The VLDB Journal (1997) 6: 132–151

The VLDB Journal
© Springer-Verlag 1997

Parametric query optimization

Yannis E. Ioannidis [1], **Raymond T. Ng** [2], **Kyuseok Shim** [3], **Timos K. Sellis** [4]

[1] Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA; yannis@cs.wisc.edu
[2] Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada; rng@cs.ubc.ca
[3] Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA; shim@research.bell-labs.com
[4] Dept. of Electrical and Computer Engineering, Computer Science Division, National Technical University of Athens, Zographou 157 73, Athens, Greece; timos@theseas.ntua.gr

Edited by S. Zdonik / Received June 1993 / Accepted April 1996

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 at compile time several execution plans, each one of which is optimal for a subset of all possible values of the run-time parameters. The goal is that at run time, when the actual parameter values are known, the appropriate plan should be identifiable with essentially no overhead. 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 and with essentially zero run-time overhead.

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 execution 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 positions where the choose-plan operators should be placed. A complete method based on this approach has been subsequently developed by Cole and Graefe [CG94]. Due to its importance, it is discussed in detail in Sect. 8. Its main difference from our approach is that it does have some optimization overhead at run time, and that it uses dynamic programming instead of randomized algorithms.

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 has been pointed out elsewhere [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 and their buffer allocations in a transaction environment. Even though their concern is different from ours, their technique still produces only one plan per query, and that plan is susceptible to changes in buffer allocations.

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 also general [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 performance goals.

This paper is organized as follows. As a background, Sect. 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, discusses several of their characteristics, and provides evidence on how they are expected to perform. Section 5 contains the results of several experiments with these algorithms, showing their effectiveness with respect to both running time and output quality. Section 6 discusses several issues related to our study. Section 7 gives some ideas on how our approach can be used for parameters other than the buffer size, as well as for multiple parameters. Section 8 presents an informal and preliminary comparison of randomized algorithms and dynamic programming in the context of optimization with unknown run-time parameters. Finally, Sect. 9 summarizes our overall approach and presents some directions for future work. The appendix lists the cost formulas used in the algorithms.

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, 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, 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, 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[1] [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. *Join method choice:* $A \bowtie_{method_i} B \rightarrow A \bowtie_{method_j} B$
2. *Join commutativity:* $A \bowtie B \rightarrow B \bowtie A$
3. *Join associativity:* $(A \bowtie B) \bowtie C \leftrightarrow A \bowtie (B \bowtie C)$
4. *Left join exchange:* $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$
5. *Right join 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[2] 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.

[1] The exclusion of cross-products follows the experience with both dynamic programming and randomized algorithms on conventional query optimization and does not affect the results of this paper in any way. Its only effect is that it removes large parts of the plan space that include almost always suboptimal plans, thus making the optimization algorithms significantly more efficient.

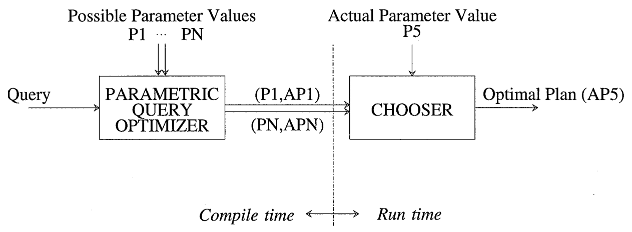

Fig. 1. Overall architecture of parametric query optimization

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 $\bar{\mathbf{c}}$ 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 $c(s, \bar{\mathbf{c}})$. The goal of any conventional optimization algorithm is to find the plan s_0 in S that satisfies the condition $c(s_0, \bar{\mathbf{c}}) = \min\{c(s, \bar{\mathbf{c}}) \mid s \in S\}$. In reality, many parameters that are part of $\bar{\mathbf{c}}$ in the above formulation do not remain constant between optimization and execution time. Hence, if we use $\bar{\mathbf{p}}$ to denote the parameters that can change, the cost of a plan s is more appropriately written as $c(s, \bar{\mathbf{p}}, \bar{\mathbf{c}})$. The task of parametric query optimization is to optimize the cost of query answering for all possible values of the $\bar{\mathbf{p}}$ vector. More formally, a *plan function* $s()$ is of the form $s() : \bar{\mathbf{P}} \rightarrow S$, where $\bar{\mathbf{P}}$ denotes the domain of $\bar{\mathbf{p}}$. Hereafter, we use the notation $\mathscr{S}_0$ to denote the set of all such plan functions. Parametric query optimization finds the optimal plan function in $\mathscr{S}_0$, i.e., the one that generates as output the optimal plan for any vector of values of $\bar{\mathbf{p}}$ that may be given as input; given the vector of actual values of $\bar{\mathbf{p}}$ at run time, the plan function returns the plan that should be used by the query processor. This is schematically shown in Fig. 1, where $P1, \ldots, PN$ denote possible parameter values, and $AP1, \ldots, APN$ denote the corresponding optimized plans.

In general, for every plan function $s()$, $\bar{\mathbf{P}}$ can be partitioned so that, for all $\bar{\mathbf{p}}_1$, $\bar{\mathbf{p}}_2$ in the same partition, the plans $s(\bar{\mathbf{p}}_1)$ and $s(\bar{\mathbf{p}}_2)$ are identical. These partitions are called *image partitions*. The image partitions in the optimal plan func-

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

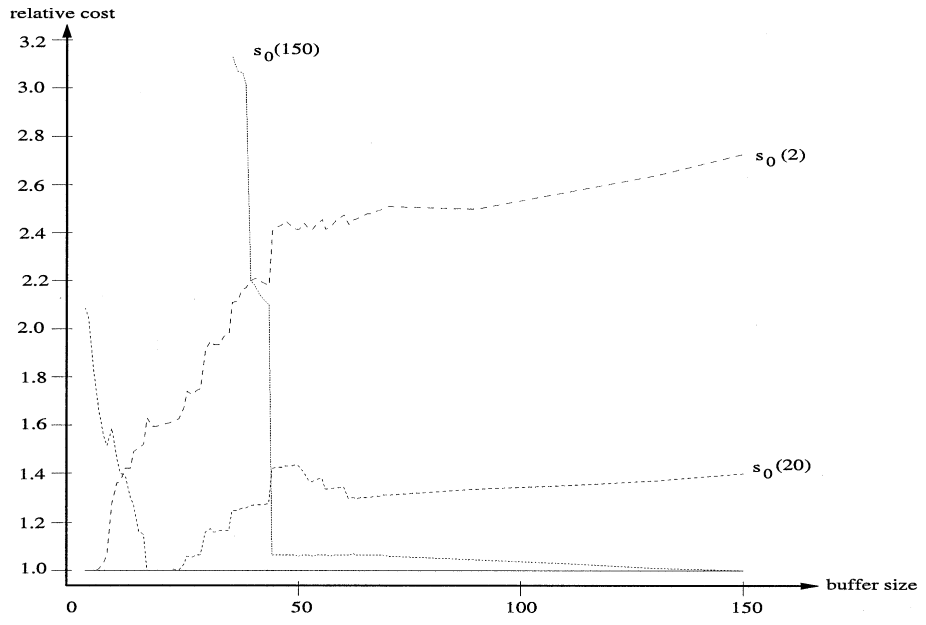

Fig. 2. Relative costs of plans $s_0(2)$, $s_0(20)$, and $s_0(150)$

tion are not known a priori but are identified by the parametric query optimizer as part of the process of identifying the optimal plan for each parameter value.

Having defined these notations, we introduce below two equivalent formulations of parametric query optimization.

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

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

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

$$\text{find } s_0() \in \mathscr{S}_0$$
$$\text{s.t. } \forall \overline{\mathbf{p}} \in \overline{\mathbf{P}}\ c(s_0(\overline{\mathbf{p}}), \overline{\mathbf{p}}, \overline{\mathbf{c}}) = \min\{c(s(\overline{\mathbf{p}}), \overline{\mathbf{p}}, \overline{\mathbf{c}}) | s() \in \mathscr{S}_0\} .$$

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 $\overline{\mathbf{P}}$ is the cross product $B \times I$ and $\overline{\mathbf{p}} = \langle 15, no\_index \rangle$ 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 $\overline{\mathbf{p}}_0$ produces a plan that is (close to) optimal for all vectors $\overline{\mathbf{p}} \in \overline{\mathbf{P}}$. We present experimental results to show that, at least for the buffer size parameter, the above is not the case; this justifies 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 $\overline{\mathbf{c}}$ 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 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 2 shows that this difference can be quite high as buffer size changes. The x-axis is the 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 2 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 Sect. 5.2. 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 may increase 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 Fig. 2, parametric query optimization appears to be necessary for efficient processing of queries at all buffer sizes.

4 Randomized algorithm 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 $\bar{\mathbf{c}}$ of constants) results in the following problem:

$$\text{Find } s_0() \in \mathscr{S}_0 \text{ s.t.}$$
$$\forall b_{min} \leq b \leq b_{max} \; c(s_0(b), b) = \min\{c(s(b), b) | s() \in \mathscr{S}_0\} \,.$$

Let R be any randomized algorithm of the type described in Sect. 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 $G[b]$ to identify the optimal plan for the given query when b buffers are available[3]. 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 (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. 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.

[3] The graph structure of $G[b]$ is the one described in Sect. 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.

```
procedure sipR($k$)
    begin
    $B$ := $\{b \mid b_{min} \leq b \leq b_{max}\}$;
    $s$ := random plan in $S$;
    foreach $b \in B$ do $s(b)$ := $s$;
    while $B \neq \emptyset$ do
        begin
        $b$ := random buffer size in $B$;
        $t$ := neighborR[$b$]($s(b)$);
        foreach $b^- - k \leq b' \leq b^+ + k$ do
            begin
            compare&moveR[$b'$]($s(b'), t$);
            if movedR[$b'$] then $B$ := $B \cup \{b'\}$;
            end
        if finishedR[$b$] then $B$ := $B - \{b\}$;
        end
    end
```

Fig. 3. Algorithm sipR(k)

Consider the image partition of the current plan function $s()$ in which b belongs. Let b^- and b^+ be the minimum and maximum buffer size, respectively, of that image partition. 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 $b^- - k \leq b' \leq b^+ + k$, $k \geq 0$. 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 A1 (separate optimizations for each buffer size), since at any point co-routines of buffer sizes in the same image partition are always friends and exchange information. 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.[4] 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 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 Fig. 3 pseudo-code for sipR(k), as it traverses a single random path. The code fully captures SA (or the second phase of 2PO), whereas it captures a single local optimization of II (or of 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 3 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

[4] Except that, in the non parametric case, independent optimization for different buffer sizes may generate different random neighbors for a plan, whereas in our case, if the plans belong in the same image partition, they always attempt to move to the same neighbor.

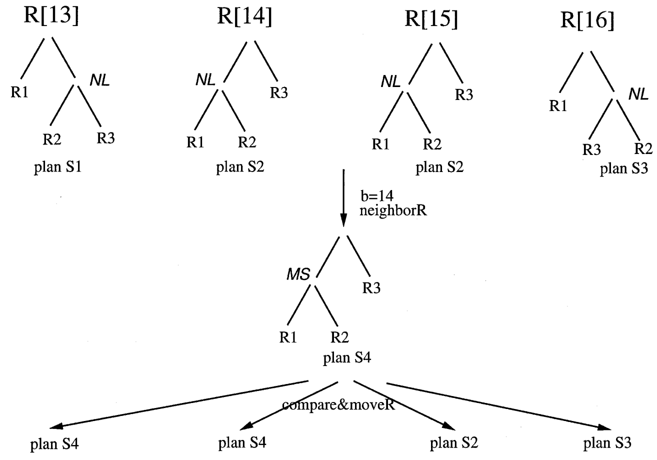

Fig. 4. Illustration for Example 4.1

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 succeeded. 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 partition are allowed to share information.

Example 4.1. Suppose at the beginning of the current iteration of the while-loop there are three image partitions for the buffer sizes from 13 to 16. As illustrated in Fig. 4, 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 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 Fig. 4. 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 three 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 Fig. 3, 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 b_{min} 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 Maintenance of image partitions

Since the sipR algorithm may start from any random plan function, we choose to generate a random plan and start with a plan function with a single image partition that covers the entire range of parameter values. As the sipR algorithm proceeds, the set of image partitions is enlarged or shrunk with each call to the compare&moveR function. The current set of image partitions can be maintained in several ways, three of which are given below:

1. Keeping a separate copy of a plan (as a tree) for each parameter value;
2. Keeping a separate copy of a plan (as a tree) for each image partition; and
3. Keeping a global graph of plans that combines the trees corresponding to the plans for all image partitions, where common subplans (subtrees) are shared.

Solution 2 simply improves space efficiency compared to solution 1, since by definition all parameter values in an image partition correspond to the same plan. Solution 3 improves space efficiency even further, since, by the nature of the transformation rules, one expects to have many subplans that are common among plans of various image partitions. On the other hand, solution 3 makes the maintenance of plans quite a bit more complex and expensive, as common nodes may have to be separated and separate nodes may have to be merged during transformations. Based on the overall trade-off, we have decided to adopt solution 2.

4.4 Plan space abstraction

Any analysis of the performance and behavior of randomized algorithms requires that the three problem-specific parameters mentioned in Sect. 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

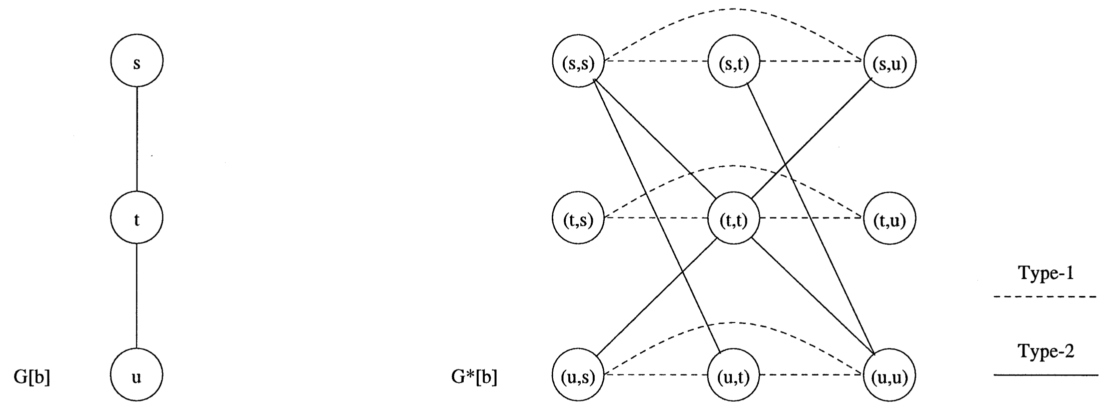

Fig. 5. Constructing the graph abstraction $G^*[b]$ from the conventional plan space $G[b]$

$G[b]$, 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 5 shows a simple example of how the above graph abstraction is constructed from a conventional plan space[5]. Note that, because of the cliques formed by the 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') is equal to $c(s, b)$ for all s', which implies that each aforementioned clique forms a plateau.

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 Fig. 3, 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. Fig. 5). 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 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.

[5] Strictly speaking, some nodes, such as (t, s) and (t, t), in Fig. 5 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 two types of edges discussed here are merely for the purpose of presentation.

Based on the above, in the next subsection, we use all the results derived for each conventional optimization 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 $G^*[b]$.

4.5 'Well' shape of cost function

As mentioned in Sect. 2, the key factor that determines 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 as above forms a very definitive 'well'. 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, and

$$c((s, t), b) \geq c((g, g), b),$$

because g is a global minimum in $G[b]$.

Hence, the only local minimums in the new graph $G^*[b]$ are also global minimums, and they are all mutually connected. The above implies that a 'perfect well' is formed.

The effectiveness of randomized algorithms does not only depend on the formation of a 'well', but also on the precise node connections that give rise to the 'well'. In the above case, the 'well' was formed in somewhat of a brute-force way, by having all pairs of nodes of the form (s, s') and (s, t) for some s be connected into a clique with the same cost. In general, such a 'well' should not be expected to be useful, since choosing the appropriate neighbor (s, t) of (s, s') so that a downhill move to (g, g) can then become possible is equivalent to choosing randomly among all conventional plans. Thus, the graph structure does not naturally guide a randomized algorithm towards the 'well'-bottom and the global minimum. Nevertheless, we claim that $G^*[b]$ is very good for executing R in a way that captures

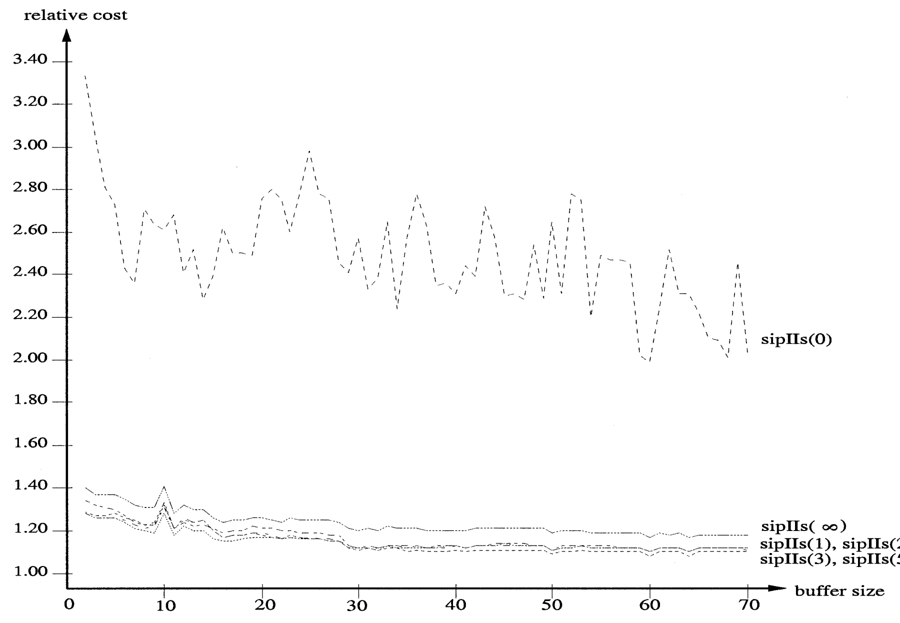

Fig. 6. Average relative cost of local minimum functions found by sipIIs(k)

the behavior of R[b] on $G[b]$ under the control of sipR(k), because, when moving from (s, s') to (s, t), t is the current plan of some friend of R[b]. Hence, the choice is not random among all plans, but actually guided by the sideways information passing. Given the fact that the cost to buffer ratio is monotonically decreasing, good plans for one buffer size are likely, though not necessarily, to be good for similar buffer sizes as well. Thus, the expectation is that the combination of the 'well'-shape of $G^*[b]$ with execution control that is based on sideways information passing is very powerful for effective parametric query optimization.

To verify the above theory on the 'well'-shape of a cost function and also the effectiveness of sideways information passing, we present below two sets of experimental results. The first experiment involved running many local optimizations under sipIIs(k) for $k = 0, 1, 2, 3, 5$, and ∞, and comparing the cost of the *average* local minimum plan functions obtained by sipIIs(k), $k > 0$, with those obtained by running sipIIs(0). As mentioned above, sipIIs(0) captures the conventional case where no nontrivial sideways information passing occurs.

Before presenting the results of this experiment, we first describe the approximation that we used for identifying local minimum plan functions. When no sideways information passing occurs, a plan is considered to be a local minimum in $G[b]$ after n randomly chosen neighbors of it are tested (possibly with repetition), where n is the actual number of its neighbors, none of which has lower cost. Note that this does not guarantee that all neighbors are tested, since some may be chosen multiple times. A plan that satisfies the above operational definition is called an *r-local minimum* (*r* for *r*andom, since it is based on examining random neighbors), to distinguish it from an actual local minimum. When sideways information passing occurs, we need to approximate entire local minimum plan functions. We again use an operational definition, but this time $G^*[b]$ is the underlying graph. A plan function $s()$ is an *r-local minimum function* if, for every buffer size b, $(s(b), s(b))$ is an r-local minimum in $G^*[b]$ and, for all buffer sizes b' such that R[b'] is a friend of R[b], $(s(b), s(b'))$ is also an r-local minimum in $G^*[b]$. This definition has been used to end a local optimization both in this experiment and the ones reported in Sect. 5 on the behavior of the algorithms. In all results presented in the rest of the paper, whenever we refer to a local minimum (individual plan or plan function), the intended meaning is always that of an r-local minimum.

Using the r-local minimum approximation, we run sipIIs(k) for each value of k on five 20-join queries for the buffer range [2,70]. (See Sect. 5.2 for more details on the experiments.) Typically, around 25 local minimum functions were found for each run of sipIIs(k). Figure 6 shows the average relative cost of local minimum plan functions found by sipIIs(k), $k = 0, 1, 2, 3, 5$, and ∞. (Recall from Sect. 3.2 that the relative cost is the ratio of the actual cost over the cost of the plan function found by individual conventional query optimizations for each buffer size.)

Note the dramatic drop in the cost when k increases from 0 to higher values. This is due to sideways information passing, which allows plans that are perceived as local minima by sipIIs(0) to be compared with plans of lower costs, and therefore to be no longer regarded as local minima. This has the effect of removing some of the high-cost local minima, which also reduces the average local minimum cost. As shown in Fig. 6, for all buffer sizes, the cost reduction between $k = 0$ and $k = 1$ is substantial. On the other hand, further increases in the value of k do not give further reductions in cost. This is due to the fact that, as long as there is sideways information passing, the plan associated with some buffer size is influenced directly or transitively by changes in the plans associated with all other buffer sizes.

One additional interesting result of the above experiment is that, with sideways information passing, not only are the average local minimum costs lower, but their standard deviations are also dramatically reduced. Table 1 shows the standard deviation of the plan costs over all buffer sizes of a local minimum plan function when sipIIs(k) is used

Table 1. Average standard deviation of local minimum functions found by sipIIs(k)

sipIIs(0)	sipIIs(1)	sipIIs(2)	sipIIs(3)	sipIIs(5)	sipIIs(∞)
70.0%	12.8%	12.8%	13.6%	10.7%	15.4%

Table 2. Percentage of local minima replaced due to hill-jumping for sipIIs(k)

sipIIs(1)	sipIIs(2)	sipIIs(3)	sipIIs(5)	sipIIs(∞)
25%	26%	27%	31%	32%

with $k = 0, 1, 2, 3, 5$, and ∞. The standard deviations are expressed as percentages to their respective averages. For example, over the buffer range [2,70], the average standard deviation for sipIIs(0) is 70% of the corresponding average cost, while that for sipIIs(1) is only about 13% of the corresponding average cost. As these figures indicate, the 'well' of the cost function without sideways information passing is much bumpier than the corresponding 'well' with sideways information passing.

The second experiment that we performed is based on the following observation. Due to sideways information passing, a plan s in $G[b]$ may be replaced by another plan t in $G[b]$ such that the following condition holds: for any path connecting s and t, there exists some third plan s' whose cost is higher than both the costs of s and t. This kind of replacement, which we refer to as *hill-jumping*, is one major reason why sideways information passing is beneficial. Ideally, a direct way to verify the benefits of sideways information passing would be to count the number of times hill-jumping occurs. This is, however, very costly to implement. Instead, while running the experiment described above, we counted the occurrences of a special kind of hill-jumping, i.e., replacement of an r-local minimum in $G[b]$ (or equivalently replacement of an r-local minimum of the form (s, s) in $G^*[b]$). More specifically, we counted the number of times that a plan that has been identified[6] as an r-local minimum in $G[b]$ is replaced due to hill-jumping. Table 2 shows the average of that number over all buffer sizes in the form of a percentage over the total number of r-local minima identified for sipIIs(k), $k = 1, 2, 3, 5$, and ∞. Clearly, this special kind of hill-jumping occurs quite frequently, thus verifying once more the 'well'-shape of the cost function and the benefits of sideways information passing.

Very similar results with respect to the average cost of local minima, its standard deviation, and the occurrence of hill-jumping were obtained when sipIIr was used instead of sipIIs, so we comment on those experiments no further.

5 Algorithm behavior

5.1 Algorithm implementation

We implemented all four algorithms sipIIs/r(k) and sip2POs/r(k) for several values of the depth k. In this section, we discuss the most important details of these implementations, in particular, the overall data structure for a state, the data structure for maintaining the costs of the various plans in a state and the individual joins in each plan for different buffer sizes, and the timing of when these costs were updated. The values of various parameters of the randomized algorithms are also important to their implementation. For those, we adopted the setup that has been used for conventional query optimization [IK90, Kan91].

Recall that a state in parametric query optimization is a plan function. As we discussed in Sect. 4.3, the plans in different image partitions of the current plan function may have common subplans which can be shared. However, the overhead of maintaining a plan graph where sharing common subplans may be shared across image partitions is quite expensive. We thus implemented sharing only within the same image partition, i.e., the plan function is an array, with one entry for each image partition, holding the corresponding plan. The entries in the array are in sorted order of the buffer sizes of the corresponding partitions. This reduces the time taken to find the friends of a given partition.

The plan in each entry of the above list is maintained as a tree structure, with several pieces of pertinent information stored in each node. The key difference between the above and the corresponding tree structure for conventional query optimization is that, for parametric query optimization, each join node maintains cost information for many buffer sizes. Specifically, for each join node, an array is used with as many entries as the number of buffer sizes examined in the experiment. Each entry of the array holds the cost of that join for the buffer size corresponding to the array entry. A similar array is associated with the entire plan to hold its cost for the different buffer sizes.

An important question is whether or not we should always maintain the costs in all entries for the arrays in all join nodes of a plan. Doing so has the advantage that, when a plan is transformed to one of its neighbors, the cost in each array entry for only a couple of join nodes must be updated (except when interesting sort orders change and the change must be propagated to more join nodes). This update can be done in constant time for each array entry. The disadvantage of this approach is that many of these entries are overwritten before ever being used, and thus computing them represents pure overhead. A small set of experiments showed that the overhead outweighed the benefits of the constant-time cost calculation. Hence, we decided to calculate cost entries on an as-needed basis. Specifically, when a neighbor plan is generated, the costs of all the join nodes and the total cost of the new plan are calculated and stored only for the buffer sizes in the range $[b^- - k, b^+ + k]$, where b^- and b^+ are the minimum and maximum buffer sizes of the image partition of the current plan, respectively, and k is the depth of the algorithm. If for a buffer size b' in the above range the relevant costs of the current plan are known, then the corresponding costs of the new plan can be found in constant time. (This is true at least for all $b^- - k \leq b' \leq b^+ + k$.) Otherwise, all these costs are calculated from scratch in a bottom-up fashion for the entire plan tree.

Figure 7 is used to illustrate some of the data structures mentioned above, the costs that are calculated at each step of the algorithms, and in general their overall flow. Specifically,

[6] It is possible that before its minimality can be detected, an r-local minimum is replaced due to sideways information passing. We are not able to capture this phenomenon in our counts, so the numbers included in Table 2 represent lower bounds.

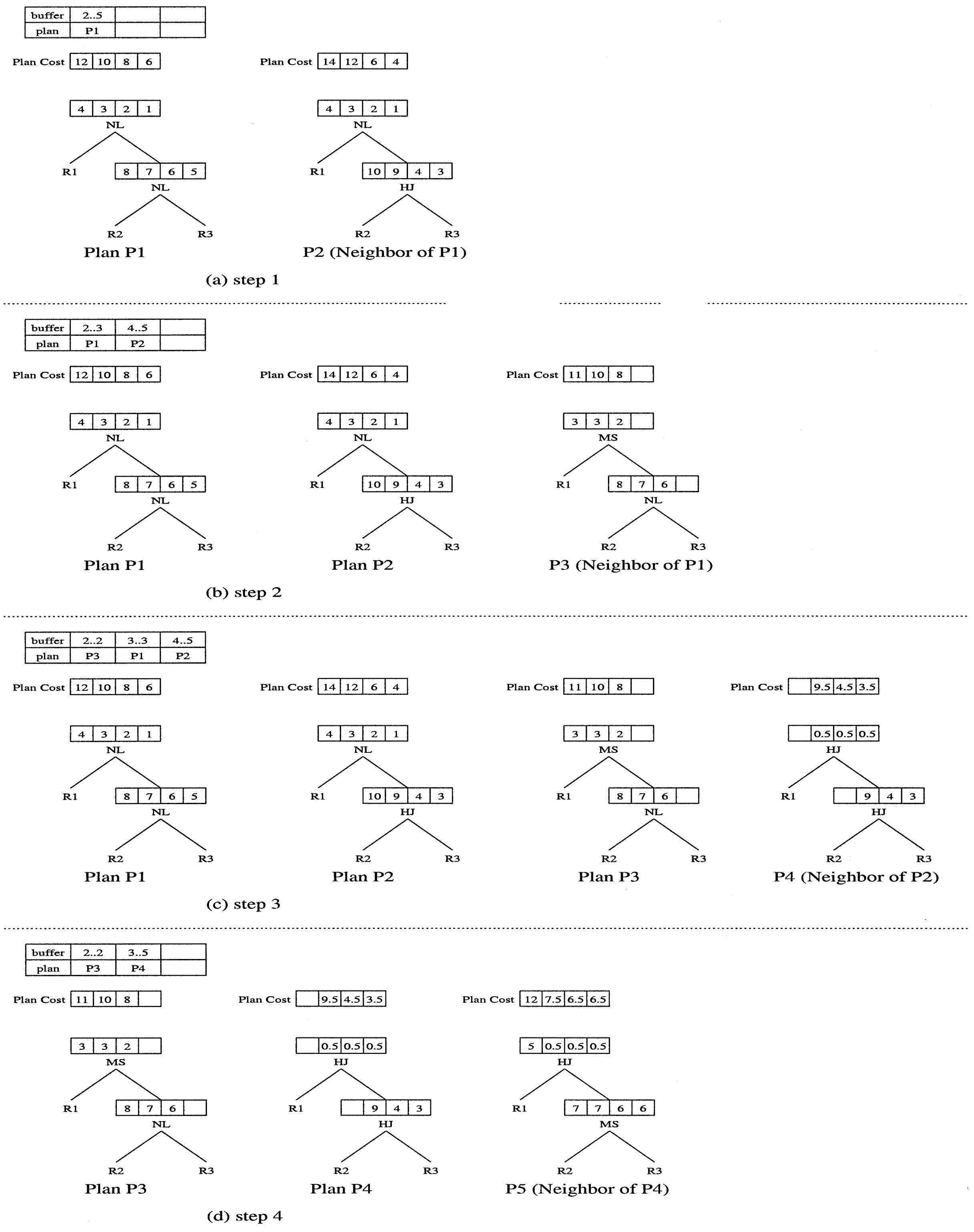

Fig. 7a–d. Steps of sipIIs(1)

it shows four steps of sipIIs(1) starting from a state with a single random plan. The buffer range in the figure is [2..5] and all other numbers for costs are artificial and are used for illustrative purposes only. To keep the example simple, we also confined ourselves to using only the join method transformation rule to create neighbors.

The table at the top of each step represents the plan function corresponding to the current state. For example, the table in Fig. 7a indicates that $P1$ is the current plan for buffer size range [2..5]. The array called ‘Plan Cost’ above each plan represents the total cost of the plan for each buffer size, and the array in each join node of a plan represents the

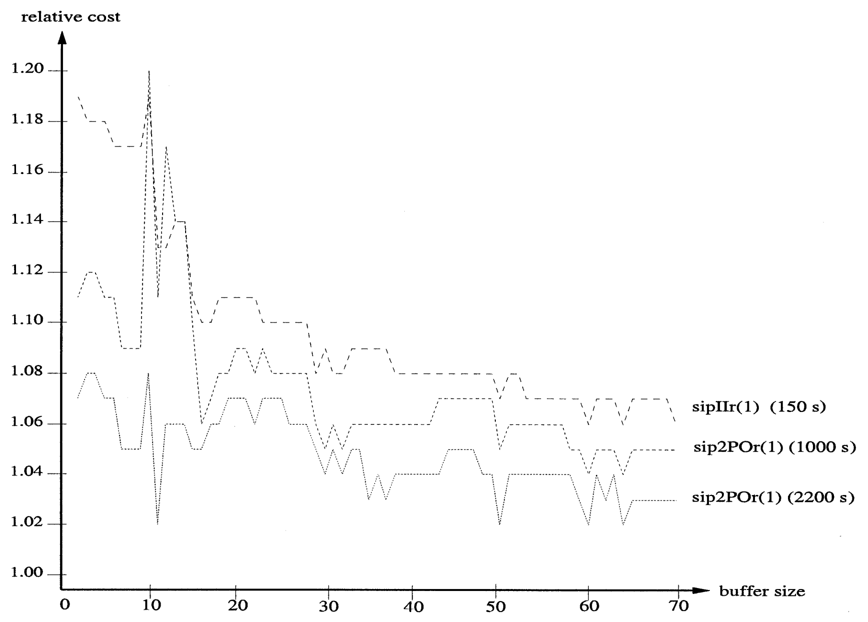

Fig. 8. Output quality of sip2POr(1) and sipIIr(1)

cost of the corresponding join for each buffer size. Since the range of operation has four buffer sizes, all these arrays have length 4 as well. In plan $P1$, the join costs of the two joins for buffer size 2 are 4 and 8. Thus, the total cost of the plan for buffer size 2 is equal 12 (the sum of the individual costs). For each step, the rightmost plan shown is a neighbor of some plan in the current state and is the one examined for a possible move to it.

In step 1, $P2$ is a neighbor of $P1$ and is obtained by changing the join method of the internal join of $P1$. Because the costs for all buffer sizes are available in $P1$, all the corresponding entries for $P2$ are calculated in constant time. A comparison of these costs results in the plan function shown at the top of Fig. 7b.

In step 2, plan $P3$ is generated, which is again a neighbor of $P1$. Since the image partition of $P1$ is [2..3] and the depth is $k = 1$ buffer size, the cost of $P3$ must be calculated for the buffer range [2..4] for comparison. Again, all these calculations may be done in constant time (assuming that no sort order needs to be propagated). The entries for buffer size 5 remain empty for all cost arrays of $P3$. If at some later point these costs are needed, they will have to be calculated from scratch. A comparison of the generated costs with those of $P1$ and $P2$ for ranges [2..3] and [4..4], respectively, results in the plan function shown at the top of Fig. 7c.

In step 3, a neighbor of $P2$ is generated (plan $P4$) and its costs for buffer size range [3..5] are calculated similarly to step 2. The resulting plan function is shown at the top of Fig. 7d. In step 4, a neighbor of $P4$ is generated (plan $P5$). Because the image partition of $P4$ is [3..5], the cost of $P5$ for the entire buffer size range [2..5] must be calculated. This is slightly more complicated than in the previous steps, because the cost of $P4$ for buffer size 2 is unknown. Hence, the cost of $P5$ for that buffer size must be calculated from scratch, while the costs for the buffer size range [3..5] may be calculated in constant time.

5.2 Experiment testbed

We ran several experiments to test the effectiveness of the implemented algorithms for query optimization. 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[7]. 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 30 ms. Cost formulas are listed in the Appendix. In what follows, unless otherwise stated, the results presented for each algorithm are averages of five runs of the algorithm on each of ten 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 Sect. 3.2) and obtained its average running time over all buffer sizes. We then allowed sipII to run for exactly that amount of time

[7] This was the most varied catalog (catalog ‘relcat3’) that we used in previous experiments [IK90].

Table 3. Output quality of sipIIr(k) and sipIIs(k)

	k = 0	k = 1	k = 2	k = 5	k = 10	$k = \infty$
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

on the query. Thus, sipII used the same amount of time to optimize a query over a range of buffer sizes as 2PO used on the average to optimize that query for a single buffer size. On the other hand, sip2PO was run for as long as its second phase (SA) needed to converge.

5.3 Sip2PO versus sipII

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, Fig. 8 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 different versions of sip2POr(1) that took around 1000 s and 2200 s, respectively. When compared with sipIIr(1), which only takes around 150 s, 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 Sect. 4.5 that $G^*[b]$ forms a ‘perfect well’. In addition, even the r-local minimum approximations found by sipIIr(k) and sipIIs(k), $k > 0$, have much lower costs than those found by sipIIr(0) and sipIIs(0), respectively (both with respect to their average and their standard deviation). 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.

5.4 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 3 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 an equal amount of time). The specific results are for 20-join queries, but similar results were obtained for other queries as well. As Table 3 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 3 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 five buffer sizes. Second, for corresponding k values, the output quality of sipIIs(k) is consistently better than that of sipIIr(k)[8]. 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.

Table 4. Average time given to sipIIs(k) for queries of different sizes

Query size (joins)	1	3	5	7	10	20
T (s)	2	14	27	42	62	158

5.5 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 average 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 4. We performed additional experiments where the amount of time given to sipIIs(1) was T/3, 2T/3, T, and 2T. Figure 9 shows the results of the combined 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.

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

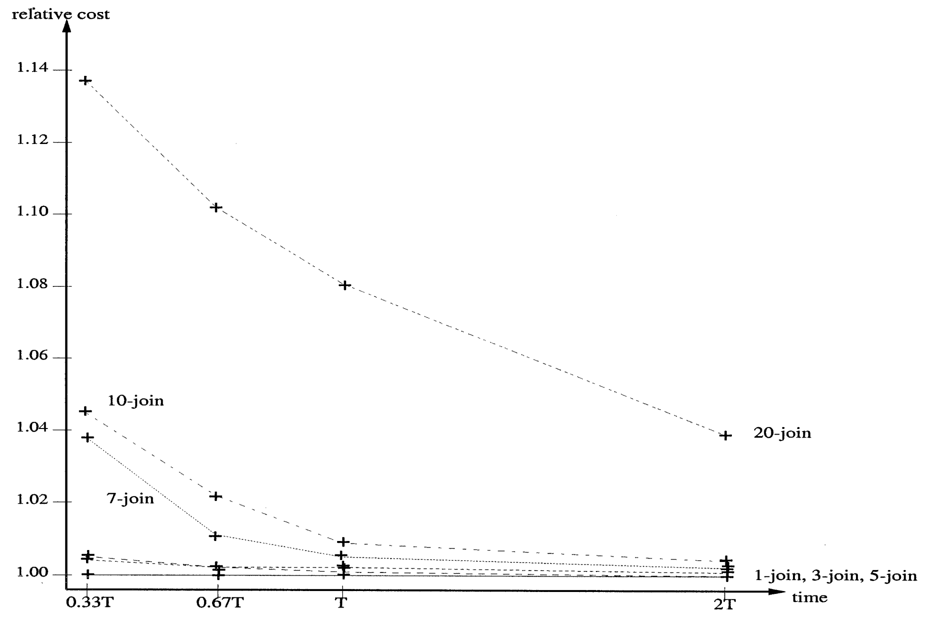

Fig. 9. Output quality of sipIIs(1) with varying experimentation times

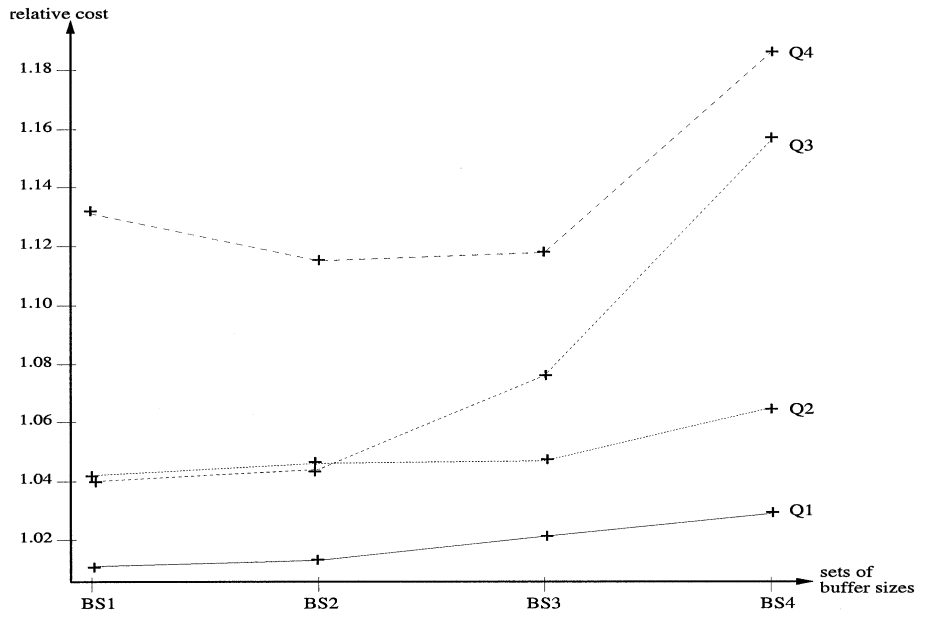

Fig. 10. Output quality of sipIIs(1) for different sets of buffer sizes

6 Other issues

6.1 Incomplete sets of buffer sizes

In this subsection, we study the effect on the optimizer effectiveness of dealing only with a subset of all possible values of the unknown run-time parameter. This may be necessary when the domain of parameter values is very large, in which case generating plans for each different value is impractical. In that case, the resulting partial plan function would be *implicitly* extended to the full domain of values by some form of 'interpolation'. For example, if the actual parameter value at run time is not in the optimized subset, then the plans generated for its 'nearest neighbors' should be examined and the best one among them should be chosen for processing.

We have used sipIIs(1) to solve this modified parametric query optimization problem for the buffer size parameter. For any set B' of buffer sizes, the friends of R[b] are all co-routines R[b'] where b' belongs to the same image partition as b or b' is the immediately largest or immediately smallest buffer size in B' outside of the image partition of b. We have experimented with B' being equal to each of the following four sets:

$BS1$ =
$\{b|2 \leq b \leq 70\}$, (canonical case based on Formulation B)
$BS2$ =
$\{b|2 \leq b \leq 70$ and b is even $\}$

$BS3 =$
$\{2,3,4,5,7,10,15,20,25,30,35,40,45,50,55,60,70\}$
$BS4 =$
$\{2,5,10,20,30,50,70\}$.

The sets have been constructed in a way so that $\forall i, |BS(i+1)| < |BSi|$. The specific choices for the 'nonuniform' sets $BS3$ and $BS4$ were motivated by the observation that the image partitions of small buffer sizes tend to be smaller than those of large buffer sizes. Hence, in both $BS3$ and $BS4$, the density of small buffer sizes is higher. For all sets $BS2 - BS4$, the plan associated with a missing buffer size is the better of the plans associated with the closest buffer sizes (one larger and one smaller) that are in the set.

For each one of the above sets, sipIIs(1) was given the same amount of time. Figure 10 shows the average over all buffer sizes of the relative cost of the output plan function for each set. It contains data for four representative 20-join queries (Q1-Q4), which differ in how close to the optimal plan function was the output plan function of sipIIs(1) for them. Each of the remaining of the ten queries with which we experimented was similar to one of these four. For almost all queries (Q1-Q3), using the contiguous range [2,70] (set $BS1$) gave the best results. Even for query Q4 where $BS2$ and $BS3$ were better, the results for $BS1$ were close. Almost always, as the buffer size set became smaller, the output quality became worse. On the other hand, the output quality degradation was not dramatic: between BS1 and BS4, the cardinality of the buffer size sets dropped by an order of magnitude, while the relative cost of the output plan barely increased by 10% in the worst case (Q3). These general observations were true not only for the average plan cost over all buffer sizes but for most individual buffer sizes as well. Hence, we conclude that at least for the buffer size parameter and for the ranges that we discussed, if given the choice, using the full domain of values is preferred, but using a much smaller subset of it is also quite effective.

6.2 Storage overhead of output plan functions

Another issue that arises in parametric query optimization is the number of image partitions of the output plan function. Ideally, one wants to have considerably fewer partitions than the domain of that function, so that only a few plans are stored in the database.

To provide some feeling for whether this is a serious problem or not, we present in Table 5 some relevant results from our experiments with the buffer size parameter. The first row of the table contains numbers of image partitions and is the focus of most of the rest of the discussion, whereas the second row contains the size of the corresponding storage space in bytes. All numbers are averages over ten 20-join queries. With respect to the columns of Table 5, the first column is provided only for reference and contains the total number of buffer sizes examined, which is an upper bound on the number of image partitions. The next column shows the average number of image partitions based on the output plan function of sipIIs(1). This number turns out to be very high (clearly impractical). However, many of these plans form groups such that the members of each group belong in a plateau and can therefore be replaced in a simple post-processing step by a single plan without affecting the output quality. The resulting average number of image partitions of such a step is shown in the third column: the dramatic drop to very reasonable numbers is clear. This number may still be high for some applications, so we have also experimented with two other postprocessing steps, which put plans in the same group if their costs for the corresponding buffer sizes differ by no more than 5% and 10%, respectively. These results are shown in the last two columns of Table 5. The 5% experiment brings another factor of 2 reduction in the number of image partitions, whereas the 10% experiment does not offer any further significant improvements, indicating that for efficient processing of a query over a large range of buffer sizes, one needs to store approximately six or seven distinct plans.

Hence, using sipIIs(1) with some postprocessing step of the form mentioned above appears to be adequate with respect to the final number of plans. In fact, the second row of Table 5 shows that, even without any postprocessing, the space required to store the alternative plans is not really significant.

6.3 Overhead for run-time decision

Recall that the primary goal of parametric query optimization is to have minimal run-time optimization overhead, i.e., all optimization occurs at compile time and a simple table look-up determines the appropriate access plan at run time (Fig. 1). As shown below, our measurements of the run-time overhead verify that this goal has been achieved.

Our implementation of the run-time optimal-plan selection is based on storing the plan of each image partition as a separate tree and building a 1-level index on top of the set of all plans, like the array at the top of every part of Fig. 7. That is, given an image partition (i.e., its boundary buffer sizes), the index returns a pointer to the corresponding plan. Even without any grouping of the final plan function, the average number of image partitions was 51 for 20-join queries (Sect. 6.2), so the entire index occupies a single disk block. In addition, each 20-join query access plan occupies one or two additional blocks. So, the entire I/O cost of the run-time decision has been two to three block accesses. The corresponding CPU time comes essentially from performing a handful of comparisons within the index block to identify the appropriate plan and is, therefore, insignificant. Hence, the overall overhead of the run-time optimal-plan selection is essentially zero (i.e., far less than 1 s); the method has achieved our original goal.

7 Algorithm extensions

7.1 Parametric query optimization for other parameters

Our focus so far has been on the single parameter such as buffer size and index parameters. To examine the effectiveness of the devised algorithms for general parametric query

Table 5. Average size of the output plan function of sipIIs(1)

	Number of buffer sizes	No grouping	= grouping	5% grouping	10% grouping
Number of image partitions	69	51.4	21.4	6.2	5.0
Storage space (Kbytes)		62.2	17.7	6.4	5.5

Table 6. Output quality of sipIIs(1) for the index parameter

Query size	1 join	3 joins	5 joins	7 joins	10 joins
Relative cost	1.00	1.00	1.00	1.01	1.13

optimization, we have performed a limited set of experiments with the index parameter. We chose this parameter because most others are similar to the buffer size parameter in that there is a natural continuity and total order in their domain of values, which are expected to make those parameters behave similarly in parametric query optimization. We experimented again with ten queries ranging from 1 join to 10 joins. For 10- and 20-join queries, changing the index of one relation does not affect the cost of plans that much, so experimenting only with 10-join queries was enough. The values of the index parameter were taken from the set $\{no\_index, non\_clustered\_Btree, non\_clustered\_hashing, clustered\_Btree, clustered\_hashing\}$. These values were considered to be ordered as in the above set, and co-routine friends were determined based on that order, as for the buffer size case. The specific ordering tried to approximately capture some monotonicity between the index value and the cost of a relation scan, although there were clearly many cases where this approximation was inaccurate.

Table 6 contains the average relative cost of the output plan function of sipIIs(1) as a function of the query size. For all queries with up to seven joins, sipIIs(1) found a plan function that is very close to the optimal. For 10-join queries, the relative cost of the output was higher than the corresponding cost found when the buffer size parameter was varied. As mentioned above, for such queries, the cost is relatively insensitive to a single index change. Therefore, the opportunities for hill-jumping are very limited, and the algorithm behaves similarly to the case where a fixed index type is considered, where the second phase of 2PO is necessary for effective optimization. Nevertheless, the overall results are very promising and indicate that randomized algorithms enhanced with sideways information passing may be the generic answer to parametric query optimization for arbitrary parameters.

Both the buffer size and the index type are discrete parameters. In reality, there are continuous run-time parameters as well, e.g., predicate selectivity. Clearly, for such parameters, one cannot deal with all their possible values, because there is an infinite number of them. One simple extension of our algorithms to handle this case is to use some selected parameter values during optimization, as was done with incomplete sets of buffer sizes in Sect. 6.1. Then, given a specific parameter value at run time, the algorithm would choose between those plans found for the selected parameter values closest to the given one.

7.2 Scaling to multiple parameters

Abstractly, applying our parametric query optimization approach to multiple parameters is straightforward. The only difference is that the parameter space for the plan functions becomes multidimensional. This requires an efficient multidimensional data structure to maintain the image partitions in the plan function, during the execution of the algorithm and also when storing the final result for use at run time. In addition, friends should be defined in multiple dimensions, but otherwise the notion of sideways information passing remains identical. Other than that, our implementation remains valid and can be easily applied. Of course, in practice, the key questions that arise include the running time of the algorithm, the cost of the resulting plan function relative to the actual optimal, and the number of resulting image partitions, which determines the run-time overhead as the multidimensional index on the plan function is traversed. Investigating the effectiveness of our parametric query optimization techniques for multiple parameters is beyond the scope of this paper, but is part of our current and future effort.

8 Dynamic programming algorithms

All current relational database systems that we are aware of use a heuristically pruning, dynamic programming algorithm for query optimization, usually an enhanced and tuned-up version of the original algorithm of System R [SAC$^+$79]. The randomized algorithms that we have discussed in this paper represent a completely different style of optimization. It has already been shown that for small queries (approximately up to ten joins), dynamic programming is superior to randomized algorithms, whereas for large queries the opposite holds [Kan91]. In this section, we provide a preliminary discussion of how the two approaches compare when dealing with unknown run-time parameters. We distinguish between the case of allowing essentially no run-time optimization overhead, which we have called parametric query optimization and has been the focus of our work, and the case where nontrivial run-time overhead is allowed, as proposed by Cole and Graefe in what they called *dynamic query optimization* [CG94].

8.1 Zero run-time overhead

We believe that, for parametric query optimization, where one wants no run-time overhead, the break-even point between randomized algorithms and dynamic programming is at smaller queries than for conventional query optimization. This belief is supported by the fact that dynamic programming or any of its heuristic variants cannot take advantage of sideways information passing. For an n-way join query, these algorithms proceed by optimizing all k-way join subqueries of the original query for all values of k from 1 to n. For parametric query optimization, consider the co-routine abstraction again, where each co-routine runs such a dynamic programming algorithm. In that case, co-routine friends are not helpful. Using the buffer size parameter as an example, we observe that knowing the best plan for a subquery found by R[b] does not provide any information to R[b'] about the best plan that it should find. R[b'] still has to compare all alternatives among themselves and proceed accordingly. Thus, any extensions to dynamic programming that remain faithful to the principles of the algorithm will have to operate in a way similar to sipR(0). The running time of the extended algorithms should increase significantly for the same size queries compared to their conventional versions. Thus, sipIIs(1) should be preferred over dynamic programming for parametric query optimization for many more (smaller) queries than 2PO is for non-parametric query optimization.

Clearly, the above needs further investigation and also experimental verification. It is conceivable that heuristics can be added to dynamic programming that not only find an approximation to the optimal plan function but which can make use of sideways information passing. Although we believe that sipIIs(1) will still remain the preferred algorithm, we plan to study these alternatives and resolve these issues.

8.2 Non-zero run-time overhead

To obtain an effective algorithm based on dynamic programming, Cole and Graefe proposed the *dynamic query optimization* approach [CG94], which leaves part of the optimization process for run time. Specifically, this approach is based on the notion of *incomparability of costs*. In particular, the cost of a plan is not a single value but an interval of values, corresponding to the minimum and maximum cost of the plan obtained over the entire range of possible parameter values (e.g., buffer sizes). Alternative plans are compared based on their cost intervals. When the minimum cost of a plan is higher than the maximum cost of another, then the former can be pruned, but when the two cost intervals overlap, then both plans are kept until run time. In other words, instead of spending significant amount of compile time to identify the exact image partitions and the corresponding optimal plans, partial ordering of plans is allowed. At run time, the cost of the partially ordered plans is re-evaluated using the actual run-time parameter values, and the optimal plan for the occasion is identified.

Clearly, this approach significantly reduces the required compile time by shifting some of the decisions to run time, where parameter values are known, and makes dynamic programming effective. The resulting run-time overhead, however, may be significant, since essentially *all* plans stored must be read from disk and have their cost re-evaluated. Although sharing of common nodes across partially ordered plans reduces not only the storage space but also the time it takes to re-evaluate the cost of the plans, that time may still be non-negligible. For example, in the experiments of Cole and Graefe, the run-time decision overhead went up to 74% of the query processing time [CG94].

Table 7. Symbols used in cost formulas

$C_{I/O}$	number of page accesses
C_{CPU}	number of CPU instructions
$P(R)$	number of pages in R
$T(R)$	number of tuples in R
$V(A,R)$	number of unique values for attribute A in R
$B(A,R)$	number of buckets in hash table for attribute A in R
$LP(I)$	number of leaf pages of index I
$D(I)$	depth of B-tree index I
C_{comp}	number of instructions to compare keys in main memory
C_{hash}	number of instructions to hash a key
C_{move}	number of instructions to move a tuple in main memory
C_{swap}	number of instructions to swap two tuples in main memory

Whether systems should strive for zero run-time overhead or not is a question whose answer may be partly based on philosophical and stylistic grounds. Beyond that, however, it also depends on the exact comparison of the compile-time costs of sipIIs(1) and the Cole-Graefe dynamic programming algorithm. Such a comparison is beyond the scope of this paper, but will be part of our future work.

9 Conclusions and future work

We have formalized the problem of parametric query optimization and studied it primarily with regard to 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 loss 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. Finally, we have provided evidence that these algorithms are applicable to the general form of parametric query optimization. Experiments with the index parameter have shown that sipIIs(1) once again can obtain multiple query plans with very comparable output quality in short times.

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, makes no assumptions about any properties of the plan costs, and incurs no run-time overhead. 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 very significant.

There are several issues that we plan to address in our future work. The most important ones are a detailed comparison of randomized algorithms and the dynamic programming algorithm (especially the Cole and Graefe approach) for parametric query optimization, and comprehensive experimentation with large vectors of diverse parameters to understand the scalability of the developed algorithms. 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.

Appendix

Cost formulas for joins

This appendix lists all the cost formulas for joins that we used in our experiments. Selectivities, relation sizes, and buffer sizes are parameters of these formulas, which include both I/O and CPU costs. These formulas are based on the assumptions that i) values in each column are uniformly distributed, ii) LRU is the policy used in page replacement, and iii) there is no data sharing in the sense that each query can only access data in its own buffers.

Table 7 defines the symbols used in the cost formulas. While most of these symbols are self-explanatory, it is worth pointing out that we express CPU costs in units of CPU instructions, and I/O costs in units of page accesses. In our experiments, we assume that each CPU instruction takes 0.001 ms, and that each I/O takes 30 ms. Furthermore, as we use Shapiro's cost formulas for hash joins [Sha86], we set $C_{comp}, C_{hash}, C_{move}$ and C_{swap} to 3, 9, 20 and 60 units, respectively, to ensure compatibility of our formulas with his. Finally, we use the notation $R \bowtie_A S$ to represent a join between the outer relation R and the inner relation S on attribute A. Joins on more than one attribute are not considered here and in our experiments.

There are three join methods considered in our experiments – hash join, nested loops, and merge scan. Here, we only include the formulas for the latter two methods; the formulas for hash join are listed in [Sha86].

A Nested-loops joins

In this section, we examine three different cases for nested loop joins $R \bowtie_A S$. We first consider the case when there is no index on attribute A for the inner relation S. Then we study the cases when there is either a hash index or a B-tree index for S. In all three cases, we use file scan for relation R.

A.1 File scan for S

If both the outer and inner relations have no index on A, then there are two cases, depending on the number of available buffers b. If $P(S) \le (b-1)$, we can read in all pages of S. Then, the formulas for CPU and I/O costs are:

$$C_{CPU} = T(R) * T(S) * C_{comp}\,,$$
$$C_{I/O} = P(R) + P(S)\,.$$

In calculating the CPU costs, we assume in our experiments that the cost of opening and closing a relation or an index is 0. When $P(S) > (b-1)$, the cost formulas become:

$$C_{CPU} = T(R) * T(S) * C_{comp}\,,$$
$$C_{I/O} = P(R) + \lceil \frac{P(R)}{b-1} \rceil * P(S)\,.$$

A.2 Hash index for S

There are two cases: one where the hash index is a primary index, and the other where the index is a secondary index.

A.2.1 Primary hash index. If we use the primary hash index for S, the CPU cost is given by:

$$C_{CPU} = T(R) * (C_{hash} + \frac{T(S)}{B(A,S)} * C_{comp})\,.$$

The I/O cost depends on the number of buffers b. If b is not larger than the average number of pages in each bucket, (more specifically, if $b < (1 + \lceil \frac{P(S)}{B(A,S)} \rceil)$), we assume that we only give one buffer to S. Then the I/O cost becomes:

$$C_{I/O} = P(R) + T(R) * (1 + \lceil \frac{P(S)}{B(A,S)} \rceil)\,.$$

Now, when $b \ge (1 + \lceil \frac{P(S)}{B(A,S)} \rceil)$, we assume that we give m times of $(1+\lceil \frac{P(S)}{B(A,S)} \rceil)$ buffers to S, where m is the largest integer such that $b \ge m * (1 + \lceil \frac{P(S)}{B(A,S)} \rceil)$. In other words, the buffers are divided into m partitions, each of which can contain all the pages of an average bucket. Then the accessing of pages in S can be viewed as a random reference to the buckets. In [NFS91], we derive the following formula for approximating (very closely) the expected number of page accesses for a random reference of length k to a file of size N using s buffers:

$$C_{I/O} \approx \begin{cases} N * [1 - (1 - 1/N)^{k_0}] & k < k_0 \\ s + (k - k_0) * (1 - s/N) & \text{otherwise} \end{cases}, \qquad (1)$$

where $k_0 = \ln(1 - s/N) \,/\, \ln(1 - 1/N)$. Applying Eq. 1 by setting $N = B(A,S)$, $k = T(R)$, and $s = m$, we obtain the I/O cost for the case when $b = m * (1 + \lceil \frac{P(S)}{B(A,S)} \rceil)$:

$$C_{I/O} \approx \begin{cases} P(R) + (1 + \lceil \frac{P(S)}{B(A,S)} \rceil) * B(A,S) * [1 - (1 - \frac{1}{B(A,S)})^{k_0}] \\ \quad T(R) < k_0 \\ P(R) + (1 + \lceil \frac{P(S)}{B(A,S)} \rceil) * [m + (T(R) - k_0) * (1 - \frac{m}{B(A,S)})] \\ \quad \text{otherwise} \end{cases}$$

where $k_0 = \ln(1 - \frac{m}{B(A,S)}) \,/\, \ln(1 - \frac{1}{B(A,S)})$. This completes our analysis when the hash index is a primary index.

A.2.2 Secondary hash index. Now consider the situation when we use a secondary hash index. Let I be the index and $B(A,I)$ denote the number of buckets for I. Then the CPU cost is given by:

$$C_{CPU} = T(R) * (C_{hash} + \frac{T(S)}{B(A,I)} * C_{comp})\,.$$

As for the I/O costs, it again depends on the number of buffer b. If $b < (3 + \lceil \frac{P(I)}{B(A,I)} \rceil)$, we give one buffer each to R, S and I. Then the I/O cost becomes:

$$C_{I/O} = P(R) + T(R) * (1 + \lceil \frac{P(I)}{B(A,I)} \rceil + \frac{T(S)}{V(A,S)}) .$$

Otherwise, when $b \geq (3 + \lceil \frac{P(I)}{B(A,I)} \rceil)$, we give 1 buffer to each of R and S, but $m * (1 + \lceil \frac{P(I)}{B(A,I)} \rceil)$ buffers to I. The I/O cost is then:

$$C_{I/O} \approx \begin{cases} P(R) + (1 + \lceil \frac{P(I)}{B(A,I)} \rceil) * B(A,I) * [1 - (1 - \frac{1}{B(A,I)})^{k_0}] \\ +T(R) * \frac{T(S)}{V(A,S)} \quad T(R) < k_0 \\ P(R) + (1 + \lceil \frac{P(I)}{B(A,I)} \rceil) * [m + (T(R) - k_0) * (1 - \frac{m}{B(A,I)})] \\ +T(R) * \frac{T(S)}{V(A,S)} \quad \text{otherwise} \end{cases}$$

where $k_0 = \ln(1 - \frac{m}{B(A,I)}) \,/\, \ln(1 - \frac{1}{B(A,I)})$. This completes our analysis when the hash index is a secondary index.

A.3 B-tree index for S

A.3.1 Primary B-tree index. If we use the primary B-tree index, the CPU cost is given by:

$$C_{CPU} = T(R) * \frac{T(S)}{V(A,S)} * C_{comp} .$$

As for the I/O cost, if $b < (3 + \lceil \frac{P(S)}{V(A,S)} \rceil)$, the cost is:

$$C_{I/O} = P(R) + T(R) * (D(I) + \lceil \frac{P(S)}{V(A,S)} \rceil) .$$

Otherwise, when $b \geq (3 + \lceil \frac{P(S)}{V(A,S)} \rceil)$, we give one buffer to R, one buffer to the root node of the index, one buffer to a leaf page of the index, but $(m * \lceil \frac{P(S)}{V(A,S)} \rceil)$ buffers to S. The I/O cost is then given by:

$$C_{I/O} \approx \begin{cases} P(R) + \lceil \frac{P(S)}{V(A,S)} \rceil * V(A,S) * [1 - (1 - \frac{1}{V(A,S)})^{k_0}] \\ +T(R) * (D(I) - 1) \quad T(R) < k_0 \\ P(R) + \lceil \frac{P(S)}{V(A,S)} \rceil * [m + (T(R) - k_0) * (1 - \frac{m}{V(A,S)})] \\ +T(R) * (D(I) - 1) \quad \text{otherwise} \end{cases},$$

where $k_0 = \ln(1 - \frac{m}{V(A,S)}) \,/\, \ln(1 - \frac{1}{V(A,S)})$. This completes our analysis when the B-tree index is a primary index.

A.3.2 Secondary B-tree index. If we use a secondary B-tree index, the CPU cost is:

$$C_{CPU} = T(R) * \frac{T(S)}{V(A,S)} * C_{comp} .$$

As for the I/O cost, if $b < (3 + \lceil \frac{LP(I)}{V(A,S)} \rceil)$, the I/O cost is:

$$C_{I/O} = P(R) + T(R) * (D(I) + \lceil \frac{LP(I)}{V(A,S)} \rceil + \frac{T(S)}{V(A,S)}) .$$

Otherwise, we give one buffer to each of relations R and S, one buffer to the root node of the index, but $(m * \lceil \frac{LP(I)}{V(A,S)} \rceil)$ buffers to the leaf pages of the index. The I/O cost then becomes:

$$C_{I/O} \approx \begin{cases} P(R) + \lceil \frac{LP(I)}{V(A,S)} \rceil * V(A,S) * [1 - (1 - \frac{1}{V(A,S)})^{k_0}] \\ +T(R) * (D(I) - 1 + \frac{T(S)}{V(A,S)}) \quad T(R) < k_0 \\ P(R) + \lceil \frac{LP(I)}{V(A,S)} \rceil * [m + (T(R) - k_0) * (1 - \frac{m}{V(A,S)})] \\ +T(R) * (D(I) - 1 + \frac{T(S)}{V(A,S)}) \quad \text{otherwise} \end{cases},$$

where $k_0 = \ln(1 - \frac{m}{V(A,S)}) \,/\, \ln(1 - \frac{1}{V(A,S)})$. This completes our analysis when the B-tree index is a secondary index.

B Merge-scan joins

B.1 M-way sort-merge

Before we proceed to present the cost formulas we used for merge-scan joins, we first give the formulas for sorting a relation. The sorting procedure we assumed is M-way sort-merge, the costs of which are summarized below.

$$C^{sort}_{CPU}(R) = \\ T(R) * \ln_2 \frac{T(R) * b}{P(R)} * (C_{comp} + C_{move}) + \\ (\lceil \ln_b P(R) \rceil - 1) * T(R) * (b * C_{comp} + C_{move}) , \qquad (2)$$

$$C^{sort}_{I/O}(R) = 2 * P(R) * \lceil \ln_b P(R) \rceil . \qquad (3)$$

Given b buffers, the merge-sort takes $\lceil \ln_b P(R) \rceil$ passes. In the first pass, $\frac{P(R)}{b}$ sorted runs are produced, each of which is b pages. To sort the $n = \frac{T(R)*b}{P(R)}$ tuples in each run, a CPU cost proportional to $n \ln_2 n$ is required. Thus, the total CPU cost for the first pass is given by the first line of Equation 2. Merging occurs in the second and subsequent passes. In each step in a merging pass, b tuples are compared, and the tuple with the minimum search-key value is output to a file for subsequent passes. This gives rise to a total (merging) cost corresponding to the second line of Equation 2. As for the I/O cost, $2 * P(R)$ page accesses are required for each pass.

B.2 No index for the outer relation R

There are three cases to be considered if there is no index for R (and thus R has to be sorted).

B.2.1 No index for S. First, consider the case when S has to be sorted as well. Then the combined costs are:

$$C_{CPU} = C^{sort}_{CPU}(R) + C^{sort}_{CPU}(S) + C^{mjoin}_{CPU}(R,S),$$
$$C_{I/O} = C^{sort}_{I/O}(R) + C^{sort}_{I/O}(S) + C^{mjoin}_{I/O}(R,S).$$

The sorting costs for R and S are given by Eqs. 2 and 3. The costs $C^{mjoin}_{CPU}(R,S)$ and $C^{mjoin}_{I/O}(R,S)$ represent the CPU and I/O costs of performing a merge join on R and S, after these relations are sorted. These costs depend on the number of available buffers b. If $b < (\min(\frac{P(R)}{V(A,R)}, \frac{P(S)}{V(A,S)}) + 1)$, we give one buffer to each of R and S. Then the costs become:

$$C^{mjoin}_{CPU}(R,S) = T(R) * \frac{T(S)}{V(A,S)} * C_{comp},$$

$$C^{mjoin}_{I/O}(R,S) = P(R) + T(R) * \frac{P(S)}{V(A,S)}.$$

Table 8. Extra symbols used in merge-scan cost formulas

$C_{I/O}^{sort}(R)$	number of page accesses for sorting R
$C_{CPU}^{sort}(R)$	number of CPU instructions for sorting R
$C_{I/O}^{mjoin}(R,S)$	number of page accesses for performing a merge join on R and S
$C_{CPU}^{mjoin}(R,S)$	number of CPU instructions for performing a merge join on R and S

Otherwise, suppose without loss of generality that $\frac{P(R)}{V(A,R)} \geq \frac{P(S)}{V(A,S)}$. Then we give one buffer to R and $\frac{P(S)}{V(A,S)}$ buffers to S. The costs are then given by:

$$C_{CPU}^{mjoin}(R,S) = T(R) * \frac{T(S)}{V(A,S)} * C_{comp}$$

$$C_{I/O}^{mjoin}(R,S) = P(R) + P(S)$$

This completes our analysis of the case when S does not have any index.

B.2.2 Primary B-tree index for S. Now consider the case when we use the primary B-tree index. Then the combined costs simply become:

$$C_{CPU} = C_{CPU}^{sort}(R) + C_{CPU}^{mjoin}(R,S),$$
$$C_{I/O} = C_{I/O}^{sort}(R) + C_{I/O}^{mjoin}(R,S),$$

where the costs $C_{CPU}^{sort}(R)$, $C_{CPU}^{mjoin}(R,S)$, $C_{I/O}^{sort}(R)$ and $C_{I/O}^{mjoin}(R,S)$ are exactly the same as those given in the previous subsection.

B.2.3 Secondary B-tree index for S. Finally, consider the case when we use a secondary B-tree index. While the CPU cost is still given by $C_{CPU} = C_{CPU}^{sort}(R)+C_{CPU}^{mjoin}(R,S)$, the I/O cost varies according to b. If $b < (2+\lceil \frac{LP(I)}{V(A,S)} \rceil)$, the cost is:

$$C_{I/O} = C_{I/O}^{sort}(R) + D(I) + P(R) + T(R) * \frac{LP(I)+T(S)}{V(A,S)} .$$

On the other hand, if $(2+\frac{LP(I)}{V(A,S)}) \leq b < (1+\frac{LP(I)}{V(A,S)}+\frac{t(S)}{V(A,S)})$, we give one buffer to each of R and S, and $\frac{LP(I)}{V(A,S)}$ buffers to the index. The I/O cost is then:

$$C_{I/O} =$$
$$C_{I/O}^{sort}(R) + D(I) + P(R) + LP(I) + T(R) * \frac{T(S)}{V(A,S)} .$$

Finally, if $b \geq (1+\frac{LP(I)}{V(A,S)}+\frac{T(S)}{V(A,S)})$, then we give one buffer to R, $\frac{LP(I)}{V(A,S)}$ buffers to the index, and as many buffers b_1 as possible to S (i.e. $b_1 = b - 1 - \frac{LP(I)}{V(A,S)}$). The I/O cost then becomes:

$$C_{I/O} = C_{I/O}^{sort}(R) + D(I) + P(R) + LP(I)$$
$$+ \begin{cases} P(S) * (1-(1-\frac{1}{P(S)})^{k_0}) & \text{if } V(A,R) * \frac{T(R)}{V(A,S)} < k_0 \\ b_1 + (V(A,R) * \frac{T(R)}{V(A,S)} - k_0) * (1-\frac{b_1}{P(S)}) & \text{otherwise} \end{cases}$$

where $k_0 = \ln(1-\frac{b_1}{P(S)}) / \ln(1-\frac{1}{P(S)})$.

B.3 Primary B-tree index for the outer relation R

There are once again three cases to be considered, depending on whether S has an index or not.

B.3.1 No index for S. The situation here is very similar to the one analyzed in Sect. B.2.2. The only difference is that instead of R, now S is the relation to be sorted. Thus, the costs are given by:

$$C_{CPU} = C_{CPU}^{sort}(S) + C_{CPU}^{mjoin}(R,S),$$
$$C_{I/O} = C_{I/O}^{sort}(S) + C_{I/O}^{mjoin}(R,S),$$

where the costs $C_{CPU}^{sort}(S)$, $C_{CPU}^{mjoin}(R,S)$, $C_{I/O}^{sort}(S)$ and $C_{I/O}^{mjoin}(R,S)$ are exactly the same as those before.

B.3.2 Primary B-tree index for S. The situation here is very similar to the one analyzed in Sect. B.2.1. The only difference is that now no sorting costs are required:

$$C_{CPU} = C_{CPU}^{mjoin}(R,S) ,$$
$$C_{I/O} = C_{I/O}^{mjoin}(R,S) ,$$

where the costs $C_{CPU}^{mjoin}(R,S)$ and $C_{I/O}^{mjoin}(R,S)$ are given by the same formulas listed in Sect. B.2.1.

B.3.3 Secondary B-tree index for S. The situation here is very similar to the one studied in Sect. B.2.3. The only difference is that now the cost for sorting R can be saved:

$$C_{CPU} = C_{CPU}^{mjoin}(R,S)$$

and $C_{I/O}$ is given by the three formulas in Sect. B.2.3, without the cost $C_{I/O}^{sort}(R)$ in each case.

B.4 Secondary index for the outer relation R

There are three cases depending on whether S has an index or not.

B.4.1 No index for S. The CPU cost is given by:

$$C_{CPU} = C_{CPU}^{sort}(S) + C_{CPU}^{mjoin}(R,S)$$

where $C_{CPU}^{sort}(S)$ and $C_{CPU}^{mjoin}(R,S)$ are the same as before. The I/O cost depends on b. If $b < (2+\frac{P(S)}{V(A,S)})$, the cost is:

$$C_{I/O} = = C_{I/O}^{sort}(S) + D(I_R) + LP(I_R)$$
$$+T(R) * (1+\frac{P(S)}{V(A,S)}) ,$$

where I_R denotes the index for R.

Otherwise, if $b \geq (2 + \frac{P(S)}{V(A,S)})$, we give one buffer to the index, $\frac{P(S)}{V(A,S)}$ buffers to S, and as many buffers b_1 as possible to R (i.e. $b_1 = b - 1 - \frac{P(S)}{V(A,S)}$). The I/O cost then becomes:

$$C_{I/O} = C^{sort}_{I/O}(S) + D(I_R) + LP(I_R) + P(S) + \begin{cases} P(R) * (1 - (1 - \frac{1}{P(R)})^{k_0}) & \text{if } T(R) < k_0 \\ b_1 + (T(R) - k_0) * (1 - \frac{b_1}{P(R)})) & \text{otherwise} \end{cases},$$

where $k_0 = \ln(1 - \frac{b_1}{P(R)}) / \ln(1 - \frac{1}{P(R)})$.

B.4.2 Primary B-tree index for S. The situation here is almost identical to the one analyzed above. The only difference is that now the CPU and I/O costs for sorting S can be saved. In other words, the CPU cost is given by:

$$C_{CPU} = C^{mjoin}_{CPU}(R, S)$$

and the I/O cost is given by the two formulas in Sect. B.4.1, without the cost $C^{sort}_{I/O}(S)$ in each case.

B.4.3 Secondary B-tree index for S. The CPU cost is exactly the same as in the above case, i.e., no sorting costs are required for both R and S. The I/O cost depends on b. If $b < (3 + \frac{LP(I_S)}{V(A,S)})$, where I_S denotes the index for S, we give one buffer each to R, I_R, S and I_S. The I/O cost is then:

$$C_{I/O} = = D(I_R) + LP(I_R) + D(I_S) + T(R) * (1 + \frac{LP(I_S) + T(S)}{V(A,S)}).$$

Otherwise, if $(3 + \frac{LP(I_S)}{V(A,S)}) \leq b < (2 + \frac{LP(I_S)}{V(A,S)} + \frac{t(S)}{V(A,S)})$, we give one buffer each to R, I_R and S, and $\frac{LP(I_S)}{V(A,S)}$ buffers to I_S. The I/O cost becomes:

$$C_{I/O} = = D(I_R) + LP(I_R) + D(I_S) + LP(I_S) + T(R) * (1 + \frac{T(S)}{V(A,S)}).$$

Otherwise, if $b \geq (2 + \frac{LP(I_S)}{V(A,S)} + \frac{t(S)}{V(A,S)})$, then we give one buffer each to R and I_R, $\frac{LP(I_S)}{V(A,S)}$ buffers to I_S, and as many buffers b_1 as possible to S. The I/O is then given by:

$$C_{I/O} = D(I_R) + LP(I_R) + D(I_S) + LP(I_S) + T(R) + \begin{cases} P(S) * (1 - (1 - \frac{1}{P(S)})^{k_0}) & \text{if } V(A,R) * \frac{T(R)}{V(A,S)} < k_0 \\ b_1 + (V(A,R) * \frac{T(R)}{V(A,S)} - k_0) * (1 - \frac{b_1}{P(S)})) & \text{otherwise} \end{cases}$$

where $k_0 = \ln(1 - \frac{b_1}{P(S)}) / \ln(1 - \frac{1}{P(S)})$. This completes our analysis on all the cases for merge-scan joins.

Acknowledgements. Y. E. Ioannidis was partially supported by NSF under Grants IRI-9113736 and IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Informix and Oracle. Most of the work done while R.T. Ng was with the Dept. of Comp. Science, Univ. of Maryland. Partially Supported by a University of Maryland Dissertation Fellowship, by NSF under Grants IRI-8719458 and IRI-9109755, and by NSERC under Grants OGP0138055 and STR0134419. The work was performed while K. Shim was with the Dept. of Comp. Science, Univ. of Maryland. Supported by a Korean Government Overseas Scholarship and by NSF under Grant IRI-9057573. T.K. Sellis was partially supported by NSF under Grants IRI-9057573 (PYI Award), by NASA under Grant NAS5-31351, by grants from DEC and IBM, and by the University of Maryland Institute for Advanced Computer Studies.

References

[CG94] Cole RL, Graefe G (1994) Optimization of dynamic query evaluation plans. In: *Proc. of the 1994 ACM-SIGMOD Conference on the Management of Data*, 150–160 pp, Washington, D.C.

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

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

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

[GW89] Graefe G, Ward K (1989) Dynamic query evaluation plans. In: *Proc. of the 1989 ACM-SIGMOD Conference on the Management of Data*, Portland, Ore., pp 358–366

[HP88] Hasan W, Pirahesh H (1988) Query rewrite optimization in starburst. Technical Report RJ 6367, IBM Almaden Research Center

[HS91] Hong W, Stonebraker M (1991) Optimization of parallel query execution plans in xprs. In: *Proc. of the 1st International PDIS Conference.* Miami, Fla., pp 218–225

[IK90] Ioannidis YE, Kang Y (1990) Randomized algorithms for optimizing large join queries. In: *Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data*, Atlantic City, N.J., pp 312–321

[IK91] Ioannidis YE, Kang Y (1991) 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*, Denver, Colo., pp 168–177

[IW87] Ioannidis YE, Wong E (1987) Query optimization by simulated annealing. In: *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*. San Francisco, Calif., pp 9–22

[Kan91] Kang Y *Randomized algorithms for query optimization.* PhD thesis, University of Wisconsin, Madison, Wis.

[KGV83] Kirkpatrick S, Gelatt Jr. CD, Vecchi MP (1983) Optimization by simulated annealing. *Science* 220(4598):671–680

[ML86] Mackert LF, Lohman GM (1986) R^* validation and performance evaluation for local queries. In: *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*. Washington, DC, pp 84–95

[NFS91] Ng R, Faloutsos C, Sellis T (1991) Flexible buffer allocation based on marginal gains. In: *Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data*, Denver, Colo., pp 387–396

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

[SAC$^+$79] Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. In: *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Boston, Mass., pp 23–34

[SG88] Swami A, Gupta A (1988) Optimization of large join queries. In: *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, Chicago, Ill., pp 8–17

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

[SKPO88] Stonebraker M, Katz R, Patterson D, Ousterhout J (1988) The design of xprs. In: *Proc. of the 14th International VLDB Conference*, Long Beach, Calif., pp 318–330,

[Ull82] Ullman JD (1982) *Principles of Database Systems*. Computer Science Press, Rockville, MD