passing and is consistent with the results on the 'well'-shape of the cost functions over $G^*[b]$ presented in the previous section.

On the other hand, as the depth k increases beyond 1, there is a gradual degradation in performance. This is due to the fact that as the number of plan cost comparisons increases, the time consumed by such comparisons more than offsets the benefits of a friend occasionally finding a lower cost plan. In general, the larger the difference between the buffer sizes of friends, the less likely that the comparison between the costs of their associated plans is beneficial. Throughout this paper, we refer to this phenomenon as *over-comparing*. Indeed, due to over-comparing, our experiments consistently find sipII(1) to be the best among sipII(k) for all values of k.

Table 1 also serves to compare sipIIs with sipIIr and identify some differences between them. First, unlike sipIIr(k), for small values of $k > 1$, the output quality of sipIIs(k) is comparable to that of sipIIs(1): any small value of the depth k is equally optimal for sipIIs(k). This result is consistent with the fact that the optimal depth k for sipIIr(k) is 1, for an image partition of the optimal plan function rarely consists of more than 5 buffer sizes. Second, for corresponding k values, the output quality of sipIIs(k) is consistently better than that of sipIIr(k)[6]. The reason is that an image partition of the current plan function in sipIIr(k) may consist of more than one buffer size, and thus sipIIr(k) is more prone to the effect of over-comparing than sipIIs(k). Since sipIIs(1) appears to be the dominant algorithm for parametric query optimization, we devote our full attention to it in the rest of the paper.

[6]The two algorithms coincide when k=0 or $k = \infty$ but behave differently for intermediate values of k.

5.4 Effect of Query Size and Running Time

In this subsection, we show the effectiveness of sipIIs(1) for optimizing queries of various sizes as well as how this is affected when the time consumed by the algorithm varies. We present results for queries with 1, 3, 5, 7, 10, and 20 joins. With respect to the running time of the algorithm, recall that for the results presented so far, the amount of time given to sipIIs(1) was equal to the time needed by 2PO to optimize a query for a single buffer size. Let T be that time. The average values of T for various query sizes is shown in Table 2. We performed additional experiments where

Query Size (Joins)	1	3	5	7	10	20
T (sec)	2	14	27	42	62	158

Table 2: Average time given to sipIIs(k) for queries of different sizes.

the amount of time given to sipIIs(1) was $T/3$, $2T/3$, T, and $2T$. Figure 6 shows the results of the combined

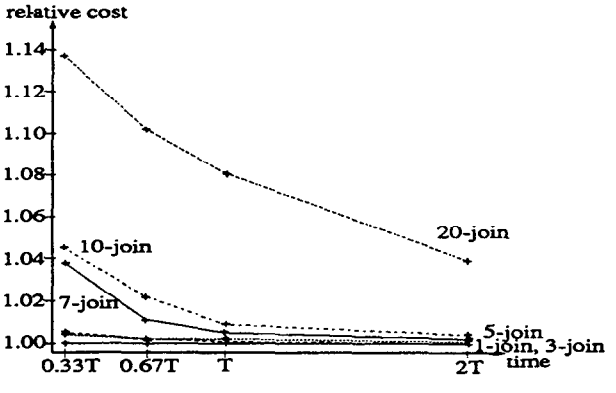

Figure 6: Output quality of sipIIs(1) with varying experimentation times.

experiments. Specifically, it shows the average over the buffer size range [2,70] of the relative cost of the output plan function of sipIIs(1). As expected, more time gives better results for any query size. The surprising result, however, is that for small queries, even a time of $T/3$ is sufficient to produce a plan function that is within 1% of the optimal (i.e., a relative cost of 1). As for larger queries, such as 20-join queries, a time of $2T$ produces a plan whose average cost is within 4% of the optimal one. These results are very

promising and indicate that, by using sipIIs(1), parametric query optimization can be efficiently supported in current systems. As in applying II to conventional query optimization, an interesting question that arises in parametric query optimization is how to determine the running time of a query optimizer for real applications. This is an issue that requires further study in the form of a comprehensive performance evaluation on sipII.

6 Conclusions and Future Work

We have formalized the problem of parametric query optimization and studied it primarily for the buffer size parameter. We have adopted randomized algorithms as the main approach to this style of optimization and have introduced sideways information passing to increase the effectiveness of these algorithms in the new task. Extensive experimentation has shown that these enhanced algorithms optimize queries for large numbers of buffer sizes in the same time needed by their conventional versions for a single buffer size, without much sacrifice in the output quality. These experiments have also identified sipIIs(1) as the most effective of the randomized algorithms for a very broad spectrum of cases.

To the best of our knowledge, the approach presented in this paper for parametric query optimization is the first of its kind, since it offers a complete query optimization algorithm that has a plan function as output and makes no assumptions about any properties of the plan costs. We believe that incorporating sipIIs(1) into a query optimizer will significantly enhance the performance of queries. When a query is ready to be executed, the database system will know the precise values of the parameters that were unknown at query optimization time. It will take a simple table look-up with the parameter values to identify the appropriate plan for the execution. The savings in execution cost of using a plan that is specifically tailored to the actual parameter values as opposed to one obtained for typical parameter values could be enormous.

There are several issues that we plan to address in our future work. First, it would be interesting to devise ways to make sipII more efficient, such as by reducing the number of buffer sizes that are being considered during optimization. Second, it would be beneficial to control the number of image partitions of the output plan function. Preliminary work in this direction indicates that whenever the number of partitions is large, simple postprocessing steps can reduce that number considerably, without a significant penalty on the output quality. Third, it would be interesting to adapt the traditional, dynamic programming algorithm for parametric query optimization and to compare it against sipII. Although we believe that by its very nature, the dynamic programming approach will not be effective for parametric query optimization, this belief needs further investigation and experimental verification. Finally, it would be important to experiment with large vectors of diverse parameters to understand the scalability of the proposed algorithms. Preliminary experiments with the index parameter of some query relation have shown that sipIIs(1) can obtain output of good quality in a short amount of time. The results of these studies will complement those presented in this paper and shed some new light into how parametric query optimization should be approached in future database systems.

References

[CY89] D. Cornell and P. Yu. Integration of buffer management and query optimization in relational database environment. In *Proc. of the 15th International VLDB Conference*, pages 247–255, Amsterdam, The Netherlands, August 1989.

[FNS91] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. of the 17th International VLDB Conference*, pages 265–274, Barcelona, Spain, August 1991.

[GM91] G. Graefe and W. McKenna. The volcano optimizer generator. Technical Report 563, University of Colorado, Boulder, December 1991.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the 1989 ACM-SIGMOD Conference on the Management of Data*, pages 358–366, Portland, OR, May 1989.

[HP88] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Technical Report

RJ 6367, IBM Almaden Research Center, 1988.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proc. of the 1st International PDIS Conference*, pages 218–225, Miami, FL, December 1991.

[IK90] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[IK91] Y. E. Ioannidis and Y. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data*, pages 168–177, Denver, CO, May 1991.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. Technical report, Univ. of Wisconsin, Madison and Univ. of Maryland, College Park, 1992.

[IW87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, pages 9–22, San Francisco, CA, May 1987.

[Kan91] Y. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin, Madison, May 1991.

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[ML86] L. F. Mackert and G. M. Lohman. *R** validation and performance evaluation for local queries. In *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, pages 84–95, Washington, DC, May 1986.

[NFS91] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data*, pages 387–396, Denver, CO, May 1991.

[NSS86] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Proc. of the 23rd Design Automation Conference*, pages 293–299, 1986.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 23–34, Boston, MA, June 1979.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, pages 8–17, Chicago, IL, June 1988.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(3):239–264, September 1986.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of xprs. In *Proc. of the 14th International VLDB Conference*, pages 318–330, Long Beach, CA, August 1988.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1982.