Design Tradeoffs of Alternative Query Tree Representations for Multiprocessor Database Machines

by

Donovan Schneider
David J. DeWitt

Computer Sciences Technical Report 869
August 1989

Design Tradeoffs of Alternative Query Tree Representations for Multiprocessor Database Machines

Donovan A. Schneider
David J. DeWitt

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, and by a research grant from Tandem Computers. Funding was also provided by a DARPA/NASA sponsored Graduate Research Assistantship in Parallel Processing.

Abstract

During the past five years the design, implementation, and evaluation of join algorithms that exploit large main memories and parallel processors has received a great deal of attention. However, the methods used to represent join queries and their corresponding effects on performance has received little attention during this same time span. In this paper we examine the tradeoffs imposed by left-deep, right-deep and bushy query trees in a multiprocessor environment. Specifically, we address potential parallelism, memory consumption, support for dataflow processing, and the cost of optimization that are dictated by a particular query tree format. Results indicate that for hash-based join algorithms, right-deep query trees provide the best potential to exploit large multiprocessor database machines.

1. Introduction

Several important trends have occurred in the last ten years which have combined to change the traditional view of database technology. First, microprocessors have become much faster while simultaneously becoming much cheaper. Next, memory capacities have risen while their costs have declined. Finally high-speed communication networks have enabled the efficient interconnection of multiple processors. All these technological changes have combined to make the construction of high performance multiprocessor database machines feasible.

Of course, as with any new technology, there are many open questions regarding the best ways to exploit these multiprocessor database machines in order to achieve the highest possible performance. Because the join operator is critical to the operation of any relational DBMS, a number of papers have addressed parallel implementations of the join operation [LU85, BARU87, BRAT87, DEWI85, DEWI87, DEWI88, KITS88, SCHN89]. However, the methods used to represent queries have received little attention in the context of this new environment. In this paper we examine the tradeoffs imposed by left-deep, right-deep and bushy query trees in a multiprocessor environment. Specifically, we address the costs of optimization, potential parallelism, memory consumption, and support for dataflow processing. In this discussion we also examine the tradeoffs between several different join methods, including those based on sorting and hashing.

Degrees of Parallelism

There are three possible ways of utilizing parallelism in a multiprocessor database machine. First, parallelism can be applied to each operator within a query. For example, ten processors may work in parallel to compute a single join operation. This form of parallelism is termed **intra-operator** parallelism and has been studied extensively by previous researchers. Second, **inter-operator** parallelism can be employed to execute several operators within the same query concurrently. Finally, **inter-query** parallelism refers to executing several queries simultaneously. In this paper, we specifically address only those issues involved with exploiting inter-operator parallelism for queries composed of many joins. We defer issues of intra-query parallelism to future work.

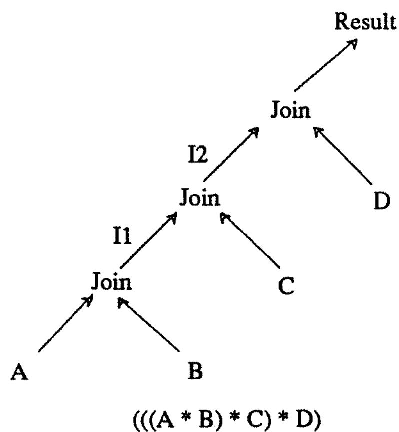
Query Tree Representations

Instrumental to understanding how to process complex queries is understanding how query plans are generated. A query is compiled into a tree of operators and several different formats exist for structuring this tree of operators. As will be shown, the different formats offer different tradeoffs, both during query optimization and

query execution.

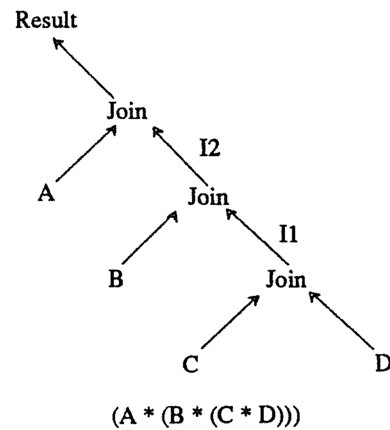
The different formats that exist for query tree construction range from simple to complex. A ''simple'' query tree format is one in which the format of the tree is restricted in some manner. There are several reasons for wanting to restrict the design of a query tree. For example, during optimization, the space of alternative query plans is searched in order to find the "optimal" query plan. If the format of a query plan is restricted in some manner, this search space will be reduced and optimization will be less expensive. Of course, there is the danger that a restricted query plan will not be capable of representing the optimal query plan.

Query tree formats also offer tradeoffs at runtime. For instance, some tree formats facilitate the use of dataflow scheduling techniques. This improves performance by simplifying scheduling and eliminating the need to store temporary results. Also, different formats dictate different maximum memory requirements. This is important



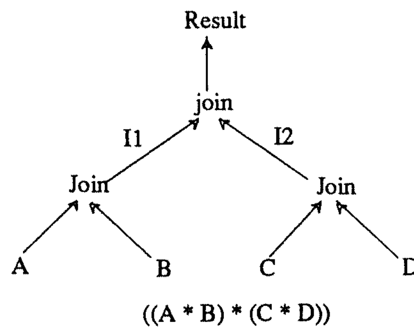
(((A * B) * C) * D)

Left-Deep Query Tree
Figure 1



(A * (B * (C * D)))

Right-Deep Query Tree
Figure 2



((A * B) * (C * D))

Bushy Query Tree
Figure 3

because, as discussed in [DEWI84, SCHN89], the performance of several of the join algorithms depends heavily on the amount of available memory. Finally, the format of the query plan is one determinant of the amount of parallelism that can be applied to the query.

Left-deep trees and right-deep trees represent the two extreme options of restricted format query trees. Bushy trees, on the other hand, have no restrictions placed on their construction. Since they comprise the design space between left-deep and right-deep query trees, they have some of the benefits and drawbacks of both strategies. They do have their own problems, though. For instance, it is likely to be harder to synchronize the activity of join operators within an arbitrarily complex bushy tree. We will examine the tradeoffs associated with each of these query tree formats more closely in the following sections. Refer to Figures 1, 2 and 3 for examples of left-deep, right-deep, and bushy query trees, respectively, for the query A join B join C join D. Note that the character * refers to the relational join operator.

Assumptions

In the following discussion we make several key assumptions. First, we assume the use of a hash-based join algorithm. Second, we assume the existence of sufficient main-memory to support as many concurrent join operations as required. We also assume that the optimizer has perfect knowledge of scan and join selectivities. In later portions of this paper we relax these assumptions.

2. Survey of Related Work

[GERB86] describes many of the issues involved in processing hash-based join operations in multiprocessor database machines. Both inter-operator and intra-operator concurrency issues are discussed. In the discussion of intra-query/inter-operator parallelism, the tradeoffs of left-deep, right-deep and bushy query tree representations with regard to parallelism, pipelined data flow, and resource utilization (primarily memory) are addressed. However, while [GERB86] defines the basic issues involved in processing complex queries in a multiprocessor environment, it does not explore the tradeoffs between the alternative query tree representation strategies in great depth. Although [GRAE89a] also supports the three alternative query tree formats in the shared-memory database machine Volcano, he does not discuss the tradeoffs in detail. In the sections below we will demonstrate that several interesting algorithms can be developed for scheduling query trees under the alternative tree formats.

[GRAE87] considers some of the tradeoffs between left-deep and bushy execution trees in a single proces-

sor environment. Analytic cost functions for hash-join, index join, nested loops join, and sort-merge join are developed and used to compare the average plan execution costs for the different query tree formats. Although optimizing left-deep query trees requires less resources (both memory and CPU), the execution times of the resulting plans are very close to those for bushy queries when the queries are of limited complexity. However, when the queries contain 10 or more joins, plan execution costs for the left-deep trees become up to an order of magnitude more expensive.

[STON89] describes how the XPRS project plans on utilizing parallelism in a shared-memory database machine. Optimization during query compilation assumes the entire buffer pool is available, but in order to aid optimization at runtime, the query tree is divided into fragments. These fragments correspond to our operator subgraphs described in Section 3. At runtime, the desired amount of parallelism for each fragment is weighed against the amount of available memory. If insufficient memory is available, three techniques can be used to reduce memory requirements. First, a fragment can be decomposed into sequential fragments. This requires the spooling of data to temporary files. If further decomposition is not possible, the number of batches used for the Hybrid join algorithm [DEWI84] can be increased. Finally, the level of parallelism applied to the fragment can be reduced.

3. Tradeoffs of Alternative Query Tree Representations

A good way of comparing the tradeoffs between the alternative query tree representations is through the construction of **operator dependency graphs** for each representation strategy. In the dependency graph for a particular query tree, a subgraph of nodes enclosed by a dashed line represent operators that should be scheduled together. The directed lines within these subgraphs indicate the producer/consumer relationship between the operators. The bold directed arcs between subgraphs show which sets of operators must be executed before other sets of operators are executed, thereby determining the maximum level of parallelism and resource requirements (e.g. memory) for the query. Either not scheduling the set of operators enclosed in the subgraphs together or failing to schedule sets of operators according to the dependencies will result in having to spool to disk tuples from the intermediate relations.

The operator dependency graphs presented in this section are based on the use of a hash-join algorithm as the join method. In this paper, we consider the use of three different hash-join methods: Simple hash-join [DEWI84], Grace join [KITS83], and Hybrid hash-join [DEWI84]. To illustrate the algorithms, consider the join of relations R and S, where R is the smaller joining relation.

With the Simple hash-join algorithm [DEWI84], the tuples from R are read from disk and staged in an in-memory hash table (which is formed by hashing on the join attribute of each tuple of R). Next, the larger joining relation S, is read from disk and its tuples probe the hash table for matches. However, when the number of tuples in R exceeds the size of the hash table, memory overflow occurs. In this case, the join operator creates a new file R′ and streams tuples to this file based on a new function h′ (an example h′ function is: join attribute value > 90,000) until all tuples in R are distributed between the hash table and R′. The h′ function is then given to the operator producing the tuples of S, the probing relation. Tuples from S corresponding to tuples in the overflow partition R′ are spooled directly to a temporary file S′; all other tuples probe the hash table to affect the join. The algorithm is then left with the task of joining the overflow partitions R′ and S′. Since R′ may still exceed the size of the hash table, this procedure repeats until the join has been fully computed.

The Grace join algorithm [KITS83] was developed to prevent the type of overflow processing just described. The algorithm first partitions relation R into N disk buckets by hashing on the join attribute of each tuple in R. Next, relation S is partitioned into N disk buckets using the same hash function. The algorithm is then left with the task of joining the respective buckets of relations R and S.

The number of buckets, N, is chosen to be very large. This reduces the chance that any bucket will exceed the memory capacity during the bucket joining phase. In the event that the buckets are much smaller than main memory, several are combined during bucket joining to form more optimal size join buckets (referred to as bucket tuning in [KITS83]). In the original description of the algorithm a hardware sort engine was used to perform the join of corresponding buckets. However, in this paper, hashing will be used to perform each of the bucket-joins.

The Hybrid hash-join algorithm [DEWI84] was developed in order to more effectively utilize large main memories. Instead of using all of memory as buffers for writing disk buckets as is done for the Grace join algorithm, the Hybrid algorithm chooses the number of buckets N such that the size of each bucket is slightly smaller than the amount of memory available for joining. Any additional memory is used for a hash table that is processed while relations R and S are being partitioned into buckets. For example, during the partitioning of R into buckets, tuples which hash to the first bucket are immediately written to a memory resident hash table, all other tuples are written to disk buckets as before. Then, during the partitioning of S, tuples which hash to the first bucket immediately probe the hash table for matches, all other tuples are written to disk buckets. The algorithm then has to join the N-1 corresponding buckets of R and S.

With this entire family of hash-join algorithms, the computation of the join operation can be viewed as

consisting of two phases. In the first phase, a hash table is constructed from tuples produced from the left input stream (relation R, in the above examples) and in the second phase, tuples from the right input stream (relation S) are used to probe the hash table for matches in order to compute the join. Since the first operation must completely precede the second operation, the join operator can be viewed as consisting of two separate operators, a **build** operator and a **probe** operator. The dependency graphs model this two phase computation for hash-joins by representing $Join_i$ as consisting of the operators B_i and P_i. In a later section we discuss the modifications necessary for supporting the sort-merge join algorithm.

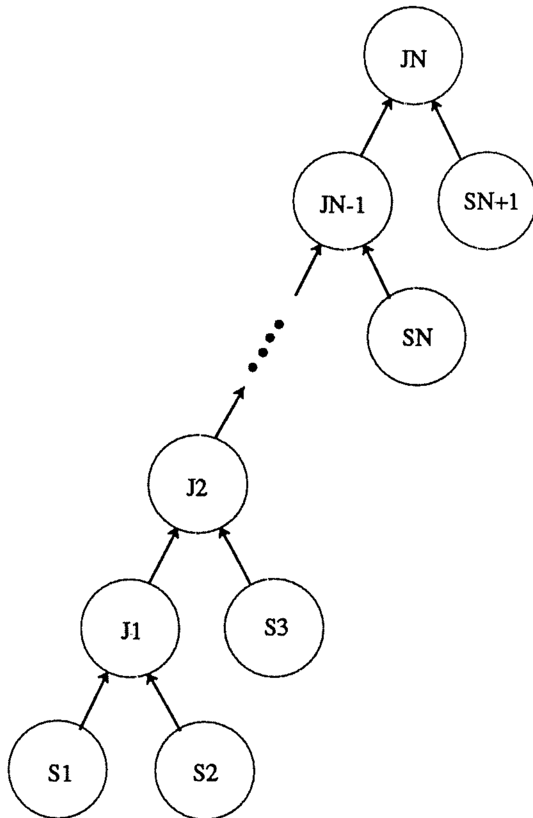
The reader should keep in mind that intra-operator parallelism issues are being ignored in this paper. That is, when we discuss executing two operators concurrently, we have assumed implicitly that **each** operator will be computed using multiple processors. For a description of parallel implementations of these hash-join algorithms, refer to [SCHN89].

3.1. Left-Deep Query Trees

Figure 4 shows a N-join query represented as a left-deep query tree. The associated operator dependency graph is shown in Figure 5. From the dependency graph it is obvious that no scan operators can be executed concurrently. It also follows that the dependencies force the following unique query execution plan:

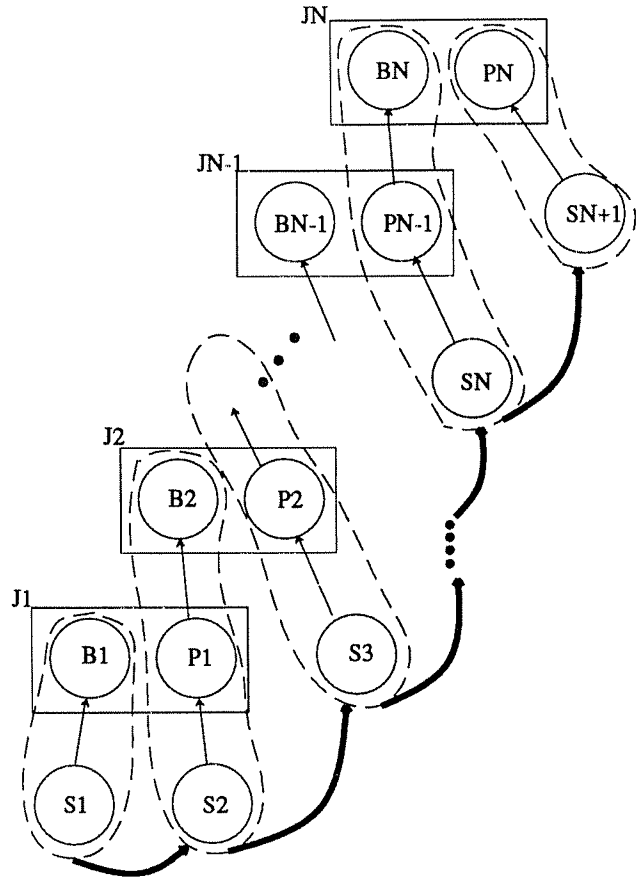
Step 1: Scan S1 - Build J1
Step 2: Scan S2 - Probe J1 - Build J2
Step 3: Scan S3 - Probe J2 - Build J3
•
•
•
Step N: Scan SN - Probe JN-1 - Build JN
Step N+1: Scan SN+1 - Probe JN

The above schedule demonstrates that at most one scan and two join operators can be active at any point in time. Consider Step N in the above schedule. Prior to the initiation of Scan SN, a hash table was constructed from the output of Join N-1. When the Scan SN is initiated, tuples produced from the scan will immediately probe this hash table to produce join output tuples for Join N. These output tuples will be immediately streamed into a hash table constructed for Join N. The hash table space for Join N-1 can only be reclaimed after all the tuples from scan SN have probed the hash table, computed Join N and stored the join computation in a new hash table. Thus, the maximum memory requirements of the query at any point in its execution consist of the space needed for the hash tables of any two adjacent join operators.



Generic Left-Deep Query Tree

Figure 4



Dependency Graph for a Left-Deep Query Tree

Figure 5

Bit Filtering

Bit vector filtering techniques [BABB79, VALD84, DEWI85] can be easily employed for left-deep query trees. As tuples are inserted into a hash table as part of the building phase of a join operation, the bit filter associated with the join operator is updated via a hashing function applied to the tuples' join attribute. After the join operator completely forms the filter, the filter is distributed to the scan operator which is located at the right child of the join operator. The scan operator then uses the bit filter to eliminate non-joining tuples. As mentioned previously, at most two join operators are active at any one point in time in the execution of a left-deep query tree. Thus, to support bit filtering in left-deep trees requires enough space for exactly two bit filters.

3.2. Right-Deep Query Trees

Figure 6 shows a generic right-deep query tree for an N-join query and Figure 7 shows the associated dependency graph. From the dependency graph it can easily be determined which operators can be executed concurrently and the following execution plan can be devised to exploit the highest possible levels of concurrency:

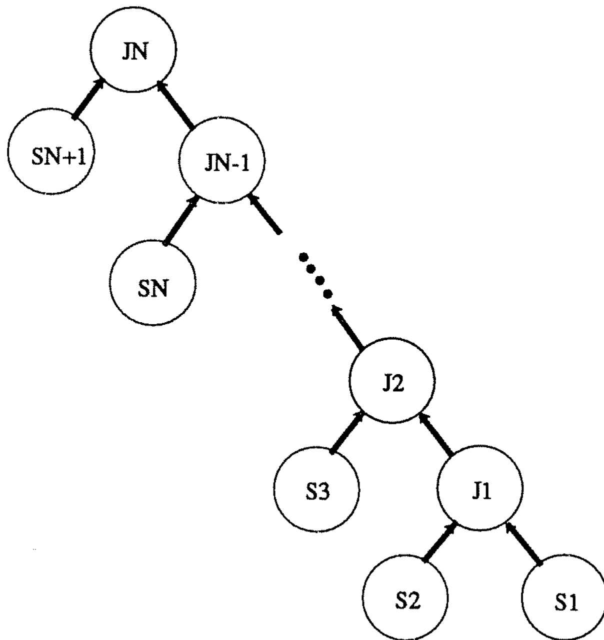
Step 1: Scan S2 - Build J1, Scan S3 - Build J2, ..., Scan SN+1 - Build JN
Step 2: Scan S1 - Probe J1 - Probe J2 -...- Probe JN

From this schedule it is obvious that all the scan operators but S1, and all the build operators can be processed in parallel. After this phase has been completed, the scan S1 is initiated and the resulting tuples will probe the first hash table. All output tuples will then percolate up the tree. As demonstrated, very high levels of parallelism are possible with this strategy (especially since every operator will also generally have intra-operator parallelism applied to it). However, the query will require enough memory to hold the hash tables of **all** N join operators throughout the duration of the query.

A question arises as to whether one would really want to schedule the scans S2 through SN+1 concurrently. If these operators access relations which are declustered [LIVN87] over the same set of storage sites, initiating all the scans concurrently may be detrimental because of the increased contention at each disk. This claim is supported by results presented in [GHAN89] in which it is demonstrated that for low degrees of data sharing an increase in the multiprogramming level from 1 to 2 can cause an increase in response time and a corresponding decrease in throughput. This occurs because continuously moving the disk heads from cylinder to cylinder to support concurrent scans adversely affects performance when the disk is the bottleneck.

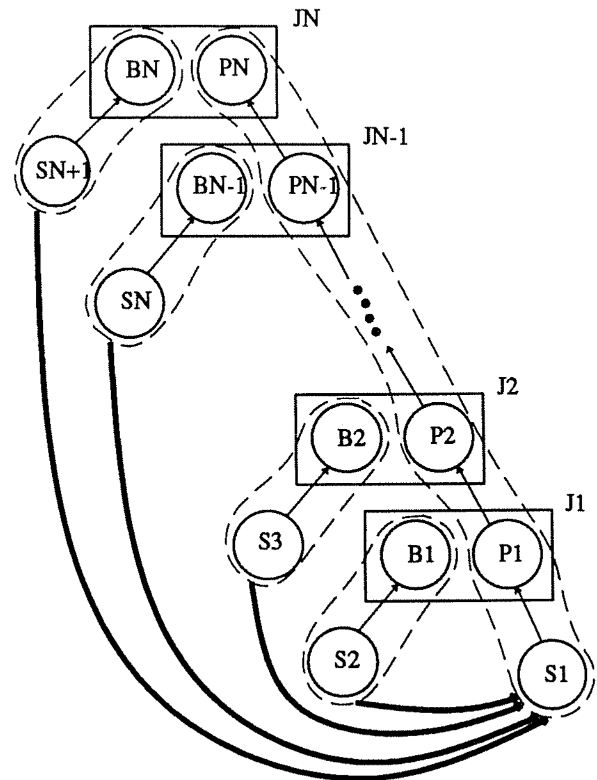
However, in a large database machine, it is **not** likely that relations will be declustered over all available storage sites [COPE88]. Further declustering eventually becomes detrimental to performance because the costs of controlling the execution of a query eventually outweigh the benefits of adding additional disk resources. This argument is supported by simulation results reported in [GERB87] which show that the response time for a 10,000 tuple selection query increases with each additional CPU/disk after 15 and by the experimental results reported in [DEWI88] which show that response time actually increases for very low selectivity selection queries that use a clustered index as the number of CPU/disk pairs is increased.

Given the situation where the database system contains a large number of disks and each relation is declustered over some subset of the disks, in order to minimize the response time for a query it makes sense to



Right-Deep Query Tree

Figure 6



Dependency Graph for a Right-Deep Query Tree

Figure 7

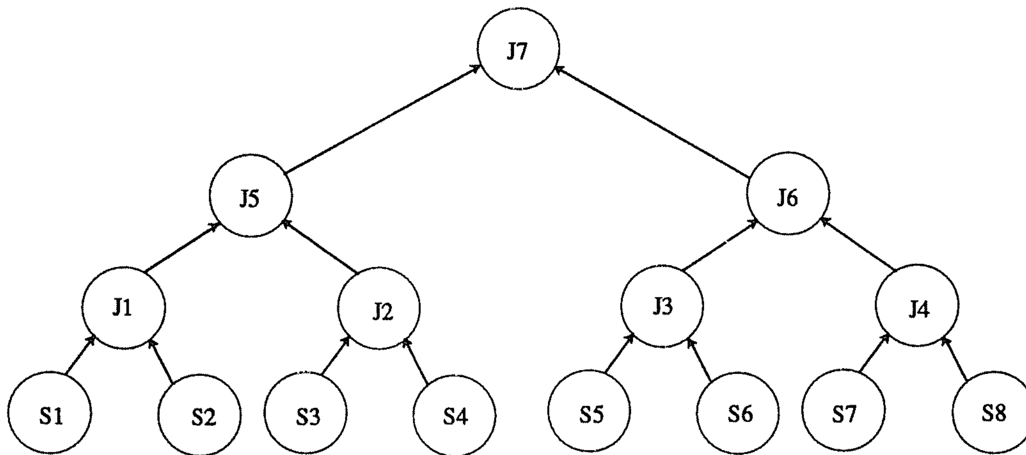
schedule in parallel only those scan operators which access data that do not overlap storage sites. Although we do not have a formal proof of this claim yet, the problem of choosing the optimal scan operators to co-schedule appears to be similar to the multiprocessor Resource Constrained Scheduling problem which is NP-complete [GARE79]. Hence, a heuristic algorithm should be designed to make this choice. As we will see in later sections, though, other factors may affect the scheduling decisions.

Bit Filtering

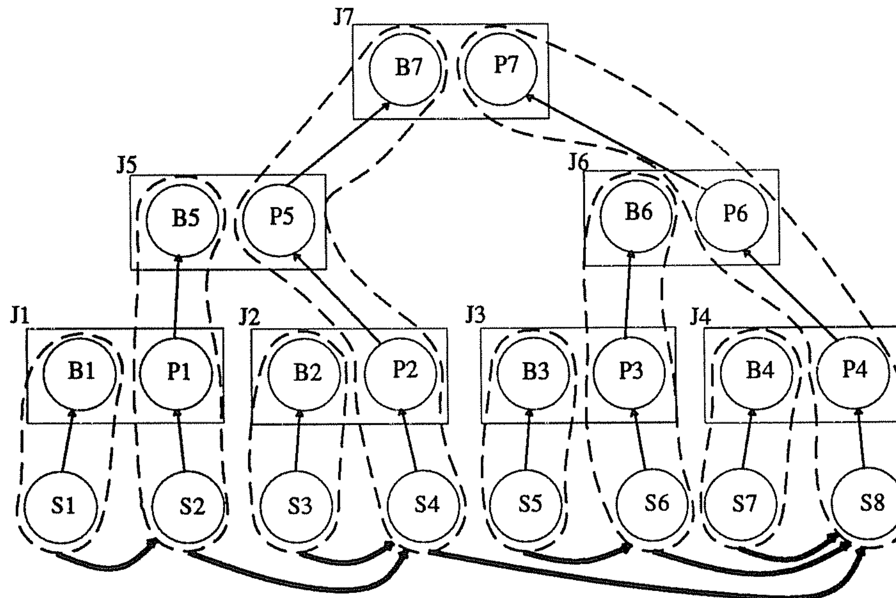
A separate bit vector filter is built by each of the join operators as tuples are inserted into its hash table. These N bit filters are then distributed to their appropriate nodes in the pipeline before Step 2 is initiated. For example, the bit filter built during the scan of S3 must be distributed to the processors that are to be used to process join J1. When an output tuple is being formed by J1 it would first apply the filter to see if the tuple has a possibility of participating in the join J2. Hence, adding bit filtering would require additional space for N bit filters.

3.3. Bushy Query Trees

With more complex query tree representations such as the bushy query tree for the eight-way join shown in Figure 8, several different schedules can be devised to execute the query. A useful way of clarifying the possibilities is again through the construction of an operator dependency graph. Figure 9 contains the dependency graph corresponding to the join query shown in Figure 8. By following the directed arcs it can be shown that the longest path through the graph is comprised of the subgraphs containing the scan operators S1, S2, S4 and S8. Since the



Bushy Query Tree
Figure 8



Dependency Graph for a Bushy Query Tree
Figure 9

subgraphs containing these operators must be executed serially in order to maximize dataflow processing (i.e., to prevent writing tuples to temporary storage), it follows that every execution plan must consist of at least four steps. One possible schedule is:

Step 1: Scan S1-Build J1, Scan S3-Build J2, Scan S5-Build J3, Scan S7-Build J4.
Step 2: Scan S2-Probe J1-Build J5, Scan S6-Probe J3-Build J6.
Step 3: Scan S4-Probe J2-Probe J5-Build J7.
Step 4: Scan S8-Probe J4-Probe J6-Probe J7.

However, notice that non-critical-path operations like Scan S7 and Build J4 could be delayed until Step 3 without violating the dependency requirements. The fact that scheduling options such as the above exist, demonstrates that runtime scheduling is more complicated for bushy trees than for the other two tree formats. As was the case with the other query tree designs, if the order in which operators are scheduled does not obey the dependency constraints, tuples from intermediate relations will have to be spooled to disk and re-read at the appropriate time. In a later section, we will demonstrate how these multiple execution strategies can be exploited in the case where memory is limited.

Bit Filtering

Bit filtering techniques are incorporated into bushy trees in exactly the same manner as described previously. As tuples are inserted into a hash table during the build phase of each join operator the filter is updated. When all the input tuples have been consumed, the filter is passed to the right child of the join operator, where the filter is used to eliminate non-joining tuples. As stated before, each join operator requires space for one bit filter.

4. Relaxed Assumptions

4.1. Non-hash-join join methods

The use of a hash-join method affected the above discussion on the achievable levels of parallelism associated with the alternative query tree designs. For example, reconsider the left-deep query tree and its associated operator dependency graph in Figures 4 and 5, respectively. With the sort-merge algorithm as the join method, the scan S1 does not necessarily have to precede the scan S2. For example, the scan and sort of S1 could be scheduled in parallel with the scan and sort of S2. The final merge phase of the join can proceed only when the slower of these two operations is completed. This is in contrast to the strictly serial execution of the two scans in order for a hash join algorithm to work properly. One possible schedule for executing the query shown in Figure 4 is:

Step 1: Sort S1, S2, ..., SN+1 (if not already sorted on the join attribute)
Step 2: Merge-join J1 (sort output if necessary)
Step 3: Merge-join J2 (sort output if necessary)
•
•
•
Step N+1: Merge-join JN

Modifying the operator dependency graphs to support the sort-merge join method is simple. First, assume that join nodes in the graph represent only the final merge-join operation (designated MJi), that is, join operation i will **not** consist of the two suboperators B_i and P_i. All dependencies will be implicitly assumed by the normal flow of data up the query tree. The algorithm for creating the entire graph is as follows.

(1) Add a Scan MJi node after each MJi node in the query tree. Create all scan nodes as before.

(2) If a base relation needs sorting, replace Scan Si with Sort Si —> Scan Si´

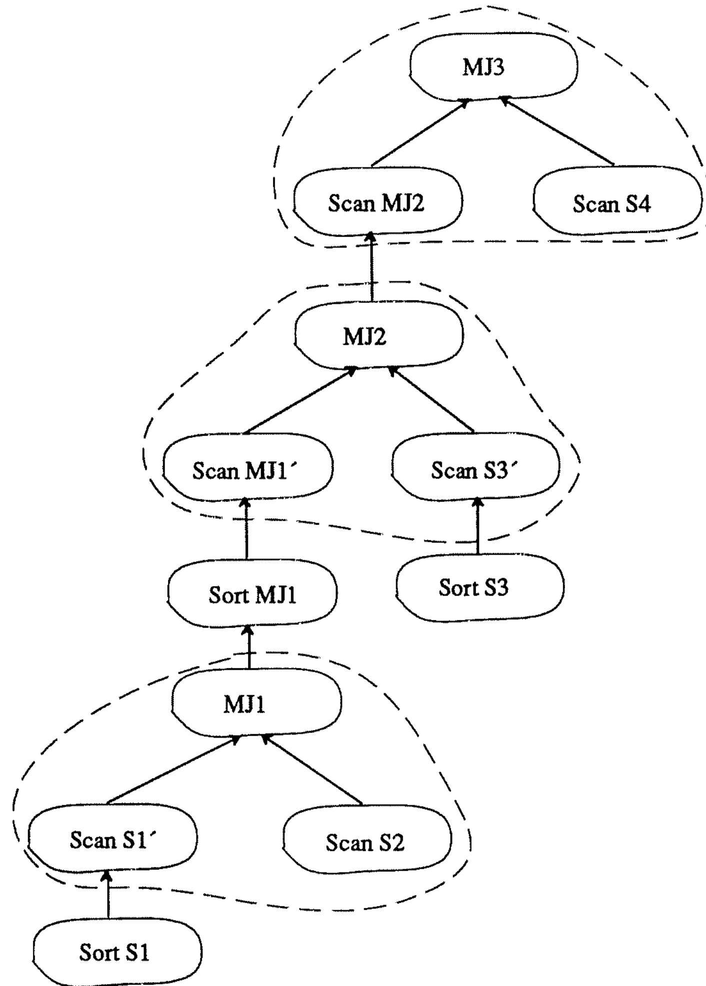
(3) If the output from MJi needs sorting, replace Scan MJi with Sort MJi —> Scan MJi´

(4) Group each MJi operator with its immediate descendants as a subgraph of operators to co-schedule.

To illustrate this algorithm, consider the query which joins relations S1, S2, S3 and S4, where S1 and S3 and the output of MJ1 must be sorted. The operator dependency graph resulting from this query is shown in Figure 10. It should be noted that the addition of scan nodes does not necessarily imply additional disk IO. If the result of a sort or a merge-join operation can be stored in memory buffers, the scan operation need never access the disk.

It is interesting to note that if the sort-merge join algorithm is used exclusively as the join method, the left-deep and right-deep query tree representations become equivalent because the relations S1 through SN+1 can be scanned/sorted concurrently in either strategy, whereas with the hash-join algorithm there is an ordering dependency which specifies that the left-child input must be completely consumed before the right-child input can be initiated.

Previous discussions of bit filtering were also influenced by the use of hash-join methods. Since all hash-join algorithms require that the entire left input be consumed before the operator producing the right input can be initiated, bit filtering works well with this family of join algorithms. However, for the sort-merge join method, the potential improvement in performance that can be obtained by the use of bit filtering must be weighed against the potential loss in performance that occurs from imposing a strict ordering on the processing of the two input relations. If the relations to be joined are distributed over different subsets of storage sites, the best performance may be achieved by scanning and sorting the relations concurrently.



Operator Dependency Graph for Sort-Merge Join
Figure 10

4.2. Reduced Memory

In the previous discussion of the alternate query tree formats, it was assumed that sufficient memory was available to hold the "building" relation for all the concurrent join operators. (We are again only addressing hash-joins.) However, is was shown that the memory requirements may be quite high for certain query plans, particularly when right-deep query trees are employed. Even with the trend towards database machines with larger numbers of processors and larger memories, it cannot be expected that large numbers of join operators which access large datasets will fit completely in memory at the same time.

4.2.1. Left-Deep Query Trees

As stated previously, left-deep query trees require that only the hash tables corresponding to two join operators be memory resident at any point in time during the execution of any complex query. However, except for the relation at the lowest level of the query tree, the relations staged into the memory resident hash tables are the result of intermediate join computations, and hence it is expected to be difficult to predict their size. And, even if the size of the intermediate relations can be accurately predicted, in a multi-user environment it **cannot** be expected that the optimizer will know the exact amount of memory that will be available when the query is executed. If memory is extremely scarce, sufficient memory may not exist to hold even one of these hash tables. Thus, even though only two join operators are active at any point in time, many issues must be addressed in order to achieve optimal performance.

[GRAE89b] proposes a solution to this general problem by having the optimizer generate multiple query plans and then having the runtime system choose the plan most appropriate to the current system environment. A similar mechanism was proposed for Starburst [HAAS89].

One possible problem with this strategy is that the number of feasible plans may grow quite large for the complex join queries we envision. Besides having to generate plans which incorporate the memory requirements of each individual join operator, an optimizer must recognize the consequences of intra-query parallelism. For example, if a join operator is optimized to use most of the memory in the system, the next higher join operator in the query tree will be starved for memory, and if no capability for modifying the query plan at runtime is provided, performance will suffer.

A simpler strategy may be to have the runtime query scheduler adjust the number of buckets for the Hybrid join algorithm [DEWI84] in order to react to changing memory availabilities. An enhancement to this strategy would be to keep statistics on the size of the intermediate join computations stored in the hash tables and use this information to adjust the number of buckets for join operators higher in the query tree. Finally, if much more memory is available at runtime than expected, it may be beneficial to perform some type of query tree transformation to more effectively exploit these resources. For example, the entire query tree, or perhaps just parts of it, could be transformed to the right-deep query tree format.

4.2.2. Right-Deep Query Trees

Since right-deep trees require that N hash tables be co-resident, they present a different set of problems because they are much more memory intensive than their respective left-deep trees. They also afford little opportunity for runtime query modifications since once the scan on S1 is initiated the data flows through the query tree to completion. However, it is expected that more accurate estimates of memory requirements will be available for a right-deep query tree since the left children (the building relations) will always be base relations (or the result of applying selection predicates to a base relation), while with a left-deep tree the building input to each join is always the result of the preceding join operation.

One would expect that if memory is limited (or the scan selectivities are under-estimated) and a number of the N hash tables experience overflow, the resulting performance will be extremely poor. In this case, the costs of overflow processing would be magnified with each succeeding join (loss of dataflow processing, overflow processing...). Since it is not likely that sufficient space will be available for all N hash tables concurrently, we are considering several alternative techniques for exploiting the potential performance advantages of right-deep query trees. One strategy (similar to that proposed in [STONE89]) involves having the optimizer or runtime scheduler *break* the query tree into disjoint pieces such that the sum of the hash tables for all the joins within each piece are expected to fit into memory. This splitting of the query tree will, of course, require that temporary results be spooled to disk. When the join has been computed up to the boundary between the two pieces, the hash table space currently in use can be reclaimed. The query can then continue execution, this time taking its right-child input from the temporary relation.

A more dynamic strategy, called **dynamic bottom-up scheduling**, schedules the scans S2 to SN+1 in a strict bottom-up manner. For example, the scan S2 is first initiated. The tuples generated by this scan will be used to construct a hash table for the join operator J1. After this scan completes the memory manager is queried to see if enough memory is available to stage the tuples expected as a result of the scan S3. If sufficient space exists, scan S3 is initiated. This same procedure is followed for all scans in the query tree until memory is exhausted. If all scans are processed, all that remains is for the scan S1 to be initiated to start the process of probing the hash tables. However, in the case that only the scans through S_i can be processed in this first pass, the scan S1 is initiated but now the results of the join computation through join J_{i-1} are stored into a temporary file S1´. The further processing of the query tree proceeds in an identical manner only the first scan to be scheduled is the scan S_{i+1}. Also, the scan to start the generation of the probing tuples is initiated on the temporary file S1´.

Both of these strategies share a common feature of dealing with limited memory by "breaking" the query tree at one or more points. Breaking the query tree has a major impact on performance because data flow processing is lost when the results of the temporary join computation must be spooled to disk. And, it was assumed that enough memory was available to hold at least each relation individually and hopefully several relations simultaneously, However, that may not always be the case. An alternative approach is to do some pre-processing on the input relations in order to reduce the memory requirement. This is what the Grace [KITS83] and Hybrid [DEWI84] join algorithms attempt to do. In this subsection, we discuss the use of the Grace and Hybrid join algorithms for processing complex query trees represented as right-deep query trees. To illustrate the algorithms, the join of relations A, B, C and D will be presented. See Figure 11 for the representation of this join in both a left-deep and a right-deep query tree format.

Grace Join Algorithm

Since during the bucket-forming phase, the Grace join algorithm writes both relations to be joined to disk [KITS83], if the join query is represented as a right-deep query tree, the resulting query execution will be equivalent to breaking the query tree after each join, i.e., every intermediate relation will be written to disk. Notice, though, that when using the Grace algorithm no part of relation C is staged into memory until I1 is fully computed and distributed into the appropriate number of buckets. Likewise, relation D is not staged into memory until I2 has been computed. Since the Grace join algorithm breaks all relations into buckets (including intermediate relations) it can

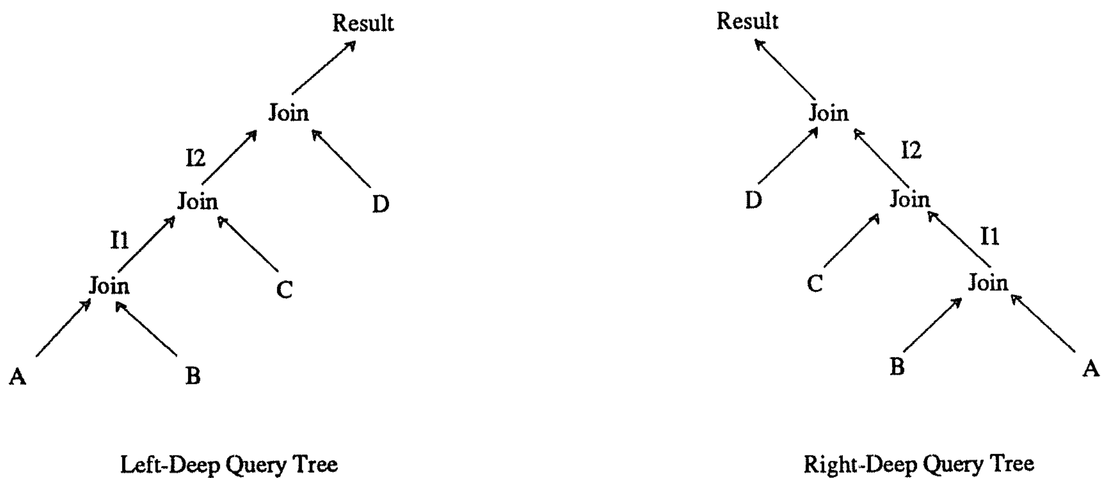


Figure 11

be shown that the amount of disk I/O is constant whether the query is represented as a left-deep or a right-deep tree. Thus, the choice between the two query representations should be made on the basis of memory availability and selectivity estimates. With right-deep trees only a single bucket of a single relation need ever be in memory and this bucket will be from a base relation.

Hybrid Join Algorithm

With right-deep query trees, query processing becomes much more interesting when using the Hybrid join algorithm (termed **Right-Deep Hybrid Scheduling**). Unlike the situation with the Grace join algorithm, the first bucket of each of the building relations will have to be staged into memory. Consider the previous join query but with each relation being subdivided into two buckets. The first bucket of A (denoted A_{b1}) will join with the first bucket of B to compute the first half of A*B. Since this is a right-deep tree the first inclination would be to probe the hash table for C (actually C_{b1}) with all these output tuples. However, this cannot be done immediately because the join attribute may be different between C and B, in which case the output tuples corresponding to A*B (I1) must be rehashed before they can join with the first bucket of C. Since I1 must use the same hash function as C, I1 must be composed of two buckets (one of which will directly map to memory as a probing segment). Thus, the tuples corresponding to B_{b1} * A_{b1} will be rehashed to $I1_{b1}$ and $I1_{b2}$, with the tuples corresponding to the first bucket (about half the A*B tuples assuming uniformity) immediately probing the hash table built from C_{b1}. Again, output tuples of this first portion of A*B*C will be written to the buckets $I2_{b1}$ and $I2_{b2}$. Output tuples will thus keep percolating up the tree, but their number will be reduced at each succeeding level based on the number of buckets used by the respective building relation. Query execution will then continue with the join B_{b2} * A_{b2}. After all the respective buckets for A * B have been joined, the remaining buckets for C * I1 will be joined. Processing of the entire query tree will proceed in this manner.

As shown above, with Right-Deep Hybrid Scheduling, it is possible for some tuples to be delivered to the host as a result of joining the first buckets of the two relations at the lowest level of the query tree. This is not possible with an analogous left-deep query tree. If a user is submitting the query, the quicker feedback will correspond to a faster response time (even though the time to compute the entire result may be identical). And in the case of an application program submitting the query, it may be very beneficial to provide the result data sooner and in a more "even" stream as opposed to dumping the entire result of the join computation in one step because the computation of the application can be overlapped with the data processing of the backend machine.

Several questions arise as how to best allocate memory for right-deep query trees with the Hybrid join algorithm. For correctness it is necessary that the first bucket of EACH of the building relations be resident in memory. However, it is NOT a requirement that all relations be distributed into the same number of buckets. For example, if relation B and D are very large but relation C is small, it would be possible to use only one bucket for relation C while using additional buckets for relations B and D. Hence, the intermediate relation I1 would never be staged to disk in any form, rather it would exist solely as a stream of tuples up to the next level in the query tree.

As can be seen, Right-Deep Hybrid Scheduling provides an alternative to using the dynamic bottom-up scheduling algorithm described earlier. Whereas the dynamic bottom-up scheduling algorithm assumed that enough memory was available to hold at least each relation individually and hopefully several relations simultaneously, the use of the Hybrid algorithm potentially reduces the memory requirements while still retaining some dataflow throughout the **entire** query tree. If Right-Deep Hybrid Scheduling can use a single bucket for every relation, the two algorithms become the same. It remains an open question as to when one scheduling strategy will outperform the other.

The Case for Right-Deep Query Trees

(1) The size of the "building" relations can be better predicted since the cardinality estimates are based on predicates applied to a base relation as opposed to estimates of the size of intermediate join computations.

(2) As shown previously, right-deep query trees provide the best potential for exploiting parallelism (N concurrent scans and hash table build operations).

(3) In the best case, the results of intermediate join computations are neither stored on disk nor inserted into memory resident hash tables, rather, intermediate join results exist only as a stream of tuples flowing through the query tree.

(4) Even though bushy trees can potentially re-arrange joins to minimize the size of intermediate relations, a best-case right-deep tree will never store its larger intermediate relations on disk (the extra CPU/network costs will be incurred for the larger intermediates, though).

(5) The proposed dynamic bottom-up scheduling algorithm looks very promising. It reacts well to memory availability and appears simple to implement.

(6) The Right-Deep Hybrid Scheduling strategy can deliver tuples **sooner** and in a more constant stream to the user/application than a similar left-deep query tree can, even in the case of reduced memory.

(7) Right-deep trees are generally assumed to be the most memory intensive query tree format but this is not always the case. Consider the join of relations A, B, C, and D as shown in Figure 11 for both a left-deep and a right-deep query tree format. Assume the size of all the relations is 10 pages. Furthermore, assume that the size of A*B is 20 pages and the size of A*B*C is 40 pages. At some point during the execution of the left-deep query tree, the results of A*B and A*B*C will simultaneously reside in memory. Thus, 60 pages of memory will be required in order to execute this query in the most efficient manner. With a right-deep query tree, however, relations B, C and D must reside in memory for optimal query processing. But these relations will only consume 30 pages of memory. In this example, the left-deep tree requires twice as much memory as its corresponding right-deep tree.

(8) The size of intermediate relations may grow with left-deep trees in the case where attributes are added as the result of each additional join. Since the intermediates are stored in memory hash tables, memory requirements will increase. Note that although the width of tuples in the intermediate relations will also increase with right-deep trees, these tuples are only used to probe the hash tables and hence they don't consume memory for the duration of the join.

4.2.3. Bushy Query Trees

Recall from Section 3 that the scheduling of "non-critical-path" joins may affect the amount of parallelism. Also, it may be possible to adapt to limited memory situations by intelligently scheduling operators such that the maximum memory requirements for the query are reduced.

Consider again the schedule for executing the 7-join bushy tree described in Section 3.3. After the execution of Step 1, four hash tables will be resident in memory. After Step 2 completes, memory can be reclaimed from the hash tables corresponding to join operators J1 and J3 but new hash tables for join operators J5 and J6 will have been constructed. Only after the execution of Step 3 can the memory requirements be reduced to three hash tables (J7, J6, and J4).

However, it may be possible to reduce the memory consumption of the query by constructing a different schedule. Consider the following execution schedule in which we have noted when hash table space can be reclaimed:

Step 1: Scan S1-Build J1.
Step 2: Scan S2-Probe J1-Build J5-Release J1, Scan S3-Build J2.
Step 3: Scan S4-Probe J2-Probe J5-Build J7-Release J2 and J5.
Step 4: Scan S5-Build J3.
Step 5: Scan S6-Probe J3-Build J6-Release J3, Scan S7-Build J4.
Step 6: Scan S8-Probe J4-Probe J6-Probe J7-Release J4 and J6 and J7.

Although this execution plan requires six steps instead of four, the maximum memory requirements have been reduced throughout the execution of the query from a maximum of 4 hash tables to a maximum of 3 hash tables. If these types of execution plan modifications are insufficient in reducing memory demands, the techniques described in the last two subsections for left-deep and right-deep query trees can also be employed.

5. Conclusions

In this paper, we have outlined many of the problems and tradeoffs associated with the task of processing complex queries in a multiprocessor database machine. In particular, we focused on how the three different query tree representation strategies, left-deep, right-deep and bushy trees, can affect response time and resource consumption. Although multiuser performance comparisons are beyond the scope of this paper, the memory consumption of a query will serve as a good indicator of potential throughput because memory is so crucial to high performance query processing, especially with the hash-join algorithms.

We have also developed several strategies for processing complex queries represented in each of the alternative query tree formats in a multiprocessor database machine. Results from this study indicate that designers of large database machines should not limit their systems to use only left-deep query trees as has commonly been done in the past (e.g., System R [LOHM85] and Gamma [DEWI86]). Rather, at the least, support for right-deep query trees should be provided when using hash-based join algorithms because in this environment right-deep query trees have the best potential for exploiting the resources (processors and memory) available in large multiprocessor database machines. One simple but flexible scheduling strategy outlined for right-deep query trees that we introduced was called Dynamic Bottom-up Scheduling. An alternative scheduling strategy called Right-Deep Hybrid Scheduling also appears attractive because of its ability to deliver output tuples sooner and in a more constant stream to the user/application program.

Our remaining work consists of more quantitatively describing the tradeoffs between the alternative query tree representations in large multiprocessor database machines (systems with up to a thousand processors and a hundred gigabytes of main memory). To evaluate the tradeoffs, we are currently in the process of developing a

simulation model for such a large multiprocessor database machine. We intend on using this environment to simulate the different scheduling strategies outlined in this paper for complex queries composed of up to 10 joins. The simulation model will be initially validated with results from running complex queries represented as left-deep query trees in the Gamma database machine. The current hardware configuration of the Gamma database machine consists of a 32 processor/32 disk Intel iPSC/2 Hypercube [INTE88].

6. References

[BABB79] Babb, E., ''Implementing a Relational Database by Means of Specialized Hardware'' ACM Transactions on Database Systems, Vol. 4, No. 1, March, 1979.

[BARU87] Baru, C., O. Frieder, D. Kandlur, and M. Segal, ''Join on a Cube: Analysis, Simulation, and Implementation'', **Database Machines and Knowledge Base Machines**, M. Kitsuregawa and H. Tanaka (eds), Kluwer Academic Publishers, 1987.

[BRAT87] Bratbergsengen, Kjell, ''Algebra Operations on a Parallel Computer -- Performance Evaluation'', **Database Machines and Knowledge Base Machines**, M. Kitsuregawa and H. Tanaka (eds), Kluwer Academic Publishers, 1987.

[COPE88] Copeland, G., W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba", Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.

[DEWI84] DeWitt, D. J., Katz, R., Olken, F., Shapiro, L., Stonebraker, M. and D. Wood, ''Implementation Techniques for Main Memory Database Systems,'' Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[DEWI85] DeWitt, D., and R. Gerber, ''Multiprocessor Hash-Based Join Algorithms,'' Proceedings of the 1985 VLDB Conference, Stockholm, Sweden, August, 1985.

[DEWI86] DeWitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K. and M. Muralikrishna, ''GAMMA - A High Performance Dataflow Database Machine,'' Proceedings of the 1986 VLDB Conference, Japan, August 1986.

[DEWI87] DeWitt, D., Smith, M., and H. Boral, ''A Single-User Performance Evaluation of the Teradata Database Machine,'' MCC Technical Report Number DB-081-87, March 5, 1987.

[DEWI88] DeWitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine", Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.

[GARE79] Garey, M. and D. Johnson, **Computers and Intractability: A Guide to the Theory of NP-Completeness**, W. H. Freeman and Company, New York, 1979.

[GERB86] Gerber, R., ''Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms,'' PhD Thesis and Computer Sciences Technical Report #672, University of Wisconsin-Madison, October, 1986.

[GERB87] Gerber, R. and D. DeWitt, ''The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine'', Computer Sciences Technical Report #708, University of Wisconsin-Madison, July, 1987.

[GHAN89] Ghandeharizadeh, S., and D.J. DeWitt, ''Performance Analysis of Alternative Declustering Strategies'', Computer Sciences Technical Report #855, University of Wisconsin-Madison, June 1989.

[GRAE87] Graefe, G., "Rule-Based Query Optimization in Extensible Database Systems", PhD Thesis and Computer Sciences Technical Report #724, University of Wisconsin-Madison, November, 1987.

[GRAE89a] Graefe, G., "Volcano: A Compact, Extensible, Dynamic, and Parallel Dataflow Query Evaluation System", Working Paper, Oregon Graduate Center, Portland, OR, February 1989.

[GRAE89b] Graefe, G. and K. Ward, "Dynamic Query Evaluation Plans", Proceedings of the 1989 SIGMOD Conference, Portland, OR., May 1989.

[HAAS89] Haas, L., Freytag, J.C., Lohman, G.M., and H. Pirahesh, "Extensible Query Processing in Starburst", Proceedings of the 1989 SIGMOD Conference, Portland, OR., May 1989.

[INTE88], Intel Corporation, **iPSC/2 User's Guide**, Intel Corporation Order No. 311532-002, March, 1988.

[KITS83] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, "Application of Hash to Data Base Machine and Its Architecture," New Generation Computing, Vol. 1, No. 1, 1983.

[KITS88] Kitsuregawa, M., Nakano, M., and M. Takagi, "Query Execution for Large Relations On Functional Disk System," to appear, 1989 Data Engineering Conference.

[LIVN87] Livny, M., S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms", Proceedings of the 1987 SIGMETRICS Conference, Banf, Alberta, Canada, May, 1987.

[LOHM85] Lohman, G., et. at., "Query Processing in R*", in **Query Processing in Database Systems**, edited by Kim, W., Reiner, D., and D. Batory, Springer-Verlag, 1985.

[LU85] Lu, H. and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network", Proceedings of the 1985 VLDB Conference, Stockholm, Sweden, August, 1985.

[SCHN89] Schneider, D. and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", Proceedings of the 1989 SIGMOD Conference, Portland, OR, June 1989.

[STON89] Stonebraker, M., P. Aoki, and M. Seltzer, "Parallelism in XPRS", Memorandum No. UCB/ERL M89/16, February 1, 1989.

[VALD84] Valduriez, P., and G. Gardarin, "Join and Semi-Join Algorithms for a Multiprocessor Database Machine" ACM Transactions on Database Systems, Vol. 9, No. 1, March, 1984.