Multi-Join Optimization for Symmetric Multiprocessors

Eugene J. Shekita
Honesty C. Young
IBM Almaden Research Center
San Jose, CA, USA
{shekita,young}@almaden.ibm.com

Kian-Lee Tan
Department of Information Systems
and Computer Science
National University of Singapore
tankl@iscs.nus.sg

Abstract

This paper looks at the problem of multi-join query optimization for symmetric multiprocessors. Optimization algorithms based on dynamic programming and greedy heuristics are described that, unlike traditional methods, include memory resources and pipelining in their cost model. An analytical model is presented and used to compare the quality of plans produced by each optimization algorithm. Experimental results show that, while dynamic programming produces the best plans, simple heuristics often do nearly as well. The same results are also used to highlight the advantages of bushy execution trees over more restricted tree shapes.

1 Introduction

Relational database systems are increasingly being used to execute decision-support queries. Such queries are used to discern sales trends, analyze marketing data, etc., and often they include multi-way joins with long execution times. One way to improve the execution times of these complex queries is through parallelism, and symmetric multiprocessors (SMPs) provide a cost-effective alternative. The best way to optimize queries for parallel execution is still largely an open problem, however [3].

This paper looks at the problem of multi-join optimization for SMPs. Recently, the authors of [5] showed how the time-honored method of optimizing database queries, namely dynamic programming [14], could be extended to include both pipelining and parallelism. We build on that work in the context of SMPs, showing how memory resources can be included as well. The importance of including memory resources during optimization was previously demonstrated in [1, 13] for shared-nothing multiprocessors and more recently in [20] for SMPs. None of those studies considered the full variety of execution trees that are examined here, however. These include left-deep, right-deep, deep, segmented right-deep, and bushy trees.

In addition to dynamic programming, we also describe optimization algorithms based on greedy heuristics that feature much lower computational complexity. An analytical model is presented and used to compare the quality of plans produced by each optimization algorithm. Experimental results show that, while dynamic programming produces the best plans, the simple heuristics often do nearly as well. The same results are also used to highlight the advantages of bushy execution trees over more restricted tree shapes.

The remainder of this paper is organized as follows: In Section 2 we introduce the terminology and notation that shall be used throughout this paper. In Section 3, we describe the process and execution models on which our optimization algorithms are based. This is followed by Section 4, where we describe optimization algorithms that use dynamic programming and greedy heuristics. Next, in Section 5, we present an analytical cost model that is used by the optimization algorithms to compute elapsed time. Finally, the results of our experimental study are presented in Section 6, and conclusions are drawn in Section 7.

2 Terminology and Notation

Throughout this paper, it is assumed that the query optimizer produces an *execution plan*, which is then executed by the database management system (DBMS). Execution plans for multi-join queries will be depicted as trees, with each internal node corresponding to a join and each leaf node corresponding to a base relation. Only hash-based join methods [4, 15] will be considered, since they generally offer superior performance. To simplify the discussion, only full-relation joins are considered.

Following the convention established in [13], each

**Proceedings of the 19th VLDB Conference,
Dublin, Ireland, 1993.**

join in an execution tree will have its *building* relation to the left and its *probing* relation to the right. Hash tables are built on building relations and probed by probing relations. If all the internal nodes of an execution tree have at least one leaf (i.e., base relation) as a child, then the tree is called *deep* [9]. Otherwise it is called *bushy*. A *left-deep* tree is a deep tree whose probing relations are restricted to base relations. Conversely, a *right-deep* tree is a deep tree whose building relations are restricted to base relations. As defined, the search space of bushy trees includes deep trees, which in turn includes left-deep and right-deep trees. Examples of left-deep, right-deep, deep, and bushy trees are shown in Figure 1.

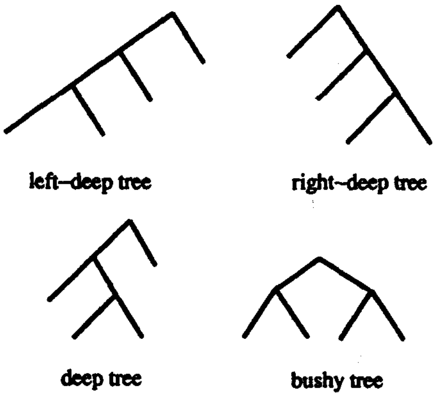


Figure 1: Examples of Different Query Trees

3 The Process and Execution Models

All the optimization algorithms described in this paper are based on the same process and execution model. This will allow us to develop just one cost model and use it to analyze all the optimization algorithms being studied.

3.1 The Process Model

To exploit inter- and intra-query parallelism as well as I/O parallelism, a threaded process model is assumed in this paper, where each processor is assigned its own dedicated CPU thread and multiplexed over some number of I/O threads. CPU threads are used for join processing, while I/O threads are used in conjunction with double buffering to initiate asynchronous I/O. To facilitate I/O parallelism, relations are assumed to be uniformly declustered over all disks.

For join processing, a simple parallelization of a uniprocessor hash join is modeled. The model assumes that a single hash table is built (in parallel) for each join and then probed by all processors. The hash tables of right-deep tree segments are probed in a parallel, pipelined fashion by all processors to achieve both inter- and intra-operator parallelism. More will be said about this shortly.

3.2 An Alternative Process Model

The advantages of this process model are its simplicity and self-balancing nature. It is self-balancing due to the fact that each processor participates in every join. Thus, within a right-deep segment, the processors will gravitate towards the join that requires the most processing. This simplifies optimization and runtime scheduling by eliminating the problem of how to assign and balance the load across processors.

One potential disadvantage of this process model is that a single hash table is used for each join, i.e., hash tables are not partitioned by processor as in shared-nothing architectures [2]. In practice, this could lead to lower cache hit ratios, since each processor would work with larger hash tables.

An alternative process model with more overhead for inter-process communication but potentially better cache behavior was described in [20]. In that process model, hash tables are partitioned by physical processor as in shared-nothing architectures [2, 13], and data is repartitioned using shared memory as described in [6]. The assignment of CPU threads to hash partitions is fixed, but physical processors are allowed to migrate among CPU threads (if necessary) to dynamically balance data flows.

To judge how sensitive our results were to the choice of the process model, we constructed analytical models for both of the above alternatives. Experimental results turned out to be nearly identical. This is due to the fact that the cost equations for the two process models are essentially the same, modulo the cost of repartitioning data in shared memory, which is small in relative terms [6]. Thus, the general results of this paper should hold regardless of which process model is used. More empirical evidence is needed to determine which alternative actually performs better in practice.

3.3 The Execution Model

All the optimization algorithms described in this paper are based on a *segmented* execution model, which is similar to the one described in [1]. In a segmented execution model, a query plan is broken down into a collection of memory-resident, right-deep segments that are executed one at a time. By memory-resident, we mean that the hash tables for all the building relations in a segment fit in memory. This execution model does not restrict the shape of a query plan — arbitrary bushy trees are allowed. However, pipelining along right-deep segments is used to achieve inter-operator parallelism rather than parallel execution of independent subtrees, i.e., ***bushy-tree parallelism*** [7].

Our segmented execution model differs slightly from the one described in [1]. There, the full result of each right-deep segment was always written to disk. Here, to exploit memory resources more fully, we allow part or all of the result to be materialized in memory if it plays the role of a building relation in the next segment to be executed.

Figure 2 shows how a segmented execution model might be used to execute an 8-relation join. As shown, the query plan, which is a bushy tree, has been broken into four right-deep segments. Segment 1 is executed first, followed by segment 2, and so on. The order of execution is determined by data dependencies. Segment 1 is not dependent on the results of any other segment and so it is executed first.

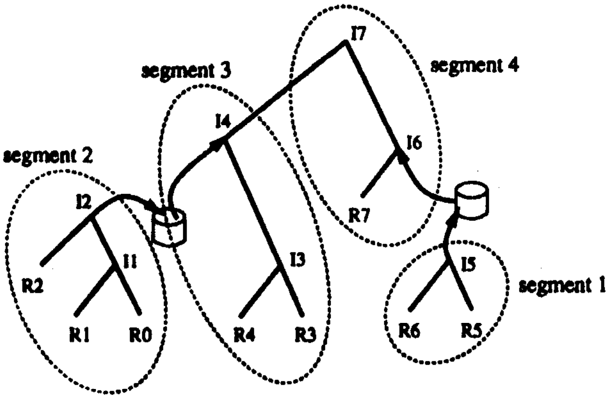


Figure 2: Segmented Execution

Using cardinality estimates, the optimizer has determined that R_6 and R_7 will not both fit in memory. Thus, the right-deep subtree rooted at I_6 is broken into segments 1 and 4. All of segment 1's result I_5 is written to disk to free up as much memory as possible for subsequent joins. The optimizer has also determined that the building relations for segment 2, namely R_1 and R_2, will both fit in memory but that there is not enough memory to hold all of the intermediate result I_2. Thus, part of I_2 is written to disk during the probing phase of segment 2, and later read when segment 3 is executed. In contrast to segment 2, the optimizer has determined that segment 3's result I_4 will fit in memory along with its building relations R_4 and I_2. Consequently, I_4 is used to build a hash table in memory during the probing phase of segment 3. Finally, the optimizer has determined that the building relations for segment 4, namely R_7 and I_4, will both fit in memory.

As mentioned, it is up to the optimizer to keep track of memory resources and determine segment boundaries as it generates a query plan. This information is then used at runtime for scheduling the execution of segments within the query plan.

3.4 The Case for a Segmented Execution Model

The main advantage of a segmented execution model is that optimization and runtime scheduling is greatly simplified. This is because only one right-deep segment is ever executed at a time. The disadvantage is that that bushy-tree parallelism is not exploited. For example, in Figure 2, it could be advantageous to execute segment 1 in parallel with segment 2.

We would argue that the opportunities to exploit bushy-tree parallelism are often limited, however. For example, it would almost never make sense to execute segments 1 and 2 in parallel. If that was done, then memory would be overcommitted, and more I/O for intermediate results would be generated. This *intermediate I/O*, as we will refer to it, is particularly onerous, since each intermediate I/O really translates into two disk accesses — a write and then a read. In [13], the authors found that it is always better to break up right-deep segments and avoid overcommitting memory for precisely this reason.

One instance where bushy-tree parallelism can offer an advantage is when the query plan looks like the one shown in Figure 3. If everything fits in memory, then it could be advantageous to execute the subtrees T_1 and T_2 in parallel. But this is only true when:

1. T_1 is I/O bound and T_2 is CPU bound or vice versa. The gains in this case accrue from a more balanced system [7].

2. The join predicates are such that T_1 and T_2 just happen to access relations which are declustered over disjoint sets of disks. The gains in this case accrue from I/O parallelism.

In addition to these limitations, note that case 1) usually requires case 2) to be true. This is because, if any of the relations accessed in T_1 and T_2 reside on the same set of disks, then there is likely to be arm contention, which would tend to make both T_1 and T_2 I/O bound.

Although it may not be obvious, note that if T_1 and T_2 were both CPU bound, then there is nothing to be gained by executing them in parallel (assuming linear speedup). To see this, let there be p processors, and let W_1 and W_2 correspond to the amount of work involved in executing T_1 and T_2, respectively. Then it should be clear that the optimal processor assignment is to proportionally allocate $p \cdot W_1/(W_1 + W_2)$ processors to

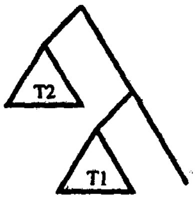


Figure 3: Candidate for Bushy-Tree Parallelism

T_1 and $p \cdot W_2/(W_1 + W_2)$ processors to T_2. Using this assignment, the time to execute the two subtrees in parallel would be:

$$max\left(\frac{W_1}{p \cdot W_1/(W_1+W_2)}, \frac{W_2}{p \cdot W_2/(W_1+W_2)}\right)$$

But this equals:

$$\frac{W_1}{p} + \frac{W_2}{p}$$

which is the same amount of time it that it would take to serially execute the two subtrees, one after the other.

In this paper, relations are assumed to be uniformly declustered over all disks, which is a reasonable assumption for the sort of small-scale parallelism that an SMP offers. Not only does it simplify system management, but it also facilitates parallel I/O, which is essential for high performance. Moreover, there is no concept of "node locality" on an SMP, which is often the motivation for alternative declustering strategies in shared-nothing architectures.

Because of our assumptions about how data has been declustered, bushy-tree parallelism offers no advantages in our study and is therefore not considered. In general, we recognize that there may be cases where bushy-parallelism may prove useful. However, because of its limitations, the scheduling problems it introduces, and its impact on the search space, it is not clear whether it is really worthwhile for an optimizer to actively seek for bushy-tree parallelism. A better alternative may be to make the optimizer cognizant of bushy-tree parallelism in its cost model and exploit it when possible but not actively search for it.

4 The Optimization Algorithms

Before picking a set of optimization algorithms to study, we identified three key issues that were of interest to us. These were:

- How could dynamic programming [14] be modified to include memory resources and elapsed time in its search strategy?
- How would the plans produced by the modified version of dynamic programming compare to heuristics that have been described in the literature [1]?
- How much could be gained by letting the optimizer consider generalized bushy trees versus restricting it to just, say, deep, left-deep or right-deep trees?

4.1 Dynamic Programming vs. Heuristics

Dynamic programming is the time-honored method of optimizing join queries [14]. It is based on exhaustive enumeration with pruning and produces "optimal" plans. We were particularly interested in dynamic programming because of the huge commercial investment that has gone into it — virtually all commercial optimizers are based on on it. We felt that if a simple modification to dynamic programming could produce competitive query plans for SMP architectures, then that would be of considerable commercial interest.

One of the problems with dynamic programming is that it has exponential complexity in terms of the number of relations in the query [11]. In practice, however, its performance has generally been found to be acceptable. This is because queries with more than 10 joins are rare. In a survey of 30 major DB2 customers [19], for example, the most complex join query from a sample of 200 queries involved "just" 8 relations.

Another problem with dynamic programming is that memory resources are usually ignored in optimizers based on it. This stems from the fact that its origins as an optimization technique date back to the days when computer memories were small and there was little opportunity for caching data in memory. As argued in [13], however, several relations can often fit in memory today. Consequently, we were interested in the problem of how to add an awareness of memory resources to the search space of dynamic programming and what its impact would be.

Yet another problem with dynamic programming is that optimizers based on it have traditionally used resource consumption (the sum of I/O and CPU usage) as their cost metric. As argued in [5], however, elapsed time is probably a better metric to use for the execution of complex multi-join queries in parallel systems. Consequently, we were also interested in what the impact of using elapsed time as a cost metric would be.

As we will show, it is not particularly hard to modify dynamic programming to include both memory resources and elapsed time. However, one can then prove that dynamic programming no longer produces optimal plans [5]. In short, it becomes nothing more than an expensive heuristic. Nonetheless, we felt that a modified version of dynamic programming could produce better query plans than other heuristics by virtue of its sheer "brute force".

4.2 Exploring the Advantages of Bushy and Deep Trees

In addition to dynamic programming, we were also interested in examining how much could be gained by letting the optimizer consider generalized bushy trees for query plans. A recent study [1] showed how a restricted form of bushy trees could offer better performance than right-deep trees. We wanted to investigate this issue more thoroughly and examine unrestricted bushy, left-deep, and deep trees as well.

The main advantage that bushy and deep trees offer over left-deep and right-deep trees is that they can reduce or eliminate I/O for intermediate results. As mentioned earlier, this *intermediate I/O* is particularly onerous because each I/O really translates into two disk accesses — a write and then a read. Moreover, in a batch environment it can cause disk arm contention that might have otherwise been avoided.

Figure 4 illustrates how a deep tree can reduce intermediate I/O. There, three possible query plans for a 4-relation join are shown. In the figure, we have assumed that R_0 is huge, R_3 is the size of memory, I_1 is 3/4 the size of memory, R_1 is 1/2 the size of memory, and that all other relations are 1/4 the size of memory. Because of R_0's size and the shape of the join graph, the join order is essentially fixed.

As shown, part of the intermediate result I_1 has to be written to disk in the left-deep tree. This is because R_1 and I_1 do not both fit completely in memory. In the right-deep tree, on the other hand, I_1 can be pipelined, and thus intermediate I/O for it is avoided. However, now the intermediate result I_2 has to be written to disk, since R_1, R_2, and R_3 do not all fit in memory. In contrast to the left-deep and right-deep trees, intermediate I/O can be avoided all together in the deep tree. This is achieved by pipelining I_1 and then materializing I_2 in memory.

In general, left-deep trees generate less I/O than right-deep trees, since intermediate I/O can be completely avoided whenever the building relation and result of a single join can fit in memory. Furthermore, the memory for the building relation can be released as soon as the join's result has been materialized. On the other hand, right-deep trees can pipeline large intermediate results that would otherwise have to be written to disk. Finally, deep and bushy trees get the best of both worlds. Like left-deep trees, intermediate results can be materialized in memory to free up space for subsequent joins, and like right-deep trees, pipelining can be used to avoid materializing large intermediate results. Compared to deep trees, bushy trees can reduce intermediate I/O further by virtue of the fact that there is more latitude in picking join orders. This can lead to smaller intermediate results and less I/O in some cases.

Based on the above observations, one would expect bushy trees to offer the best performance, followed closely by deep trees, then left-deep trees, and finally right-deep trees (assuming I/O costs dominate). Our results will show that this is indeed the case. In [13], cases were provided where right-deep trees performed much better than left-deep trees. This only happened when the building relations were declustered over disjoint sets of disks, however. Thus, more I/O parallelism was possible with right-deep trees, as all building relations could be scanned in parallel. However, when the building relations are uniformly declustered over all disks, as modeled here, this advantage disappears.

4.3 Optimization Using Dynamic Programming

To study the quality of plans produced by dynamic programming, we built a stripped-down optimizer based on it. This optimizer, which we refer to as our *DP optimizer*, takes an arbitrary join graph as input along with statistics on join selectivities and relation cardinalities and outputs a query plan. We built our DP optimizer in a general way so that output plans can be restricted to bushy, deep, left-deep, or right-deep trees by simply changing a runtime flag.

Our DP optimizer is, for the most part, a straightforward implementation of dynamic programming [14]. The only real difference is the way the cost of subplans are computed. Optimizers based on dynamic programming typically compute a single cost value for each subplan that is based on resource consumption. If one subplan for a given set of relations is cheaper in terms of resource consumption than another subplan for the same set of relations, then the more expensive plan is pruned.

As noted earlier, we wanted to replace resource consumption with elapsed time and also account for memory resources in some way. This was accomplished by simply replacing the single cost value that is usually kept for each subplan by a pair (et, mc), where et is the elapsed time to execute the subplan and mc is the memory consumption of the subplan. The way et is computed will be described shortly. The value of mc is computed using the input statistics on relation cardinalities and join selectivities. It includes the memory needed to compute the subplan's result, but not any memory released along the way when intermediate results are written to disk. The value of mc is used in pruning and also used to determine how a subplan will be broken up into right-deep segments during its execution.

To illustrate how mc is computed, we return to the deep tree in Figure 4. Letting $|R_i|$ denote the size of relation R_i, the value of mc for the subplan rooted at I_2 is computed as $|R_1| + |R_2|$. At the time the subplan is

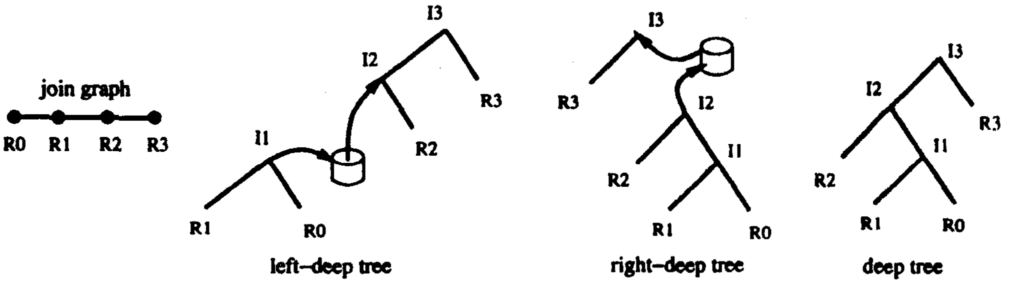


Figure 4: Example where Deep Tree can Eliminate Intermediate I/O

generated, it is not known whether I_2 will be pipelined, written to disk, or materialized in memory. Thus, $|I_2|$ is not included in mc. Following much the same logic, the value of mc for the subplan rooted at I_3 is computed as $|R_1| + |R_2| + |I_2|$.

In comparing subplans, (et, mc) pairs are used as follows: Let (et_1, mc_1) be the cost of subplan 1, and let (et_2, mc_2) be the cost of subplan 2, where both subplans are over the same set of relations. If $et_1 < et_2 \vee (et_1 = et_2 \wedge mc_1 \leq mc_2)$, then subplan 1 is considered to be less expensive. Otherwise, subplan 2 is considered to be less expensive. This gives precedence to et, using mc as a tie breaker. It is motivated by the fact that, all things being equal, the subplan that consumes less memory is probably a better choice, as plans built on it are likely to generate less intermediate I/O.

During pruning, we initially kept only one subplan for a given set of relations. However, one can construct cases where choosing the subplan with lower cost actually leads to a more expensive plan overall. This is because the chosen subplan might have a lower et value but a much greater mc value, which leads to trouble later on. Thus, pruning based on cost (as we have defined it) does not produce "optimal" plans. This observation led us to modify our DP optimizer so that up to n alternatives ordered by (et, mc) could be kept for each subplan. One can show that the complexity of dynamic programming blows up by $O(n^2)$ when n alternatives are kept. However, by trial and error, we found that $n = 2$ works well. More will be said about this later.

4.4 Optimization Using Heuristics

In addition to our DP optimizer, we also built an optimizer based on simple greedy heuristics. We refer to this as our *HR optimizer*. Simple variations on the same greedy heuristic were implemented for bushy, deep, left-deep, and right-deep trees. We also implemented an improved version of the BC heuristic that was described in [1]. The choice of which heuristic to use is determined by a runtime flag.

All but the BC heuristic are variations on the same basic idea. A join graph G is input, where each node in G corresponds to a base relation R_i and each edge corresponds to a join predicate. The query plan is then built bottom-up in a greedy fashion, join-by-join. A simple "minsize" heuristic is used to decide which join to add next to the plan.

For bushy trees, the algorithm in Figure 5 is executed. As shown, the algorithm tries all possible pairs of relations for its first join. For each starting pair, a query plan is generated bottom-up, join-by-join. At each step, all the unjoined pairs of relations connected by a join predicate are considered. The pair R_i, R_j that produces the smallest result is chosen as the next join. Both $R_i \times R_j$ and $R_j \times R_i$ are examined, and the join order with the least cost (as defined in the previous section) is the one chosen. The join is added to the current plan being constructed, and then R_i and R_j are collapsed into one node in G to reflect the fact that they have been joined. This continues until all the relations have been joined. The least costly plan generated among all starting pairs of relations forms the final output.

The algorithm shown in Figure 5 is similar to the bushy heuristic described in [1]. It is easy to show that the algorithm enumerates $O(n^3)$ joins for a fully connected join graph with n relations. Although it is not shown in Figure 5, the memory consumption of a query plan is computed as the plan is built. This information is used to choose join orders and also used to determine how the plan will be broken up into right-deep segments during its execution.

Due to space considerations, we omit detailed descriptions for the deep, left-deep, and right-deep heuristic algorithms. The algorithms are similar to the one shown in Figure 5, however, with relatively minor changes to accommodate restrictions on the shape of the query tree. For all other tree shapes, R_x in the inner-most loop is restricted to a base relation, and R_y is restricted to the node in G corresponding to the last join added to the current plan. In addition, for left-deep and right-deep trees, the join order is restricted to

```
input $G$
$BestPlan.cost = \infty$
for $R_i, R_j$ in $G$ connected by an edge do                    // try all starting pairs of relations
    $CurrentPlan =$ mincost($R_i \times R_j, R_j \times R_i$)     // pick cheapest join order for pair
    collapse $R_i, R_j$ in $G$                                    // reflect join in $G$
    while $|G| > 1$ do                                            // stop when no more relations to join
        $|MinJoin| = \infty$                                      // pick next pair using minsize heuristic
        for $R_x, R_y$ in $G$ connected by an edge do             // examine all unjoined pairs
            if $|R_x \times R_y| < |MinJoin|$ then                // remember minsize join
                $MinJoin =$ mincost($R_x \times R_y, R_y \times R_x$)
                $i = x, j = y$
            endif
        endfor
        add $MinJoin$ to $CurrentPlan$                            // add minsize join to current plan
        collapse $R_i, R_j$ in $G$
    endwhile
    if $CurrentPlan.cost < BestPlan.cost$ then                    // keep cheapest plan seen so far
        $BestPlan = CurrentPlan$
    endif
endfor
output $BestPlan$
```

Figure 5: Bushy-Tree Heuristic

$R_y \times R_x$ and $R_x \times R_y$, respectively.

Space considerations also prohibit us from describing the BC heuristic in detail. BC is, at its core, a right-deep heuristic. However, when it detects that another building relation will not fit in memory, BC chooses a new pair of relations and begins building another right-deep segment. The result of one right-deep segment can be used as the building or probing relation in another right-deep segment, so BC actually produces a bushy tree, albeit a restricted one. In [1], it was assumed that the full result of every right-deep segment was written to disk. Here, we allow part or all of the result to be materialized in memory if it plays the role of a building relation in the next segment to be executed.

Unlike dynamic programming, the heuristic algorithms do not enumerate all possible join permutations. Consequently, one would expect dynamic programming to always produce better query plans for a given tree shape. Our results will show that this is indeed the case. On the other hand, the heuristic algorithms have dramatically lower complexity, making them attractive in situations where optimization time is a concern. Moreover, the results will show that the bushy-tree heuristic often produces surprisingly good plans.

5 The Cost Model for Computing Elapsed Time

In this section, we present an analytical model for computing the elapsed time to execute a hash-based, multi-join query. All the optimization algorithms described in this paper used the cost model that will be described. Although the model computes elapsed time, it can be used to compute resource consumption as well.

The cost model assumes a segmented execution model like the one described earlier. It also assumes that I/O and CPU processing can be overlapped through double buffering and asynchronous I/O. IBM's DB2 product features such a capability [17]. Finally, a linear speedup through parallelism is assumed. On real SMP implementations, this has been shown to be an accurate assumption for small degrees of parallelism [8].

5.1 Parameters of the Cost Model

The system parameters of the cost model are shown in Table 1. In choosing the parameters, we sought to pick values that are representative of SMP platforms which are available today or will be available in the near future. Towards this end, we have modeled a 150 Mhz processor. On database workloads, a good rule of thumb is that it takes roughly 2.5 clock ticks per

Parameter	Description	Default
mhz	clock rate of individual processor	150 Mhz
cpi	average number of clocks cycles per instruction	2.5
p	number of processors in the system	4
m	memory available for join processing	128 Mbytes
io_{init}	CPU instructions to initiate an I/O	5000
io_{setup}	device time to setup an I/O	1 msec
io_{seek}	average seek time of an individual disk	10 msec
$io_{bandwidth}$	bandwidth of an I/O unit	15 Mbytes/sec

Table 1: System Parameters

Parameter	Description	Default
io_{size}	I/O block size	1 MByte
r	record size	200 bytes
probe	instructions to probe a hash table	100
compare	instructions to compare join attributes	100
hash	instructions to hash a join attribute	50
move	instructions to move a record in memory	200
project	instructions to project a record in memory	100
concatenate	instructions to concatenate two records in memory	100
lock	instructions to acquire a lock	10
unlock	instructions to release a lock	10
h	average number of probes to find match in hash table	1.2

Table 2: Algorithm Parameters

Parameter	Description
mips	effective MIPS rate of a single processor $mips = mhz/cpi$
$\|R\|$	size of an arbitrary relation R in blocks $\|R\| = \left\lceil \frac{\\|R\\| \cdot r}{io_{size}} \right\rceil$
$io_{sequential}$	time to perform a sequential block I/O $io_{sequential} = io_{setup} + io_{size}/(p \cdot io_{bandwidth})$
io_{random}	time to perform a random block I/O $io_{random} = io_{seek} + io_{sequential}$
insert	time to insert a record in a hash table $insert = lock + h \cdot (probe + compare) + move + unlock$
findmatch	time to find a matching record in a hash table $findmatch = h \cdot (probe + compare)$
generate	time to generate a result record $generate = project + concatenate$

Table 3: Derived Parameters

instruction on RISC processors [18]. Consequently, a 150 Mhz processor will execute roughly 150/2.5 or 60 MIPS on a database workload.

Note that some number of I/O "units" are assumed to be connected to the processors, with each unit delivering 15 Mbytes/sec of bandwidth. No single disk can deliver this much bandwidth, so in practice each unit would actually consist of several disks working in parallel, e.g., a RAID [12].

To keep the system balanced, we let the number of I/O units scale with the number of processors. Thus, with 4 processors, there will be 4 I/O units and the aggregate I/O bandwidth of the whole system will be 4 × 15 or 60 Mbytes/sec. This turns out to be just enough bandwidth to keep the building phase of the queries under study slightly CPU bound. An I/O bandwidth of 60 Mbytes/sec is well within the capability of existing mainframes but is aggressive by current workstation standards. Over time, however, the I/O capabilities of workstation servers will undoubtedly become more mainframe-like. Even today, a single IBM RS6000/980 server can support several microchannels, each of which can nominally handle 80 Mbytes/sec of I/O bandwidth.

Table 2 lists various algorithm parameters.[1] The values that have been chosen are similar to those used in previous studies [4, 15, 10]. A large I/O block size has been chosen to reduce the impact of disk seeks.

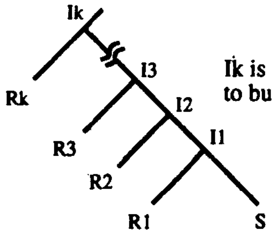
Derived parameters are listed in Table 3. Most of the equations should be self-explanatory. $||R||$ corresponds to the number of records in an arbitrary relation R, and all records are 200 bytes long. Note that, in all the equations, CPU operations are implicitly divided by the processor MIPS rate. Thus, CPU costs are in terms of time rather than instruction counts. Also note that each block I/O is assumed to be split up and performed in parallel over all I/O units. Finally, it is assumed that "smart" disk controllers supporting out-of-order reads are used. These would effectively eliminate rotational delays on large block I/O operations. Consequently, rotational delays have not been included in the cost model.

Before moving on, it is important to warn the reader not to get too caught up in debating how realistic the parameter values we chose are. Since CPU and I/O processing are overlapped, what really matters is the speed of the CPU relative to the bandwidth of the I/O subsystem. We will present results for the default I/O bandwidth, which represents a fairly balanced system, as well as comment on the results for high and low I/O bandwidths. Thus, the reader will have some feel for how the performance of the algorithms would change under different parameter settings (e.g., more expensive CPU operations).

5.2 Cost Equations

A segmented execution model makes it easy to calculate the elapsed time of a multi-join query. Since only one right-deep segment is executed at a time, only a model for the execution of a single, memory-resident segment is needed. The elapsed time to execute an arbitrary query plan is simply the sum of the elapsed times to execute each memory-resident segment within the plan.

Figure 6 shows the right-deep segment that will be used for reference. Each R_i corresponds to a base relation or an intermediate result that was previously written to disk. Each I_i corresponds to an intermediate result. Since the right-deep segment is memory-resident, the sum of the building relations must fit in memory, that is, $\sum_{i=1}^{k} |R_i| \leq m/io_{size}$.



Ik is either written to disk or used to build a hash table in memory

Figure 6: Right-Deep Segment used for Reference

If R_1 is larger than memory, then the pipeline has just one stage (i.e., $k = 1$) and a hybrid-hash join is used [4, 15]. Hybrid-hash requires hash buckets to be written to disk. A cost model for it has been covered elsewhere [10] and so we omit the analysis here. Note that our results do include many queries where hybrid-hash joins were required, however.

The elapsed time to execute a memory-resident, right-deep segment is the sum of the times to build the hash tables for each R_i and the time to probe the hash tables with S. The probe of the hash tables falls into one of two cases: Either 1) all or part of the result I_k is written to disk, or 2) the records from I_k are used to build a hash table in memory.

5.2.1 The Time to Build the Hash Tables

Since I/O and CPU processing are overlapped, the time to build a hash table for R_i is:

$$T^i_{build} = max(io^i_{build}, cpu^i_{build})$$

where io^i_{build} is the I/O time required to read R_i from disk, and cpu^i_{build} is the CPU time required to build the hash table for R_i in memory. Initially, queries are

[1] Note that a uniprocessor can be modeled by simply setting $p = 1$ and *lock* = *unlock* = 0.

assumed to run in batch mode, which is often the case for the kinds of complex queries that are of interest here. Batch-mode execution allows sequential I/O to be used during the building phase, consequently:

$$io^{i}_{build} = |R_i| \cdot io_{sequential}$$

The CPU time required to build the hash table for R_i includes the time to initiate the I/O for R_i and the time to insert each record of R_i into its hash table. Assuming linear speedup this is:

$$cpu^{i}_{build} = \frac{|R_i|}{p} \cdot io_{init} + \frac{\|R_i\|}{p} \cdot (hash + insert)$$

To avoid disk arm contention, the hash tables for each R_i are built serially, one after the other. Thus, the elapsed time to build all the hash tables in the right-deep segment is:

$$T_{build} = \sum_{i=1}^{k} T^{i}_{build}$$

5.2.2 The Time to Probe and Write the Result to Disk

We first consider the case where all or part of I_k is written to disk. When that happens, S is read from disk and used to probe for a match in R_1's hash table. When a match is found, the resulting join record is then used to probe for a match in R_2's hash table. This process is repeated until there is no match, or until all the hash tables have been probed. In the latter case, the resulting record is moved to an output buffer. Since I/O and CPU processing are overlapped, the elapsed time to perform these steps can be expressed as:

$$T_{probe+write} = max(io_{probe+write}, cpu_{probe+write})$$

where $io_{probe+write}$ is the I/O time required to read S and write I_k to disk, and $cpu_{probe+write}$ is the CPU time required to initiate I/O operations and probe the hash tables. Since S and I_k compete for the same disk arms, sequential I/O is no longer possible here, even in batch mode.

If I_k plays the role of a building relation in the next segment to be executed, then only the fraction f of I_k that does not fit in memory is actually written to disk. In this case, the value of f is calculated as:

$$f = max\left(0, 1 - \frac{m - \sum_{i=1}^{k} |R_i|}{|I_k|}\right)$$

On the other hand, if I_k plays the role of the probing relation in a subsequent segment (not necessarily the next one to be executed), then all of I_k is written to disk, i.e., $f = 1$. This is done to free up as much memory as possible for other segments and also to simplify scheduling. Using the appropriate value for f, $io_{probe+write}$ is calculated as:

$$io_{probe+write} = (|S| + f \cdot |I_k|) \cdot io_{random}$$

For each join, the CPU time includes the time to probe for a match and the time to generate the result. There is also the time to move I_k to the output buffer. Letting $I_0 = S$, we then have:

$$cpu_{probe+write} = \frac{|S| + f \cdot |I_k|}{p} \cdot io_{init} + \sum_{i=0}^{k-1} \frac{\|I_i\|}{p} \cdot (hash + findmatch) + \sum_{i=1}^{k} \frac{\|I_i\|}{p} \cdot generate + \frac{f \cdot \|I_k\|}{p} \cdot move$$

One caveat needs to be mentioned about the calculation of f. For left-deep, right-deep, and deep trees, the calculation of f is correct as it stands. For bushy trees, however, there can be cases where two or more composite subtrees, say, T_1 and T_2, appear as building relations along the same unbroken, right-deep segment (see Figure 3). In that case, the hash table for the result of T_1 may consume memory that is not available for use by T_2. This happens when $f < 1$ in T_1. If it turns out that insufficient memory is available to execute T_2, then T_2 is scheduled to be executed before T_1 and all of its result is written to disk. This greedy-like scheduling heuristic, which is not claimed to be optimal, means the optimizer never has to backtrack and recompute the cost of a subtree under different memory restrictions.

5.2.3 The Time to Probe and Build a Hash Table on the Result

If I_k plays the role of a building relation in the next join to be executed, and there is enough memory to hold $R_1 - R_k$ as well as I_k, then a hash table is built for I_k in memory as its records are produced. Following much the same logic as above, the equations for this case are:

$$T_{probe+build} = max(io_{probe+build}, cpu_{probe+build})$$

$$io_{probe+build} = |S| \cdot io_{sequential}$$

$$\begin{aligned} cpu_{probe+build} = & \frac{|S|}{p} \cdot io_{init} + \\ & \sum_{i=0}^{k-1} \frac{\|I_i\|}{p} \cdot (hash + findmatch) + \\ & \sum_{i=1}^{k} \frac{\|I_i\|}{p} \cdot generate + \\ & \frac{\|I_k\|}{p} \cdot (hash + insert) \end{aligned}$$

Here, note that sequential I/O has been modeled. Sequential I/O is possible because the disks are only accessed to read S.

6 Results

To quantitatively compare dynamic programming and the heuristic algorithms, we conducted a series of experiments. The results of those experiments are presented in this section. In each experiment, 1000 multi-join queries were optimized and the resulting plans were compared on the basis of elapsed time (as estimated by the cost model of the previous section). Results were collected for bushy, deep, left-deep, and right-deep trees using both dynamic programming and heuristics. Results were collected for the improved version of the BC heuristic as well. We first present results for the default parameter settings, and then comment on how sensitive the results were to changes in the values of key parameters.

6.1 How the Queries were Generated

The default parameter settings were for 8-relation joins. As mentioned, 1000 different multi-join queries were generated and used as input to the DP and HR optimizers. For each query, the optimizers took a join graph along with statistics on relation cardinalities and join selectivities as their input. The data for a query was generated in two steps. First a join graph was generated, then relation cardinalities and join selectivities were assigned to the graph.

Join graphs were generated using the algorithm described in [8]. Although that algorithm generates only acyclic graphs, we felt that it was adequate because in practice most multi-join queries tend to have simple join predicates [19]. Moreover, star-shaped graphs, which have nearly worst-case complexity in dynamic programming [11], are still possible.

When it came to assigning relation cardinalities and join selectivities, we tried to strike a balance between realistic values and values that would produce a large variance in the memory requirements and elapsed time of different queries. After some experimentation, we found that using three relation types (small, medium, and large) worked well. The cardinality of small, medium, and large relations were uniformly distributed over [10K, 20K], [100K, 200K], and [1M, 2M] records, respectively. Using these relation types, the algorithm to assign cardinalities and join selectivities worked as follows:

1. The type (small, medium, or large), but not the cardinality, of the final result was picked.

2. Each node (i.e., relation) R was randomly assigned a type (small, medium, or large) and then randomly assigned a cardinality in that type's range.

3. The join selectivity js of each edge (R_1, R_2) was chosen by first picking a value v for $\|R_1 \times R_2\|$ in $[0.5 \cdot min(\|R_1\|, \|R_2\|), (1.5 \cdot max(\|R_1\|, \|R_2\|)]$ and then solving $js \cdot (\|R_1 \times R_2\|) = v$

If the product of all the relation cardinalities and all the join selectivities (i.e., the cardinality of the final result) fell within the range of the final result type chosen in step 1), then the algorithm exited. Otherwise it backtracked to step 2), and tried new join selectivities. The algorithm restarted itself if it found that it had backtracked over 500 times. The calculation for js reflects the fact that a join's size is often a function of its input relations. The multipliers of 0.5 and 1.5 were added to increase the variance in the size of intermediate results. All join selectivities were treated as independent.

This algorithm, which is admittedly ad hoc, worked quite well in practice. In fact, the elapsed times of queries varied so much that we had to use 1000 queries in each experiment to get the 95% confidence intervals for our results down to the ±1% range.

6.2 How Results were Averaged

The following abbreviations will be used in describing the results:

DP.BY = dynamic programming, bushy trees
DP.DE = dynamic programming, deep trees
DP.LD = dynamic programming, left-deep trees
DP.RD = dynamic programming, right-deep trees
HR.BY = heuristic, bushy trees
HR.DE = heuristic, deep trees
HR.LD = heuristic, left-deep trees
HR.RD = heuristic, right-deep trees
HR.BC = heuristic, improved BC, bushy trees

Since elapsed times varied wildly from query to query, results were scaled by the elapsed time for DP.BY. For example, let DP.BY_i and DP.DE_i equal the (estimated) time to execute the plan generated by DP.BY and DP.DE for query i, respectively. Then the scaled average for DP.DE was calculated as:

$$\frac{1}{1000} \cdot \sum_{i=1}^{1000} \frac{\text{DP.DE}_i}{\text{DP.BY}_i}$$

Finally, note that the time to write the final result of each query to disk was not included in elapsed time calculations. We felt that this was more realistic because aggregates are often used on complex join queries to condense the final result into a readable report. Moreover, the time to write the final result to disk would be the same for each optimization algorithm.

6.3 Results for the Default Parameter Settings

Results for the default parameter settings are shown in Figure 7. Most of the results can be explained simply on the basis of intermediate I/O. The algorithms that performed well generated plans requiring less intermediate I/O, while the the algorithms that performed poorly generated plans requiring more intermediate I/O.

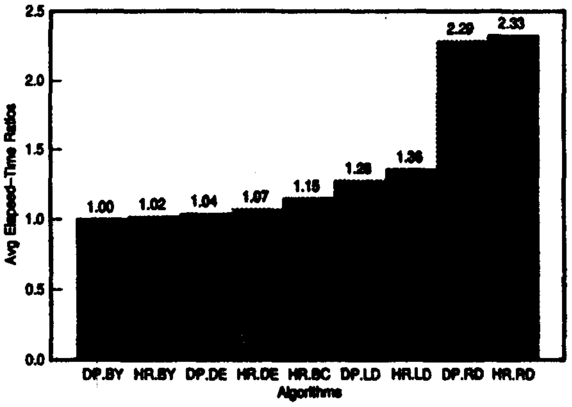


Figure 7: Average Elapsed-Time Ratios for the Default Parameter Settings (8-Relation Joins)

Looking at the results for the DP algorithms, one can see that DP.BY and DP.DE generated equally good plans. DP.BY performed slightly better by virtue of the fact that it had more latitude in picking join orders, which led to less intermediate I/O in some cases. In contrast to DP.DE, the (estimated) execution times of plans generated by DP.LD were, on average, about 28% greater than the execution times of plans generated by DP.BY. This is because pipelining cannot be used in left-deep trees to avoid materializing large intermediate results. Sometimes these have to be written to disk, causing more intermediate I/O. Finally, DP.RD did significantly worse than all the other DP algorithms, generating plans with execution times that were, on average, 129% greater than the execution times of plans generated by DP.BY. This is because intermediate results cannot be materialized in right-deep trees to free up memory, and once all of memory is consumed by building relations, intermediate I/O is required.

Looking at the results for the HR algorithms, one sees much the same trends that showed up in the DP algorithms. In all cases, the heuristic algorithms performed slightly worse than their DP counterparts, however. This is because the HR algorithms do not enumerate all possible join permutations, which can sometimes lead to join orders that produce more intermediate I/O. We were surprised by how well both HR.BY and HR.DE performed, while the relative performance of HR.BC is similar to the findings described in [1].

In addition to execution times, it is also interesting to look at the average resource consumption and number of joins enumerated by each optimization algorithm. This data is shown in Table 4. As shown, the algorithms with the best performance in terms of execution time also had the best performance in terms of resource consumption. Not surprisingly, the HR algorithms enumerated far fewer joins on average. The fact that HR.BY and HR.DE produced competitive plans while enumerating fewer joins make them very attractive. Despite their simplicity, HR.BY and HR.DE were still able to do a good job of optimizing I/O, and this was enough to make them competitive.

Algorithm	Resource Consumption	Joins Enumerated
DP.BY	24.92	873
HR.BY	25.80	98
DP.DE	25.93	487
HR.DE	27.07	98
HR.BC	29.74	71
DP.LD	30.56	251
HR.LD	32.25	56
DP.RD	52.92	251
HR.RD	53.96	56

Table 4: Avg Resource Consumption (in sec) and Avg Number of Joins Enumerated for the Default Parameter Settings (8-Relation Joins)

In Table 4, note that we set up the DP optimizer so that it kept 2 alternatives for each subplan. This increased the number of joins enumerated by a factor of 4, and was done to ensure that DP.BY produced near-optimal plans which could be used as the basis for comparison. (Recall that, here, dynamic programming is only an expensive heuristic.) Through experimentation, we found that 2 alternatives sufficed and that 3 or more alternatives offered virtually no improvement. When just 1 alternative was kept, there was a slight degradation in performance, but usually on the order of only a few percentage points. Note that multiple plan alternatives could have been kept for the heuristic algorithms as well. This would have improved their performance somewhat.

6.4 Worst-Case Results for the Default Parameter Settings

The mark of a good optimization algorithm is that it always produces optimal or near optimal execution plans without breaking down on certain queries. Figure 8 is used to show how well each optimization algorithm met this criteria. That figure shows the maximum elapsed-time ratios found for each algorithm among all queries. It further illustrates the advantages of bushy execution trees over more restrictive tree shapes. Considering that the HR algorithms enumerate far fewer joins, it should come as no surprise that the maximum elapsed-time ratios were uniformly higher for them. What is surprising is how well HR.BY performed, even at its worst.

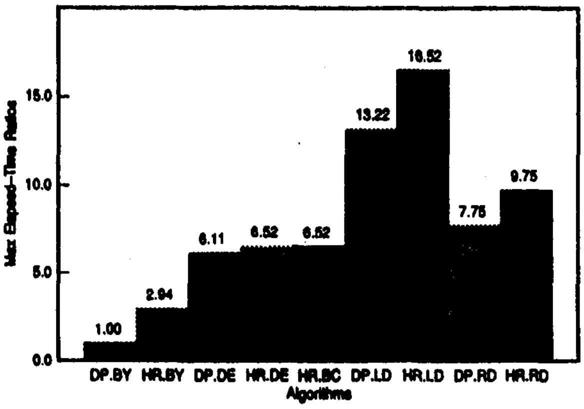


Figure 8: Max Elapsed-Time Ratios for the Default Parameter Settings (8-Relation Joins)

6.5 Results for Other Parameter Settings

To judge how sensitive our results were to changes in the values of key parameters, we also generated results for six other parameter settings. In each case, these differed from the default settings by just one parameter value. Results were generated for 4-relation joins, 12-relation joins, 10 Mbytes of I/O bandwidth (all random I/O), 20 Mbytes of I/O bandwidth, 64 Mbytes of memory, and 256 Mbytes of memory.

Due to space limitations we cannot show those results here (see [16] for the full set of results). The trends that were observed for the default settings remained largely unchanged, however. In most cases, the average elapsed-time ratios differed from those of the default parameter settings by less than 5%. Most of the results could still be explained simply on the basis of intermediate I/O. More relations, more I/O bandwidth, and less memory tended to amplify the relative differences in the algorithms, while fewer relations, less I/O bandwidth, and more memory tended to reduce the relative differences in the algorithms.

Before leaving this section, it is worth noting that, as the amount of memory available for join processing was changed, both the DP and HR optimizer picked very different plans. This contradicts the hypothesis put forth in [8], where it was argued that optimizers can effectively ignore memory resources. While this may be true for execution models in which the result of every join is written to disk, it is certainly not true if memory resources are used more effectively and taken into account during optimization.

7 Conclusion

This paper looked at the problem of multi-join query optimization for symmetric multiprocessors. A segmented execution model that uses pipelining to achieve inter- and intra-operator parallelism was presented, and then optimization algorithms using dynamic programming and greedy heuristics were described for that execution model. Unlike traditional methods, the optimization algorithms that were described include both memory resources and pipelining in their cost model. A model of memory resources is needed to fully exploit available memory resources. Otherwise, suboptimal execution plans may be generated.

An analytical model was derived and used to compare the quality of plans produced by each optimization algorithm. Experimental results for 1000 different 8-relation join queries were presented. The results showed that, although dynamic programming produced the best execution plans, the simple heuristics that were described often did nearly as well while enumerating far fewer joins during optimization. The same results also highlighted the advantages of using bushy and deep execution trees. Execution plans for bushy and deep trees were able to do a better job of exploiting memory resources. Consequently, they were able to generate less I/O for intermediate results, and this led to a reduction in execution times.

In terms of numbers, for dynamic programming, the elapsed time to execute left-deep query plans averaged 28% longer than bushy plans, while the elapsed time to execute right-deep plans averaged 129% longer. In contrast, the elapsed time to execute deep plans was within 5% of the time to execute bushy plans, even though roughly half as many joins were enumerated during optimization. Results for the heuristic algorithms followed similar trends. Since the shape of the query tree was the dominant factor in determining execution time, these general results should also hold for uniprocessors and shared-nothing multiprocessors.

References

[1] M. Chen, M. Lo, P. Yu, and H. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proc. of the 18th VLDB Conf.*, August 1992.

[2] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of the 11th VLDB Conf.*, August 1985.

[3] D. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad. *Sigmod Record*, 19(4), December 1990.

[4] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the 1984 SIGMOD Conf.*, June 1984.

[5] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the 1992 SIGMOD Conf.*, June 1992.

[6] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. of the 1990 SIGMOD Conf.*, May 1991.

[7] W. Hong. Exploiting inter-operation parallelism in XPRS. In *Proc. of the 1992 SIGMOD Conf.*, June 1992.

[8] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proc. of the 1st Int'l. Conf. on Parallel and Distributed Information Systems*, December 1991.

[9] Y. Ioannidis and Y. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *Proc. of the 1991 SIGMOD Conf.*, June 1991.

[10] H. Lu, K. Tan, and M. Shan. Hash-based join algorithms for multiprocessor computer with shared memory. In *Proc. of the 16th VLDB Conf.*, August 1990.

[11] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the 16th VLDB Conf.*, August 1990.

[12] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 SIGMOD Conf.*, June 1988.

[13] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. of the 16th VLDB Conf.*, August 1991.

[14] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database system. In *Proc. of the 1979 SIGMOD Conf.*, June 1979.

[15] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Sys.*, 11(3), September 1986.

[16] E. Shekita, H. Young, and K. Tan. Multi-join optimization for symmetric multiprocessors. Research Report, IBM Almaden Research Center, 1993. In preparation.

[17] J. Teng. DB2 buffer pool management. In *1992 IBM DB2 Technical Conference*, October 1992.

[18] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the cache performance and synchronization behavior of a multiprocessor operating system. In *Proc. of the 5th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[19] A. Tsang and M. Olschanowsky. A study of Database 2 customer queries. Technical Report 03.413, IBM Santa Teresa Laboratory, April 1991.

[20] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel query processing in DBS3. In *Proc of the 2nd Int'l. Conf. on Parallel and Distributed Information Systems*, January 1993.