Tradeoffs in Processing Multi-Way Join Queries via Hashing in 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, by a Digital Equipment Corporation External Research Grant, and by a research grant from the Tandem Computer Corporation. 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, most of this work has addressed the problem of executing joins involving only two relations. In this paper we examine the problem of processing multi-way join queries through hash-based join methods in a shared-nothing database environment.

We first discuss how the choice of a format for a complex query can have a significant effect on the performance of a multiprocessor database machine. Experimental results obtained from a simulation study are then presented to demonstrate the tradeoffs of left-deep and right-deep scheduling strategies for complex join query evaluation. These results demonstrate that right-deep scheduling strategies can provide significant performance advantages in large multiprocessor database machines, even when memory is limited.

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 feasible the construction of high performance multiprocessor database machines.

Of course, as with any new technology, there are many open questions regarding the best ways to exploit the capabilities of 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 including [BARU87, BRAT87, DEWI85, DEWI88, KITS88, LU85, SCHN89a]. However, these papers have not addressed the processing of queries with more than one or two joins. Also, the performance impact of alternative formats for representing multi-way join queries has received little attention in the context of this new environment. The related work that has been done is discussed in Section 2.

In this paper we examine the tradeoffs imposed by left-deep, right-deep and bushy query trees in a multiprocessor environment. We focussed on hash-based join methods because their performance has been demonstrated bo be superior in systems with large memories [BRAT87, DEWI84, SCHN89a, SHAP86], although we include a brief discussion of the sort-merge join algorithm. The tradeoffs we consider include the potential for exploiting intra-query parallelism (and its corresponding effect on performance), resource consumption (primarily memory), support for dataflow processing, and the cost of optimization. The examination of these tradeoffs demonstrated the feasibility of the right-deep representation strategy and resulted in several new algorithms for processing query trees in this format. As well as providing superior opportunities for exploiting parallelism within a query tree, a right-deep representation strategy can also reduce the importance of correctly estimating join selectivities. This analysis of the tradeoffs of the alternative query tree representation strategies and a description of several algorithms for processing right-deep query trees is presented in Section 3.

Because, left-deep and right-deep representation strategies present the extreme cases among the alternative query representation strategies, we were interested in quantitatively examining the performance tradeoffs between these two strategies. To perform this analysis, we constructed a multiprocessor database machine simulator and implemented scheduling algorithms for both of these tree formats. The description of the simulation model, its validation, and the experimental results obtained are contained in Section 4. Results from this experimental analysis confirm the more qualitative results which indicate that right-deep trees can indeed provide substantial performance improvements under many different experimental conditions but that this strategy is not optimal under all circumstances. Our conclusions and plans for future work are presented in Section 5.

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 or select 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 inter-query parallelism to future work.

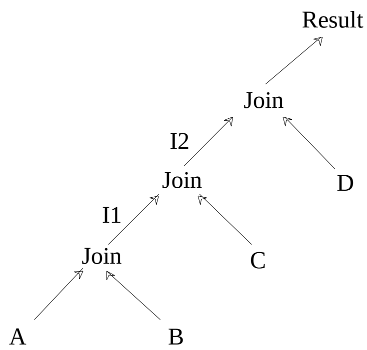
Query Tree Representations

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.

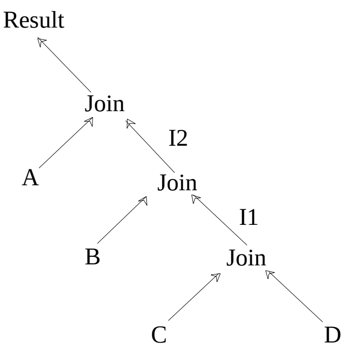
Query tree formats also offer tradeoffs at runtime. For instance, some tree formats facilitate the use of dataflow scheduling techniques. This improves performance by simplifying scheduling and eliminating the need to store temporary results. Also, different formats dictate different maximum memory requirements. This is important because the performance of several of the join algorithms depends heavily on the amount of available memory [DEWI84, SCHN89a, SHAP86]. 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,



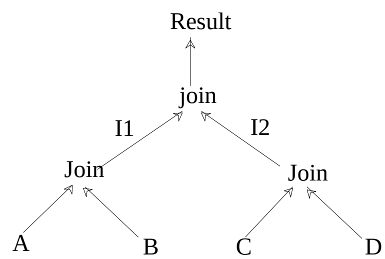
(((A * B) * C) * D)

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

right-deep, and bushy query trees, respectively, for the query A join B join C join D. (Note that the character * is used to denote the relational join operator.)

Assumptions

In the following discussion we make several key assumptions. First, we assume 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, the tradeoffs are not discussed in detail. In this paper 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 processor 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 queries are of limited complexity. However, when the queries contain 10 or more joins, execution costs for the left-deep trees can 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

In this section we discuss how each of the alternative query tree formats affects memory consumption, dataflow scheduling, and the ability to exploit parallelism in a multi-way join query. The discussion includes processing queries in the best case (unlimited resources) to more realistic situations where memory is limited. Strategies are proposed to process multi-way join queries when memory is limited.

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 tuples from the intermediate relations to disk.

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 two different hash-join methods, Simple hash-join and Hybrid hash-join [DEWI84]. It is assumed that the reader is familiar with these join methods although a brief description of them is included here. For this description, consider the join of relations R and S, where R is the smaller joining relation.

The Simple hash-join method is a naive algorithm that assumes that all the tuples from the **smaller** joining relation R (termed the **Building** relation) can be staged into a main memory hash table.[1] If this assumption fails, overflowing tuples from R are dynamically staged to a temporary file on disk. Once all the tuples from R have been processed, the tuples from the **larger** relation S (the **Probing** relation) are processed. As each tuple from S is read from disk, the tuple is either used to probe the hash table containing tuples from R or is written back to disk, if hash-table overflow occurred while building the hash table with R (see [SCHN89a for more details). If hash-table overflow has occurred, the overflow partitions of R and S are recursively joined using this same procedure.

The Hybrid hash-join algorithm was developed in order to prevent the overflow processing discussed above while still effectively utilizing as much memory as is available. The key idea is to recognize a priori that the join will exceed the memory capacity and partition each joining relation into enough disjoint buckets such that each bucket will fit into the available memory. Main memory hashing is used to compute each of the smaller bucket joins. As an enhancement, a portion of the join result is computed while the two joining relations are being partitioned into buckets.

With 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. See [SCHN89a] for a description of parallel implementations of these hash-join algorithms.

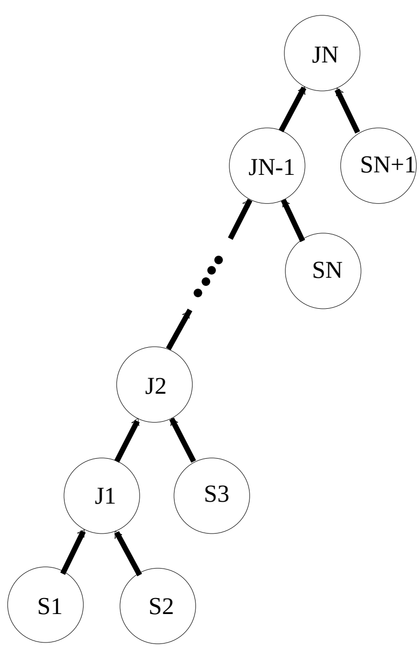
[1] The smaller relation is always used as the building or inner relation in a hash join algorithm in order to minimize the number of times the outer relation must be read from disk. Also by using the smaller relation as the inner or building relation, one maximizes the probability that the outer relation will only have to be read once.

3.1. Left-Deep Query Trees

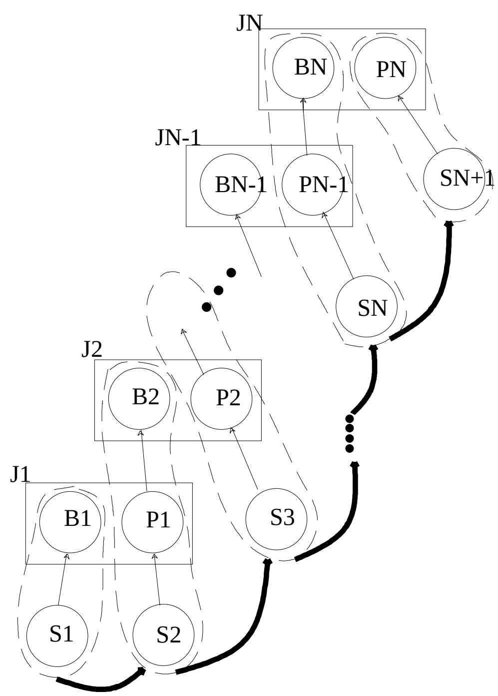
Figure 4 shows a N-join query represented as a left-deep query tree. The associated operator dependency graph is presented in Figure 5. From the dependency graph it is obvious that no scan operators can be executed concurrently. It also follows that the dependencies force the following unique query execution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



Generic Left-Deep Query Tree
Figure 4



Dependency Graph for a Left-Deep Query Tree
Figure 5

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.

Limited Memory

Although, left-deep query trees require that only the hash tables corresponding to two adjacent join operators be memory resident at any point in time during the execution of any complex query, 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. Furthermore, 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 be 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. If it is not possible to modify the query plan at runtime, performance will suffer.

A simpler strategy may be to have the runtime query scheduler adjust the number of buckets for the Hybrid join algorithm in order to react to changes in the amount of memory available. 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 significantly 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.

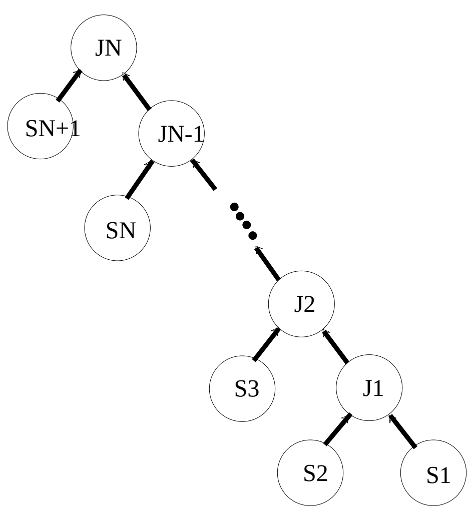
3.2. Right-Deep Query Trees

Figure 6 shows a generic right-deep query tree for an N-join query and Figure 7 presents 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:

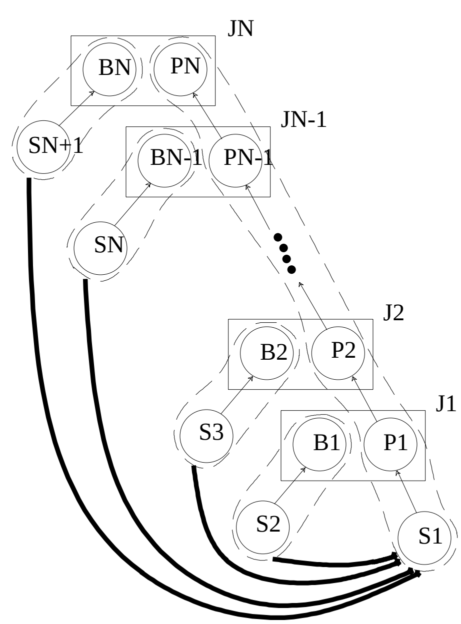
Step 1: Scan S2 - Build J1, Scan S3 - Build J2, ..., Scan SN+1 - Build JN
Step 2: Scan S1 - Probe J1 - Probe J2 -...- Probe JN

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 the performance implications of scheduling scans S2 through SN+1 concurrently. If these operators access relations which are declustered over the same set of storage sites, initiating all the scans concurrently may be detrimental because of the increased contention at each disk [GHAN89]. 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 [GERB87, COPE88, DEWI88]. In Section 4 we present experimental results which illustrate the performance implications of the data declustering strategy.



Right-Deep Query Tree
Figure 6



Dependency Graph for a Right-Deep Query Tree
Figure 7

Limited Memory

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 cannot always be assumed that sufficient space will be available for all N hash tables concurrently, we have developed 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. This scheduling strategy is called **static right-deep scheduling**.

A more dynamic strategy, called **dynamic bottom-up scheduling**, schedules the scans S2 to SN+1 (Figure 6) 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 the scans have been 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´. Further processing of the query tree proceeds in an identical manner only the first scan to be scheduled is 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 significant impact on performance because the benefits of data flow processing are lost when the results of the temporary join computation must be spooled to disk. Although, we have assumed that enough memory is available to hold at least each relation individually and, hopefully, several relations simultaneously, this may not always be the case. An alternative approach is to preprocess the input relations in order to reduce memory requirements. This is what the Hybrid join algorithm attempts to do. Below, we discuss the use of the Hybrid join algorithm for processing complex query trees represented as right-deep query trees. To illustrate the algorithm, the join of relations A, B, C and D will be presented as shown in Figure 10.

Right-Deep Hybrid Scheduling

With right-deep query trees, query processing becomes much more interesting when using the Hybrid join algorithm (termed **Right-Deep Hybrid Scheduling**). Consider the right-deep join query in Figure 10 and assume that each join will be broken into two buckets, with the first bucket being staged immediately into memory. 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, the 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 (RDHS), it is possible for some tuples to be

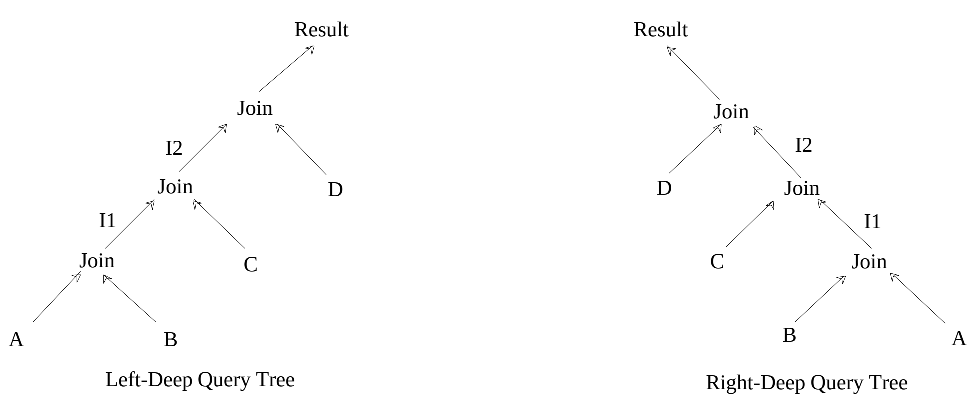


Figure 10

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 RDHS 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, RDHS provides an alternative to the static and dynamic bottom-up scheduling algorithms described above. Whereas these algorithms assumed that enough memory was available to hold at least each relation individually and hopefully several relations simultaneously, the use of the RDHS algorithm potentially reduces the memory requirements while still retaining some dataflow throughout the **entire** query tree. If RDHS can use a single bucket for every relation, it becomes the same as the static right-deep scheduling algorithm. It remains an open question as to when one scheduling strategy will outperform the other.

The Case for Right-Deep Query Trees

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

(2) 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.

(3) The size of the "building" relations can be more accurately 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.

(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) Several strategies proposed to deal with limited memory situations offer a range of interesting tradeoffs. "Breaking" the query tree represents a static approach while the dynamic bottom-up scheduling algorithm reacts much better to the amount of memory available at run-time. The RDHS strategy can deliver tuples **sooner** and in a more constant stream to the user/application than a similar left-deep query tree can.

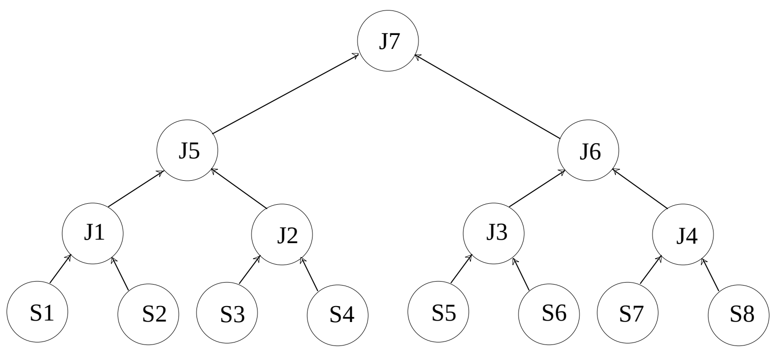
(6) Right-deep trees are generally assumed to be the most memory intensive query tree format but this is not always the case. Consider the join of relations A, B, C, and D as shown in Figure 10 for both a left-deep and a right-deep query tree format. Assume the size of all the relations is 10 pages. 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.

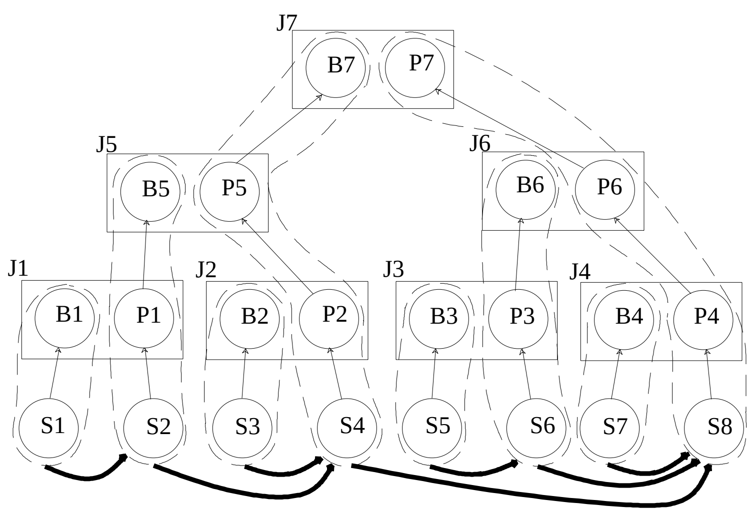
(7) The size of intermediate relations may grow with left-deep trees in the case where attributes are added as the result of each additional join. Since the intermediates are stored in memory hash tables, memory requirements will increase. Note that although the width of tuples in the intermediate relations 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.

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 must be spooled to disk and re-read at the appropriate time.

Limited Memory

However, by intelligently scheduling operators it is possible to reduce the memory demands of a query. Consider again the previous schedule for executing the 7 join query. 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.

3.4. Issues When Using the Sort-Merge Join Algorithm

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.

The modifications to the operator dependency graphs required to support the sort-merge join method can be found in [SCHN89b]. These modifications are very simple but are not presented here due to space limitations. One interesting point to note about using the sort-merge join algorithm is that the left-deep and right-deep query tree representations become equivalent because all the base 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.

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.

The modifications to the operator dependency graphs required to support the sort-merge join method can be found in [SCHN89b]. These modifications are very simple but are not presented here due to space limitations. One interesting point to note about using the sort-merge join algorithm is that the left-deep and right-deep query tree representations become equivalent because all the base 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.

4.2. Limited Memory Environments

In the discussion of the alternate query tree formats in Section 3, we assumed that sufficient memory was available to hold the "building" relation for all concurrent join operators. (We are again only addressing hash-joins.) However, even with the trend towards database machines with larger numbers of processors and larger memories, it cannot be expected that the hash tables for complex join queries accessing large datasets will always fit completely in memory at the same time, especially when right-deep query trees are employed.

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. Furthermore, 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 be 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. If it is not possible to modify the query plan at runtime, performance will suffer.

A simpler strategy may be to have the runtime query scheduler adjust the number of buckets for the Hybrid join algorithm in order to react to changes in the amount of memory available. 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 significantly 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 cannot always be assumed that sufficient space will be available for all N hash tables concurrently, we have developed 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. This scheduling strategy is called **static right-deep scheduling**.

A more dynamic strategy, called **dynamic bottom-up scheduling**, schedules the scans S2 to SN+1 (Figure 6) 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 the scans have been 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´. Further processing of the query tree proceeds in an identical manner only the first scan to be scheduled is 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 significant impact on performance because the benefits of data flow processing are lost when the results of the temporary join computation must be spooled to disk. Although, we have assumed that enough memory is available to hold at least each relation individually and, hopefully, several relations simultaneously, this may not always be the case. An alternative approach is to preprocess the input relations in order to reduce memory requirements. This is what the Hybrid join algorithm attempts to do. Below, we discuss the use of the Hybrid join algorithm for processing complex query trees represented as right-deep query trees. To illustrate the algorithm, the join of relations A, B, C and D will be presented as shown in Figure 10.

Right-Deep Hybrid Scheduling

With right-deep query trees, query processing becomes much more interesting when using the Hybrid join algorithm (termed **Right-Deep Hybrid Scheduling**). Consider the right-deep join query in Figure 10 and assume that each join will be broken into two buckets, with the first bucket being staged immediately into memory. 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, the 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

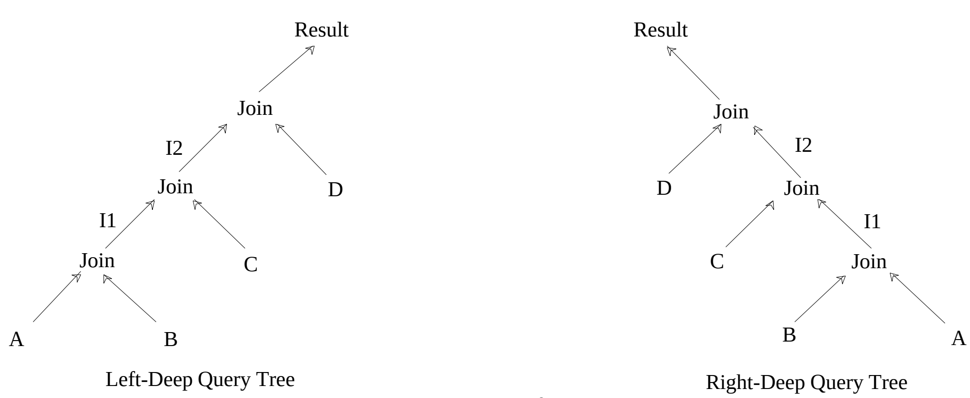


Figure 10

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 (RDHS), 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 RDHS 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, RDHS provides an alternative to 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 RDHS algorithm potentially reduces the memory requirements while still retaining some dataflow throughout the **entire** query tree. If RDHS 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) As shown previously, right-deep query trees provide the best potential for exploiting parallelism (N concurrent scans and hash table build operations).

(2) 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.

(3) The size of the "building" relations can be more accurately 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.

(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) Several strategies proposed to deal with limited memory situations offer a range of interesting tradeoffs. "Breaking" the query tree represents a static approach while the dynamic bottom-up scheduling algorithm reacts much better to the amount of memory available at run-time. The RDHS strategy can deliver tuples **sooner** and in a more constant stream to the user/application than a similar left-deep query tree can.

(6) 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 10 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.

(7) 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. An Initial Performance Evaluation of Left-Deep vs. Right-Deep Query Trees

The preceding discussion indicated that a multi-way join query represented in a right-deep query tree can potentially offer significant performance advantages over the same query represented in a left-deep tree. In this section we focus on quantitatively measuring the extent of this performance advantage. The reader should note that the goal of the performance evaluation is to determine the range of possible performance tradeoffs between left-deep and right-deep query trees and does not purport to encompass all possible situations. Rather, the analysis will serve to show the feasibility of the strategies proposed for processing multi-way join queries.

As the experimental vehicle for our analysis we chose the shared-nothing database machine Gamma [DEWI86, DEWI90]. Gamma currently runs on a 32 processor iPSC/2 Intel hypercube [INTE88] with one 330 megabyte MAXTOR 4380 (5 1/4") disk directly attached to each Intel 80386 processor. One deficiency of the iPSC/2's I/O system is that it does not provide DMA support for disk transfers. Rather, disk blocks are instead transferred by the disk controller into a FIFO buffer, from which the CPU must copy the block into memory.[2] A high-speed hypercube connected network topology using specially designed hardware routers is used for communication between processors.

Instead of implementing the scheduling algorithms directly on Gamma and measuring their performance, we chose to build a simulation model of Gamma. There were two reasons for this decision. First, we felt it would be simpler and faster to construct new scheduling algorithms in a simulator than in the actual system. Second, a simulation model allows us to study the different algorithms in hardware configurations much larger than Gamma currently provides. In the rest of this section, we will describe the simulation model, the validation of the simulation model, and some experimental results from this model comparing the performance tradeoffs of scheduling left-deep and right-deep query trees.

5.1. Simulation Model

The simulation model of Gamma is constructed as follows. Each node in the multiprocessor system consists of a Disk module, a CPU module, a Query Scheduler module, and multiple instances of an Operator module. Additionally, three stand-alone modules are provided: a Network module, a File Manager module, and a Terminal module. The DeNet simulation language [LIVN88] was used to construct the simulator.

The Disk module schedules disk requests according to an elevator algorithm. In order to accurately reflect the hardware currently being used by Gamma, the disk module interrupts the CPU when there are bytes to be transferred from the I/O channel's FIFO buffer to memory or vice versa. The CPU module enforces a FCFS non-preemptive scheduling paradigm on all requests, with the exception of these byte transfers to/from the disk's fifo buffer. Operator modules are responsible for modeling the relational operators select and join. These modules repeatedly make requests to the CPU, Disk and Network modules in order to perform their particular operation. An instance of this module is required for each select and join operator in a query tree. The Query Scheduler module implements the algorithms to process left-deep and right-deep query trees. It simply assigns operators of the query tree to Operator managers on the relevant processing nodes depending on the scheduling algorithm being used. The

---

[2] Intel was forced to use such a design because the I/O system was added after the system had been completed and the only way of doing I/O was by using an empty socket on the board which did not have DMA access to memory.

Network module currently models a fully connected network and the Terminal module provides the entry point of new queries into the system. Finally, the File manager keeps track of how many files are defined, what disks each file is declustered over, and the number of pages of each file on each disk. An assignment of each page in a file to a physical disk address is maintained, although it is assumed that all the pages of a file are contiguous. This physical assignment of file pages allows for more accurate modeling of sequential as well as random disk accesses.

5.2. Simulation Model Validation

In order to provide more faith in the results from the multiprocessor database machine simulator, we validated the simulator against results produced by Gamma. For the validation procedure, the system was configured to use 18 KByte disk pages and 8 KByte network pages. Costs associated with basic operations on this machine and relevant system parameters are summarized in Table 1.

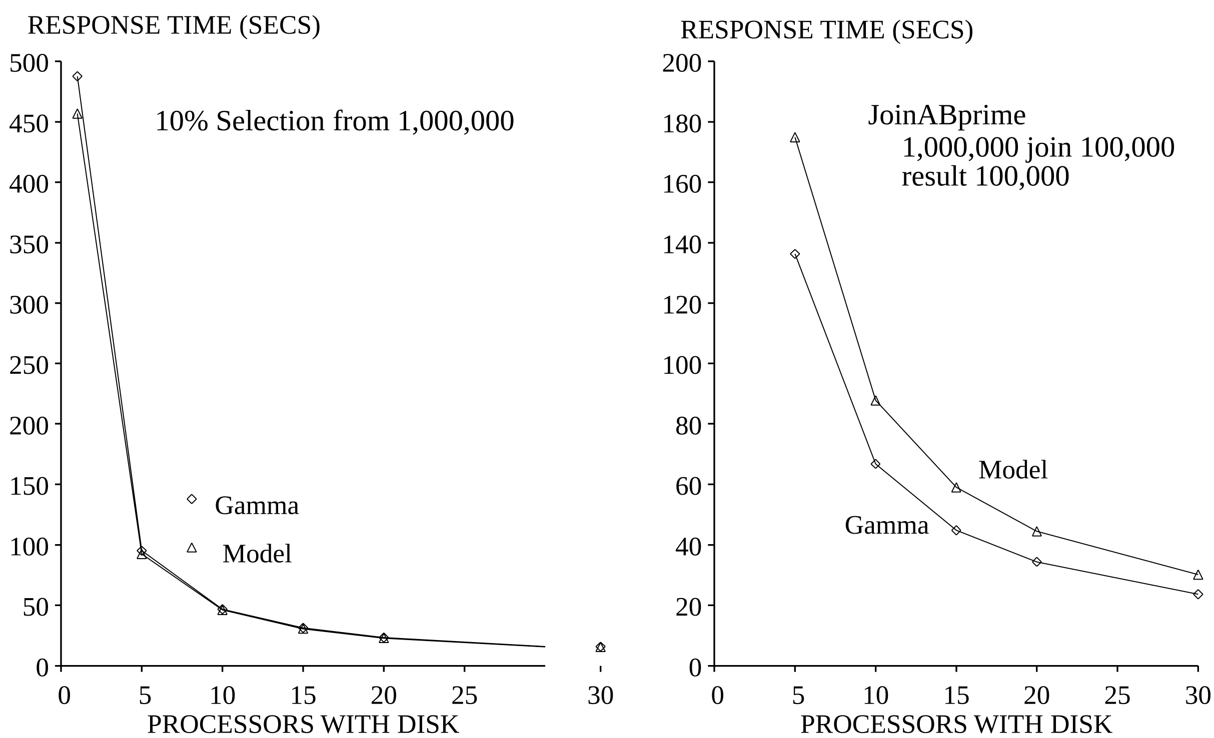
To validate the simulation model we present the performance of both a 10% selection query and a join query in a system with 1-30 processors with disks. Expanded versions of the Wisconsin Benchmark relations [BITT83] serve as the test database. The selection query retrieves 100,000 tuples from a 1,000,000 tuple relation and stores the resulting tuples back into the database. As shown in Figure 11, the simulation model matches with observed Gamma performance very closely for this query. However, the actual error is greater than implied by Fig-

Disk Parameters	
Average Seek Time	16 msec
Average Settle Time	2 msec
Average Latency	0-16.667 msec (Uniform)
Transfer Rate	1.8 MBytes/sec
Track Size	18 KBytes
Disk Page Size	18 KBytes
Xfer Disk Page from SCSI to memory	9000 instructions
Network Parameters	
Maximum Packet Size	8 KBytes
Send 100 bytes	0.6 msec
Send 8192 bytes	5.6 msec
Cpu Parameters	
Instructions/Second	4,000,000
Read 18K Disk Page	32,800 instructions
Write 18K Disk Page	61,500 instructions
Miscellaneous	
Tuples/Network Packet	36
Tuples/Disk Page	82
Number of Sites	1-30
Number of CPUs/site	1
Number of Disks/site	1

Simulation Parameters for Model Validation
Table 1

ure 11 because Gamma uses a one page readahead mechanism when reading pages from a file sequentially. The performance implications of this mechanism are discussed in more detail below.

In order to validate join performance in the model, we joined a 1,000,000 tuple relation (200 megabytes) with a 100,000 tuple relation (20 megabytes) to produce a 100,000 tuple result relation (40 megabytes). As illustrated by Figure 11, the simulation model overestimates the response time for this query by a constant factor of 20% over the range of 5 to 30 processors with disk. We attribute much of this modeling inaccuracy to be related to Gamma's use of a one page readahead mechanism when scanning a file sequentially. Since join queries are very CPU intensive operations, Gamma can effectively overlap most of the CPU costs of constructing and probing the hash table with the disk I/O necessary for reading the joining relations. This should not imply, though, that the model is overpredicting performance by 20% for the 10% selection query presented in Figure 11. The CPU requirements of this query are much lower than that of the join query and thus the extent of the overlap of CPU and disk processing is much more limited. This claim is further supported by the fact that the simulation model accurately predicts execution times for selection queries which use a non-clustered B-tree access. These queries generate a series of random disk requests and hence readahead is not employed.



Validation of Selection and Join Performance
Figure 11

5.3. Experimental Design

As stated in the beginning of the section, the experiments were designed to present the range of performance differences between left-deep and right-deep query trees. For the experiments conducted, the query suite consisted of join queries composed of 1, 2, 4, and 8 joins. In order to simplify the analysis, though, the queries were highly constrained. For example, the queries were designed such that the size of the result relation is constant regardless of the number of joins in the query or the query tree representation. This was accomplished by making all relations the same size and by setting the join "probe-ability" factor to 1 for every join in the query tree. That is, each probing tuple joins with exactly one building tuple. A parallel version of the Simple hash-join algorithm [DEWI84, SCHN89a] was used as the join method and, unless otherwise stated, it was assumed that enough main memory exists such that hash table overflow never occurs, regardless of the number of concurrent join operations.

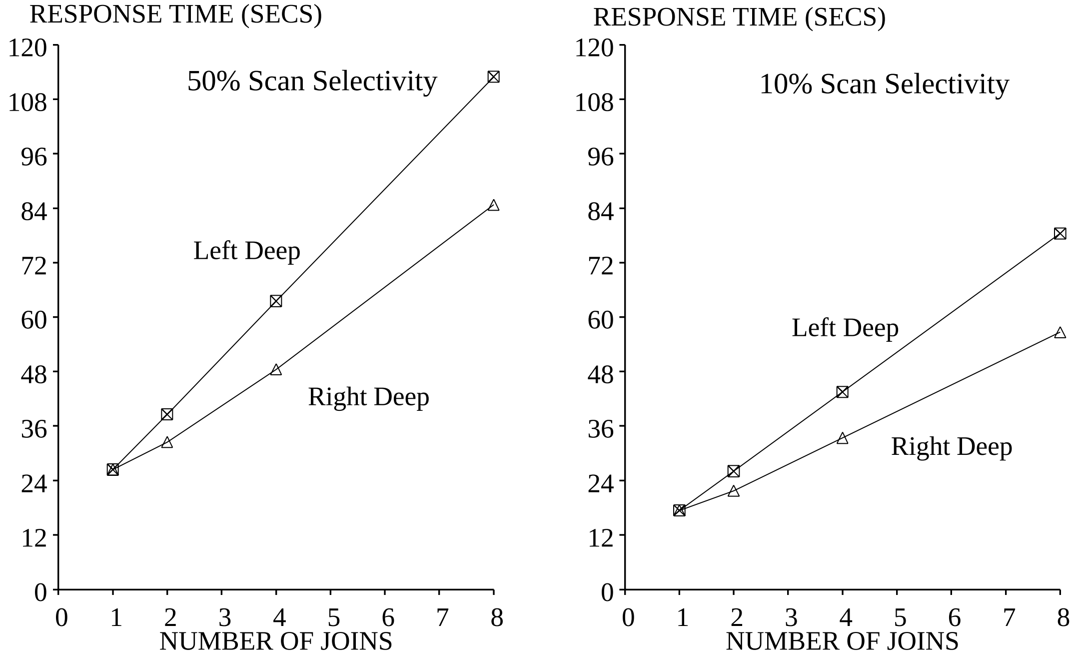
The database was composed of nine 1,000,000 tuple relations and each relation has a selection predicate applied to it which reduces the output cardinality to 500,000 tuples. Since input tuples are 208 bytes wide and attributes are not added with each successive join, the result cardinality of ALL the joins was 500,000 tuples, each 208 bytes wide. All result relations were written back into the database. In order to more accurately predict performance for “typical” database machines, a 25% buffer pool hit ratio was specified in order to model a disk prefetch mechanism. Other system parameters are identical to those specified in Table 1. Response time for the queries is measured from the time the query plan is submitted to the database machine until the query is fully computed.

Four major experiments are reported here. The first experiment considers performance differences in an environment where the declustering of the joining relations forces a high level of resource contention. In the second experiment, the environment is changed to ensure a low level of resource contention. In the third and fourth experiments, the first two experiments are repeated in an environment where memory for joining is not unlimited.

5.3.1. High Resource Contention Environment

In a database machine with a relatively small number of processors, it is not unlikely that large relations will be declustered over ALL the available nodes. Thus, executing multiple scan and join operators concurrently will result in a high degree of resource contention. For these experiments, the system was configured such that each relation was declustered over 50 nodes. Each join in the query tree was also processed on all 50 nodes.

The results of these full declustering experiments are shown in Figure 12. For each different join query, the performance of the right-deep query tree is approximately 15-20% faster than for its analogous left-deep query tree (left-deep and right-deep query trees are identical for single join queries). This performance improvement occurs because the disks are not fully utilized. Thus, executing the scans in parallel for the right-deep trees provides some performance improvement. However, if the scan of a declustered relation fully utilizes each of its associated



Full Declustering - 50 nodes
Figure 12

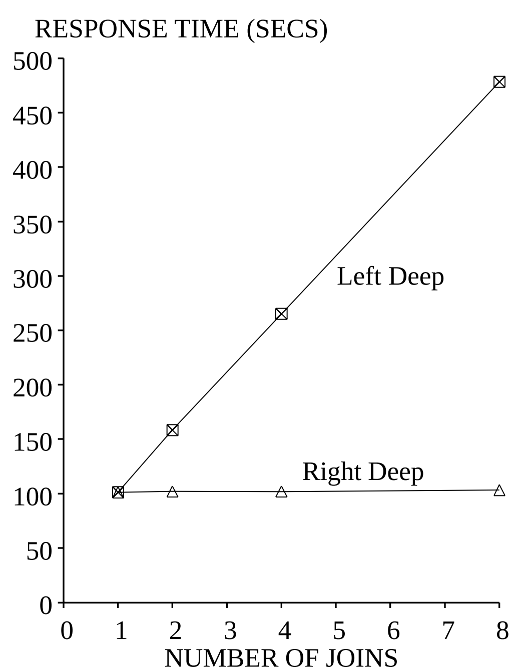
disks, a right-deep query tree will not demonstrate a performance advantage under these experimental conditions. Note, that the maximal memory requirements for left-deep and right-deep query trees is identical for 1 and 2 join queries, but is twice as high for right-deep queries with 4 joins and four times as high with 8 joins. Thus, when all the relations to be joined are declustered over the same set of nodes, the benefits provided by using right-deep query trees cannot be maintained relative to the amount of memory consumed as the number of joins increases.

5.3.2. Low Resource Contention Environment

In this next set of experiments we wanted to demonstrate the performance tradeoffs between the two query representation strategies in a configuration with more processors. In such a machine, it is likely that relations will be declustered over a subset of nodes [GERB87, COPE88] and hence, resource contention will be reduced when executing several operators concurrently. The system was configured in the following manner. Each of the nine 1,000,000 tuple relations was declustered over 10 distinct, non-overlapping nodes. Each join was also processed at 10 nodes. The processors used to execute each join operator were assigned such that they were identical to the processors over which each "building" relation was declustered. Given these conditions, the number of processors actively participating in each query during the scanning of relations and the building/probing of hash tables increased as the number of joins in the query increased. For example, in a 2-join query 30 nodes were used, and in an 8-join query 90 nodes were used. Regardless of the number of joins in the query, each **result** relation was declustered over all 90 nodes.

As illustrated by Figure 13, left-deep query trees are unable to take advantage of the hardware resources that become available as additional joins are added to the query. This is to be expected because relations must be scanned one at a time when a left-deep query tree is employed. However, for right-deep query trees, a nearly constant response time is maintained as the number of joins is increased from one to eight. This may appear startling but can be easily explained given the experimental parameters. Consider the first step in executing the query - scanning the "building" relations and constructing the corresponding hash tables. Since all relations are the same size and have the same selectivity factor applied, and since all the relations are declustered over distinct nodes and the join nodes correspond to the base relation declustering nodes, each scan can be executed completely in parallel and without interference. Thus, the cost of this operation is constant regardless of the number of joins (disregarding the small overhead necessary for initiating the operators).

The second phase (the "probing" phase) can scale with the number of joins due to the effect of pipelining. As tuples are produced from a lower join they are immediately sent across the network to participate in the next level of the join. Thus, processing of tuples in the upper and lower levels of the tree are overlapped with each other. Viewed another way, the throughput of the pipeline is constant regardless of the depth of the join tree and the difference in response time as the number of join levels increases is due to the increased latency to initiate and terminate



Partial Declustering - 90 nodes
Figure 13

the pipeline.

The results contained in Figure 13 represent best-case performance improvements of right-deep versus left-deep query trees. All experimental parameters were set to allow the parallelism potential of the right-deep strategy to be exploited to its fullest. Under more realistic conditions, the performance improvements of right-deep query trees will fall between the extremes presented in Figures 12 and 13. Also, it should be noted that the right-deep query with eight joins required four times more memory than any of the left-deep join queries. However, the results do demonstrate the extremely high performance benefits that can be obtained by using a right-deep query representation strategy.

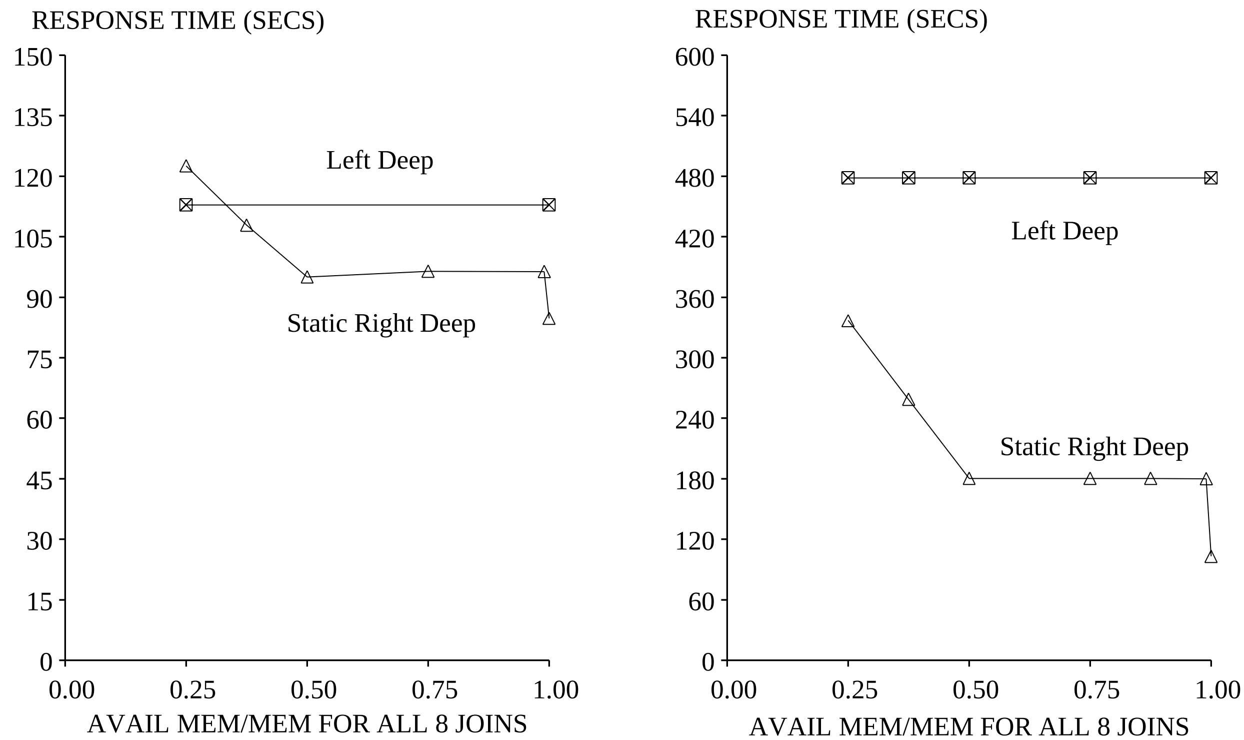
5.3.3. Limited Memory Experiments

In this set of experiments we relaxed the assumption that an unlimited amount of memory exists for joining. All query and model parameters are identical to those reported in the previous section with the exception that for this analysis we concentrated on the query consisting of 8 joins. High resource and low resource contention (full declustering and disjoint declustering) experiments were again conducted. The static right-deep scheduling strategy (see Section 4.2.2) was used for processing right-deep query trees.

In order to model a limited memory environment, we modified the aggregate amount of memory available for joining relative to the memory required to stage all eight "building" relations into memory simultaneously. Response time was plotted for left-deep and right-deep strategies for x-axis values ranging from a value of 0.25, where only 2 of the 8 building relations could co-reside in memory, to a value of 1.00, where all 8 building relations can fit in memory simultaneously. X-axis values less than 0.25 would have required resolution of memory overflow for left-deep query trees and are not reported. For the static right-deep strategy it was assumed that the optimizer could perfectly predict scan selectivities and thus could always choose the optimal places to "break" the query tree.

5.3.3.1. Limited Memory - High Resource Contention

In Figure 14, the performance of the left-deep and static right-deep scheduling algorithms are shown as the amount of available memory is varied in an environment where all base relations are declustered across all 50 sites. Three observations should be noted from this figure. First, it is obvious that the left-deep scheduling algorithm is not able to take advantage of memory as it is added. In contrast, the static right-deep scheduling algorithm does demonstrate some significant performance advantages by using any additional memory available. Next, the crossover point in the graphs demonstrates the fact that "breaking" the query tree into too many pieces can be detrimental to the performance of right-deep scheduling algorithms. For example, at the x-axis value 0.25, the tree had to be broken into three pieces to ensure that the right-deep strategy did not experience memory overflow, requiring the writing and subsequent reading of temporary join computations to and from the disk at three points during query



Limited Memory - Full and Disjoint Declustering
Figures 14 & 15

execution. The last point to note is the flatness of the right-deep scheduling graph from 0.5 to just before 1.0. Over this range, the query tree had to be broken into only two pieces. Since the joins in the queries tested produced intermediate relations of a constant size regardless of the number of joins in the query, the placement of the "break" has no effect on performance because the same number of tuples are temporarily staged to disk. Under more likely conditions of growing or diminishing temporary join size results, the selection of the break points for a query will almost certainly have some effect on the execution time of the query. This will be explored in a future performance analysis.

5.3.3.2. Limited Memory - Low Resource Contention

In Figure 15, we present the execution time of the left-deep and right-deep scheduling strategies for the 8-join query when the relations to be joined are declustered over mutually disjoint processors with disks. The simulation parameters are identical to those reported in Section 4.3.2.

The results are very similar to those shown in Figure 14, i.e., the shape of the curves is identical. It is obvious though, that the partial declustering environment offers additional performance advantages for right-deep scheduling strategies even when memory is not unlimited. This is very encouraging because, as stated earlier, it is likely that relations will be partially declustered in database machines with large numbers of processors/disks.

6. Conclusions

In this paper, we have described many of the problems and tradeoffs associated with the task of processing queries composed of many joins in a multiprocessor database machine. In particular, we focused on how the strategy chosen to represent a query tree affects the degree to which parallelism can be applied within a query and the corresponding effects on performance, the resource consumption of the query, the extent that dataflow processing techniques can be applied, and the cost of query optimization. Hash based join methods were assumed for much of this analysis although the sort-merge join method was discussed briefly.

Results obtained from this analysis indicate that the right-deep query representation strategy is well suited to exploit the parallelism found in large multiprocessor database machine configurations. As an added benefit, the importance of accurately estimating join selectivities is potentially reduced when using right-deep query trees.

In order to quantitatively measure the performance benefits that right-deep query trees can provide over their corresponding left-deep query trees, we constructed a simulation model of a parallel database machine and implemented scheduling algorithms for processing queries represented in these formats. Experimental results from the analysis of these algorithms confirmed that right-deep query trees can offer very significant performance advantages in large database machines. However, the extent of the performance improvement is strongly dictated by the physical placement of base relations and comes at the cost of increased resource consumption. Also, when memory is limited, the performance difference between the scheduling algorithms diminishes.

Our future work includes analyzing the performance tradeoffs of these query tree representation strategies and the proposed scheduling algorithms under multiuser workloads and with skewed data distributions. Although multiuser performance comparisons were beyond the scope of this paper, the memory requirements 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 hope to apply the technique of Adaptive Sampling [LIPT90a, LIPT90b] to help tackle the problem of skew.

Acknowledgments

We would like to thank Jeff Naughton for his many helpful discussions concerning this work.

7. References

[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.

[BITT83] Bitton, D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, October, 1983.

[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.

[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.

[DEWI90] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., and R. Rasmussen, “The Gamma Database Machine Project”, to appear IEEE Transactions on Knowledge and Data Engineering, March, 1990.

[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.

[GHAN90] Ghandeharizadeh, S., and D.J. DeWitt, “A Multiuser Performance Analysis of Alternative Declustering Strategies”, Proceedings of the 6th International Conference on Data Engineering, 1990.

[GRAE87] Graefe, G., “Rule-Based Query Optimization in Extensible Database Systems”, Ph.D. 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.

[KITS88] Kitsuregawa, M., Nakano, M., and M. Takagi, “Query Execution for Large Relations On Functional Disk System,” Proceedings of the 5th International Conference on Data Engineering, 1989.

[LIPT90] Lipton, R., J. Naughton, and D. Schneider, “Practical Selectivity Estimation through Adaptive Sampling”, Proceedings of the 1990 SIGMOD Conference, Atlantic City, New Jersey, May, 1990.

[LIVN88] Livny, M., **DeNet User’s Guide**, Version 1.0, Computer Sciences Department, University of Wisconsin, Madison, WI, 1988.

[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.

[SCHN89a] 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.

[SCHN89b] Schneider, D. and D. DeWitt, “Design Tradeoffs of Alternative Query Tree Representations for Multiprocessor Database Machines”, Computer Sciences Technical Report #869, University of Wisconsin-Madison, August, 1989.

[SHAP86] Shapiro, L., “Join Processing in Database Systems with Large Main Memories”, ACM Transactions on Database Systems, Vol. 11, No. 3, September, 1986.

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