COMPLEX QUERY PROCESSING IN MULTIPROCESSOR DATABASE MACHINES

by

Donovan Schneider

Computer Sciences Technical Report #965
September 1990

COMPLEX QUERY PROCESSING IN MULTIPROCESSOR DATABASE MACHINES

by

DONOVAN A. SCHNEIDER

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1990

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 DARPA/NASA sponsored Graduate Research Assistantship in Parallel Processing, by a Digital Equipment Corporation External Research Grant, and by a research grant from the Tandem Computer Corporation.

ABSTRACT

High performance multiprocessor database machines have been made feasible with the advent of cheap, powerful microprocessors and large main memories. However, exploiting these platforms to support high speed complex query processing has lagged behind the hardware technology. The thrust of this dissertation has concentrated on developing strategies for efficiently processing join queries consisting of on the order of 10 joins in a parallel database machine with hundreds of processors. Although the algorithms were developed with a shared-nothing architecture in mind, the algorithms can be applied to shared-memory systems with little modification.

For queries that join only a few relations, we have found that the parallel Hybrid hash-join algorithm dominates under most circumstances, except when the join attribute values of the building relation are highly skewed.

For multi-way join queries, a subset of the optimization search space of query plans called right-deep query plans is identified as being particularly important in this highly-parallel environment. Several algorithms are proposed for processing right-deep query plans and results from a simulation model are presented that demonstrate that right-deep plans can indeed offer significant performance advantages over the more traditional left-deep plans under many conditions.

ACKNOWLEDGMENTS

This dissertation was made possible through the expert guidance and support of Professor David DeWitt. David provided a stimulating research environment, opportunities for consulting, and financial support for attending both domestic and international conferences. Most importantly, he taught me how to do research. I have the utmost respect for David and am honored to be counted among his Ph.D. graduates.

I would also like to thank Professor Mike Carey for "turning me on" to databases with his excellent teaching. His comments and suggestions concerning this dissertation have greatly improved its quality and are much appreciated. Having the opportunity to work with Professor Jeff Naughton has also been a pleasure. The work we did on sampling has helped broaden my perspective of databases. I would also like to thank the other members of my committee, Professors Yannis Ioannidis and John Beetem, for their helpful comments.

Thanks should also be given to the many friends and colleagues I have had the pleasure to meet during my graduate studies. Bob Gerber, M. Muralikrishna, and Goetz Graefe made working on the Gamma project a great experience; special thanks to Bob for smoothing my transition onto the project. I would also like to thank Lester McCann, Scott and Lisa Leutenegger, Mark Allmen, George and Cheryl Bier, Bob Kabler, Mike Skarpelos, Rajiv Jauhari and many others for their friendship and for providing much needed diversions from graduate school. Thanks are also due to the members of the infamous Hackers basketball team, the SPAM volleyball team, and the bicycling gang. I would especially like to thank Shahram Ghandeharizadeh; we've shared an office, an advisor, many painful programming nights, many Gamma highlights, innumerable technical discussions, several great conference trips, some crazy stunts, and many beers over the last five years.

Sheryl Pomraning has been an invaluable aid during graduate school. Her help in preparing technical papers and in other areas is much appreciated. Paul Beebe, Mitali Lebeck, and the rest of the Computer Science Lab also deserve thanks for providing a great working environment; special thanks to Rick Rasmussen for getting the Hypercube going when we really needed it.

I would also like to thank Professor Michael Meeker from the University of Wisconsin — Oshkosh for encouraging me to go to graduate school at Madison.

And finally, special thanks go to my parents. Without their support, encouragement, and inspiration these years of graduate study would not have been possible. I would also like to thank my brother, David, and my sisters, Dawn and Deb, for all their love and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

Several important events 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 the cost of memory has declined. Finally high-speed communication networks have enabled the efficient interconnection of large numbers of 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 operation is the cornerstone of a relational database management system (RDBMS), we are interested in studying how to process join queries in order to take advantage of the resources available in these database machines. Furthermore, as a result of the increased demands being placed on database systems in recent years, queries have become much more complicated and hence support must be provided for efficiently executing queries composed of many join operations. The main contribution of this dissertation is to address the efficient execution of complex join queries, especially queries involving many relations, on multiprocessor database machines.

1.2. Outline

Chapter 2 surveys the literature related to join query processing. Emphasis is placed on parallel join algorithms and on strategies for processing multi-way join queries.

Before addressing the problem of efficiently executing queries composed of many join operations, we needed to first understand all the problems and tradeoffs of executing queries consisting of a single join in a multiprocessor database machine environment. This problem has been studied previously by several researchers including [BARU87, BRAT87, DEWI85, DEWI87, KITS88, LU85]. However, the performance of the more popular parallel join algorithms has not been compared in a common hardware/software environment. In Chapter 3, we present the results of how a variety of alternative join algorithms perform when implemented in the shared-nothing multiprocessor database machine Gamma [DEWI86, DEWI90]. The algorithms studied were parallel versions of sort-merge, GRACE [KITS83], Simple [DEWI84], and Hybrid [DEWI84]. These algorithms cover the spectrum from hashing, to looping with hashing, and finally to sorting. The performance analysis of these different algorithms includes factors such as the effects of different tuple distribution policies, the presence of bit vector filters, varying amounts of memory available for joining, and non-uniformly distributed join attribute values. The main result from this analysis is that the Hybrid hash-join algorithm dominates all the other join algorithms, except when the join attribute values of the inner, "building" relation are highly skewed.

In Chapter 4, the results of Chapter 3 are extended to encompass the problem of executing queries composed of many joins in a multiprocessor database machine. In particular, the ramifications of choosing a particular query plan format in which to optimize a multi-way join query is addressed. Two restricted query plan formats, left-deep and right-deep, as well as the fully general bushy-tree format are considered. The tradeoffs that we studied include the potential for exploiting intra-query parallelism (and its corresponding effect on performance), resource consumption (primarily memory), support for dataflow query processing, and the cost of query optimization. From this analysis, several different algorithms are proposed for executing complex join queries when formatted in each of the alternative query trees.

In Chapter 5, the performance of two of the query processing algorithms for processing right-deep query plans is studied in depth. A simulation model of a shared-nothing multiprocessor database machine is used to conduct this performance analysis. Additionally, Chapter

6 presents a comparison of the performance of algorithms that employ left-deep query plans with that of right-deep query plans. Our conclusions and future research directions are presented in Chapter 7.

CHAPTER 2

SURVEY OF RELATED WORK

Considerable attention has been directed to the efficient processing of join queries since the formulation of the relational model. Initially, nested loops and sort-merge were the algorithms of choice [BLAS77]. Later, work by [BRAT84, DEWI84, KITS83, SHAP86] demonstrated the effectiveness of hash-based join methods for centralized relational database systems when large amounts of memory are available.

Many studies have also addressed parallel implementations of the join operation. [QADA85, VALD84] compare a variety of multiprocessor join algorithms based on analytical models, while [DEWI85] used simulation to study hash-based multiprocessor join algorithms. Several researchers, including [BARU87, BRAT87, DEWI87, DEWI88, KITS88, LU85, SCHN89], reported measurements taken from implementations of a variety of parallel join algorithms. However, none of these papers addressed the processing of queries with more than two joins.

[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 inter-operator parallelism, the tradeoffs of left-deep, right-deep and bushy query tree representations with regard to parallelism, pipelined data flow, and memory consumption are addressed. However, while the basic issues involved in processing complex queries in a multiprocessor environment are discussed, the tradeoffs between the alternative query tree optimization strategies are not studied in depth and no algorithms for processing the different query trees are proposed.

[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 these two 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.

[TAY90] studies the problem of finding the optimal query plan for a multi-way join query in a single-processor environment. The goal is to describe the conditions under which the best plan can be found by searching only a subspace of all possible plans (e.g., when a linear strategy such as a left-deep tree will always represent the best plan). A limiting factor of this work is the simplistic cost model employed. Costs are determined solely by the number of tuples generated at the interior levels of the candidate query plans. The use of this cost measure disregards such effects as using large disk pages, disk page readahead, and most importantly, the amount of work that can be done concurrently.

[STON88, STON89] describes how the XPRS project plans on utilizing parallelism in a shared-memory database machine. This research has several points in common with ours. First, hash joins are used for all equi-join queries. Second, examples of both left-deep and bushy query trees are provided; however, the paper does not discuss the tradeoffs between these two strategies or the impact of each on system performance. It is also not clear if XPRS intends to use right-deep query trees. Optimization during query compilation assumes the entire buffer pool is available, but in order to simplify runtime optimization, the query tree is divided into fragments. These fragments correspond to our operator subgraphs described in Chapter 4. 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. [GRAE90] also supports each of the three alternative query tree formats in the shared-memory database machine Volcano, but the tradeoffs between the different formats are not discussed.

In [BABA87] an algorithm is proposed that accepts as input a set of data flow graphs (optimized query trees), a set of parameters describing the multiprocessor environment (e.g. disk times, CPU costs, network costs, and number of processors), and a set of parameters describing the test database. The algorithm, using heuristics, generates an assignment of relational operators to processors in the system such that overall response time for the collection of data flow graphs is minimized. As part of the assignment, operators may be replicated (parallelized) across multiple processors to reduce response time. Our proposed research differs from theirs in several ways. First, a major objective of our research is to determine how to structure a query tree for a particular query and the tradeoffs between choosing different strategies in a multiprocessor system. [BABA87], however, assumes that their algorithm can easily decompose all input query trees. We consider this assumption to be very simplistic, especially because their algorithm doesn't recognize many of the difficulties involved in synchronizing subtrees in a complex bushy query tree. Also, we are interested in identifying all the relevant costs associated with parallelizing relational operators, especially joins. Again, [BABA87] makes a simplifying assumption that there is NO cost in further replicating an operator. That is, they ignore the cost of starting, terminating, and otherwise scheduling an operator. Perhaps most seriously, a naive memory model is employed. It is assumed that all operators consume identical amounts of memory and that adding a second occurrence of the same operator (although potentially from a different query) on the same processor requires no additional memory. Their algorithm also does not account for operators such as hash joins that can run significantly faster when given additional memory resources.

[MURP89] proposes to increase performance of general purpose shared-memory database systems by using intra-query parallelism and minimizing resource requirements. To process a join operation, it is assumed that a **page connectivity graph** exists. This graph can be constructed from a join index [VALD87] and consists of a node for each page of each relation and an edge between each pair of pages that have at least one matching attribute. [MURP89] develops an algorithm for scheduling a join operation across multiple processors given such a graph. Processors are scheduled according to **page joins** and disk I/O is modeled at the page level. In addition, lower bounds are presented for the join execution time as well as the number of processors required to complete processing in a minimum time. If enough memory does not exist to hold the entire page connectivity graph, the scheduling algorithm attempts to minimize the number of pages that must be re-read.

CHAPTER 3

EVALUATION OF SINGLE JOIN QUERIES

3.1. Introduction

In this chapter, we examine the performance of a variety of parallel join algorithms for processing queries composed of a single join operation. This work is important for two reasons. First, although some researchers, including [BARU87, BRAT87, DEWI87, DEWI88, KITS88], have previously presented performance measurements for several parallel join algorithms, implementations of the more popular parallel join algorithms have not been compared in a common hardware/software environment. And second, a comprehensive understanding of the performance of single-join queries is critical to understanding how to process queries composed of multiple join operations.

The join algorithms we studied were parallel versions of sort-merge, Grace [KITS83], Simple [DEWI84], and Hybrid [DEWI84]. These algorithms cover the spectrum from hashing, to looping with hashing, and finally to sorting. The Gamma database machine [DEWI86, DEWI90] served as the experimental vehicle. We feel that Gamma is a good choice for a comparison environment because its shared-nothing architecture is becoming increasingly popular [COPE88, LORI88, TERA83, TAND88].

The experiments were designed to test the performance of each of the join algorithms under several different conditions. First, we compare the performance of the join algorithms as the amount of memory for joining is varied.[1] We then discuss how bit vector filtering

[1] This set of experiments can also be viewed as predicting the relative performance of the various algorithms when the size of memory is constant and the algorithms are required to process relations larger than the size of available memory.

techniques [BABB79, VALD84] improve performance for each of the parallel join algorithms. Finally, the join algorithms are analyzed in the presence of non-uniformly distributed join attribute values.

The remainder of this chapter is organized as follows. In Section 3.2, we discuss the hardware and software platform used as the testbed for the performance analysis. Next, the four parallel join algorithms that we tested are described in Section 3.3. Section 3.4 contains the results of the experiments that we conducted and our conclusions appear in Section 3.5.

3.2. Overview of the Gamma Database Machine

3.2.1. Hardware Configuration

Currently, Gamma runs on a 32 processor Intel iPSC/2 hypercube [INTE88]. Each processor is configured with an 80386 CPU (4 MIPS), 8 megabytes of memory, and a 330 megabyte MAXTOR 4380 (5 1/4") disk drive. Each disk drive has an embedded SCSI controller that provides a 45 Kbyte RAM buffer that acts as a disk cache on read operations.

The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight[2] full-duplex, serial, reliable communication channels operating at 2.8 megabytes/s. Small messages ($\leq$ 100 bytes) are sent as datagrams. For large messages, the hardware builds a communications circuit between the two nodes over which the entire message is transmitted without any software overhead or copying. After the message has been completely transmitted, the circuit is released. Table 3.1 summarizes the transmission times from one Gamma process to another (on two different hypercube nodes) for a variety of message sizes.

In 1989, we conducted similar experiments to those reported in this chapter using a different hardware platform for Gamma [SCHN89]. At that time Gamma was running on 17

[2]On configurations with a mix of compute and I/O nodes, one of the eight channels is dedicated for communication to the I/O subsystem.

Packet Size (in bytes)	Transmission Time
50	0.74 ms.
500	1.46 ms.
1000	1.57 ms.
4000	2.69 ms.
8000	4.64 ms.

Hypercube Packet Transmission Times
Table 3.1

VAX/750 processors connected via a ring. Only eight of the seventeen processors had an attached disk. The processors were also much slower (0.5 MIPS) and had only 2 megabytes of memory each. Throughout this chapter, we will draw comparisons with the results reported in [SCHN89] to point out the effects of the underlying hardware.

3.2.2. Software Overview

3.2.2.1. Physical Database Design

In Gamma, all relations are **horizontally partitioned** [RIES78] across all disk drives in the system. The key idea behind horizontally partitioning each relation is to enable the database software to exploit all the I/O bandwidth provided by the hardware. By declustering[3] the tuples of a relation, the task of parallelizing a selection/scan operator becomes trivial as all that is required is to start a copy of the operator on each processor.

Three alternative ways of distributing the tuples of a relation are provided: round-robin, hashed, and range partitioning. As implied by its name, in the first strategy when tuples are loaded into a relation, they are distributed in a round-robin fashion among all disk drives. If the hashed strategy is selected, a randomizing function is applied to the "key" attribute of each tuple to select a storage unit. In the third strategy the user specifies a range of key values for each site.

---

[3]Declustering is another term for horizontal partitioning that was coined by the Bubba project [LIVN87].

3.2.2.2. Query Execution

Gamma uses traditional relational techniques for query parsing, optimization [SELI79, JARK84], and code generation. Queries are compiled into a tree of operators with predicates compiled into machine language. After being parsed, optimized, and compiled, the query is sent by the host software to an idle scheduler process through a dispatcher process. The scheduler process, in turn, starts operator processes at each processor selected to execute the operator. The task of assigning operators to processors is performed in part by the optimizer and in part by the scheduler assigned to control the execution of the query. For example, the operators at the leaves of a query tree reference only permanent relations. Using the query and schema information, the optimizer is able to determine the best way of assigning these operators to processors.

In Gamma, the algorithms for all operators are written as if they were to be run on a single processor. The input to an operator process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a **split table**. Once the process begins execution, it continuously reads tuples from its input stream, operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table. Consider, for example, the case of a selection operation that is producing tuples for use in a subsequent hash join operation. If the join is being executed by N processes, the split table of the selection process will contain N entries. For each tuple satisfying the selection predicate, the selection process will apply a hash function to the join attribute to produce a value between 0 and N-1. This value is then used as an index into the split table to obtain the address (e.g. machine_id, port #) of the join process that should receive the tuple. When the process detects the end of its input stream, it first closes the output streams and then sends a control message to its scheduler indicating that it has completed execution. Closing the output streams has the side effect of sending *end of stream* messages to each of the destination processes. Except for three control messages, the execution of an operator is completely self-scheduling. Data flows among the processes executing a query tree in a dataflow fashion. If the result of a query is a new relation, the operators at the root of the query tree distribute the result tuples on a round-robin basis to store operators at each disk site.

3.2.2.3. Operating and Storage System

The NX/2 operating system provided by Intel for the hypercube was modified to better support database applications. A thread package was added that provides lightweight processes with shared memory. Messages between two processes on the same processor are *short-circuited* by the communications software.

File services in Gamma are based on the Wisconsin Storage System (WiSS) [CHOU85]. These services include structured sequential files, B^+ tree indices, byte-stream files as in UNIX, long data items, a sort utility, and a scan mechanism. A one page readahead mechanism is used when scanning a file sequentially.

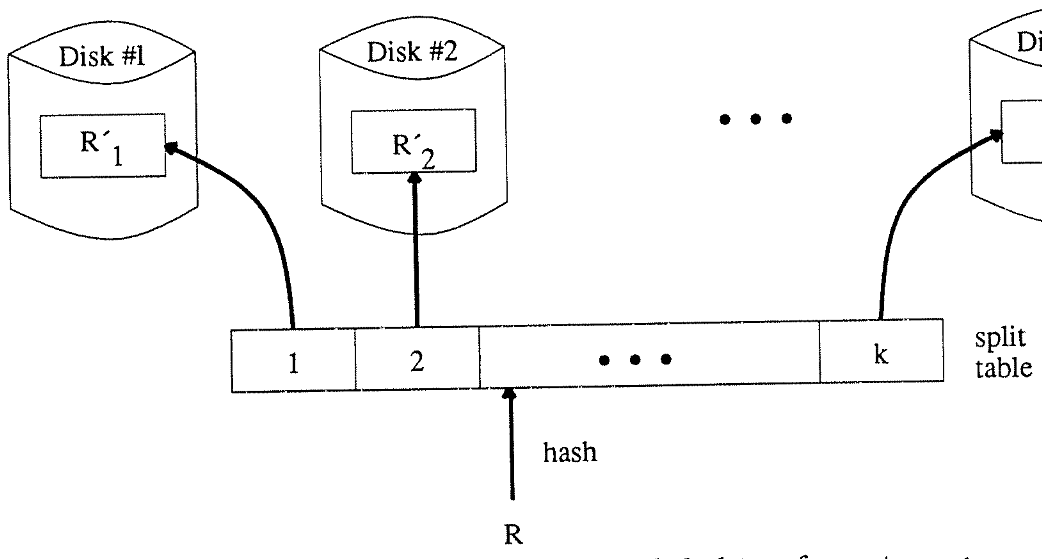
3.3. Parallel Join Algorithms

We implemented parallel versions of four join algorithms: sort-merge, Grace [KITS83], Simple hash [DEWI84], and Hybrid hash-join [DEWI84]. A common feature of the parallel versions of each of these algorithms is the use of a hash function to partition each relation being joined into a collection of disjoint subsets that can be processed independently and in parallel. This partitioning is performed by applying a hash function to the join attribute of each tuple. The actual join computation depends on the algorithm: building and probing of hash tables is used for the Simple, Grace, and Hybrid algorithms, whereas sorting and merging is used for the sort-merge algorithm. As part of the partitioning process, the Grace and Hybrid join algorithms first partition the two relations being joined into additional fragments when the inner relation is larger than the amount of available main memory. This is referred to as the **bucket-forming phase** [KITS83]. More details on each algorithm are presented in the following sections.

In the following discussion, R and S refer to the relations being joined. R is the smaller of the two relations and is always the inner joining relation.

3.3.1. Sort-Merge

Our parallel version of the sort-merge join algorithm is a straightforward adaptation of the traditional single processor version of the algorithm and is essentially identical to the algorithm employed by the Teradata machine [TERA83, DEWI87]. The smaller of the two joining relations, R, is first partitioned through a split table that contains an entry for each processor with an attached disk. A hash function is applied to the join attribute of each tuple to determine the appropriate disk site. As the tuples arrive at a site they are stored in a temporary file. When the entire R relation has been redistributed, each of the local files is sorted in parallel. As an example, Figure 3.1 depicts R being partitioned across K disk nodes into relation R′. Notice that each relation fragment of R on each disk will be passed through the same split table for redistribution. Relation S is then processed in the same manner. Since the same hash function is used to redistribute both relations, only tuples within fragments at a particular site have the possibility of joining [KITS83]. Thus, a local merge join performed in parallel across the disk sites will fully compute the join.



Partitioning of relation R across K disk drives for sort-merge.
Figure 3.1

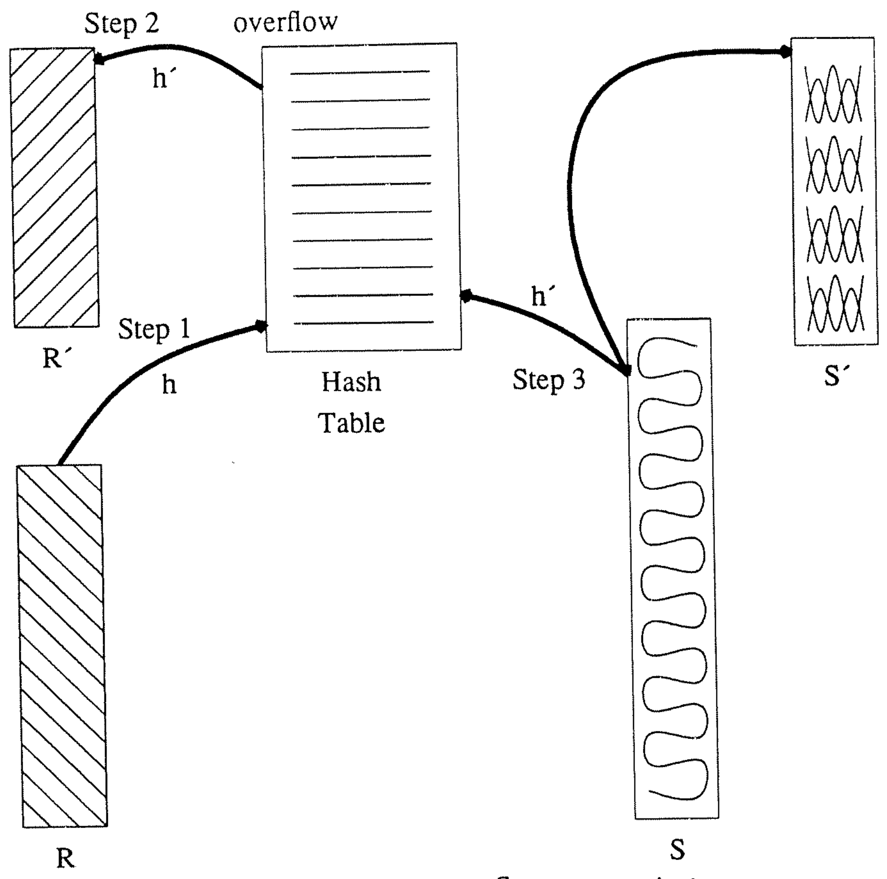
To increase intra-query parallelism, it would be possible to partition (or sort) both relations concurrently. However, this could cause performance problems due to disk head and network interface contention. Another issue is that bit filters must be created from the entire inner relation before they can be applied to the outer relation; we therefore chose to partition the relations serially.

3.3.2. Simple Hash-Join

A centralized version of the Simple hash-join [DEWI84] operates as follows. First, the smaller joining relation, R, is read from disk and staged into 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. When the number of tuples in R exceeds the size of the hash table, memory overflow occurs. Figure 3.2 depicts the steps taken in order to handle this overflow. In step 1, relation R is used to build the hash table. When the hash table space is exceeded, the join operator creates a new file R´ and streams tuples to this file based on a new function, h´, until the tuples in R are distributed between R´ and the hash table (step 2)[4]. The query scheduler then passes the function h´ to the operator producing the tuples of S, the outer relation. In step 3, 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. We are now left with the task of joining the overflow partitions R´ and S´. Since R´ may also exceed the capacity of the hash table, the same process continues until no new overflow partitions are created, at which time the join will have been fully computed.

To parallelize this algorithm we inserted a split table in step 1 which routes tuples (via hashing) to their appropriate joining site. Of course, hash table overflow is now possible at **any** (or all) of these join sites. Overflow processing is still done, though, as described in step 2

[4]An example h´ function is: join attribute value > 60,000.



Simple hash-join overflow processing.
Figure 3.2

of the centralized algorithm. In fact, each join site that overflows has its own locally defined h´ and its own associated overflow file R´. Although each overflow file is stored entirely on a single disk (i.e., not horizontally partitioned) which may or may not be a local disk, different overflow files are assigned to different disks. Although it would have been possible to horizontally partition each overflow file across all nodes with disks, if one assumes that the R tuples are uniformly distributed across the join nodes, all nodes should overflow to about the same degree. Hence, the final result will be as if the aggregate overflow partition was horizontally partitioned across the disk sites. For step 3 of Figure 3.2, the split table used in step 1 to route tuples to their appropriate joining sites is augmented with the appropriate h´ functions. When relation S is passed through this split table, tuples will be routed to either one of the joining sites for immediate joining or directly to the S´ overflow files for temporary storage. As with the centralized algorithm, the overflow partitions R´ and S´ are recursively joined until no overflow occurs on any of the joining sites. Finally, it should be noted that there is no constraint that the processors used for executing the join operator must have disks attached to them.

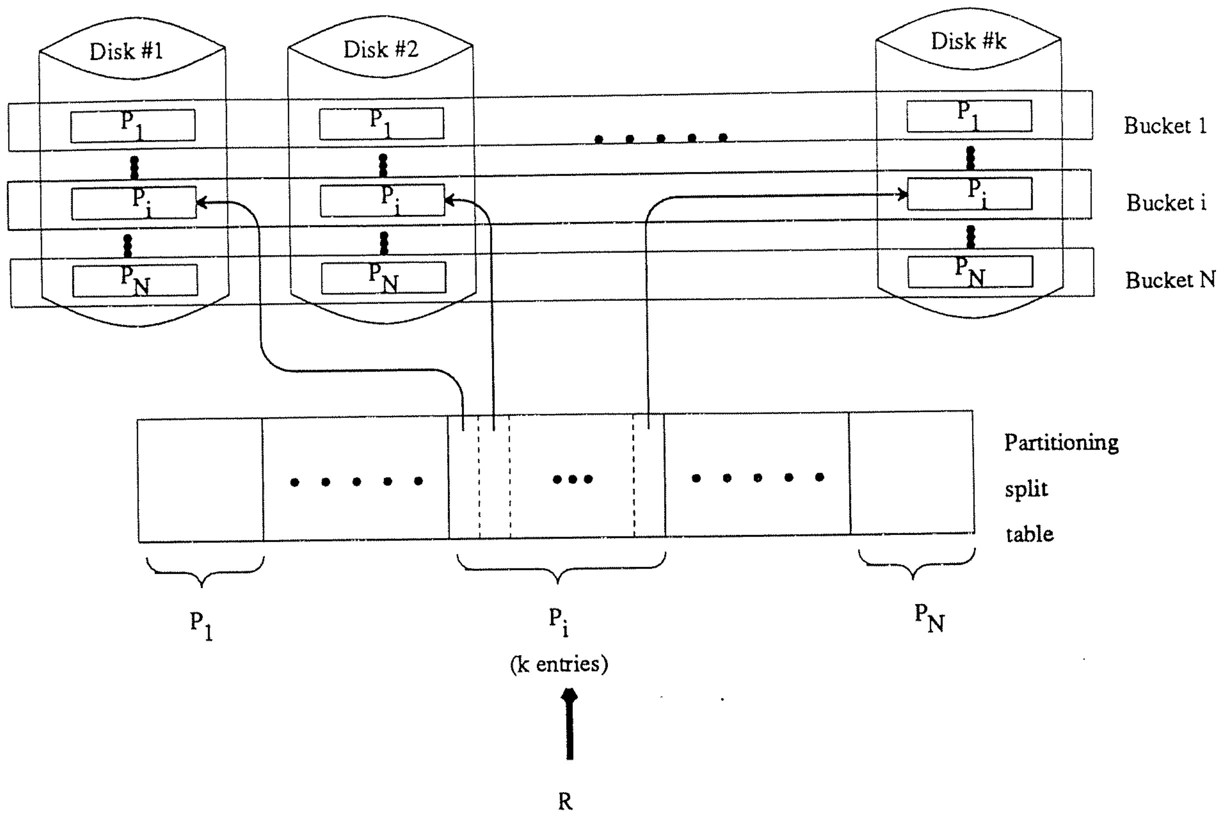
The Simple hash-join algorithm is also used as the overflow resolution method for our parallel implementations of the Grace and Hybrid algorithms.

3.3.3. Grace Hash-Join

A centralized Grace join algorithm [KITS83] works in three phases. In the first phase, the algorithm partitions relation R into N disk buckets by hashing on the join attribute of each tuple in R. In phase 2, relation S is partitioned into N buckets using the same hash function. In the final phase, the algorithm joins the respective matching buckets from 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 of the processors used to actually affect the join of two buckets. If the buckets are much smaller than main memory, several will be combined during the third phase to form more optimally sized join buckets (referred to as bucket tuning in [KITS83]).

The Grace algorithm differs fundamentally from the sort-merge and Simple hash-join algorithms in that data partitioning occurs at two different stages - during **bucket-forming** and during **bucket-joining**. Parallelizing the algorithm thus must address both these data partitioning stages. To insure maximum utilization of available I/O bandwidth during the bucket-joining stage, each bucket is partitioned across all available disk drives. A **partitioning split table**, as shown in Figure 3.3, is used for this task. When it is time to join the i-th bucket of R with the i-th bucket of S, the tuples from the i-th bucket in R are distributed to the available joining processors using a **joining split table** (which will contain one entry for each processor used to effect the join). As tuples arrive at a site they are stored in in-memory hash tables. Tuples from bucket i of relation S are then distributed using the same joining split table and, as tuples arrive at a processor, used to probe the hash table for matches.



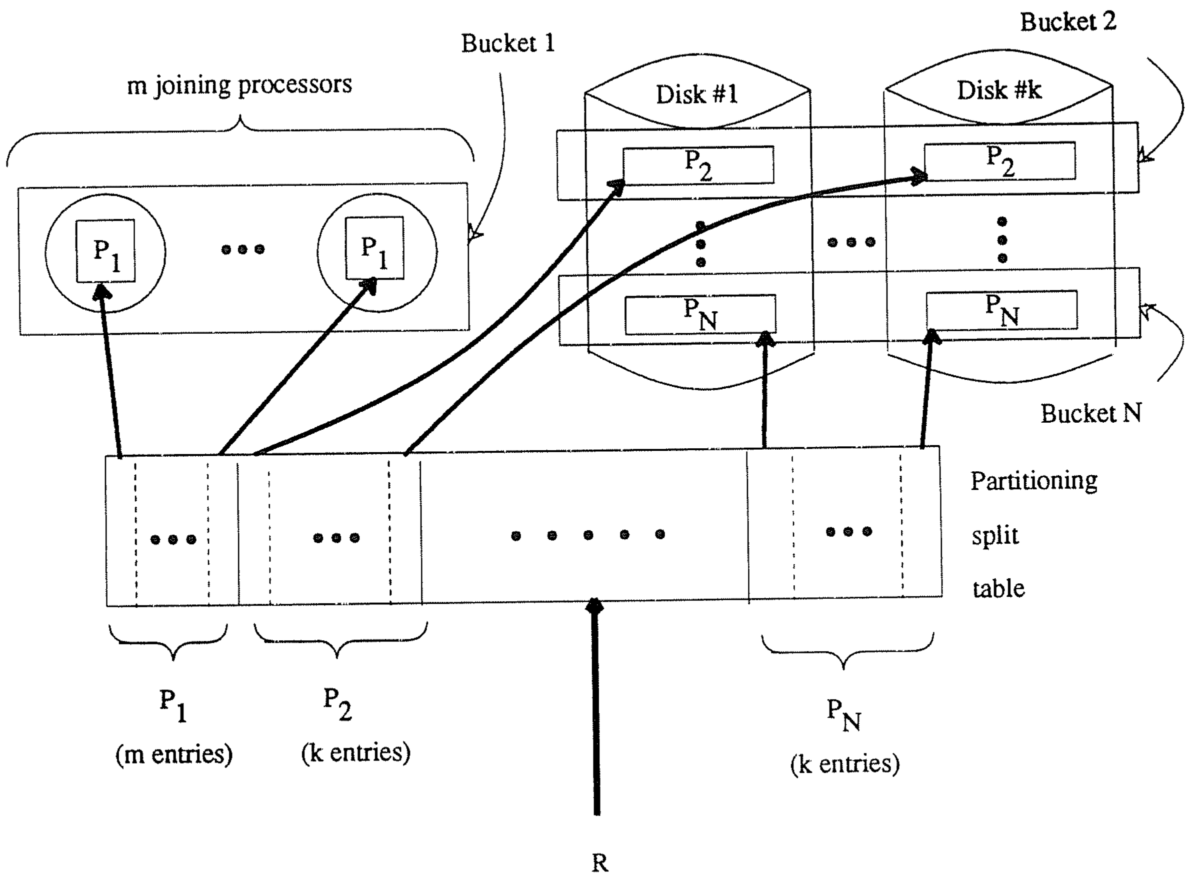
Grace Partitioning of R into N Logical Buckets
Figure 3.3

The bucket-forming phase is completely separated from the bucket-joining phase under the Grace join algorithm. This separation of phases forces the Grace algorithm to write both the joining relations back to disk before beginning the join stage of the algorithm.

Currently, our parallel implementation of the Grace join algorithm does not use bucket tuning. Instead, the number of buckets is determined by the query optimizer, which tries to ensure that the size of each bucket is just less than the total amount of main-memory of the joining processors.

3.3.4. Hybrid Hash-Join

A centralized Hybrid hash-join algorithm [DEWI84] also operates in three phases. In the first phase, the algorithm uses a hash function to partition the inner relation, R, into N buckets. The tuples of the first bucket are used to build an in-memory hash table while the remaining N-1 buckets are stored in temporary files. A good hash function produces just enough buckets to ensure that each bucket of tuples will be small enough to fit entirely in main memory. During the second phase, relation S is partitioned using the hash function from step 1. Again, the last N-1 buckets are stored in temporary files while the tuples in the first bucket are used to immediately probe the in-memory hash table built during the first phase. During the third phase, the algorithm joins the remaining N-1 buckets from relation R with their respective buckets from relation S. The join is thus broken up into a series of smaller joins, each of which hopefully can be computed without experiencing join overflow. As with the Grace join algorithm, the size of the smaller relation determines the number of buckets; this calculation is independent of the size of the larger relation. Whereas the Grace join



Partitioning of R into N logical buckets for Hybrid hash-join.
Figure 3.4

algorithm uses additional memory during the bucket-forming phase in order to produce extra buckets, Hybrid exploits this additional memory to immediately begin joining the first two buckets.

Our parallel version of the Hybrid hash join algorithm is similar to the centralized algorithm described above. A **partitioning split table** first separates the joining relations into N logical buckets. The number of buckets is chosen such that the tuples corresponding to each bucket will fit in the **aggregate** memory of the joining processors. The N-1 buckets intended for temporary storage on disk are each partitioned across all available disk sites as with the Grace algorithm. Likewise, a **joining split table** will be used to route tuples to their respective joining processor (these processors do not necessarily have attached disks), thus parallelizing the joining phase. Furthermore, the partitioning of the inner relation, R, into buckets is overlapped with the insertion of tuples from the first bucket of R into memory-resident hash tables at each of the join nodes. In addition, the partitioning of the outer relation, S, into buckets is overlapped with the joining of the first bucket of S with the first bucket of R. This requires that the partitioning split table for R and S be enhanced with the joining split table, as tuples in the first bucket must be sent to those processors being used to effect the join. Of course, when the remaining N-1 buckets are joined, only the joining split table will be needed. Figure 3.4 depicts relation R being partitioned into N buckets across k disk sites where the first bucket is to be joined on m processors.

3.4. Experimental Results

We tested the various parallel algorithms under several conditions. First, the performance of the algorithms are studied when the amount of memory available for joining is varied. Next, we compare the effects of bit vector filtering on the different algorithms. Finally, the impact of non-uniformly distributed join attribute values on the performance of each of the algorithms is studied.

The benchmark relations are based on the Wisconsin Benchmark [BITT83]. Each relation consists of thirteen 4-byte integer values and three 52-byte string attributes. Except where noted otherwise, hashing on the unique1 attribute (an integer field) was used to determine each tuple's destination site during loading of the database. The benchmark join query is joinABprime, which joins a 1,000,000 tuple relation (approximately 208 megabytes) with a 100,000 tuple relation (approximately 20 megabytes) and produces a 100,000 tuple result relation (over 40 megabytes). 8 kilobyte disk pages were used in all experiments. The hardware environment consists of 30 processors with disks, with one additional processor reserved for query scheduling and global deadlock detection. All relations, including output relations, are declustered across all 30 processors.

Join performance (for each of these parallel join algorithms) is sensitive to the amount of available memory relative to the size of the joining relations. In designing the set of experiments described below, the first decision to make was how to capture this aspect of the performance of the different algorithms. One approach was to keep the amount of available memory constant while varying the size of the two relations being joined. The other choice was to keep the size of the joining relations constant while (artificially) varying the amount of available memory. We rejected the first choice because increasing the size of the joining relations has the side-effect of increasing the number of I/O's needed to execute the query.

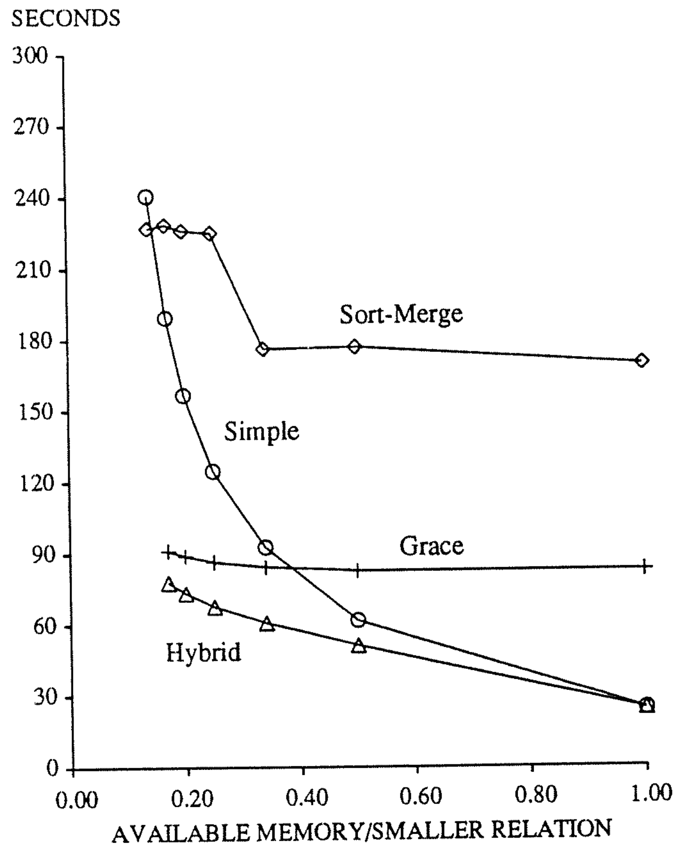
Our experiments analyze join performance over a wide range of memory availability. All results are graphed with the x-axis representing the ratio of available memory to the size of the smaller relation. Note that available memory is the sum of the memory that is available for computing the join on the joining processors. For the hash-based join algorithms, this memory is used to construct an in-memory hash table, while with the sort-merge join algorithm, this memory is used for both sorting and merging. In the case of the sort-merge join algorithm, we simply reduced the amount of sort/merge space and for the Simple-hash join algorithm we reduced the amount of hash table space, accordingly. For the Grace and Hybrid algorithms, however, a data point at 0.5 relative memory availability, for instance, equates to a two-bucket join. Likewise, a data point at 0.20 was computed using 5 buckets. Thus, neither Grace or Hybrid joins ever experienced hash table overflow. At the end of Section 3.4.1 we analyze the performance of the Grace and Hybrid join algorithms at data points not corresponding to an integral number of buckets.

3.4.1. Parallel Join Algorithm Performance

Figure 3.5 displays the execution times of the joinABprime query as a function of the amount of memory available relative to the size of the inner relation. Several points should be made about this graph. First, when the smaller relation fits entirely in memory (at 1.0), the Hybrid and Simple algorithms have, as expected, identical execution times. One might expect, however, that at the 50% available memory data point Simple hash would also be equal to Hybrid hash because their respective I/O behavior is identical (both algorithms write approximately one-half of the joining relations to disk). Simple hash is slower, however, because it first sends all tuples to the join sites for processing (where it will turn out that 1/2 of the tuples belong to the overflow partition - see Section 3.3.2) while the Hybrid algorithm writes the tuples belonging to the second bucket directly to disk.

As expected, Grace joins are relatively insensitive to decreasing the amount of available memory, but Hybrid is very sensitive, especially when large amounts of memory are available. This occurs because the Grace algorithm is not using the extra memory for joining and hence decreasing memory simply increases the number of buckets, each of which incurs a small scheduling overhead. However, for Hybrid, decreasing the amount of memory available from a ratio of 1.0 to 0.5 forces the algorithm to stage half of each joining relation back to disk. Furthermore, note that the response time for the Hybrid algorithm approaches that of the Grace algorithm as memory is reduced. Hybrid derives its benefits from exploiting extra memory and, when this memory is reduced, the relative performance of the algorithm degrades.

The Hybrid algorithm dominates over the entire available memory range. Between the memory ratios of 0.5 and 1.0, Simple hash outperforms Grace and sort-merge because a decreasing fraction of the larger joining relation is written back to disk. However, as memory availability decreases, Simple hash degrades rapidly because it repeatedly reads and writes the same data. While the performance of the sort-merge algorithm is relatively stable, it is



Parallel Join Algorithm Performance
30 Processors with Disks
Figure 3.5

dominated by the Hybrid and Grace algorithms over the entire memory range. The upward steps in the response time curves for sort-merge result from the cost of the additional merging passes that are required to sort the larger source relation as memory is reduced. However, it should not be concluded from Figure 3.5 that Simple hash-join will outperform sort-merge join for all queries and under most situations of limited memory availability. If the curves were extended to use even less memory, the sort-merge algorithm would maintain near level performance while Simple hash-join will continue its rapid performance degradation. Also, if the test query joined two equi-size relations, the differences between all the algorithms would diminish.

It is important to point out that the trends observed in these graphs and their general shape are almost identical to the analytical results reported in [DEWI84] and the experimental results in [DEWI85] for **single-processor** versions of the same algorithms. There are several reasons why we find this similarity encouraging. First, it demonstrates that each of the algorithms parallelizes well. Second, it serves to verify that our parallel implementation of each algorithm was done in a fair and consistent fashion.

It should also be noted that these results are almost identical to those reported in [SCHN89] when Gamma was using an older hardware platform (see Section 3.2.1). However, there are a few important differences. When using the older version of Gamma, it was found that exploiting knowledge about the manner in which the relations were declustered could result in a significant performance difference. If the two relations being joined were both hash-partitioned on their joining attributes, the partitioning phase of each of the parallel algorithms described in the previous section could be eliminated. Rather than special-case this situation, Gamma relied on the operating system to "short-circuit" packets between two processes on the same machines and the software was designed to maximize the extent to which tuples were mapped to hash-join buckets on the same processor. The benefits were significant in this environment because the time to reliably transmit a 2 Kbyte message between processes on different processors was 12.4 milliseconds compared to only 4.4 milliseconds for a message between processes on the same processor [GERB86]. In the current hardware configuration this special case is not significant because message costs are a small fraction of the total cost of computing the query and because it costs at most an additional millisecond to send a message to a process on a different processor.

Another difference between the results in Figure 3.5 and those reported in [SCHN89] is a greater degradation of performance of the sort-merge algorithm with respect to the other algorithms, especially Simple hash-join. Part of this effect is due to the larger relations used in these current tests, but the most important factor is the hardware cache that is installed on each of the disk controllers in the current machine. With the addition of the disk cache, scanning a file sequentially became twice as fast (in single-user mode). With the Grace, Hybrid, and Simple join algorithms, whenever a file is read it is accessed sequentially. However, with sort-merge join, disk I/O is basically random during both run generation and merging because any request made to a sector not in the hardware cache invalidates the cache.

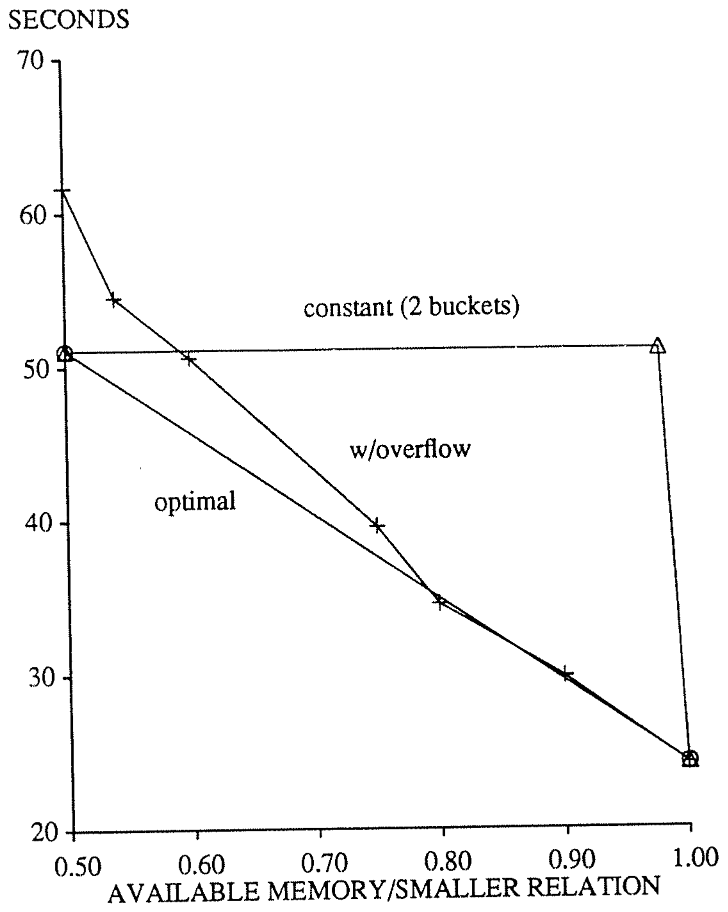
Grace and Hybrid Performance over Intermediate points

As stated in the introduction of this section, we chose to plot response times for the Hybrid and Grace algorithms when the available memory ratio corresponded to an integral number of buckets. Thus, instead of the straight lines connecting the plotted points in Figure 3.5, the curves should actually be step functions. Alternatively, we could have chosen to let the algorithms overflow at non-integral memory availabilities and used the Simple hash-join algorithm to process the overflow. This decision represents the tradeoff between being pessimistic and always electing to run with one additional bucket, or being optimistic and hoping the overflow mechanism is cheaper than using an extra bucket. Clearly, with the Grace algorithm the pessimistic choice is the best choice since extra buckets are inexpensive. However, the tradeoffs are not as obvious for the Hybrid algorithm.

Figure 3.6 presents a more detailed examination of the performance of the Hybrid join algorithm between the memory ratios of 0.5 and 1.0[5]. Performance is optimal at the memory ratios of 0.5 and 1.0 because memory is fully utilized and no unnecessary disk I/O is performed. The line connecting these points thus represents the optimal achievable performance if perfect partitioning of the joining relations was possible. The horizontal line reflects the pessimistic option of executing with one extra bucket. Explaining the performance of the Hybrid algorithm with overflow requires more details of how the Simple algorithm processes overflows.

When tuples of R (the inner relation) arrive at a join site they are inserted into a hash table based on the application of the hash function, and a histogram based on the application of the hash function to the tuple's join attribute value is updated. This histogram records the number of tuples between successive ranges of hash values. When the capacity of the hash table is exceeded, a procedure is invoked to clear some number of tuples from the hash table and write them to an overflow file (this is the h´ function described earlier). We currently try to

[5]This snapshot of performance will also show the expected behavior between the other plotted points on the graphs.



Performance over Intermediate Points
Figure 3.6

clear 10% of the hash table memory space when overflow is detected. This is accomplished by examining the histogram of hash values. For example, the histogram may show us that writing all tuples with hash values above 90,000 to the overflow file will free up 10% of memory. Given this knowledge, the tuples in the hash table are examined and all qualifying tuples are written to the overflow file. As subsequent tuples arrive at the join site, they are first compared to the present cutoff value. If their hash values are above the cutoff mark they are written directly to the overflow file; otherwise, they are inserted into the hash table. Notice that the hash table could again overflow if the heuristic of clearing 10% of memory turns out to be insufficient. In this case, an additional 10% of the tuples are removed from the hash table. The 10% heuristic may seem overly optimistic at first but notice that each successive application of the heuristic really increases the percentage of incoming tuples being written to the overflow file. For example, after the second invocation of the heuristic, incoming tuples will only have about an 80% chance of even being directed to the hash table; the other 20% are immediately sent to the overflow file. Thus, more tuples will need to be examined before the now available 10% of free memory is filled than was required for the previous invocation of the heuristic.

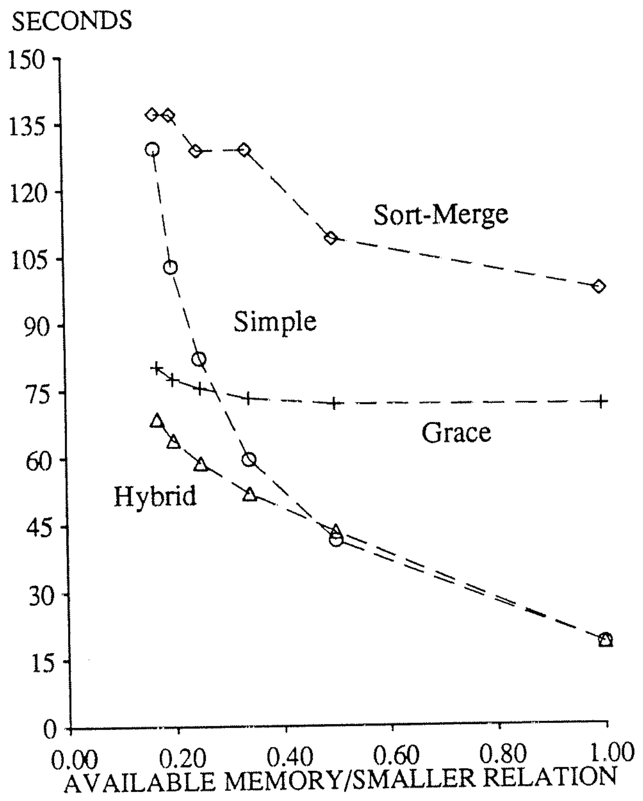
The shape of the response time curves for the Hybrid algorithm in Figure 3.6 illustrates how these heuristics impact performance. The overflow curve becomes worse than that of Hybrid with two buckets because of the CPU overhead required to repeatedly search the hash table and also because the heuristic forces more than 50% of the tuples to be written to the overflow file (hence incurring additional disk I/Os). Also, there is the cost of sending tuples to the join site, finding that they now belong to the overflow partition, and re-sending them to the process that will write them to disk (even though the transmission of the overflow tuples is short-circuited, the protocol processing costs are still incurred). These results show that a tradeoff exists between being pessimistic and increasing the number of buckets versus being optimistic and counting on Simple hash-join to resolve any memory overflow.

3.4.2. Multiprocessor Bit Vector Filtering

In this next set of experiments we studied the performance of the parallel join algorithms when bit filters [BABB79, VALD84] were used. The concept of bit filtering is simple. Initially an array of bits is set to zero. A hashing function is applied to the join attribute of each tuple processed in the first joining relation and the appropriate bit is set to one. The fully constructed bit filter is then passed to the second joining relation. The same hashing function is applied to the joining attribute of each tuple in the second joining relation and the appropriate bit in the bit filter is checked. If the bit is zero, there is no possibility that the tuple can participate in the join and it can safely be eliminated from further processing.

With our parallel sort-merge join algorithm, a bit filter is built at each disk site as the inner (smaller) relation is partitioned across the network and stored in temporary files. With the hash-based algorithms, a bit filter is built at each of the join sites as tuples from the inner relation are being inserted into the hash tables. The bit filter is then used to eliminate non-joining tuples from the outer relation.

In our implementation, bit filtering of tuples is only applied during the joining phase. With the Simple hash-join algorithm this means that as the number of overflows increases, the opportunities for filtering out non-joining tuples also increases (because each overflow resolution is treated as a separate join). With Grace and Hybrid joins, each bucket-join is treated as a separate join. Thus, a new bit filter will be built as a part of processing each bucket. Since each join uses the same size bit filter, increasing the number of buckets (or overflows) increases the effective size of the aggregate bit filter across the entire join operation.



Effects of Bit Filtering
Figure 3.7

Because of implementation constraints we did not use the joinABprime query as described earlier for these tests. Instead, we modified this query to add a selection predicate on the smaller joining relation which reduced its size to half its previous value. We were forced into modifying the query, because in the current implementation, only a single 8 Kbyte packet which is shared among all the joining processors is reserved for bit filtering. With the regular joinABprime query, each site receives 3,333 building tuples. However, with a total bit filter size of 8 Kbytes, only 2,184 bits are available for the bit filter at each site. Obviously, the bit filter would have been useless in this environment because most of the bits would have been set to one. We would have preferred to use the regular joinABprime query and simply increase the size of the bit filter, but the required implementation effort was prohibitive.

Figure 3.7 shows the results with bit filtering applied. Notice that the relative positions of the algorithms have not changed, but the execution times have dropped in comparison to the results shown in Figure 3.5. The performance improvements from bit filtering for each individual algorithm are shown in Figures 3.8 through 3.11.

Figures 3.8 and 3.9 show the effects of using bit filters with the Hybrid and Grace join algorithms, respectively. The improvement in performance with bit filtering is limited to about 5% for both of these algorithms. This is somewhat disappointing, but it can easily be explained. First, the bit filters are filled to 77% occupancy and thus not all of the non-joining tuples are eliminated. And second, by using bit filtering only during bucket-joining, the bit filters only reduce unnecessary network traffic and CPU cycles for "probing". With the relatively fast communication provided by the hypercube, these benefits are necessarily limited. If bit filtering was extended to the bucket-forming stage, the speedups would be much more impressive because substantial amounts of disk I/O would also be eliminated.

The results for the Simple hash-join algorithm shown in Figure 3.10 support the conclusion that, in the current hardware environment, reducing disk traffic results in significantly better performance. Since the Simple algorithm "loops" over its joining relations, applying filtering techniques eliminates the writing and reading of useless tuples potentially many

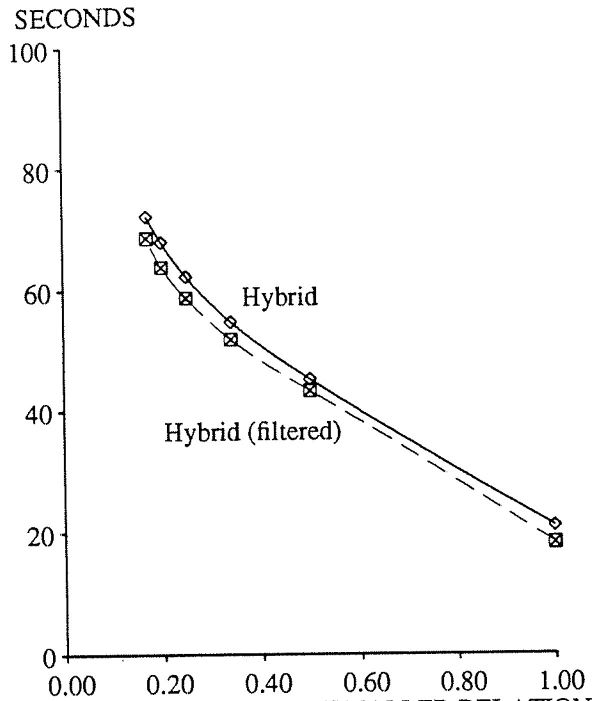


Figure 3.8

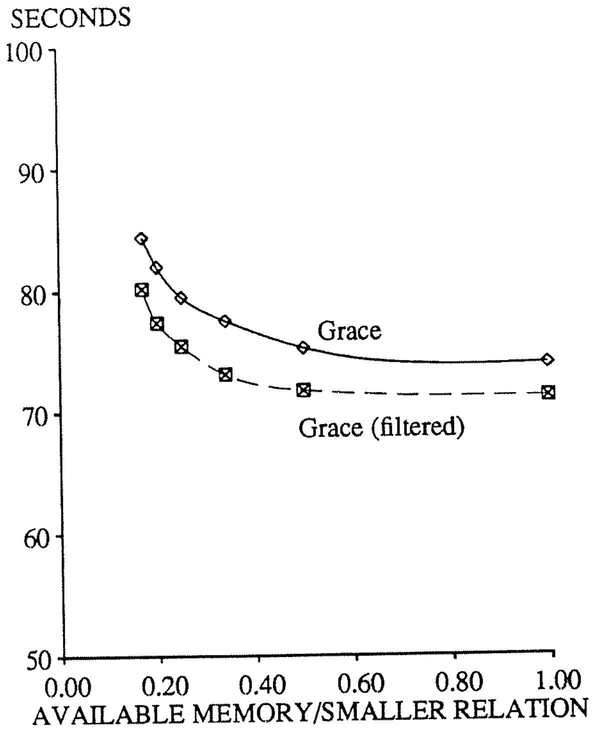


Figure 3.9

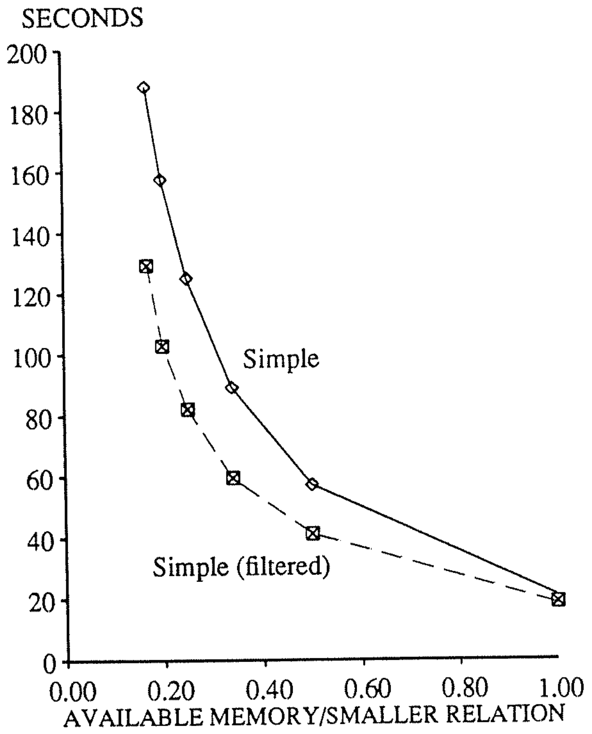


Figure 3.10

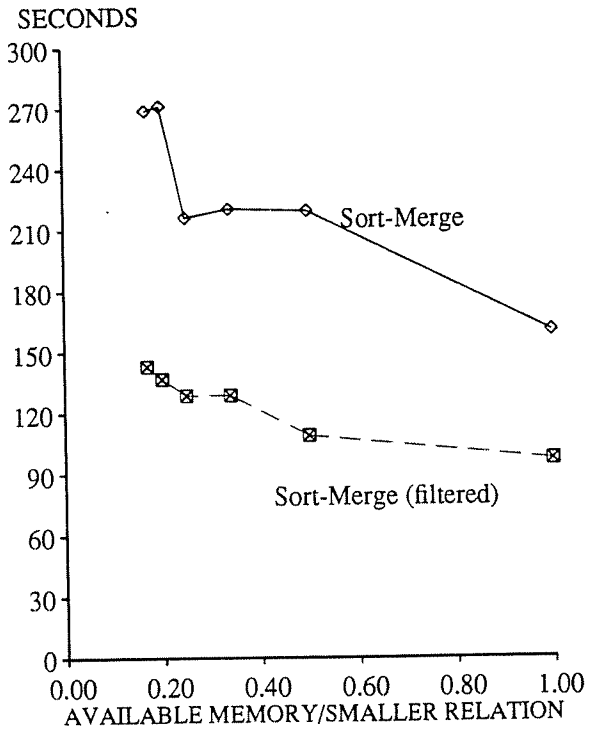


Figure 3.11

times. The performance improvement for this algorithm is approximately 30% whenever overflow processing is required. Using a larger bit filter would provide even better performance.

The performance of the sort-merge join algorithm also benefits significantly from the use of bit filters, as shown in Figure 3.11. Tuples of the larger relation that are eliminated by the filter do not need to be written to disk, sorted, and later read during merging. Obviously, using a larger bit filter would further improve the performance of this algorithm.

These results should also be contrasted with the more optimistic results reported in [SCHN89] when Gamma was running on a collection of VAX 750's. In this older Gamma configuration, sending a message from node to node was much more expensive (12.4 ms. versus 4.6 ms.) and hence reducing network traffic was more beneficial. Also, due to experimental conditions and implementation constraints, the bit filters in the older configuration were only 66% occupied as opposed to 77% in the current configuration.

3.4.3. Non-Uniform Data Distributions

In this set of experiments we wanted to analyze the performance of the four parallel join algorithms in the presence of non-uniformly distributed join attribute values. In order to isolate the effects of non-uniform data distributions, we varied the distribution of the two join attribute values independently. Figure 3.12 shows the four possible combinations that comprised our experimental design space. The key we are using in this figure is XY, where X (Y) represents the attribute value distribution of the inner (outer) relation; U = Uniform distribution and N = Non-uniform distribution.

The time to compute the joinABprime query on Gamma was used as the performance metric for this set of experiments. Recall that this query joins a 1,000,000 tuple relation (208 megabytes) with a 100,000 tuple relation (20 megabytes) to produce a 100,000 tuple output relation (over 40 megabytes). The 100,000 tuple relation is the inner (building) relation and the 1,000,000 tuple relation is the outer (probing) relation. For the non-uniform distribution we chose the **normal** distribution with a mean of 500,000 and a standard deviation of 8,333. These parameters resulted in a highly skewed distribution of values over the domain 0-

		Outer Relation	
		uniform	non-uniform
Inner Relation	uniform	UU	UN
	non-uniform	NU	NN

Figure 3.12

999,999. In fact, 33,333 tuples had join attribute values between 500,009 and 500,707. However, no single attribute value occurred in more than 79 tuples. The 100,000 tuple relation was created by randomly selecting 100,000 tuples from the 1,000,000 tuple relation. Thus the 100,000 tuple relation's primary key had values uniformly distributed from 0 to 999,999. Also, the normally distributed attribute had the same distribution characteristics for the 100,000 tuple relation as it did for the 1,000,000 tuple relation.

To ensure that each processor did the same amount of work during the initial scan of the two relations being joined, we distributed each of the relations on their join attribute by using the round-robin partitioning strategy provided by Gamma. This resulted in an equal number of tuples on each of the thirty sites.

As in the previous experiments we used the amount of memory relative to the size of the smaller joining relation as a basis for comparing the join algorithms. Remember, though, that the amount of memory at each joining processor is enough to hold its share of tuples from the inner relation **only** if the tuples are distributed evenly across the joining processors. Thirty processors with disks were used for these experiments.

Table 3.2 presents the results for each of the parallel join algorithms for the cases of 100% and 17% memory availability. The UU joins produced a result relation of 100,000 tuples as did the NU joins. The UN joins produced 99,406 result tuples. Results for the NN joins are not presented because the query produced over three million tuples. We could find no way of normalizing these NN results to meaningfully compare them with the results from the other three join types.

We begin by analyzing the effects of using non-uniformly distributed join attribute values for the inner (building) relation by comparing the NU results with the UU results. For the hash-join algorithms two major factors result from "building" with a non-uniformly distributed attribute. First, tuples will not generally be distributed equally among the physical bucket fragments. Second, chains of tuples will form in the hash tables due to duplicate attribute values. The first factor is significant because the aggregate memory allocated for the join was sufficient to hold the tuples from the building relation **only** if the tuples were uniformly spread across the joining processors. With the normally distributed join attribute values used in our experiments, the hash function could not distribute the relation uniformly and memory overflow resulted. However, the hash function did not perform that poorly and only one pass of the overflow mechanism was necessary to resolve the overflow of each join bucket. The second factor, chains of tuples forming in the hash tables, also materialized with our normally distributed join attribute values. In fact, chains of 3.3 tuples were found on the average, with a maximum hash chain length of 16.

Algorithm	100% memory			17% memory		
	UU	UN	NU	UU	UN	NU
Hybrid	26.60	27.55	32.98	77.33	79.88	98.16
Grace	78.73	80.50	81.94	86.03	90.32	91.80
Sort-Merge	199.50	203.09	185.89	249.89	256.73	233.54
Simple	26.59	27.51	33.01	188.70	195.96	194.86

Join results with non-uniform join attribute value distributions.
(All response times are in seconds)
Table 3.2

Table 3.2 shows that NU is indeed slower than UU for the hash-based join algorithms. Since the philosophy behind the Grace join algorithm is to use many small buckets to prevent buckets from overflowing, we executed this algorithm using one additional bucket so that memory overflow would not occur. However, performance is still degraded due to the non-uniform size of the buckets. At 100% memory availability, the Hybrid algorithm processes the overflow fairly efficiently. However, as memory is reduced, each bucket-join in the Hybrid algorithm experiences overflow. As shown by the 17% memory availability results, the cost of overflow resolution becomes more significant when memory is limited.

Why, though, does the sort-merge join algorithm run NU faster than UU and UN? The hashing function will distribute tuples unevenly across the joining processors exactly as it did for the hash-join algorithms, thereby requiring a subset of the sites to store to disk, sort, and subsequently read more tuples than others. One would expect this to have a negative impact on performance. The explanation has to do with recognizing how the merge phase of the join works. Since the join attribute of the inner relation is so skewed (the maximum join attribute value is only 541,641) the merge phase does not need to read all of the outer (1000K) relation. In this case, the semantic knowledge inherent in the sort-order of the attributes allowed the merge process to determine the join was fully computed **before** all the joining tuples were read. A similar effect occurs for UN, but it is not significant in this case because only part of the inner (100K) relation can be skipped from reading; all of the 1000K outer relation must still be read.

The effects of having the outer (probing) relation's join attribute non-uniformly distributed can be determined by comparing the UN results with the UU results. Again, having a non-uniform data distribution for the outer join attribute will result in unequal numbers of tuples being distributed to the joining processors. At 100% memory availability, the differences are small. However, the differences increase slightly as memory is reduced. This occurs because as memory is reduced the joining relations are first divided up into several disjoint buckets. Because the outer join attribute is non-uniformly distributed the outer relation will not be uniformly divided across the buckets. This will result in some of the disk sites requiring additional disk I/Os during bucket forming **and** during bucket joining. Thus, as the number of buckets increases the more significant the effects of having non-uniformly distributed join attribute values become.

We feel these UN results are very encouraging for the Hybrid join algorithm. One can argue that many joins are done to re-establish relationships. These relationships are generally one-to-many and hence one join attribute will be a key. Since the relation on the "one" side is probably smaller, it will be the building relation in a hash-join algorithm. Thus, this case will result in a UN type of join which we have shown can be efficiently processed.

The results from applying bit filtering techniques are very similar to those of the previous experiments when the join attribute values were uniformly distributed, although the NU joins experienced slightly better improvements because the normally distributed attribute values resulted in more collisions in the bit filter. Thus, fewer bits were set in the filter and more outer (probing) tuples were eliminated.

3.5. Summary of Results

Several conclusions can be drawn from these experiments. First, for uniformly distributed join attribute values the parallel Hybrid algorithm appears to be the algorithm of choice because it dominates each of the other algorithms at all degrees of memory availability. Second, bit filtering should be used because it is cheap and can significantly reduce response times.

However, non-uniformly distributed join attribute values alter the relative performance of the parallel join algorithms. The performance of the hash-based join algorithms, Hybrid, Grace and Simple, degrades when the join attribute values of the inner, building relation are non-uniformly distributed. When the join attribute values of the building relation are **highly** skewed and the available memory relative to the size of the smaller relation is limited, a non-hash-based algorithm such as sort-merge or nested loops should be used. We find it very encouraging, though, that the Hybrid join algorithm still performs best when the joining attribute of the outer relation is non-uniformly distributed. We expect this type of join to be very common in the case of re-establishing one-to-many relationships.

CHAPTER 4

MULTI-WAY JOIN QUERY PROCESSING

4.1. Introduction

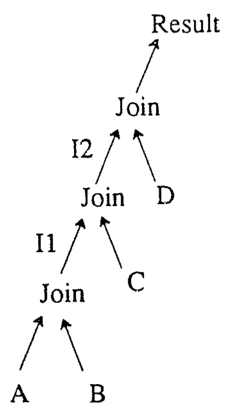
In this chapter, we address the problems and tradeoffs of processing multiple-join queries in a multiprocessor environment. We focus on the use of hash-based join algorithms because the results from Chapter 3 demonstrate that they dominate under most circumstances.

4.1.1. Degrees of Parallelism

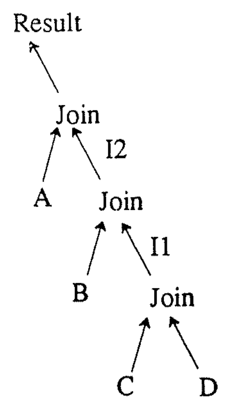
There are three possible ways of utilizing parallelism in a multiprocessor database machine. First, parallelism can be applied to each operator within a query. For example, ten processors can work in parallel to compute a single join or select operation. This form of parallelism is termed **intra-operator** parallelism and has been studied by previous researchers and was covered in depth in Chapter 3. 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 chapter, 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.

4.1.2. Query Tree Representations

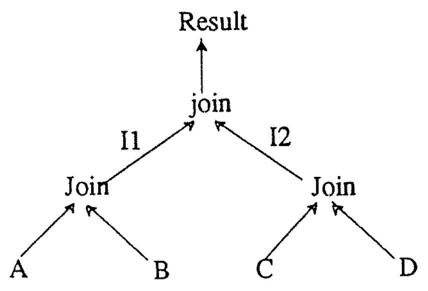
Instrumental to understanding how to process complex queries is understanding how query plans are generated. A query is compiled into a tree of operators and several different formats exist for structuring this tree of operators. As will be shown, the different formats offer different tradeoffs, both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



(((A * B) * C) * D)
Left-Deep Query Tree



(A * (B * (C * D)))
Right-Deep Query Tree



((A * B) * (C * D))
Bushy Query Tree

Figure 4.1

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 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 have different maximum memory requirements. This is important because the performance of hash-based join algorithms depends heavily on the amount of available memory, as was demonstrated in Chapter 3. 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. These two tree formats are symmetrical and differ only under the assumption that when a hash-based join algorithm is being used the left operand is used to build the hash table and the right operand is used to probe the hash table. Bushy trees 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 Figure 4.1 for examples of left-deep, right-deep, and bushy query trees for the query: $A * B * C * D$. (Note that the character * denotes the relational join operator.)

4.2. Tradeoffs of Alternative Query Tree Representation Strategies

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 memory resources) to more realistic situations where memory is limited.

A good way of comparing the tradeoffs between the alternative query tree formats is through the construction of **operator dependency graphs** for each optimization 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 for efficient pipelining. 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 others, 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.

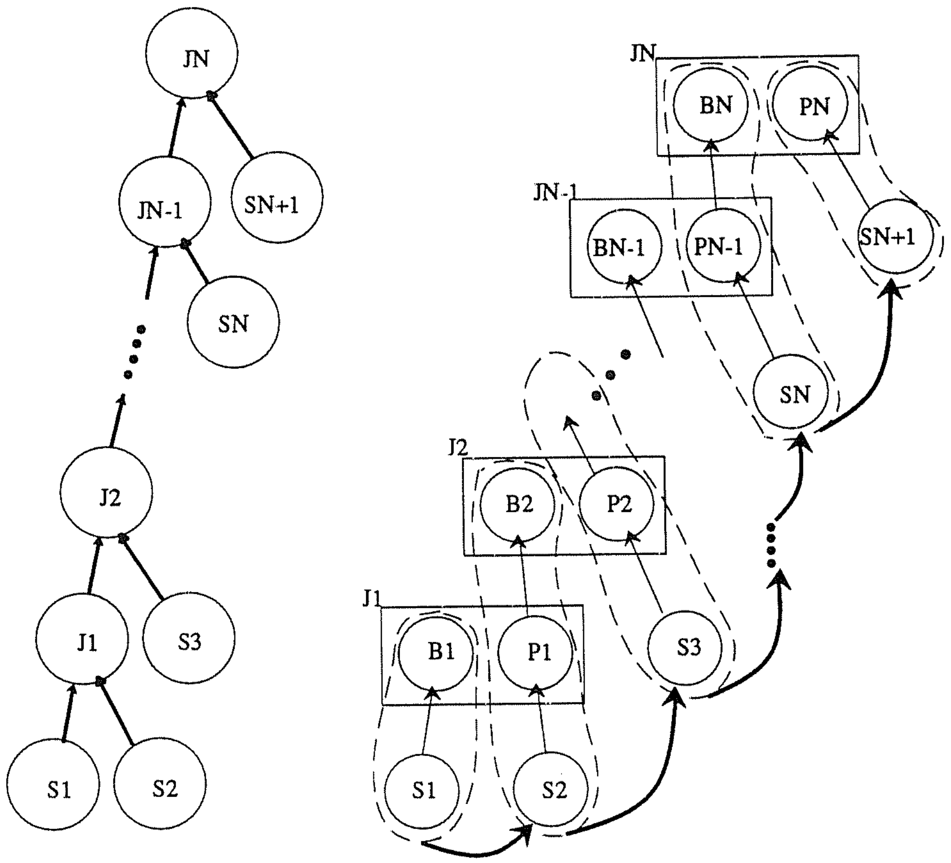
The operator dependency graphs presented in this section are based on the use of a hash-join algorithm as the join method. In this chapter, we consider two different parallel hash-join methods, Simple hash-join and Hybrid hash-join. Recall from the discussion in Chapter 3 that with hash-join algorithms, the computation of the join operation can be viewed as consisting of two phases. First, a hash table is constructed from tuples produced from the left input stream. In the second phase, tuples from the right input stream probe the hash table for matches to compute the join. Since the first operation must completely precede the second, 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. The base relations to be joined are represented in the operator dependency graphs as S^i, signifying the scan of relation i.

The reader should keep in mind that intra-operator parallelism issues are being ignored in this chapter. That is, when we discuss executing two operators concurrently, we have assumed implicitly that **each** operator will be computed using multiple processors as described in Chapter 3.

4.3. Left-Deep Query Trees

Figure 4.2 shows a generic N-join query represented as a left-deep query tree and its associated operator dependency graph. 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:

1) Scan S^1 - Build J^1
2) Scan S^2 - Probe J^1 - Build J^2
3) Scan S^3 - Probe J^2 - Build J^3
●
●
N) Scan S^N - Probe J^{N-1} - Build J^N
N+1) Scan S^{N+1} - Probe J^N



Left-Deep Query Tree and Dependency Graph
Figure 4.2

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 S^N, a hash table was constructed from the output of *Join*$^{N-1}$. When the Scan S^N is started, 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 S^N have probed the hash table, *Join*^{N} has been computed, and the join computation has been stored 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.

Although left-deep query trees require that only the hash tables corresponding to two adjacent join operators be memory resident at any point during the execution of any complex query, the relations staged into the hash tables are the result of intermediate join computations, and hence it is difficult to predict their size. Furthermore, even if the size of the intermediate relations can be accurately predicted, in a multi-user environment it can not 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, enough 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.

[GRAE89] 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 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 that 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 memory availability. An enhancement to this strategy would be to keep statistics on the size of the intermediate join computations as they are computed 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 transform the query tree 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.

Bit Filtering

It is simple to extend the bit filtering techniques described in Chapter 3 to apply to multi-way join queries. Conceptually, a distinct bit filter is maintained for each join operator in the query tree. As described before, each bit filter is constructed as tuples are "built" into the respective hash tables. Whenever a scan that produces probing tuples is about to be started, the bit filter associated with that join operation is used to eliminate non-joining tuples. For example, in Figure 4.2, the scan S^3 would use the filter that was produced from the join of S^1 and S^2.

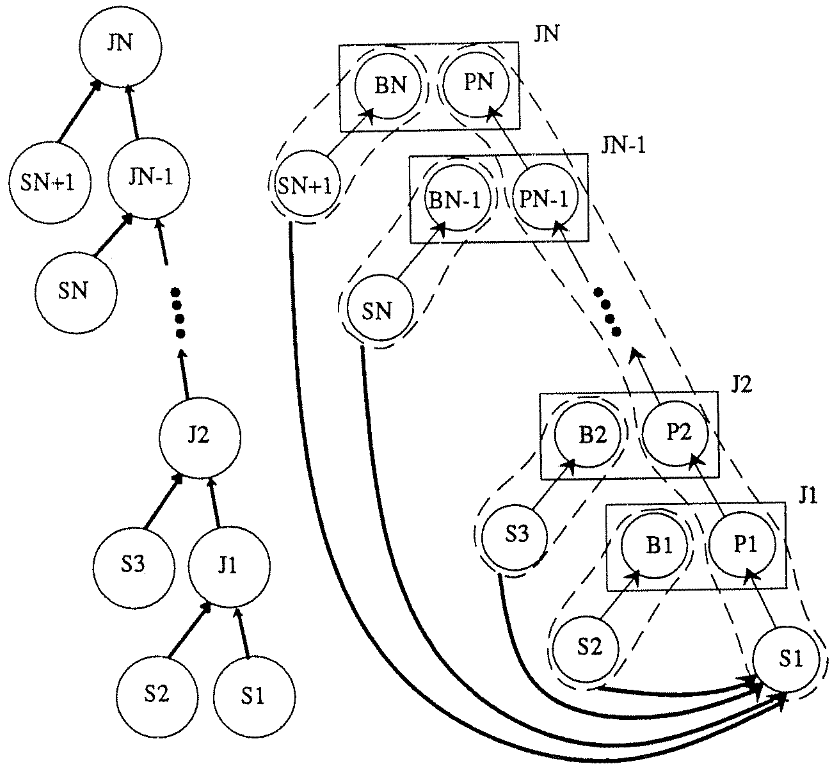
The benefit of bit filtering for left-deep trees is that probing tuples that cannot possibly produce output tuples for a join are eliminated as soon as possible, thus saving the overhead of sending them over the network to participate in the join. Note that it is not necessary to reserve memory for bit filters for all N joins. As was shown above in the schedule for executing left-deep trees, only two join operators are active at any point in time. Thus, only two bit filters need to be maintained at any time.

4.4. Right-Deep Query Trees

Figure 4.3 shows a generic right-deep query tree for an N-join query and its 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:

1) Scan S^2-Build J^1, Scan S^3-Build J^2, ...,Scan S^{N+1}-Build J^N
2) Scan S^1-Probe J^1-Probe J^2-...-Probe J^N

From this schedule it is obvious that all the scan operators but S^1, and all the build operators can be processed in parallel. After this phase has been completed, the scan S^1 is started and the resulting tuples will probe the first hash table. All output tuples will then per-



Right-Deep Query Tree and Dependency Graph
Figure 4.3

colate 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. Executing a right-deep query plan in this manner is termed **optimistic right-deep scheduling**.

There are two obvious questions concerning the optimistic right-deep scheduling algorithm as just described. First, can all the parallelism specified by the algorithm be effectively utilized? And second, what happens when memory is limited and all N hash tables cannot reside in memory simultaneously? Both of these questions are addressed below.

According to Step 1 in the execution plan, the scans S^2 through S^{N+1} should be performed concurrently to achieve maximum parallelism. The performance implications of this policy should be considered, though. If these scan operators access relations that are declustered over the identical set of storage sites, starting all the scans concurrently may be detrimental because of 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. 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 Chapter 5 we present experimental results that illustrate the performance implications of the data declustering strategy for right-deep query plans.

Dealing with limited memory for joining is expected to be a bigger problem with right-deep trees than with left-deep trees because more hash tables must co-reside in memory. Also, there is little opportunity for runtime query modifications since once the scan on S^1 is started the data flows through the query tree to completion. However, more accurate estimates of memory requirements can be obtained 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 operations. In the following sections, we propose several alternative algorithms for exploiting the potential performance advantages of right-deep query trees when memory is limited.

4.4.1. Static Right-Deep Scheduling

One strategy for dealing with limited memory (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 termed **static right-deep scheduling**. This scheduling strategy is equivalent to the optimistic right-deep scheduling strategy when the query tree does not need to be divided.

Bit filtering techniques can also be applied to right-deep query trees as well as left-deep query trees. With static right-deep scheduling, filtering can be performed in two different situations. First, as each building relation is read into memory, a separate bit filter is constructed. This filter is subsequently used to eliminate probing tuples. However, the presence of break points in the query tree provides another opportunity to exploit filtering. As each intermediate relation is being written to disk, a bit filter is constructed. This filter is then used to eliminate "building" tuples from the lowermost building relation when query processing continues. Thus, a form of *double-filtering* can be applied at each break point in the query tree. This is significant because filters applied to a building relation eliminate unnecessary memory consumption as well as unnecessary network traffic.

4.4.2. Dynamic Bottom-Up Scheduling

A more dynamic way of dealing with limited memory, called **dynamic bottom-up scheduling**, schedules the scans S^2 to S^{N+1} (see Figure 4.3) in a strictly bottom-up manner. The scan S^2 is first started and the resulting tuples are used to build a hash table for the join operator J^1. After this scan completes the memory manager is queried to check if enough

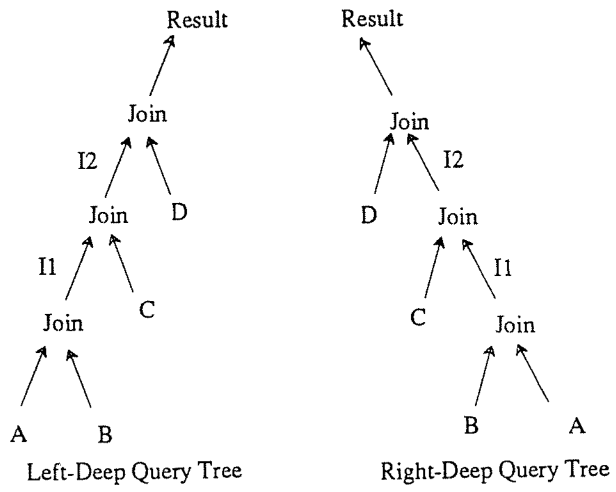


Figure 4.4

memory exists to stage the tuples expected as a result of the scan S^3. If sufficient space exists, scan S^3 is started. 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 S^1 to be initiated to start the process of probing the hash tables. However, when only the scans through S^i can be processed in this first pass, the scan S^1 is started but now the results of the join computation through join J^{i-1} are stored into a temporary file $S^{1'}$ (i.e., the query tree is broken). 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 started from the temporary file $S^{1'}$.

Dynamic bottom-up scheduling differs from static right-deep scheduling in the policy used for processing the "building" relations. With static right-deep scheduling, a **set** of building relations is read concurrently based on an estimate of the amount of available memory and selectivity estimates of the building relations. This is contrasted with the dynamic bottom-up strategy which reads building relations one at a time until available memory is exhausted. However, if only enough memory exists to hold a single building relation at a time, the two algorithms become identical.

Dynamic bottom-up scheduling is more robust to errors in selectivity estimation because it processes building relations in a purely sequential manner. If size estimates are grossly in error, only one relation will experience overflow as opposed to a set of relations for the static right-deep scheduling algorithm. However, by processing the building relations in a sequential order the dynamic bottom-up strategy sacrifices parallelism in scanning the building relations. As a compromise, it would be simple to **block** several relations together and treat each group as a single schedulable unit. This would result in an algorithm more optimistic than dynamic bottom-up scheduling while still more pessimistic than static right-deep scheduling.

The dynamic bottom-up scheduling algorithm has some very important properties when bit filtering techniques are applied [GERB90]. Consider, for example, the right-deep query tree shown in Figure 4.4. When relation B is staged into memory, a bit filter will be constructed on its joining attribute. Now, as relation C is being read into memory, the bit filter from relation B can be used to eliminate tuples from relation C (assuming there is a join clause between the two relations). A bit filter would likewise be formed from the tuples in C which survived the filtering process. This filter would then be applied to relation D. Since building tuples are being eliminated, the resulting bit filters are more restrictive than was the case with the static right-deep scheduling algorithm. Thus, when the bit filters are applied to the probing tuples, more tuples will be eliminated. In summary, bit filtering provides even more significant performance gains than seen with the optimistic and static right-deep scheduling algorithms because more non-joining building and probing tuples are eliminated. If the join selectivity is low, these savings could be substantial.

4.4.3. Right-Deep Hybrid Scheduling

Both the static right-deep and dynamic bottom-up scheduling strategies deal 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. Also, these algorithms require 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 to reduce memory requirements. This is what the Hybrid join algorithm attempts to do when it partitions its input relations into multiple buckets. When this technique is applied to complex right-deep query trees, several interesting results arise. We refer to the resulting algorithm as **right-deep hybrid scheduling** (RDHS).

The right-deep hybrid scheduling algorithm is shown in Figure 4.5. It takes as input a generic N-join right deep query tree as shown in Figure 4.3. Recall from Figure 4.3, that S^i represents the scan of the i^{th} base relation and J^i represents the i^{th} join operation. Additionally, each join operation is computed using NB(J^i) buckets. The k^{th} bucket of the i^{th} join is designated $J^i{}_{Bk}$. LC(J^i) and RC(J^i) return the operators that are the left child and right child inputs, respectively, of the i^{th} join operator. Finally, the function JOIN(x,y) specifies the join of two sets of tuples, x and y.

The RDHS algorithm is very simple. In the first four steps, each of the building relations is physically partitioned into buckets. The first bucket for each relation maps tuples to a memory resident hash table, while the remaining buckets map tuples to temporary files on disk. The intermediate relations that result from each join operation are also partitioned into buckets. This is a logical partitioning since at this point none of the intermediate relations have been materialized. The first bucket of each intermediate relation maps tuples to probe the hash table of the parent join operator while the remaining buckets map tuples to disk. The partitioning of the building and intermediate relations can be done in parallel.

In step 5, the probing base relation, S^1, is partitioned into buckets. Tuples that map to the first bucket are sent to probe the hash table built from the tuples in the first bucket of S^2. Output tuples will likewise flow up the tree. Tuples not mapping to the first bucket are staged to disk in the appropriate disk bucket.

In steps 6-10, the entire query tree is traversed from the bottom up, and the corresponding buckets for each join operator are joined.

By adjusting the number of buckets used for each join in a query tree, the RDHS algorithm can tune the amount of memory that it consumes. Assume that $|S_i|$ is the number of pages in relation i. The percentage of memory consumed relative to the amount of memory needed for the best case scenario (all joins use one bucket) is computed by the following formula.

$$M = \frac{\sum_{i=1}^{N} \left[\frac{1}{NB(J^i)} \times |LC(J^i)| \right]}{\sum_{i=1}^{N} |LC(J^i)|} \times 100\%$$

As an example of the RDHS algorithm, assume that each join in the right-deep query tree shown in Figure 4.4 will be divided into two buckets, with the first 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

```
Input:
  a "generic" $N$-join right-deep query tree (see Figure 4.3)

Algorithm:
1)  for $i$ = 1 to $N$
2)      phys_part ( $S^{i+1}$, NB( $J^i$ ))  (* partition building relations *)
3)      logic_part ($J^i$, NB( $J^i$ ))  (* partition intermediate relations *)
4)  end
    (* partition probing base relation *)
5)  phys_part ( $S^1$, NB( $J^1$ )) (* JOIN ($S^2{}_{B1}$, $S^1{}_{B1}$ )  *)

6)  for $j$ = 1 to $N$        (* for all joins in query *)
7)      for $k$ = 2 to NB( $J^j$)        (* for each bucket in join j *)
8)          JOIN ( $LC(J^j)_{Bk}$, $RC(J^j)_{Bk}$ )  (* join pair of buckets *)
9)      end
10) end
```

Right-Deep Hybrid Scheduling Algorithm
Figure 4.5

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.

With RDHS, some tuples can 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 or bushy query tree. If a user is submitting the query, the quicker feedback will result in a faster initial response time (even though the time to compute the entire result may be identical). And, when an application program is submitting the query, it may be very beneficial to provide the result data sooner and in a more "even" stream, as opposed to producing the entire result in one step, because the computation of the application can be overlapped with the processing of the join query.

Several questions arise regarding 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 relations B and D are 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 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 equivalent to the static right-deep scheduling algorithm. In Chapter 5, we explore under which conditions a particular scheduling strategy for a right-deep query tree will perform the best.

The technique of bit filtering can also be applied to the RDHS algorithm. When the building relations are being partitioned into buckets, a bit filter should be constructed for every bucket of every relation. These bit filters can then be applied to eliminate non-joining tuples as the probing tuples from the corresponding buckets are read from disk. Furthermore, the technique of double-filtering can be applied to each of the disk buckets. As tuples are written to the disk buckets of the intermediate relations, bit filters should again be constructed for each bucket. These filters can then be applied to eliminate non-joining tuples as the building tuples from the corresponding buckets are read from disk and staged into memory. In this instance, filtering saves both network traffic and hash table memory consumption.

4.4.4. A Taxonomy of Right-Deep Scheduling Algorithms

A simple taxonomy can be developed to classify the alternative algorithms for evaluating complex right-deep query trees based on how the algorithms deal with limited memory. Breaking a query tree between successive join operations can be viewed as *horizontally* partitioning the query. Conversely, dividing individual join operations into multiple buckets while still allowing data to flow throughout the query tree can be viewed as *vertically* partitioning the query. Using this taxonomy, static right-deep scheduling and dynamic right-deep scheduling are horizontal query partitioning algorithms. Right-deep hybrid scheduling is a vertical query partitioning algorithm.

4.4.5. The Case for Right-Deep Query Trees

(1) Right-deep query trees provide the best potential for exploiting parallelism.

(2) In the best case, 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.

(5) Several strategies exist to deal with limited memory situations. "Breaking" the query tree represents a static approach, while the dynamic bottom-up scheduling algorithm is more responsive to the amount of memory available at run-time. The RDHS strategy can potentially 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 4.4 for both a left-deep and a right-deep query tree format. Assume that the size of each relation 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. With a right-deep query tree, however, relations B, C and D must reside in memory, but these relations will only consume 30 pages of memory.

(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.

4.5. Bushy Query Trees

With more complex query tree representations, such as the bushy query tree for the eight-way join shown in Figure 4.6, 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 4.7 contains the dependency graph corresponding to the join query shown in Figure 4.6. 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 S^1, S^2,

S^4 and S^8. Since the 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:

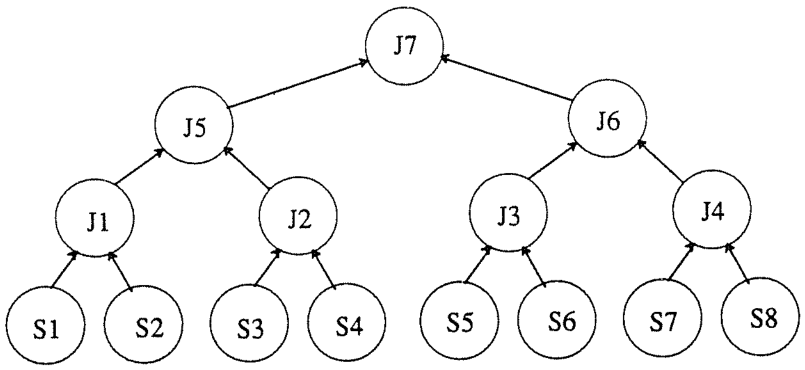
1) Scan S^1-Build J^1, Scan S^3-Build J^2, Scan S^5-Build J^3, Scan S^7-Build J^4.
2) Scan S^2-Probe J^1-Build J^5, Scan S^6-Probe J^3-Build J^6.
3) Scan S^4-Probe J^2-Probe J^5-Build J^7.
4) Scan S^8-Probe J^4-Probe J^6-Probe J^7.

However, notice that non-critical-path operations like Scan S^7 and Build J^4 could be delayed until Step 3 without violating the dependency requirements. The existence of scheduling options such as the above 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.

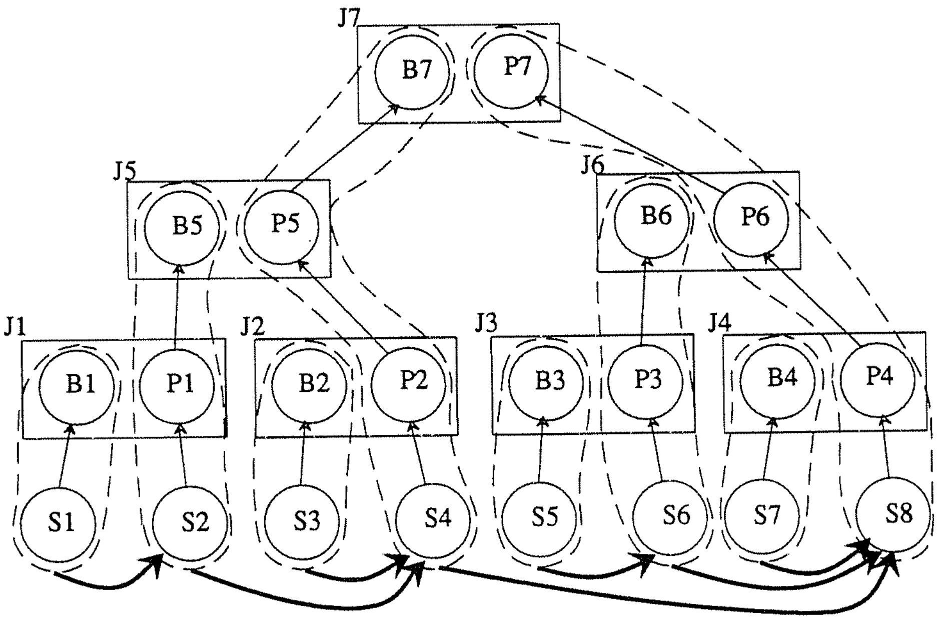
By intelligently scheduling operators it is possible to reduce the memory demands of a query optimized as a bushy tree. Consider again the previous schedule for executing the 7 join query. After the execution of Step 1, four hash tables will be resident in memory. After Step 2 completes, memory can be reclaimed from the hash tables corresponding to join operators J^1 and J^3, but new hash tables for join operators J^5 and J^6 will have been constructed. Only after the execution of Step 3 can the memory requirements be reduced to three hash tables (J^7, J^6, and J^4). 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:

1) Scan S^1-Build J^1.
2) Scan S^2-Probe J^1-Build J^5-Release J^1, Scan S^3-Build J^2.
3) Scan S^4-Probe J^2-Probe J^5-Build J^7-Release J^2 and J^5.
4) Scan S^5-Build J^3.
5) Scan S^6-Probe J^3-Build J^6-Release J^3, Scan S^7-Build J^4.
6) Scan S^8-Probe J^4-Probe J^6-Probe J^7-Release J^4, J^6 and J^7.

Although this execution plan requires six steps instead of four, the maximum memory require-



Bushy Query Tree
Figure 4.6



Dependency Graph for a Bushy Query Tree
Figure 4.7

ments 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 previous two subsections for left-deep and right-deep query trees can also be employed.

The optimization search space for bushy trees is also much larger than that of left-deep or right-deep trees. For an N-join query, the number of possible join orderings is $\frac{(2N)!}{N!}$ for bushy trees as opposed to $(N+1)!$ for either left-deep or right-deep trees [SWAM88]. However, by examining the larger search space, it is possible that the resulting query plan will process fewer tuples.

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.

4.6. Issues When Using the Sort-Merge Join Algorithm

If the sort-merge join algorithm is used instead of a hash-based join algorithm, the preceding discussion of the impact of the alternative query tree formats does not apply. For example, reconsider the left-deep query tree and its associated operator dependency graph in Figure 4.2. With the sort-merge algorithm as the join method, the scan S^1 does not necessarily have to precede the scan S^2. For example, the scan and sort of S^1 could be scheduled in parallel with the scan and sort of S^2. 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 4.2 is:

1) Sort S^1, S^2, ..., S^{N+1} (if not already sorted on the join attribute)
2) Merge-join J^1 (sort output if necessary)
3) Merge-join J^2 (sort output if necessary)
•
•
•
N+1) Merge-join J^N

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 MJ^i), 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 MJ^i node after each MJ^i node in the query tree. Create scan nodes as before.

2) If a base relation needs sorting, replace Scan S^i with Sort S^i —> Scan $S^{i'}$

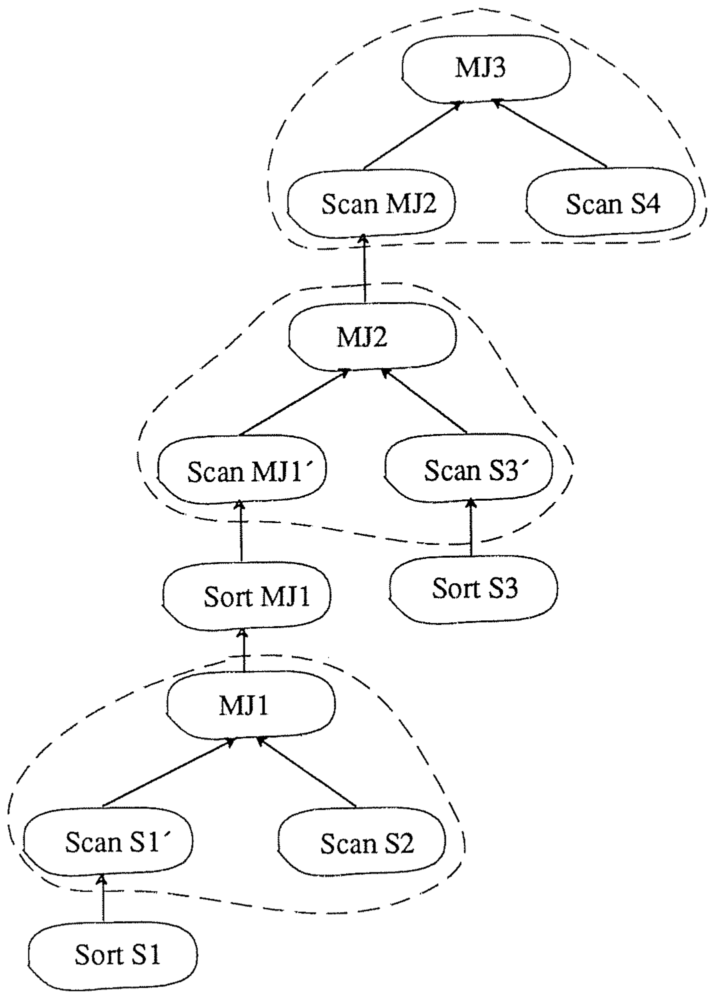
3) If the output from MJ^i needs sorting, replace Scan MJ^i with Sort MJ^i —> Scan $MJ^{i'}$

4) Group each MJ^i operator with its immediate descendants as a subgraph of operators.

To illustrate this algorithm, consider the query that joins relations S^1, S^2, S^3 and S^4, where S^1 and S^3 and the output of MJ^1 must be sorted. The operator dependency graph resulting from this query is shown in Figure 4.8. It should be noted that the addition of scan nodes does not necessarily imply additional disk I/O. 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.

One interesting point to note about using the sort-merge join algorithm is that the left-deep and right-deep query tree optimization alternatives become equivalent because all the base relations (S^1 through S^{N+1}) can be scanned/sorted concurrently in either strategy, whereas with the hash-join algorithm there is an ordering dependency that specifies that the left-child input must be completely consumed before the right-child input can be started.

Previous discussions of bit filtering were also influenced by the hash-based join methods. Since all hash-join algorithms require that the entire left input be consumed before the operator producing the right input can be started, 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 from 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 and foregoing the use of bit filters.



Operator Dependency Graph for Sort-Merge Join
Figure 4.8

CHAPTER 5

EVALUATION OF RIGHT-DEEP SCHEDULING STRATEGIES

In this chapter, we analyze the performance of the right-deep hybrid scheduling (RDHS) algorithm and the static right-deep (staticRD) scheduling algorithm for processing multi-way join queries. These algorithms were chosen because they present the two extremes among the algorithms for processing right-deep query trees when memory is limited. We first present a comparison based on a simplified analytical model. This comparison is followed by a more detailed simulation study.

5.1. Analytical Comparison of RDHS and StaticRD

In this section, we present a simple analytical model for comparing the RDHS and staticRD algorithms. The cost measure that will be used is the number of **relation** I/O's. That is, the number of times a base relation (BR) or intermediate relation (IR) is read or written. For simplicity, it is assumed that all relations are the same size and a fixed selectivity factor is applied to all relations (both base relations and intermediate relations). The parameters for the comparison are:

N : number of joins in the query (N+1 joining relations)
M : available memory (fraction of each building relation, e.g. 0.5 = 1/2 of each relation will fit in memory)
sf: selectivity factor (e.g. 0.90 = 90% of tuples qualify)

The cost formula for the RDHS algorithm is:

$$RDHS_{IO} = [((N{+}1) + 2*sf*(1-M)*(N{+}1))] \; + \; [(2*sf*(1-M)*(N-1))]$$

The total I/O cost for the RDHS algorithm is broken into two components. The first component is the number of times the base relations are read or written and the second component is the number of times the intermediate relations are read or written. The N+1 term in the base relation cost formula reflects the cost to read all the base relations. The $2*sf*(1-M)*(N+1)$ term reflects the cost to partition the base relations into buckets and to later read the buckets from disk. This term is broken down as follows. The N+1 cost component is the number of base relations to be joined. A selectivity factor, sf, reduces the number of qualifying tuples from each of the base relations. The RDHS algorithm is very sensitive to the amount of available memory because it uses the Hybrid hash-join algorithm for each individual join operation. Recall that the Hybrid join algorithm tries to stage as much of the building relation into memory as possible. Thus, the fraction of each of the relations written to disk is $1-M$. Finally, the factor of 2 reflects the fact that the tuples written to disk must be read when the disk buckets are subsequently joined. The IR cost component, $2*sf*(1-M)*(N-1)$, reflects the cost of partitioning the intermediate relations into buckets and their subsequent read accesses. It is very similar to the second half of the BR cost component, with the exception that there are only $N-1$ intermediate relations.

The cost formula for the staticRD algorithm is:

$$staticRD_{IO} = (N+1) \; + \; (2*sf*k) \qquad where \quad k = \left\lceil \frac{N}{\lfloor M*N \rfloor} \right\rceil - 1$$

The cost formula for staticRD is also broken into two components, base relation I/O and intermediate relation I/O. The first component, $(N+1)$, reflects the cost to read all the base relations. The second component reflects the fact that two intermediate relation I/O's are required each time the query tree is broken, one to write the intermediate relation to disk and a second to read it back. The number of times the query tree is broken, k, is computed under the constraint that the tree can only be broken an integral number of times, and only between adjacent join operations. Thus, for an N-join query, k must be between 0 and $N-1$.

The derivation of the formula for k is as follows. There are N building relations. M is the fraction of each building relation that fits into memory and thus $M*N$ is the number of building relations that fit into memory. Since the staticRD algorithm cannot deal with fractional numbers of relations, the term is truncated to yield $\lfloor M*N \rfloor$. But, we want to compute the number of times to "break" the query tree. If we have N building relations and $\lfloor M*N \rfloor$ fit into memory simultaneously, then $\frac{N}{\lfloor M*N \rfloor}$ is the number of subtrees that the query tree needs to be broken into to adhere to the memory requirements. Again, this must be an integral value, so the ceiling function must be applied, yielding $\left\lceil \frac{N}{\lfloor M*N \rfloor} \right\rceil$. However, this computes the number of subtrees in the query, not the number of breaks. The number of breaks, k, is one less than the number of subtrees.

These cost functions can be used to explore the expected performance of the staticRD and RDHS query processing strategies. Table 5.1 presents results for various memory availabilities for queries with 3, 4, and 8 joins, each having a selectivity factor of 50%. There are several observations to be drawn from this table. As expected, at 100% memory availability (M = 1.00)

		RDHS			STATICRD			
N	M	BR	IR	TOTAL I/O	BR	IR	k	TOTAL I/O
3	1.00	4.00	0.00	4.00	4.00	0.00	0	4.00
3	0.90	4.40	0.20	4.60	4.00	1.00	1	5.00
3	0.67	5.32	0.66	5.98	4.00	1.00	1	5.00
3	0.50	6.00	1.00	7.00	4.00	2.00	2	6.00
3	0.34	6.64	1.32	7.96	4.00	2.00	2	6.00
3	0.25	7.00	1.50	8.50	-	-	-	-
4	1.00	5.00	0.00	5.00	5.00	0.00	0	5.00
4	0.90	5.50	0.30	5.80	5.00	1.00	1	6.00
4	0.75	6.25	0.75	7.00	5.00	1.00	1	6.00
4	0.67	6.65	0.99	7.64	5.00	1.00	1	6.00
4	0.50	7.50	1.50	9.00	5.00	1.00	1	6.00
4	0.49	7.55	1.53	9.08	5.00	3.00	3	8.00
4	0.33	8.35	2.01	10.36	5.00	3.00	3	8.00
4	0.25	8.75	2.25	11.00	5.00	3.00	3	8.00
4	0.20	9.00	2.40	11.40	-	-	-	-
8	1.00	9.00	0.00	9.00	9.00	0.00	0	9.00
8	0.90	9.90	0.70	10.60	9.00	1.00	1	10.00
8	0.75	11.25	1.75	13.00	9.00	1.00	1	10.00
8	0.50	13.50	3.50	17.00	9.00	1.00	1	10.00
8	0.40	14.40	4.20	18.60	9.00	2.00	2	11.00
8	0.25	15.75	5.25	21.00	9.00	3.00	3	12.00
8	0.24	15.84	5.32	21.16	9.00	7.00	7	16.00
8	0.20	16.20	5.60	21.80	9.00	7.00	7	16.00
8	0.10	17.10	6.30	23.40	-	-	-	-

Table 5.1

the two algorithms perform the same[1]. For the staticRD algorithm, as memory is reduced, the total I/O cost is always a step function. This occurs because the tree can only be broken horizontally in an integral number of places. If hash table overflow was allowed, the results would follow the curves presented in Chapter 3. The missing entries in Table 5.1 for the staticRD algorithm reflect the limitation of only allowing horizontal memory partitioning. Regardless of the number of times a query tree is broken, at least enough memory must exist to stage a single building relation into memory. This is not a constraint with the RDHS algorithm.

The results for the staticRD algorithm may seem strange because the number of breaks in the query tree, *k*, does not always increase by one. For example, with the 4-join query, at 50% memory availability only one break is needed, while at 49% memory availability three breaks are required. At 50% memory availability, the query tree is broken between the second and third joins. This reduces the query tree to two, 2-join sub-queries, each of which requires 50% memory availability. However, if the query tree is broken in two places, three sub-queries will be formed. And regardless of where the two breaks are taken, two of the sub-queries will each be composed of only one join and the remaining sub-query will consist of two joins. Hence, the more complex sub-query will require 50% memory availability while each of the other two sub-queries will only need 25% memory availability. But, if 50% memory availability is required for one of the sub-queries a smart query optimizer would recognize that memory would be used more efficiently by breaking the tree only after the second join, thus producing two sub-queries each of which requires 50% memory availability.

The most important conclusion to draw from Table 5.1 is that the staticRD algorithm requires less total disk I/O than the RDHS algorithm for all degrees of query complexity except when very large amounts of memory are available ($M > 90\%$). And, as the complexity of the query increases (N gets larger), the difference in total I/O between the two algorithms grows.

[1]At 100% memory availability, the staticRD and RDHS algorithms are identical to each other and also to the optimistic right-deep scheduling algorithm.

Thus, based on an I/O cost metric, the staticRD algorithm should almost always outperform the RDHS algorithm. However, there are three observations that should temper this conclusion. First, a cost metric of total I/O does not take into account the effects of parallelism. This is a crucial aspect of the RDHS algorithm. For instance, when a base relation is being partitioned into buckets the disk buckets are being written as the relation is being read. If this operation can be done efficiently the resulting cost will be less than that implied by the total I/O cost metric. Second, the RDHS algorithm generally performs less intermediate relation I/O than the staticRD algorithm. If the size of intermediate relations dominates the size of the relations to be joined, the RDHS algorithm will do better. And finally, the cost formulas have no way of estimating initial response time (the time to produce the first output tuple). As was discussed in Chapter 4, this may be an important performance criteria.

Sensitivity to Selectivity Factor

The selectivity factor also has a significant impact on the difference between the two algorithms. Table 5.2 presents results for different selectivity factors applied to the base relations of the 4-join query (N=4). Note that a new column has been added which shows the ratio of total I/O required for the RDHS algorithm to that required for the staticRD algorithm. The main conclusion to draw is that as the selectivity factor decreases (more tuples eliminated), the difference in total I/O between the two algorithms diminishes.

5.2. Simulation Comparison of RDHS and StaticRD

As mentioned above, the analytical model had serious shortcomings in that it could not encompass the effects of parallelism and could not produce response times or resource utilizations for the query processing algorithms. To address these deficiencies, we faced either implementing the algorithms on a multiprocessor database machine or in a simulator. We chose to build a simulation model for two main reasons. First, we felt it would be simpler and faster to write new scheduling algorithms in a simulator than in an actual system. And second, a simulation model provides more flexibility in that the alternative algorithms can be studied in

SF	M	RDHS BR	RDHS IR	RDHS TOTAL I/O	STATICRD BR	STATICRD IR	STATICRD TOTAL I/O	DIFF
75%								
	0.90	5.75	0.45	6.20	5.00	1.50	6.50	0.95
	0.75	6.88	1.12	8.00	5.00	1.50	6.50	1.23
	0.67	7.47	1.48	8.96	5.00	1.50	6.50	1.38
	0.50	8.75	2.25	11.00	5.00	1.50	6.50	1.69
	0.49	8.82	2.29	11.12	5.00	4.50	9.50	1.17
	0.33	10.02	3.01	13.04	5.00	4.50	9.50	1.37
	0.25	10.63	3.38	14.00	5.00	4.50	9.50	1.47
50%								
	0.90	5.50	0.30	5.80	5.00	1.00	6.00	0.97
	0.75	6.25	0.75	7.00	5.00	1.00	6.00	1.17
	0.66	6.70	1.02	7.72	5.00	1.00	6.00	1.29
	0.50	7.50	1.50	9.00	5.00	1.00	6.00	1.50
	0.49	7.55	1.53	9.08	5.00	3.00	8.00	1.14
	0.33	8.35	2.01	10.36	5.00	3.00	8.00	1.30
	0.25	8.75	2.25	11.00	5.00	3.00	8.00	1.38
10%								
	0.90	5.10	0.06	5.16	5.00	0.20	5.20	0.99
	0.75	5.25	0.15	5.40	5.00	0.20	5.20	1.04
	0.66	5.34	0.20	5.54	5.00	0.20	5.20	1.07
	0.50	5.50	0.30	5.80	5.00	0.20	5.20	1.12
	0.49	5.51	0.31	5.82	5.00	0.60	5.60	1.04
	0.33	5.67	0.40	6.07	5.00	0.60	5.60	1.08
	0.25	5.75	0.45	6.20	5.00	0.60	5.60	1.11

Effect of Selectivity Factor - N=4
Table 5.2

hardware configurations different from that of the chosen database machine.

As the basis for our simulation model we chose the shared-nothing database machine Gamma (see Chapter 3). 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. Instead, disk blocks are 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

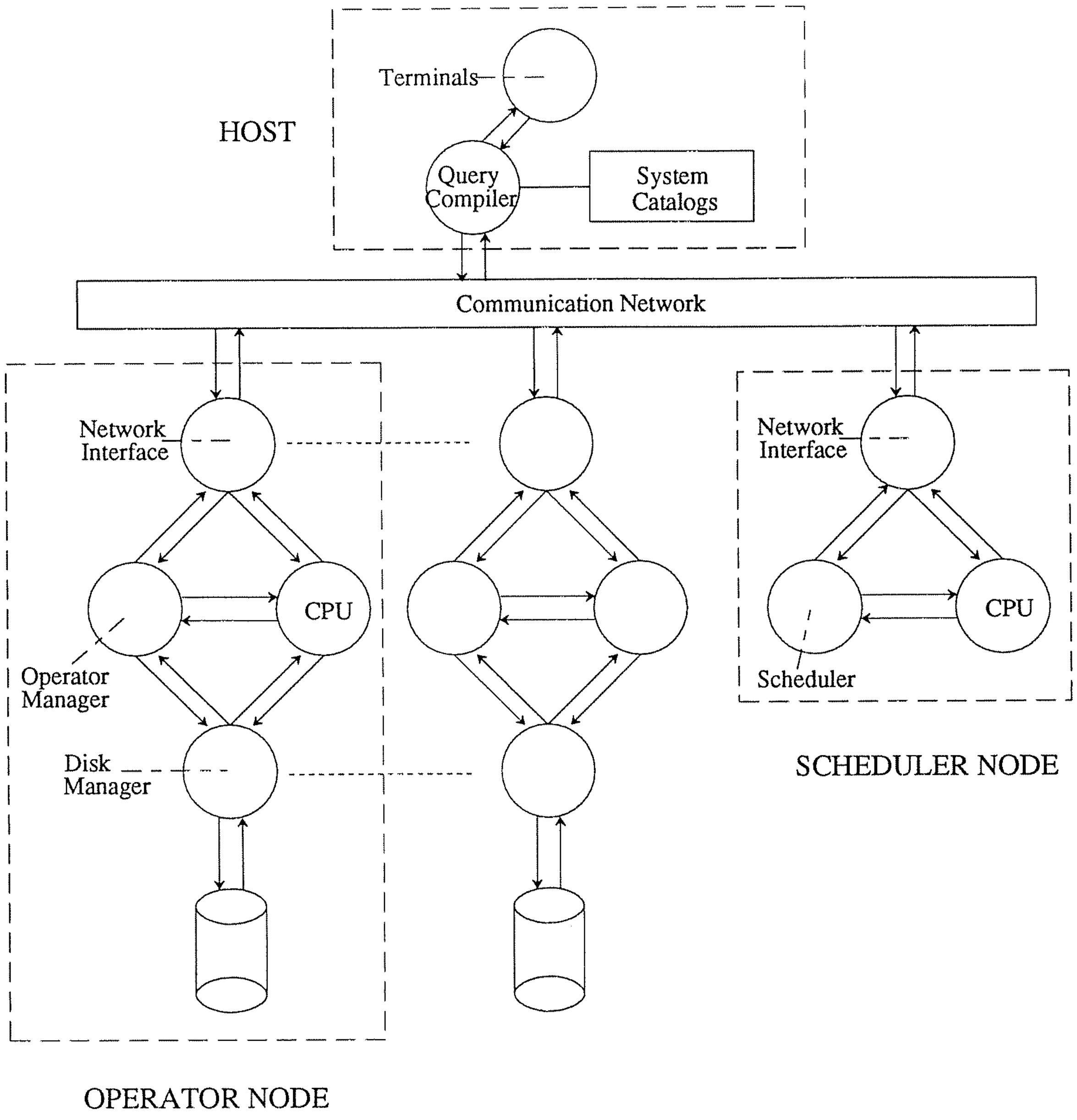
[2]Intel was forced to use such a design because the I/O system was added after the system had been completed, and the only way of doing I/O was by using an empty socket on the board which did not have DMA access to memory.

communication between processors.

5.2.1. Simulation Model

The simulation model of Gamma is organized as follows. Each node in the multiprocessor is composed of a Disk manager, a CPU manager, an Operator manager, and a Network Interface manager. Additionally, five stand-alone modules are provided: a Network manager, a Query Compiler, a Terminal module, a Query Scheduler and the System Catalog manager. See Figure 5.1 for a picture of the entire simulator. The DeNet simulation language [LIVN88] was used to construct the simulator.

The Disk Manager schedules disk requests to an attached disk according to the elevator algorithm [TEOR72]. In order to accurately reflect the hardware currently being used by Gamma, the disk manager 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, except for byte transfers to/from the disk's FIFO buffer. An Operator manager is responsible for modeling the relational operators, e.g., select and join. This manager repeatedly make requests to the CPU, Disk and Network interface managers to perform its particular operation. The Network Interface manager enforces a FCFS protocol for access to the global communications network. The Network module currently models a fully connected network and the Terminal module provides the entry point for new queries. The Query Compiler takes a description of a multiple-relation join query and produces a query plan in one of the alternative query plan formats. The Query Scheduler implements the algorithms for processing queries optimized in the alternative query plan formats. Finally, the System Catalog 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. For each file, a mapping from logical page numbers to physical disk address is also maintained. This physical assignment of file pages allows for more accurate modeling of sequential as well as random disk accesses.



Simulation Model
Figure 5.1

5.2.2. Validation of Simulation Model

As a check of the accuracy of 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. The costs associated with basic operations on this machine and relevant system parameters are summarized in Table 5.3. The values for the disk settle time, latency, and transfer rate were taken from MAXTOR disk specifications. The disk seek time is computed by multiplying the seek factor by the square root of the number of tracks to seek [BITT88]. The cost to transmit various size packets were taken from measurements of Gamma. In general, control packets are 100 bytes and data packets are 8,192 bytes.

The costs of various CPU operations were taken via measurement and also estimated by code inspection. Approximately 400 instructions are required to extract a single tuple off of a disk page and apply any local predicates. An additional 150 instructions are needed to place the tuple into a hash table; 200 instructions to probe a hash table. Finally, about 750 instructions are required to write a tuple to a disk or network page.

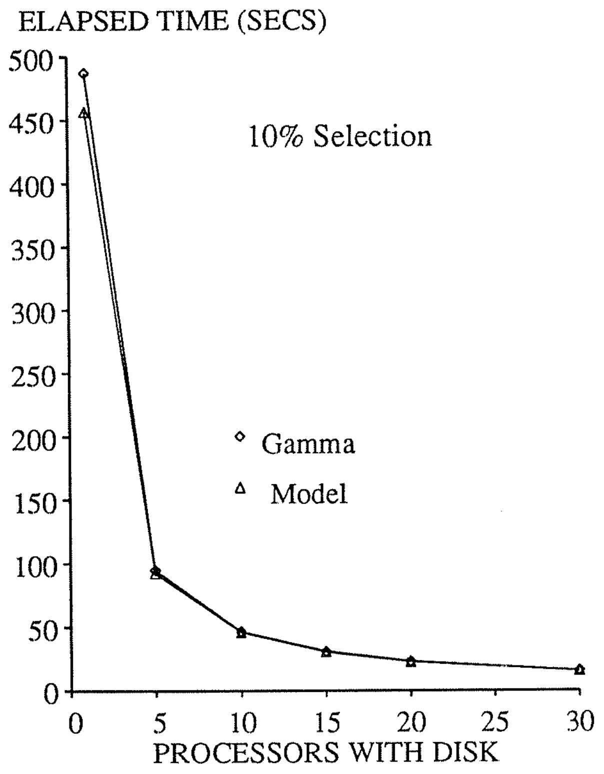
To validate the simulation model we present the performance of both a 10% selection query and a join query in a system with 1-30 processors with disks. Expanded versions of the Wisconsin Benchmark relations [BITT83] serve as the test database. The selection query

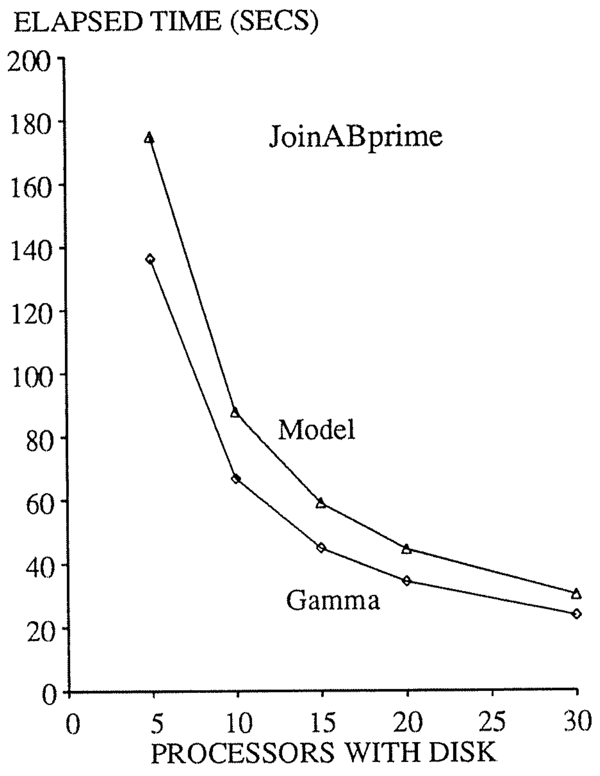
Disk Parameters	
Average Settle Time	2 msec
Average Latency	0-16.67 msec (Unif)
Transfer Rate	1.8 MBytes/sec
Seek Factor	0.78 msec
Disk Page Size	18 Kbytes
Xfer Disk Page from SCSI to mem	9000 instructions
Network Parameters	
Maximum Packet Size	8 Kbytes
Send 100 bytes	0.6 msec
Send 8192 bytes	5.6 msec
Cpu Parameters	
Instructions/Second	4,000,000
Read 18K Disk Page	32,800 instructions
Write 18K Disk Page	61,500 instructions
Miscellaneous	
Tuple Size	208 bytes
Tuples/Network Packet	36
Tuples/Disk Page	82
Number of Sites	1-30

Simulation Parameters for Model Validation
Table 5.3

retrieves 100,000 tuples from a 1,000,000 tuple relation via a sequential scan and stores the resulting tuples back into the database across all available processors. Figure 5.2 shows that the results from the simulation model and Gamma are very close for this query. However, the actual error is greater than implied because Gamma uses a one page readahead mechanism when reading pages from a file sequentially and this functionality is not modeled by the simulator. 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 (208 megabytes) with a 100,000 tuple relation (20 megabytes) to produce a 100,000 tuple result relation (40 megabytes). As illustrated by Figure 5.3, the simulation model overestimates the response time for this query by a factor of about 20% over the range of 5 to 30 processors with disks. Most of this inaccuracy is related to Gamma's use of a one page readahead mechanism when scanning a file sequentially. Since join queries are very CPU intensive





Validation of Selection and Join Performance
Figures 5.2 and 5.3

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 selection query presented in Figure 5.2. The CPU requirements of this query are much lower and thus the extent of the overlap of CPU and disk processing is much more limited. This claim is further supported by simulation results that accurately predict execution times for selection queries that use a non-clustered B^+-tree access method [HSIA90]. These queries generate a series of random disk requests and hence readahead is not employed.

5.2.3. Experimental Design

The experiments in this chapter were designed to analyze the performance differences between the staticRD and RDHS algorithms for processing right-deep query plans. We specifically address how several factors affect the comparison, including the effect of the data declustering strategy, sensitivity to selectivity factors and query complexity, and sensitivity to system parameters such as CPU speed and the size of network packets.

For the experiments conducted, the query suite consisted primarily of queries composed of four joins, although queries of other complexity are presented as well. In order to simplify the analysis, 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 tree. 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.

The database was composed of nine 1,000,000 tuple relations, and each relation has a selection predicate applied to it that reduces the output cardinality to 500,000 tuples. Each intermediate relation also consists of 500,000 tuples. Tuples are 208 bytes wide and attributes are not added with each successive join, so 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. In the following results, both initial response time and elapsed time are reported for each algorithm. Recall that initial response time is the time to produce the first output tuple, while elapsed time is the time to compute the entire query result.

The performance of each of the algorithms is computed as a function of available memory. That is, the x-axis represents the fraction of the total memory in the database machine for joining to the amount of memory required to hold all of the building relations (after any selectivity factors have been applied). Thus, at the x-axis value of 1.0, all of the building relations can be staged into memory concurrently without any memory overflow (assuming uniformity). At a value of 0.5, only half of the building relations can reside in memory concurrently.

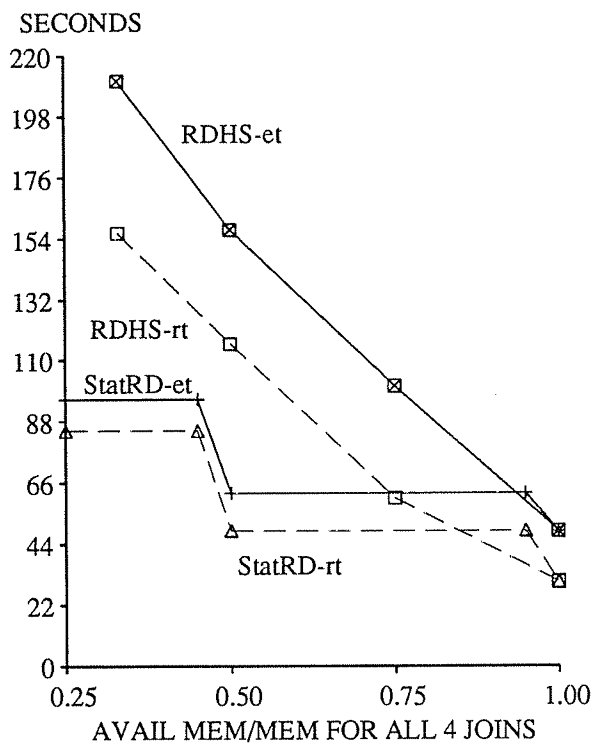
We were especially interested in how effectively each algorithm could exploit the resources found in a parallel database machine. As such, each experiment was conducted under two different data declustering strategies. The first configuration models a database machine with a modest number of processors. In such an environment it is likely 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 this particular environment, the system was configured such that each relation was declustered over the same 50 nodes. Each join in the query tree was also processed on all 50 nodes. Since all processors are used for each relational operator, this configuration will be referred to as the **full declustering** configuration.

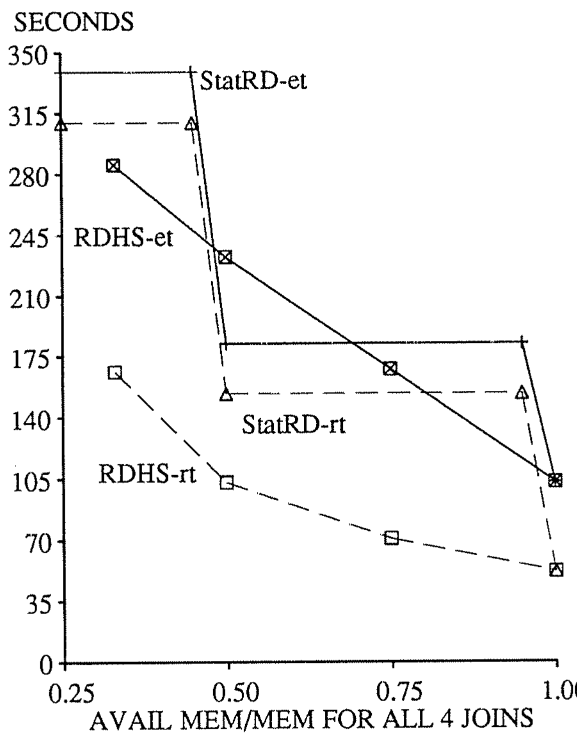
The second configuration was designed to model a database machine where relations are declustered over a subset of nodes. This scenario is likely to be true in a database machine with many processors and is referred to as **partial declustering**. The effect of having partial declustering is that resource contention is reduced when executing several operators concurrently. For this environment the system was configured as follows. Each of the relations to be joined was declustered over 10 distinct, non-overlapping nodes. Each join was also processed on the 10 processors on which its "building" relation was declustered. Each temporary relation, as well as the output relation, was declustered across all available processors.

5.2.4. Experimental Results

5.2.4.1. Sensitivity to Resource Availability

Figures 5.4 and 5.5 present the results for executing the 4-join query in a system with full relation declustering and partial relation declustering, respectively. The results for the full declustering experiment verify the analytical results presented above[3]. That is, the staticRD algorithm outperforms the RDHS algorithm in almost all cases when elapsed time is the cost metric. The difference is lessened when initial response time is the cost metric.





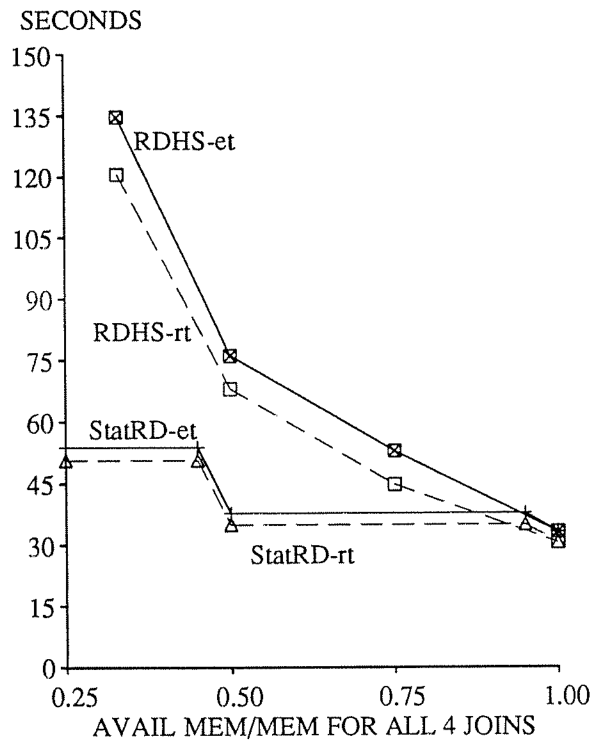
Full and Partial Declustering - 4 joins
Figures 5.4 and 5.5

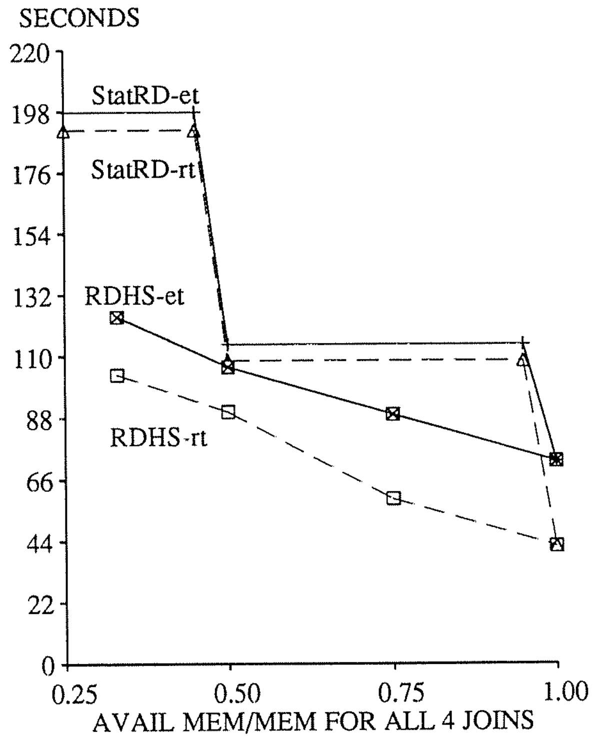
[3]The difference in I/O between the algorithms in the simulator matches that predicted by the analytical model.

However, a change in data placement can significantly alter the performance comparison as the partial declustering results in Figure 5.5 show. Here, the RDHS algorithm outperforms the staticRD algorithm over much of the range of memory availability using both the elapsed time and initial response time cost metrics. This demonstrates that when the appropriate resources are available, the parallelism inherent in the RDHS algorithm can overcome the expense of the additional disk I/O. The difference in the two algorithms with respect to initial response time is particularly striking.

5.2.4.2. Sensitivity to Selectivity Factor

Figures 5.6 and 5.7 show the results of modifying the 4-join query to use a selectivity factor of 10% for each joining relation as opposed to 50% as was shown in Figures 5.4 and 5.5. The reduction in selectivity factor reduced the size of the intermediate relations from 50,000 tuples to 10,000 tuples. As predicted by the analytical model in section 5.1, the difference in



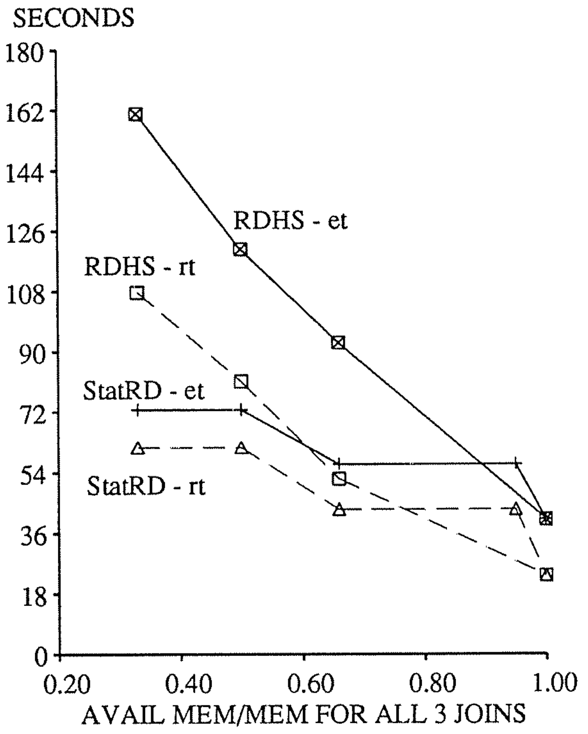


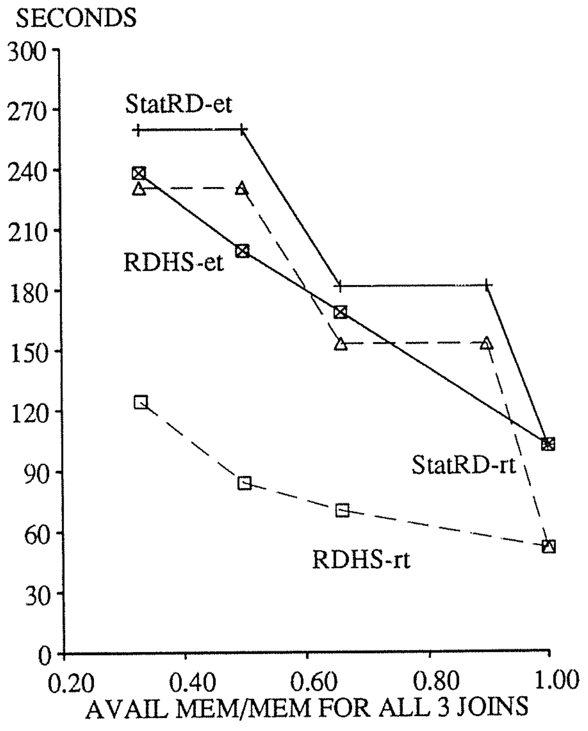
Full and Partial Declustering - 10% Selectivity Factor
Figures 5.6 and 5.7

the number I/Os performed by each algorithm decreased as the selectivity factor decreased (more tuples eliminated), reducing the difference between the two algorithms. The difference between the elapsed time and the initial response time for each algorithm also diminishes as the selectivity factor is reduced. This occurs because the temporary join results are smaller due to the lower selectivity factor, and these tuples cannot propagate up the entire query tree as fast as before because of network packet buffering. This is explained in more detail in the section on the sensitivity to network packet size.

5.2.4.3. Sensitivity to Query Complexity

In this set of experiments, we analyze the effects of query complexity, i.e., the number of relations to be joined, on the performance of the two right-deep scheduling strategies. Figures 5.8 and 5.9 show the results for executing a 3-join query in a system with full relation declustering and partial relation declustering, respectively. These results should be compared with

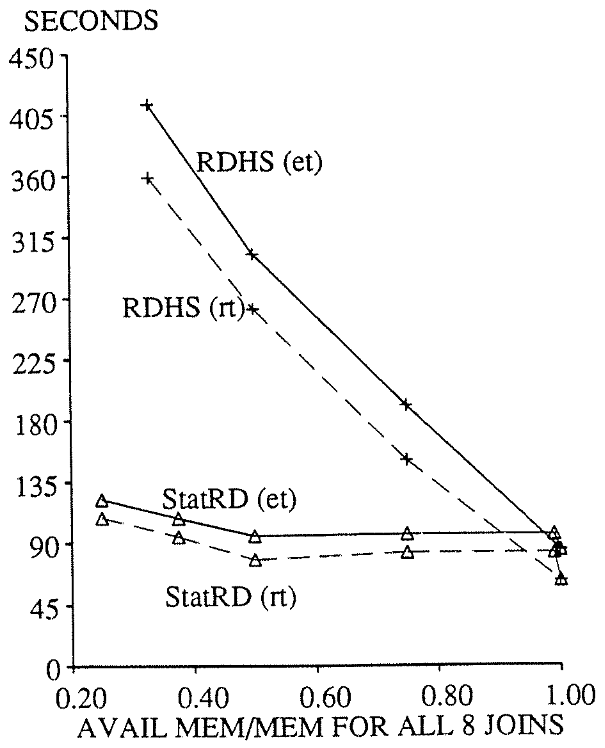




Full and Partial Declustering - 3 joins
Figures 5.8 and 5.9

Figures 5.4 and 5.5 where the join query was composed of 4 joins. The change in crossover points in Figures 5.4 and 5.8 and in Figures 5.5 and 5.9 clearly shows that having a less complex query favors the RDHS algorithm more than the staticRD algorithm. The difference in initial response time between Figures 5.5 and 5.9 is particularly striking. In this low resource contention environment, the RDHS algorithm is producing tuples in less than half the time required for the staticRD algorithm for most degrees of memory availability.

The results from running an 8-join query confirm the trends shown with the 3-join query. Figure 5.10 shows the results from running the staticRD and RDHS algorithms on an 8-join query in a full declustering environment. When compared with Figures 5.4 and 5.8 , it is obvious that increasing the complexity of the query causes substantial performance problems for the RDHS algorithm when resource contention is high. These experimental results also support the analytical results from Section 5.1 that showed that increasing the complexity of the

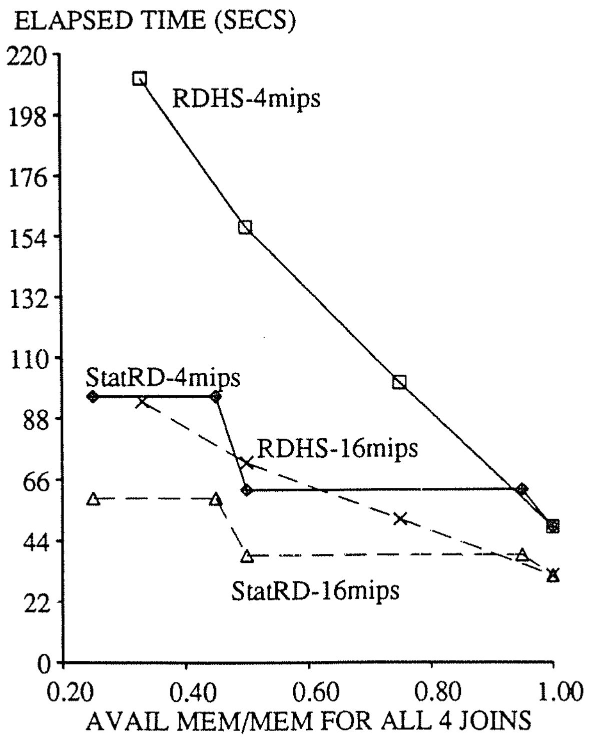


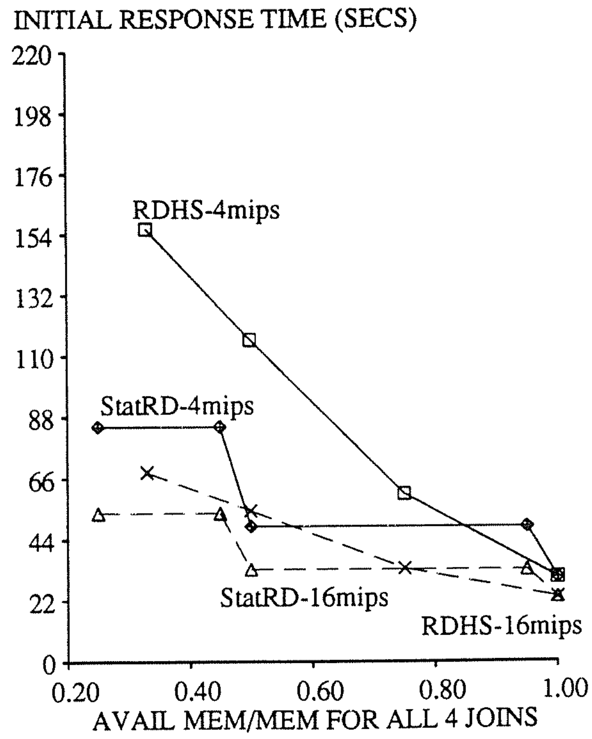
Full Declustering - 8 joins
Figure 5.10

query serves to increase the difference in disk I/O requirements between the two right-deep scheduling algorithms.

5.2.4.4. Sensitivity to CPU Speed

The speed of the CPUs has a significant effect on the performance of both of the algorithms. Figures 5.11 and 5.12 show the performance improvements in elapsed time and initial response time, respectively, for the 4-join query in the full declustering environment when the CPU speed is increased from 4 to 16 mips. As shown, both algorithms show substantial speedup from having faster CPUs. The RDHS algorithm receives more benefit, especially as memory becomes increasingly scarce, because it becomes CPU bound with 4 mip CPUs due to the cost of writing tuples to disk and later reading them, and also because it sends more network packets. The speedups for the partial declustering environment from having the faster





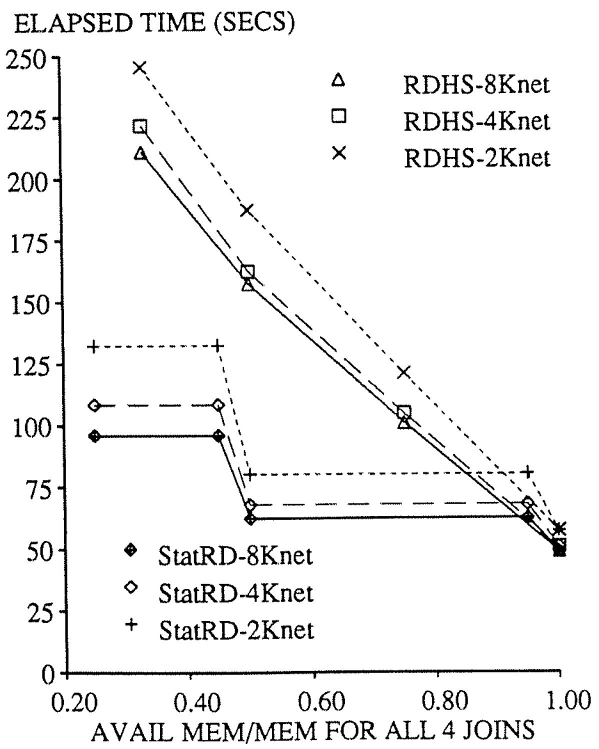
Effect of CPU Speed on Elapsed Time and Initial Response Time
Full Declustering
Figures 5.11 and 5.12

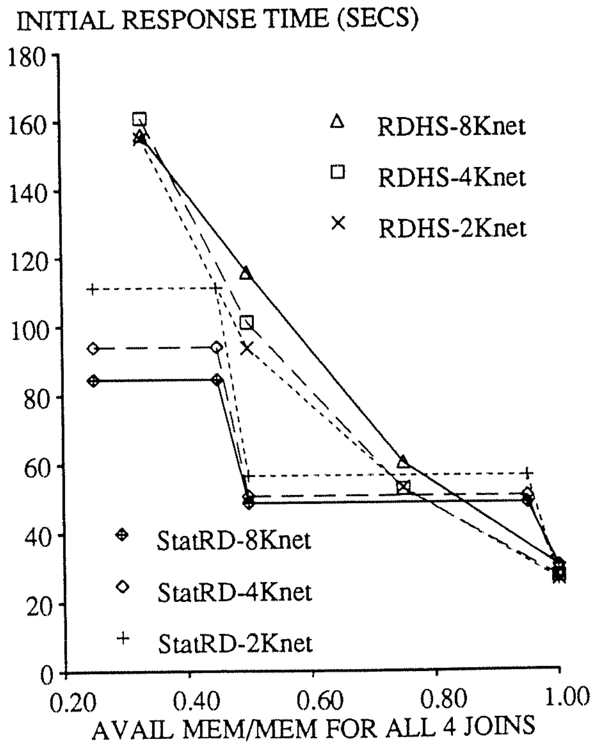
CPUs are basically identical to the speedups for full declustering and are thus not presented.

5.2.4.5. Sensitivity to Network Packet Size

The choice of the size of the network packets also has a significant effect on the performance of the two algorithms. Each of the previous experiments used a network packet size of 8 Kbytes. Figures 5.13 and 5.14 present the effects on elapsed time and initial response time, respectively, when the size of the network packets is reduced from 8 Kbytes to 4 Kbytes and 2 Kbytes. Only the results from the full declustering environment are shown because the results from the partial declustering environment follow the same trends.

Figure 5.13 shows that reducing the network packet size increases the elapsed time for both algorithms. Two factors led to this degradation in elapsed time. First, there are economies of scale in the time to transmit packets. For example, instead of taking twice as





Effect of Network Packet Size on Elapsed Time and Response Time
Full Declustering
Figures 5.13 and 5.14

long to send an 8 Kbyte packet as opposed to a 4 Kbyte packet, it only takes 1.5 times as long. Thus, reducing the size of the network packets increases the time to compute the query because the cost to transmit each byte is increased. Second, the transmission of a packet incurs a fixed CPU cost for protocol processing. A third possible factor, although it did not occur in these experiments, is that as the network packet size is reduced the proportion of "useful" bytes diminishes because of packet overheads. For example, if 36 tuples can be stored in an 8 Kbyte network packet, only 17 may fit in a 4 Kbyte network packet. Thus, decreasing the packet size by a factor of two could possibly cause more than two times as many packets to be sent.

The effect on initial response time when the network packet size is reduced, as shown in Figure 5.14, differs dramatically for the two algorithms. With the staticRD algorithm, initial response time increased when less than 100% memory was available. This occurred because of the reasons described above. However, the initial response time for the RDHS algorithm **decreased** for most memory availabilities with the reduction in the size of the network packets. This reduction in initial response time occurred, even though the elapsed time increased, because the first tuples were able to propagate up through the levels of the query tree more rapidly when the size of the network packets was reduced. This happens because output tuples are not sent to the next join operation in the query tree until a network buffer is filled. With smaller network packets, buffers fill much faster, hence the tuples in the probing pipeline propagate upwards much faster. Another benefit of the smaller network packet sizes is that the memory required for network buffers is reduced. For systems with large numbers of processors, this savings could be substantial.

5.3. Summary

The results from this chapter demonstrate the performance tradeoffs between the staticRD and RDHS algorithms for processing right-deep query plans. The analytical model predicted that the RDHS algorithm would be much more disk I/O intensive than the staticRD algorithm except in situations of very high memory availability. Results from the database machine simulator verified this aspect of the analytical model.

In a high resource contention environment, the staticRD algorithm outperforms the RDHS algorithm because it is less resource intensive. However, in a low resource contention environment, the RDHS generally performs better because it takes advantage of the greater resource availability. The experimental results also corroborate the prediction from Chapter 4 that the RDHS algorithm would perform quite well with respect to the initial response time cost metric. In a low resource contention environment, the difference in response time between the two algorithms is particularly striking.

The complexity of the query (in terms of the number of relations to join) was also shown to have a significant effect on the performance comparison. The RDHS algorithm fares much better when the number of relations to join is relatively small. As the number of joining relations grows, the resource requirements of the RDHS algorithm rise much faster than those of the staticRD algorithm and hence performance suffers.

We also saw how several system factors affect the performance of the two right-deep query processing algorithms. Both algorithms received significant speedups from having faster CPUs, but the RDHS algorithm received more benefit because it is more resource intensive. The experimental results also showed that decreasing the size of the network packets increased the amount of time required to compute the query for each algorithm. However, the reduction in packet size also had the effect of decreasing the initial response time of the RDHS algorithm because it allowed result tuples to propagate more quickly through the join tree. If query response time is important, either smaller network packets should be used or some type of early flush of network buffers should be implemented.

CHAPTER 6

LEFT-DEEP VERSUS RIGHT-DEEP QUERY PLANS

In this chapter, we explore the performance tradeoffs between left-deep query plans and right-deep query plans in a multiprocessor database machine. The discussion in Chapter 4 implied that a right-deep plan could potentially offer significant performance advantages over the same query optimized in a left-deep plan. The goal of this chapter is to ascertain under what conditions each plan type is advantageous.

In Chapter 5, we explored the performance of two strategies for processing right-deep query plans, right-deep hybrid scheduling (RDHS) and static right-deep scheduling (staticRD). This analysis will include both of these algorithms. The algorithm used for processing left-deep query plans is identical to that presented in Chapter 4. The simulation model described in Chapter 5 is used for the performance analysis.

Comparing left-deep and right-deep query plans in a fair and consistent manner is difficult. For the best comparison, a collection of "real" queries would be chosen and the optimal query plan compared for each of the left-deep and right-deep optimization strategies. Unfortunately, this type of comparison is not possible for several reasons. First, an optimizer uses cost functions to compare candidate query plans in its search for the best plan. However, as was shown in Chapter 5, the right-deep scheduling strategies attempt to increase performance through additional intra-query parallelism. If the hardware resources are available to support the increased parallelism, these algorithms will benefit considerably. Thus, for an optimizer to find the best right-deep query plan, it must have cost functions that are able to model resource contention. The development of these cost functions is an open research problem.

Another problem is finding a collection of "real" queries with which to make the performance comparison. Finding real queries, like finding real data, is an elusive task. In addition, the problem is even more complicated because we must guess what queries will look like as users' needs become more complex.

Another difficult task is deciding the proper test environment. It is common when comparing the performance of several algorithms to make conclusions like algorithm A runs faster than algorithm B by some percentage. If one algorithm consumes more resources such as CPU cycles or disk I/O, this is usually just mentioned as a side-effect of the algorithm. However, when comparing left-deep plans to right-deep plans, memory consumption is crucial to the analysis. If memory consumption is not a constraint, the right-deep plan produced by "mirroring"[1] the **optimal** left-deep plan for a particular query will generally be at least as good, and usually better, than the optimal left-deep plan. By mirroring the optimal left-deep plan, the resulting right-deep plan will have the same join selectivities and hence will process the same number of tuples. Without any memory constraints, the total amount of disk I/O will be equivalent for the left-deep and right-deep plans. If the dynamic bottom-up scheduling strategy processes the right-deep plan, each relation will be read sequentially and performance will closely match that of the left-deep plan. However, by using the optimistic right-deep scheduling strategy or the blocked version of the dynamic bottom-up strategy, several of the relations will be read concurrently; if the disks can efficiently support the scanning of multiple relations concurrently, the resulting performance will be better.

Our solution for comparing the performance of left-deep and right-deep query plans is to first delimit the performance differences by showing results obtained under extreme conditions. We then analyze how different aspects of the query, for example, the size of the building relations, the physical placement of the joining relations, the join selectivities, etc., affect the

[1]To mirror a left-deep plan in order to produce a right-deep plan, simply interchange all the building and probing relations from the left-deep plan.

performance comparison of left-deep and right-deep query plans. Through the combination of all these tests, the entire scope of performance for the two alternative optimization strategies will be covered for a single-user environment. Additionally, CPU and disk utilizations are reported for the algorithms and the ramifications of a multi-user environment are discussed.

6.1. Experimental Design

As was shown in Chapter 5, contention for CPUs and disks and the availability of memory are prime determinants of performance of the right-deep scheduling algorithms. As such, the comparison of left-deep plans with right-deep plans has to capture both of these situations.

As an attempt to delimit the range of performance differences, the first set of experiments compare the performance of left-deep and right-deep scheduling strategies under both high and low degrees of resource contention, but where memory is unlimited. The second set of experiments relaxes the unlimited memory assumption.

One potential problem with right-deep plans occurs with queries that have several large building relations, low join selectivities, and no highly restrictive selection predicates. With such queries, the memory requirements of a right-deep plan will be dramatically higher than those of a left-deep plan. Given a limited memory environment where the right-deep plan can only consume as much memory as a left-deep plan, the performance of the right-deep plan is likely to suffer. The third set of experiments demonstrate the performance implications of this type of query.

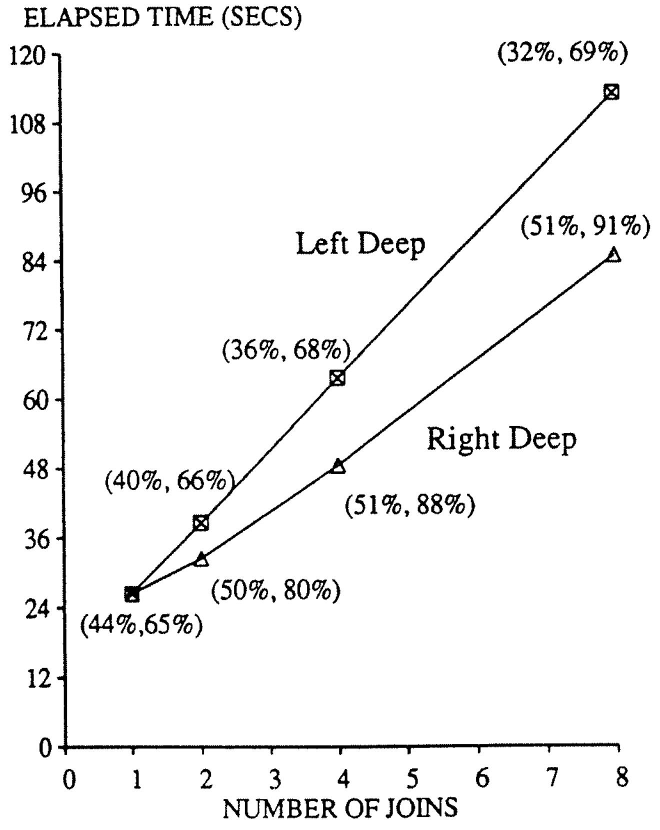
The remaining experiments show the performance of left-deep query plans and right-deep query plans for queries with high join selectivities and for different data placement strategies.

6.2. Experiment 1: Unlimited Memory

This first set of experiments is designed to present the range of performance differences between left-deep and right-deep query plans when the amount of memory for joining is unlimited. The database is identical to that reported in Chapter 5, i.e., each relation contains 1,000,000 tuples and each tuple is 208 bytes long. The queries are also similar to those described in Chapter 5, with the exception that the queries are composed of 1, 2, 4 and 8 joins. Other system parameters are identical to those outlined in Table 5.3. As in Chapter 5, we consider two database machine environments: a **full declustering/high resource contention** environment and a **partial declustering/low resource contention** environment. Recall from Chapter 4 that in an unlimited memory environment, the RDHS, staticRD and optimistic right-deep scheduling algorithms are identical.

6.2.1. High Resource Contention Environment

As described in Chapter 5, in the full declustering environment each joining relation is declustered over the same set of disk sites and each join is computed using all available processors. 50 processors with disks are used in this environment.



Unlimited Memory - Full Declustering
Figure 6.1

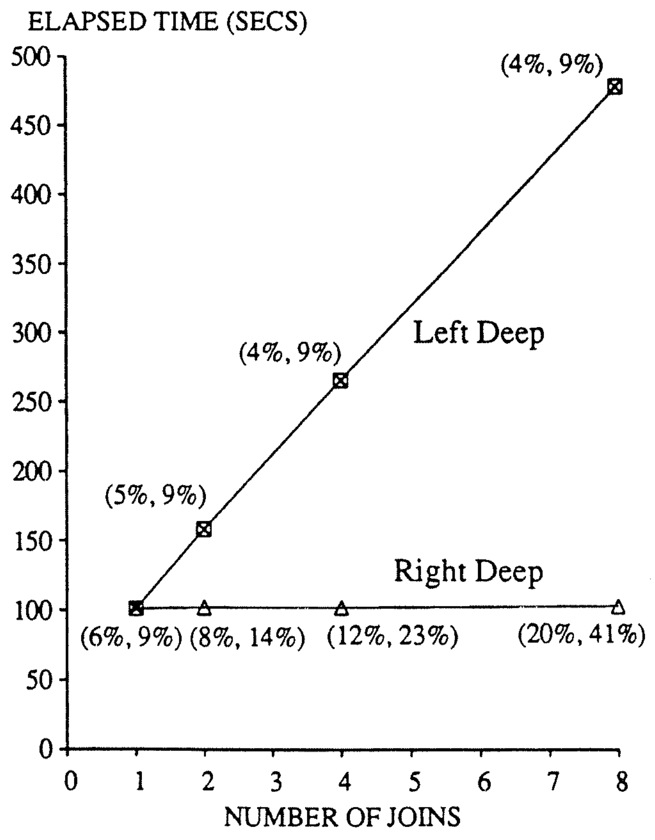
The elapsed time results for the full declustering experiments are shown in Figure 6.1 as the number of joins in the query is increased from one to eight. The graphs also indicate disk and CPU utilizations, respectively, at each of the data points. These resource utilizations reflect the average utilization over all the respective resources in the system.

For each different complexity of the join query, the performance of the right-deep query plan is approximately 15-20% faster than its analogous left-deep query plan (of course, left-deep and right-deep query plans are identical for single join queries). This performance improvement occurs because the disks and CPUs are not fully utilized and executing the scans in parallel for the right-deep plans provides some performance improvement. However, if a single scan of a declustered relation fully utilizes each of its associated disks or CPUs, a right-deep query plan will not demonstrate a performance advantage under these experimental conditions. Note that the maximum memory requirements for the left-deep and right-deep query plans are identical for queries with 1 and 2 joins, but are twice as high for right-deep plans with 4 joins and four times as high with 8 joins. These results are discouraging for right-deep plans with many joins because the speedup gained from using two and four times as much memory is limited to less than 20%. Also, the disk and CPU utilizations for the right-deep algorithm are significantly higher than the left-deep algorithm. Thus, if multiple queries were run concurrently, the throughput would be higher for the left-deep algorithm.

6.2.2. Low Resource Contention Environment

In this next set of experiments we wanted to demonstrate the performance tradeoffs between the two query optimization strategies in a database machine with more processors. The system was configured in the following manner (as in Chapter 5). 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 the elapsed times in Figure 6.2, left-deep query plans 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 are scanned one at a time when a left-deep query plan is employed. However, for right-deep query plans, a nearly constant response time is maintained as the number of joins is increased from one to eight. Given the experimental parameters, this result was expected. 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



Unlimited Memory - Disjoint Declustering
Figure 6.2

declustered over distinct nodes and the join nodes correspond to the base relation declustering nodes, each of the scans and hash table builds 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 starting the operators). The second phase (the probing phase) scales with the number of joins due to 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 elapsed time as the number of join levels is increased is due to the increased latency to initiate and terminate the pipeline. In Figure 6.2, we see that this overhead is negligible for up to 8 joins.

The results contained in Figure 6.2 represent best-case performance improvements of right-deep versus left-deep query plans. 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 plans will fall between the extremes presented in Figures 6.1 and 6.2. Also, it should be noted that the right-deep query plan with eight joins required four times more memory than any of the left-deep join plans. However, the results do demonstrate the extremely high performance benefits that can be obtained by using a right-deep query optimization strategy under conditions of low resource contention. In addition, the throughput of the two strategies should be comparable in this environment.

6.3. Experiment 2: Limited Memory

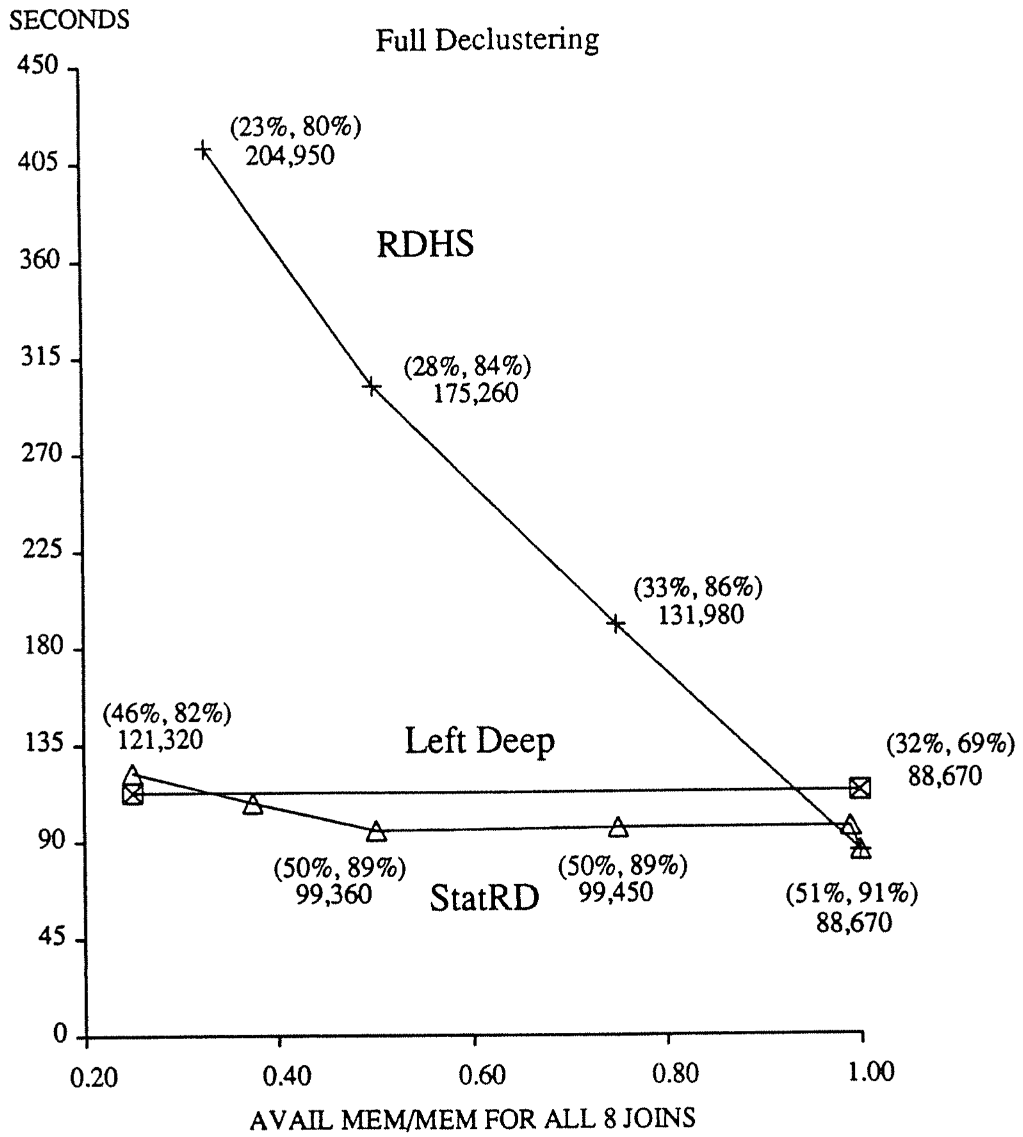
In this set of experiments we relax the assumption that an unlimited amount of memory exists for joining. All query and model parameters are identical to those reported in the previous experiment with the exception that for this analysis we concentrate on the query consisting of 8 joins. High resource and low resource contention (full declustering and disjoint declustering) experiments are again conducted.

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. Elapsed time is plotted for the left-deep and right-deep strategies for x-axis values ranging from 0.25, where only 2 of the 8 building relations can co-reside in memory, to 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 plans and are not reported. The graphs also indicate disk and CPU utilizations, respectively, as well as the total number of disk I/O's performed at selected data points.

For the static right-deep scheduling algorithm it was assumed that the optimizer could perfectly predict the scan selectivities and thus could always choose the optimal place(s) to "break" the query tree. For RDHS, each of the building relations was split into the same number of buckets at each of the different memory availabilities. Thus, at 0.33, each of the building relations was partitioned into 3 buckets; at 0.5, each relation was partitioned into 2 buckets. All temporary files for the right-deep scheduling strategies were declustered across all available disks.

6.3.1. Limited Memory - High Resource Contention

In Figure 6.3, the performance of the left-deep, staticRD, and RDHS scheduling algorithms is shown as the amount of available memory is varied in an environment where all base relations are declustered across all 50 sites. Several 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. Once enough memory is available to hold any two adjacent join operators, performance will be constant regardless of the presence of any additional memory. In contrast, staticRD does demonstrate some significant performance improvements by using any additional memory for joining. Next, the cross-over point between the staticRD and left-deep scheduling algorithms demonstrates that "breaking" the query tree into too many pieces can be detrimental to the performance of staticRD. For example, at the x-axis value 0.25, the tree had to be broken into three pieces to insure that the right-deep strategy did not experience



Limited Memory - Full Declustering
Figure 6.3

memory overflow, requiring the writing and subsequent reading of temporary join computations to and from the disk at three points during query execution. The flatness of staticRD from 0.5 to just before 1.0 occurs because 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 factor is shown in Experiment 3.

The performance of the RDHS algorithm can be easily explained by examining the total number of disk I/Os needed to compute the query. As Figure 6.3 shows, RDHS is performing substantially more disk I/O than the other two algorithms. Both staticRD and RDHS reduce their number of disk I/O's as the amount of memory for joining is increased. The disk I/O metric for the left-deep algorithm remains constant over the range of memory plotted because it is not using any of the extra joining memory.

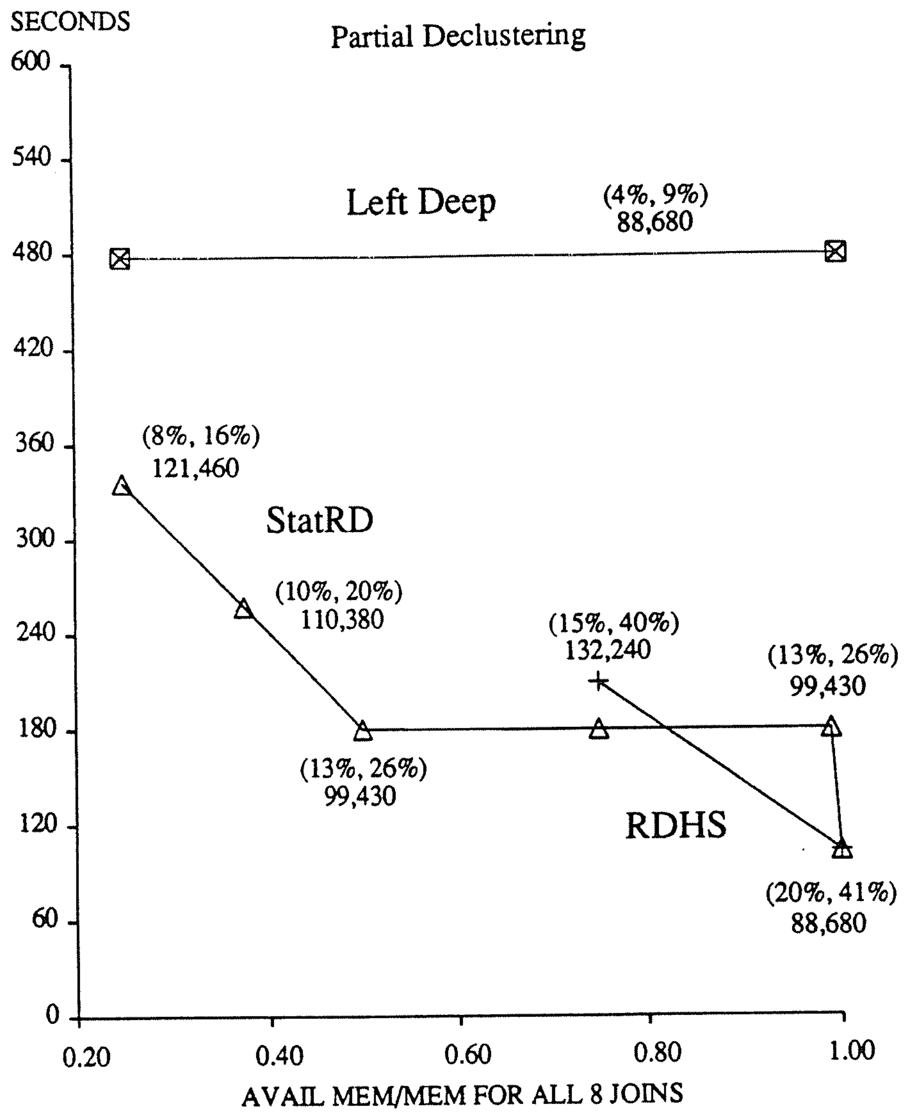
For this join query and test environment, the left-deep algorithm is still outperforming the right-deep algorithms when memory consumption for joining is identical (at x-axis value 0.25). Additionally, the left-deep algorithm has lower CPU and disk utilizations and hence will allow higher throughput when executed concurrently with other queries.

6.3.2. Limited Memory - Low Resource Contention

In Figure 6.4, we present the execution times, resource utilizations, and disk I/O counts 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 (as in the low resource contention environment of Experiment 1).

The results are very similar to those shown in Figure 6.3, i.e., the shape of the curves is identical. It is obvious though, that the low resource contention environment offers significant performance advantages for right-deep scheduling strategies even when memory is limited. This is encouraging because, as stated earlier, it is likely that relations will be partially declustered in database machines with large numbers of processors/disks. All the data points for RDHS are not included because of memory limitations when running the simulations.

The disk and CPU utilizations in Figure 6.4 show the tradeoff that exists between elapsed time and throughput for the staticRD and left-deep algorithms. At the x-axis value of 0.25 (equivalent memory consumption), the response time is lower for staticRD but more left-deep



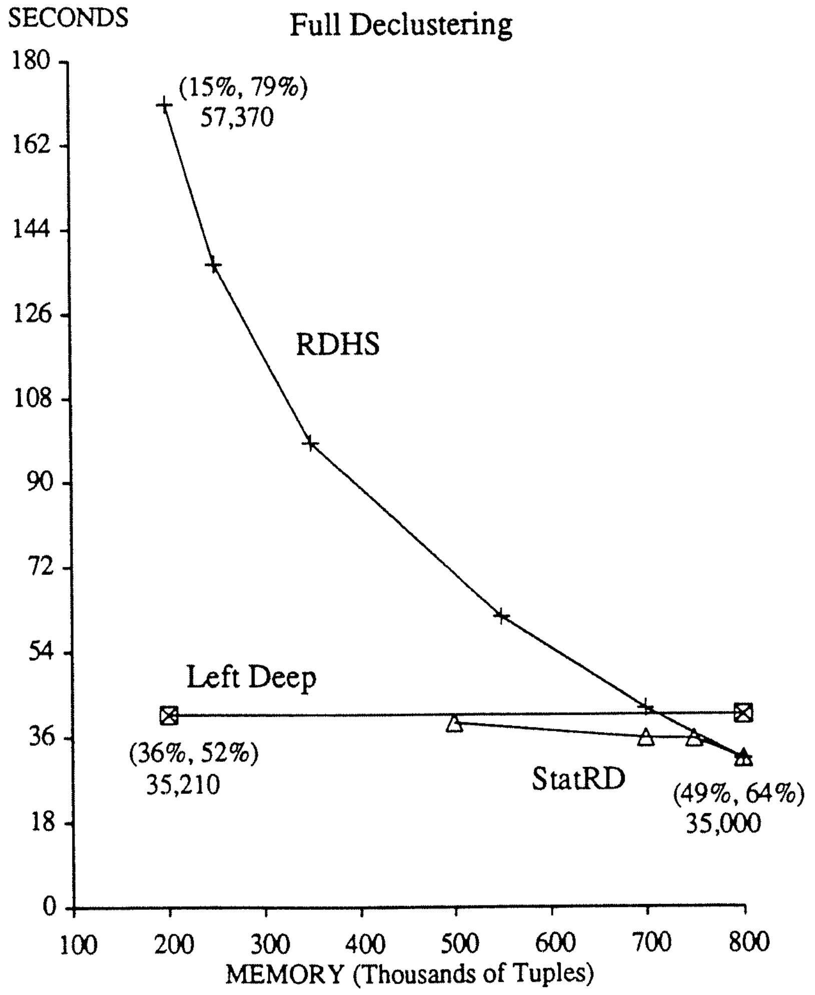
Limited Memory - Disjoint Declustering
Figure 6.4

plans can be run concurrently because of the lower resource utilizations. At 1.00, the throughput will be comparable between the alternative algorithms but the right-deep algorithms will consume four times more memory.

6.4. Experiment 3: Large Building Relations

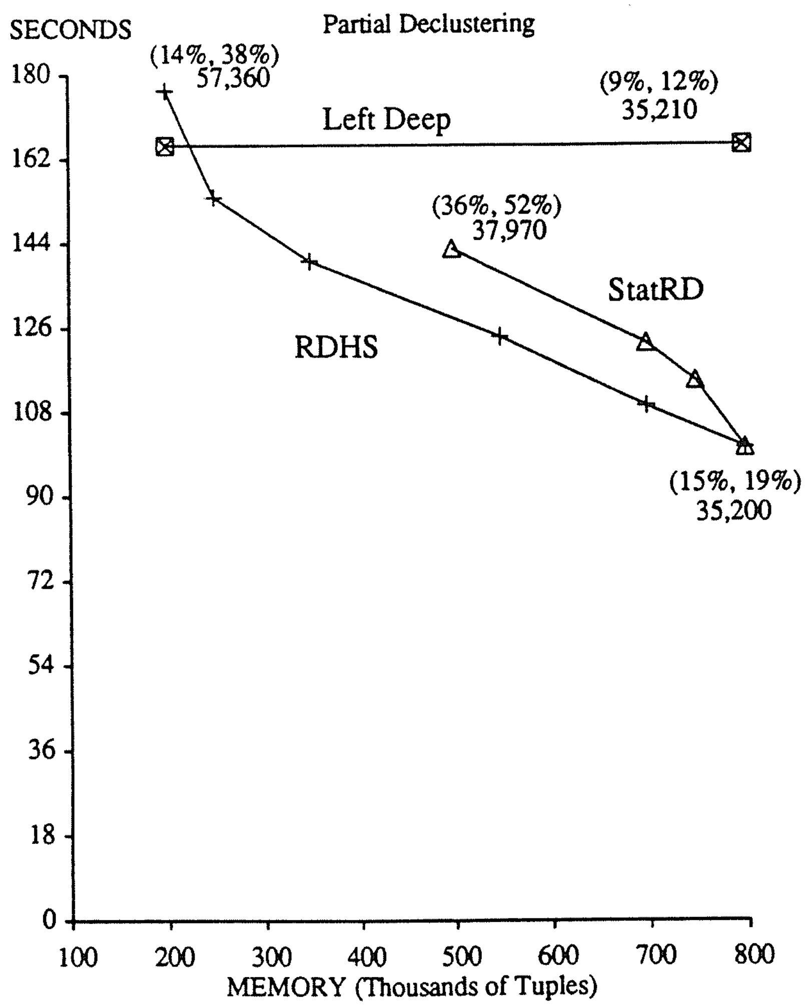
In this set of experiments, we wanted to analyze performance when the complex join query consists of several large relations without highly restrictive selection predicates and when join selectivities are low. This type of query is expected to be especially disadvantageous to right-deep plans when memory consumption is limited.

As the test query we chose a query that joins 5 relations. The relation cardinalities and selectivity factors are: 1,000,000 with 50% selectivity, 1,000,000 with 50% selectivity, 1,000,000 with 20% selectivity, 500,000 with 10% selectivity, and 200,000 with 25% selectivity. The join selectivities were designed such that the size of the intermediate relations was 50,000, 50,000, 100,000 and 100,000 tuples, and the size of the output relation was 100,000 tuples.



Large Building Relations - Full Declustering
Figure 6.5

Two environments, high and low resource contention, are again considered. The database machine consisted of 50 processors for each environment but, in the low resource contention environment each relation and join was declustered over 10 disjoint processors while in the high resource contention environment each relation and join was declustered across all 50 processors. The graphs plot elapsed time versus the amount of memory that is available for joining (shown as thousands of tuples). Additionally, disk and CPU utilizations, respectively, as well as the total number of disk I/O's performed are shown at selected data points.



Large Building Relations - Disjoint Declustering
Figure 6.6

The results for the high resource contention environment shown in Figure 6.5 are particularly revealing. It took 41 seconds to compute the query using a left-deep query plan. The RDHS algorithm processing a right-deep plan and consuming the same amount of memory as the left-deep plan took 171 seconds to compute. The staticRD algorithm fared better by executing the query in only 39 seconds but it used 2.5 times as much memory as the left-deep plan. Given the constraints of the staticRD algorithm, we were unable to reduce memory consumption to be equal to that of the left-deep plan. If the relations were even larger, the join selectivities further reduced, the selection predicates less restrictive, or more relations were to be joined the difference between the left-deep plans and the right-deep plans would continue to grow.

The results from running the same query in the low resource contention environment present a much different picture of performance, though. With this environment (see Figure 6.6), the right-deep algorithms perform significantly better as memory for joining is added. However, it should be noted that the left-deep algorithm is still outperforming the right-deep algorithms when memory consumption for joining is identical (at 200,000 tuples). Of course, even in a low resource contention environment such as here, if the relations were even larger, the selection predicates less restrictive, the join selectivities lower, or more relations were to be joined the performance of the algorithms for processing right-deep plans would further suffer when compared to the left-deep scheduling algorithm.

6.5. Experiment 4: Sensitivity to Data Placement

In the previous experiments, we have considered two extreme data placement alternatives: either all joining relations are declustered over the same set of disks or all relations are declustered over disjoint sets of disks. For this next experiment, we explore the performance ramifications for left-deep and right-deep plans at more intermediate degrees of data placement overlap, i.e., when only a subset of the joining relations are declustered over the same disks.

All query and system parameters are equivalent to the partial declustering case outlined in Experiment 3, with the exception that instead of declustering the three, 1,000,000 tuple relations over separate disk sites they are declustered over the same set of 10 disk sites.

As expected, the left-deep scheduling algorithm is unaffected by the change in placement of the joining relations. Again, this is because the joining relations are scanned one at a time. However, RDHS suffered a performance loss of about 6% due to the change in data placement causing increased contention for the CPUs and disks during the building phase. The degradation would have been worse except for the fact that one of the million tuple relations was used as the probing relation and hence its placement had no effect on the response time of the building phase. The staticRD algorithm was unaffected by the change in data placement for this experiment because the placement of the breaks in the query plan were such that the overlapping building relations were in different sub-joins and thus did not compete for resources. Of course, if two overlapping building relations had been in the same sub-join, the degradation in performance would have been similar to that of RDHS.

For the next intermediate declustering experiment, we further modified the data placement strategy. Instead of storing the million tuple relations on the same set of ten disk sites, we placed the relations such that they only overlapped at two disk sites. The results from running the identical query in this new intermediate declustering environment are identical to the previous results. This is expected for the left-deep scheduling algorithm and the staticRD algorithm because they were unaffected by the previous change in the data placement strategy. However, RDHS was unaffected by this change in data placement because the speed of the entire building phase is determined by the slowest site. In the case of this partial overlap environment, the load on the two sites that contained the overlapping building relations was significantly higher than that of the other sites. Since the building phase cannot conclude until these two overloaded sites finish, these sites determined the overall response time.

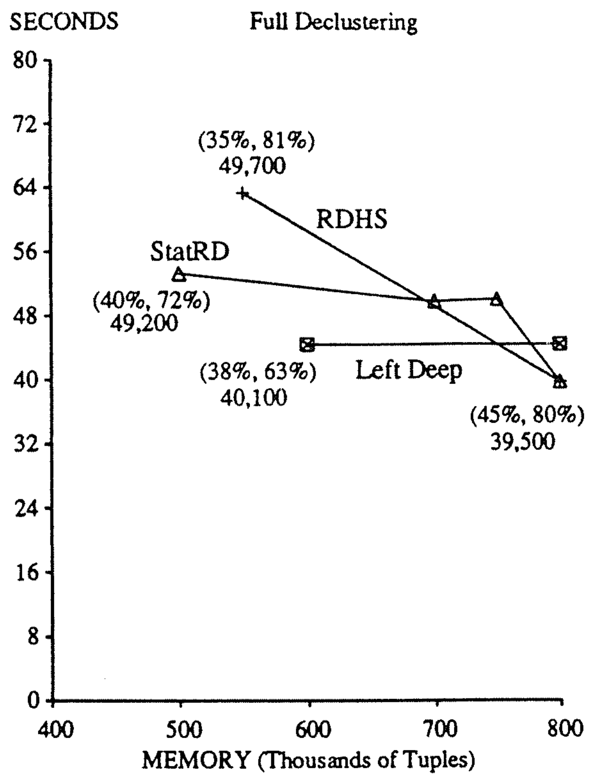
These intermediate declustering results show that an optimizer that produces right-deep query plans must be aware of where (relative to one another) the building relations of a join query are declustered in the database machine, while an optimizer for left-deep plans need not be concerned with this factor. This overlap factor is more important for RDHS than staticRD because **all** the building relations must be considered while with staticRD only those relations within the same "break" of the query plan compete for resources. These results also provide further evidence that optimizing fully-general bushy plans will likely be very difficult; besides the normal problems of synchronizing sub-joins, the resource contention of the various sub-joins being executed concurrently must be accounted for.

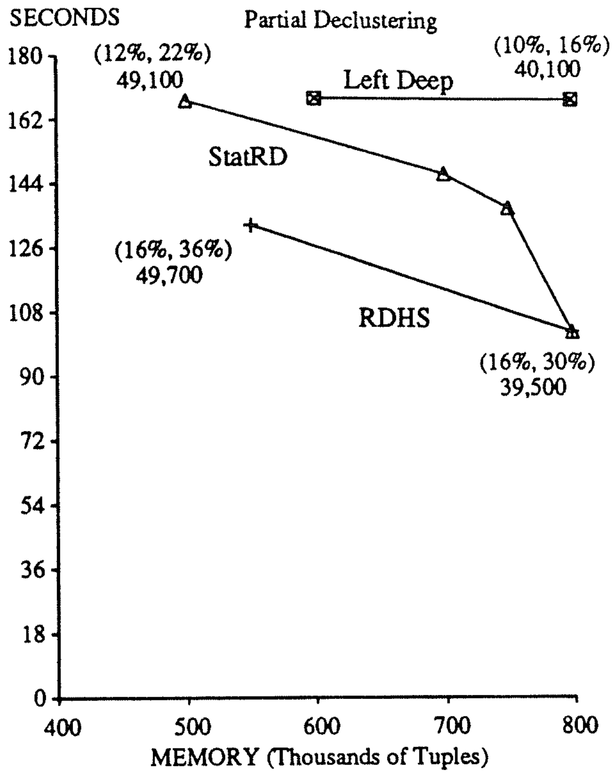
6.6. Experiment 5: High Join Selectivities

In this experiment, we explore the effect of changing the join selectivities on the performance of the left-deep and right-deep query processing algorithms. The basic query from Experiment 3 is again used, although the join selectivities are modified to generate a much larger output relation and intermediate relations. The output relation consists of 500,000 tuples and, because of the larger intermediate relations, the minimal memory requirements for the left-deep plan to avoid hash table overflow is 600,000 tuples (temporary join results of 200,000 and 400,000 tuples). The right-deep plan for the RDHS algorithm required that two buckets be used for the 1,000,000 tuple relation with the 50% scan selectivity. One bucket was used for the other three building relations, giving a memory requirement of 550,000 tuples.

In comparison to the full declustering results from Experiment 3 (see Figures 6.5 and 6.7), RDHS is now only a factor of 1.4 slower than the left-deep algorithm as opposed to a factor of 4.2 slower when memory consumption is equivalent. The change in join selectivities also allowed staticRD to achieve the same degree of memory consumption as the left-deep algorithm, although performance still lags. In the partial declustering case, RDHS went from being 10% slower to 21% faster (see Figures 6.6 and 6.8).

The right-deep scheduling algorithms benefited more from the larger join selectivities in both environments because the size of the intermediate relations determines the amount of memory consumption for the left-deep scheduling algorithm. When the size of these temporary results is increased, RDHS can increase the amount of each building relation that is staged in memory. That is, the number of buckets used to compute each join can be reduced. This has





High Join Selectivities - Full and Disjoint Declustering
Figures 6.7 and 6.8

the effect of reducing the fraction of each building relation that needs to be written to disk and later read. The staticRD algorithm also benefitted (relative to left-deep scheduling) from having a query with higher join selectivities because the number of times the query plan needed to be broken was reduced. For this particular query the query plan needed to be broken into two pieces. This reduced the minimal memory requirement to 500,000 tuples (less than that for the RDHS and left-deep scheduling algorithms). In the full declustering environment, staticRD outperformed RDHS by about 10%. This is to be expected because the results from Chapter 5 showed that staticRD performed better than RDHS when resource contention was high. However, the left-deep scheduling algorithm still outperformed staticRD by about 10%. In the partial declustering environment, staticRD took about the same amount of time as the left-deep scheduling algorithm when memory consumption was equivalent, and hence was slower than RDHS.

6.7. Summary

The experimental results in this chapter support the conclusion that left-deep plans are a more resource conservative optimization strategy while right-deep plans are more resource intensive. Under conditions of ample resources (CPU, disk I/O, and memory), the algorithms for processing right-deep plans generally outperform the left-deep scheduling strategy.

However, the experiments have shown that the right-deep plans are not the best choice under several different conditions. First, the performance of the right-deep processing algorithms is dependent on the physical placement of the relations to be joined. If several of the relations are declustered over the same set of disks, performance will suffer due to increased CPU and disk contention. Also, in an environment where memory is limited, right-deep plans should be avoided for queries with low join selectivities and several large relations without restrictive selection predicates. Left-deep plans are much better suited for this type of query. However, it is important to note that staticRD and RDHS benefit from having extra memory for joining. As long as the left-deep scheduling algorithm has enough memory to hold the results of any two adjacent join operators, its performance will not improve with additional memory. And finally, in general, a response time versus throughput tradeoff exists between the left-deep and right-deep algorithms. In general, the right-deep algorithms gain a response time performance advantage at the expense of potential throughput (based on CPU and disk utilizations).

These results also indirectly support the conclusion that optimizing and processing fully general bushy-plans in a multi-processor database machine is likely to be difficult. Because several sub-joins in a bushy-plan could be executing in parallel, the negative effects of data placement (where relations that are scanned concurrently share some disk sites) that afflicted the right-deep plans would affect the processing of bushy-plans. These data contention issues would make the synchronization of multiple sub-trees in a bushy-plan even more difficult to achieve during query processing.

CHAPTER 7

SUMMARY

7.1. Conclusions

In this dissertation, we have studied the problem of how to process large, ad-hoc join queries in a multiprocessor database machine environment. For queries that join only a few relations, we have found that the parallel Hybrid hash-join algorithm dominates under most circumstances, except when the join attribute values of the building relation are highly skewed.

We then extended the scope of the research problem to include queries that join on the order of 10 relations. Efficiently answering these more complex queries is becoming increasingly important as the demands of database users increase. Given such a complex query, the manner in which the resulting query plan is formatted has significant performance implications. In Chapter 4, we studied the tradeoffs between left-deep, right-deep, and bushy query plans. Through this analysis, we identified right-deep query plans as having the most potential to achieve high performance in a highly parallel environment. We proposed several algorithms for processing queries optimized into right-deep query plans.

In Chapter 5, we used both an analytical model and a simulation model to compare the performance of two of the right-deep query processing algorithms. We found that the staticRD algorithm outperformed the RDHS algorithm in a high resource contention environment and when the complexity of the query is relatively high. The RDHS algorithm performed well in a low resource contention environment and has the potential to deliver output tuples to the user/application much earlier and in a more constant stream than the staticRD algorithm.

Finally, we compared the performance of the two right-deep query processing algorithms with an algorithm that processes left-deep query plans. When ample resources were available (CPU, disk, and memory), the right-deep algorithms outperformed the left-deep algorithm. However, the right-deep algorithms were shown to be inferior under several different conditions. They are generally more resource intensive and when resource contention is high, performance suffers. Performance also suffers dramatically for queries that have several large relations, no highly restrictive selection predicates, and low join selectivities. Additionally, performance is more sensitive to the physical placement of the relations to be joined because of increased resource contention when the right-deep query algorithms attempt to use additional parallelism. As such, an optimizer for a highly parallel database machine should be capable of producing either a left-deep or a right-deep query plan depending on the query and the resource capabilities of the system.

7.2. Future Research Directions

Using the multiprocessor database machine simulator that we developed, we intend to explore a variety of issues. As left-deep query plans and right-deep query plans each have their strengths, we plan to develop and analyze a new subclass of query plans called **piecewise linear plans**. These plans are formed by "grafting" together pieces of left-deep plans and right-deep plans. Through this combination, piecewise linear plans will retain many of the advantages of right-deep plans while lessening the extent of some of the disadvantages. Additionally, by restricting the query plan format to a combination of linear strategies, simple and effective scheduling algorithms should be easily implementable; thereby eliminating the complexities of scheduling fully general bushy-trees.

Single-user performance evaluations are important because they delimit many of the performance differences between alternative algorithms and because many database systems are still used for batch-style processing. However, the trend is for more systems to be used in a multi-user environment. Although we did make throughput predictions based on CPU and disk utilizations, a thorough multi-user analysis should be conducted.

We also intend to analyze more closely the modification of query plans during execution. By making changes at runtime, more information will be available on system resources like disk, CPU and memory utilizations. Also, better knowledge will be available on query sizes and selectivities.

The performance analysis assumed that perfect information was available for all scan and join selectivities. In general, estimating join selectivities is hard, especially when the complexity of the query increases. We intend to explore how sampling techniques such as Adaptive Sampling [LIPT90a, LIPT90b] can be incorporated into both the right-deep and left-deep query processing algorithms. We are also interested in applying Adaptive Sampling to help tackle the problem of data skew.

REFERENCES

[BABA87] Baba, T., S. Yao, and A. Hevner, "Design of a Functionally Distributed, Multiprocessor Database Machine Using Data Flow Analysis", IEEE Transactions on Computers, Vol. C-36, No. 6, June 1987.

[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.

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

[BITT88] Bitton, D., and J. Gray, "Disk Shadowing", Proceedings of the 1988 VLDB Conference, August, 1988.

[BLAS77] Blasgen, M. W., and K.P. Eswaran, "Storage and Access in Relational Data Bases", IBM Systems Journal, No. 4, 1977.

[BRAT84] Bratbergsengen, Kjell, "Hashing Methods and Relational Algebra Operations", Proceedings of the 1984 VLDB Conference, August, 1984.

[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.

[CHOU85] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, "Design and Implementation of the Wisconsin Storage System (WiSS)", Software Practices and Experience, Vol. 15, No. 10, October, 1985.

[COPE88] Copeland, G., W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba", Proceedings of the 1988 SIGMOD Conference, 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, June, 1984.

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

[DEWI86] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K. and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," Proceedings of the 1986 VLDB Conference, 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, June 1988.

[DEWI90] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., and R. Rasmussen, "The Gamma Database Machine Project", IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March, 1990.

[GERB86] Gerber, R., "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," Ph.D. 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.

[GERB90] R. Gerber, Personal Communication, March 1990.

[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.

[GRAE89] Graefe, G. and K. Ward, "Dynamic Query Evaluation Plans", Proceedings of the 1989 SIGMOD Conference, May 1989.

[GRAE90] Graefe, G., "Encapsulation of Parallelism in the Volcano Query Processing System", Proceedings of the 1990 SIGMOD Conference, May 1990.

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

[HSIA90] Hsiao, Hui, Ph.D. thesis (to appear), University of Wisconsin-Madison, 1990.

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

[JARK84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," ACM Computing Surveys, Vol. 16, No. 2, June, 1984.

[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," Proceedings of the 5th International Conference on Data Engineering, 1989.

[LIPT90a] Lipton, R. and J. Naughton, "Query Size Estimation by Adaptive Sampling", Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1990.

[LIPT90b] Lipton, R., J. Naughton, and D. Schneider, "Practical Selectivity Estimation through Adaptive Sampling", Proceedings of the 1990 SIGMOD Conference, May, 1990.

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

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

[LORI88] Lorie, R., J. Daudenarde, G. Hallmark, J. Stamos, and H. Young, "Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience", RJ 6165, IBM Almaden Research Center, San Jose, California, March 1988.

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

[MURP89] Murphy, M. and D. Rotem, "Effective Resource Utilization for Multiprocessor Join Execution", Proceedings of the 1989 VLDB Conference, August, 1985.

[QADA85] G. Qadah, "The Equi-Join Operation on a Multiprocessor Database Machine: Algorithms and the Evaluation of Their Performance", Proceedings of the 1985 International Workshop on Database Machines, March, 1985.

[RIES78] Ries, D. and R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems," UCB/ERL Technical Report M78/22, UC Berkeley, May, 1978.

[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, June 1989.

[SCHN90] Schneider, D. and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines", Proceedings of 1990 VLDB Conference, August 1990.

[SELI79] Selinger,P. G., et. al., "Access Path Selection in a Relational Database Management System," Proceedings of the 1979 SIGMOD Conference, May 1979.

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

[STON88] Stonebraker, M., R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS", Proceedings of the Fourteenth International VLDB Conference, August, 1988.

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

[SWAM88] Swami, A., and A. Gupta, "Optimization of Large Join Queries", Proceedings of the 1988 SIGMOD Conference, June 1988.

[TAND88] Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction," Proceedings of the 1988 SIGMOD Conference, June 1988.

[TAY90] Y.C. Tay, "On the Optimality of Strategies for Multiple Joins", Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1990.

[TEOR72] Teorey T. J., and Pinkerton T.B., "A Comparative Analysis of Disk Scheduling Policies", Communications of ACM, 15:3, March 1972.

[TERA83] Teradata Corp., *DBC/1012 Data Base Computer Concepts & Facilities*, Teradata Corp. Document No. C02-0001-00, 1983.

[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.

[VALD87] Valduriez, P., "Join Indices", ACM Transactions on Database Systems, Vol. 12, No. 2, June, 1987.