

**A Performance Evaluation of Four Parallel Join Algorithms
in a Shared-Nothing Multiprocessor Environment**

Donovan A. Schneider
David J. DeWitt

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grants DCR-8512862, MCS82-01870, and MCS81-05904, and by a Digital Equipment Corporation External Research Grant.

Abstract

The join operator has been a cornerstone of relational database systems since their inception. As such, much time and effort has gone into making joins efficient. With the obvious trend towards multiprocessors, attention has focused on efficiently parallelizing the join operation. In this paper we analyze and compare four parallel join algorithms. Grace and Hybrid hash represent the class of hash-based join methods, Simple hash represents a looping algorithm with hashing, and our last algorithm is the more traditional sort-merge. The Gamma database machine serves as the host for the performance comparison. Gamma's shared-nothing architecture with commercially available components is becoming increasingly common, both in research and in industry.

1. Introduction

During the last 10 years, a significant amount of effort has been focused on developing efficient join algorithms. Initially, nested loops and sort-merge were the algorithms of choice. However, work by [KITS83, BRAT84, DEWI84] demonstrated the potential of hash-based join methods. Besides being faster under most conditions these hash-based join algorithms have the property of being easy to parallelize. Thus, with the trend towards multiprocessor database machines, these algorithms have received a lot of attention. In fact, several researchers, including [BARU87, BRAT87, DEWI87, DEWI88, KITS88], have presented performance timings for a variety of distributed join algorithms. However, the more popular parallel join algorithms have never been compared in a common hardware/software environment. In this paper we present a performance evaluation of parallel versions of the sort-merge, Grace [KITS83], Simple [DEWI84], and Hybrid [DEWI84] join algorithms within the context of the Gamma database machine [DEWI86, DEWI88, GERB86]. These algorithms cover the spectrum from hashing, to looping with hashing, and finally to sorting.

We feel that Gamma is a good choice for the experimental vehicle because it incorporates a shared-nothing architecture with commercially available components. This basic design is becoming increasingly popular, both in research, for example, Bubba at MCC [COPE88, BORA88] and in the commercial arena, with products from Tera-data [TERA83] and Tandem [TAND88].

The experiments were designed to test the performance of each of the join algorithms under several different conditions. First, we compare the algorithms when the join attributes are also the attributes used to distribute the relations during loading. In certain circumstances it is possible to exploit this situation and drastically reduce inter-processor communications. Next, the performance of each of the join algorithms is analyzed when the size of the relations being joined exceeds the amount of available memory. Our goal was to show how the performance of each algorithm is affected as the amount of available memory is reduced.¹ The join algorithms were also analyzed in the presence of non-uniform join attribute values. We also considered how effectively the different join algorithms could utilize processors without disks. Finally, bit vector filtering techniques [BABB79, VALD84] were evaluated for each of the parallel join algorithms.

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 the size of available memory.

The remainder of this paper is organized as follows. First, in Section 2, a brief overview of the Gamma database machine is presented. The four parallel join algorithms that we tested are described in Section 3. Section 4 contains the results of the experiments that we conducted. Our conclusions appear in Section 5.

2. Overview of the Gamma Database Machine

In this section we present a brief overview of the Gamma database machine. For a complete description of Gamma see [DEWI86, GERB86]. A performance analysis of Gamma can be found in [DEWI88].

2.1. Hardware Configuration

Currently², Gamma consists of 17 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring [PROT85] connects the processors to each other and to another VAX 11/750 running Berkeley UNIX. This processor acts as the host machine for Gamma. 333 megabyte Fujitsu disk drives (8") provide database storage at eight of the processors. One diskless processor is reserved for query scheduling and global deadlock detection. The remaining diskless processors execute join, projection, and aggregate operations. Selection and update operations execute only on the processors with attached disk drives.

2.2. Software Overview

Physical Database Design

In Gamma, all relations are **horizontally partitioned** [RIES78] across all disk drives in the system. Four alternative ways of distributing the tuples of a relation are provided: round-robin, hashed, range partitioned with user-specified placement by key value, and range partitioned with uniform distribution. 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. In the last partitioning strategy the user specifies the partitioning attribute and the system distributes the tuples uniformly across all sites.

² We are currently in the process of porting the Gamma software to an Intel iPSC-II Hypercube with 32 386 processors and 32 disk drives.

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 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. With the exception of these three control messages, 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.

To enhance the performance of certain operations, an array of bit vector filters [BABB79, VALD84] is inserted into the split table. In the case of a join operation, each join process builds a bit vector filter by hashing the join attribute values while building its hash table using the inner relation [BRAT84, DEWI84, DEWI85, VALD84]. When the hash table for the inner relation has been completed, the process sends its filter to its scheduler. After the scheduler has received all the filters, it sends them to the processes responsible for producing the outer relation of

the join. Each of these processes uses the set of filters to eliminate those tuples that will not produce any tuples in the join operation.

Operating and Storage System

The operating system on which Gamma is constructed provides lightweight processes with shared memory and reliable, datagram communication services using a multiple bit, sliding window protocol [TANE81]. 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⁺ indices, byte-stream files as in UNIX, long data items, a sort utility, and a scan mechanism.

3. Parallel Join Algorithms

We implemented parallel versions of four join algorithms: sort-merge, Grace [KITS83], Simple hash [DEWI84], and Hybrid hash-join [DEWI84]. Each of the algorithms share a **data partitioning stage** in which tuples from the joining relations are distributed across the available processors for joining. The actual join computation depends on the algorithm: hashing is used for the Grace, Simple, and Hybrid algorithms whereas sorting and merging is used for the sort-merge algorithm. Additionally, 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]. The degree of interaction between the bucket-forming phase and the joining phase differs between the Grace and Hybrid algorithms, though. 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 always the smaller of the two relations and is always the inner joining relation.

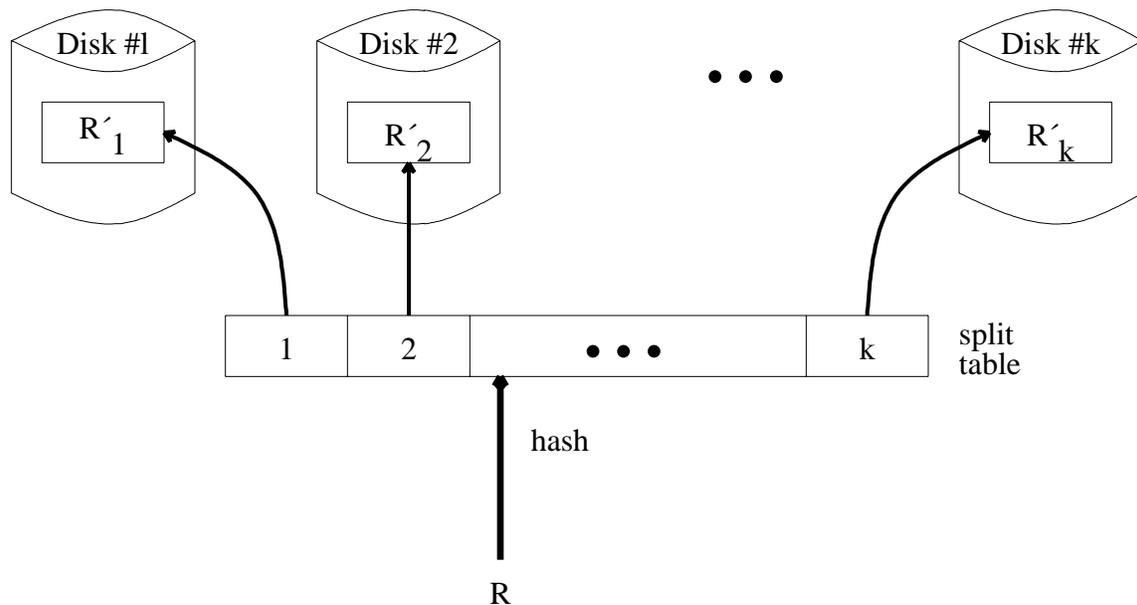
3.1. Sort-Merge

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 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 performed in parallel across the disk sites will fully compute the join. Of course, if we were required to present a sorted relation to a user, a final merge of the locally merged relation fragments would be necessary.

Although it is possible to send tuples to remote, diskless processors for merging, we felt the difficulties involved (especially when the inner relation contains tuples with duplicate attribute values and hence is required to backup) outweighed the potential benefits of offloading CPU cycles from the sites with disks. Thus, the join processors will always correspond exactly to the processors with disks.

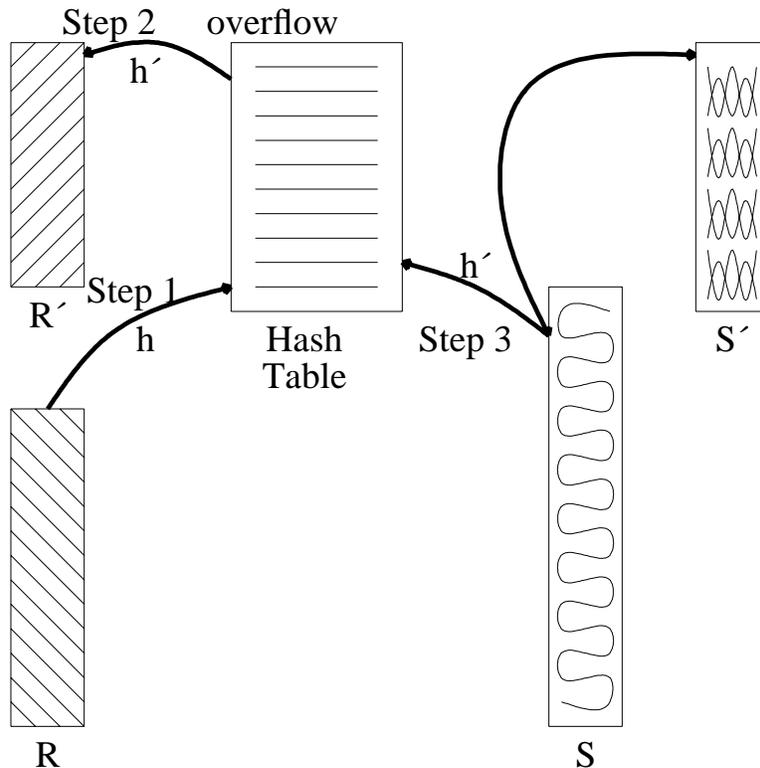
To increase intra-query parallelism, it would be possible to partition (or sort) both relations concurrently. However, this could cause performance problems due to disk head and network interface contention. The other problem is that since bit filters must be created from the complete inner relation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



Partitioning of relation R across K disk drives for sort-merge.
Figure 1

3.2. Simple Hash

A centralized version of the Simple hash-join [DEWI84] operates as follows. First, the smaller joining relation, R , is read from disk and staged in an in-memory hash table (which is formed by hashing on the join attribute of each tuple of R). Next, the larger joining relation, S , is read from disk and its tuples probe the hash table for matches. When the number of tuples in R exceeds the size of the hash table, memory overflow occurs. Figure 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 hash function, h' , until the tuples in R are distributed between R' and the hash table (step 2). The query scheduler then passes the function h' to the operator producing the tuples of the outer relation (i.e., relation S). In step 3, tuples from S corresponding to tuples in the overflow partition 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.



Simple hash-join overflow processing.

Figure 2

To parallelize this algorithm we inserted a split table in step 1 to route tuples 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 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), 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 approximately 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 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 the processors used for executing the join operation are not constrained to having disks as was the case for the sort-merge algorithm.

Until very recently, Simple hash was the only join algorithm employed by Gamma and is currently used as the overflow resolution method for our parallel implementations of the Grace and Hybrid algorithms.

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 .

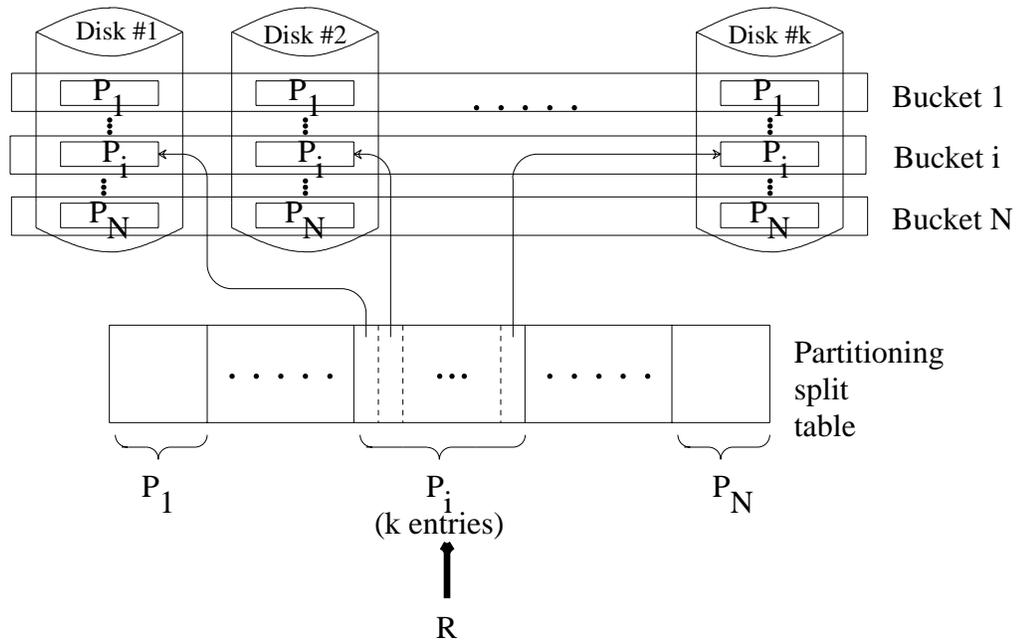
The number of buckets, N , is chosen to be very large. This reduces the chance that any bucket will exceed the memory capacity of the processors used to actually effect the join of two buckets. In the event that the buckets are much smaller than main memory, several will be combined during the third phase to form more optimal size 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 must be partitioned across all available disk drives. A **partitioning split table**, as shown in Figure 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 they probe the hash table for matches.

The bucket-forming phase is completely separated from the joining phase under the Grace join algorithm. This separation of phases forces the Grace algorithm to write both the joining relations tuples 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 in order to ensure that the size of each bucket is less than the aggregate amount of main-memory of the joining processors.

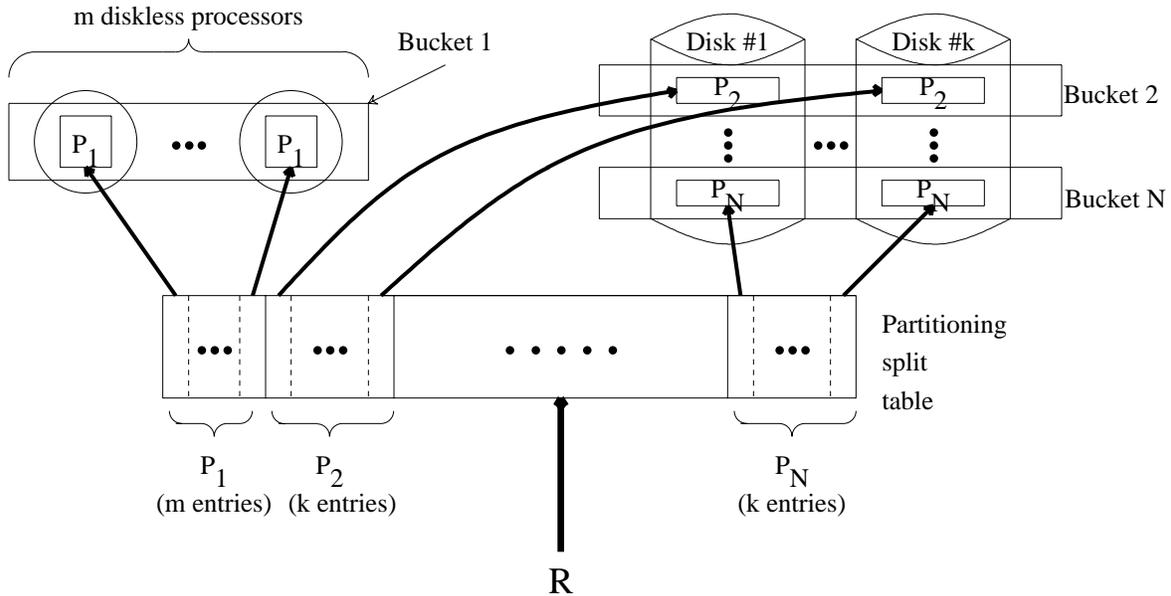


Grace Partitioning of R into N Buckets
Figure 3

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. Notice that 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 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.

The 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 buckets. 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 4 depicts relation R being partitioned into N buckets across k disk sites where the first bucket is to be joined on m diskless processors.



Partitioning of R into N buckets for Hybrid hash-join.
Figure 4

4. Experimental Results

We tested the various parallel algorithms under several conditions. First, the performance of the algorithms are studied when the attributes used to partition the two relations being joined are also used as the joining attributes. Next, we wanted to compare the effects of bit vector filtering on the different algorithms. Third, the use of diskless processors by each of the hash-based algorithms is explored. Finally, the impact on non-uniformly distributed join attribute values on the performance of each of the algorithms is studied.

The benchmark relations are based on the standard Wisconsin Benchmark [BITT83]. Each relation consists of thirteen 4-byte integer values and three 52-byte string attributes. Thus, each tuple is 208 bytes long. Except where noted otherwise, hashing on the unique1 attribute was used to determine each tuple's destination site during loading of the database. The benchmark join query is joinABprime, which joins a 100,000 tuple relation (approximately 20 megabytes in size) with a 10,000 tuple relation (approximately 2 megabytes in size) and produces a 10,000 tuple result relation (over 4 megabytes in size). We ran the experiments with the other benchmark join queries, joinAselB and joinCselAselB, but the trends were the same so those results are not presented. 8 kilobyte disk pages were used in all experiments. The **default** hardware environment consists of eight processors with disks and a single diskless processor reserved for query scheduling and global deadlock detection. We refer to this configuration as **local** because joins will be performed on the sites with attached disks. In Section 4.3, we explore

the performance of the Simple, Grace, and Hybrid algorithms when 8 diskless processors are used. This configuration is termed **remote**.

Join performance (for each of these parallel join algorithms) is very sensitive to the amount of available memory relative to the size of the joining relations. In designing the set of experiments described below, one of the first decisions we had to make was how to capture this aspect of the performance of the different algorithms. One approach would have been 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 memory available. We rejected the first choice (and hence accepted the second alternative) 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 on the joining processors that is available for use to effect the join. In the case of the hash-based join algorithms, this memory is used to construct an in-memory hash table. For 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 4.1 we analyze the performance of the Grace and Hybrid join algorithms at data points not corresponding to an integral number of buckets.

4.1. Partitioning vs. Non-Partitioning Attribute Joins: Local configuration

In Gamma and Teradata, when a relation is created, the database administrator is required to specify a partitioning or "key" attribute and, in the case of Gamma, a partitioning strategy: round-robin, hashed, or range (see Section 2 for more details). If two relations being joined are both hash-partitioned on their joining attributes, the partitioning phase of each of the parallel algorithms described in the previous section is no longer necessary and can be eliminated. Rather than special-case the system to handle a join operation on the partitioning attribute, Gamma relies on its operating system to "short-circuit" packets between two processes on the same machine and the design

of the partitioning split table to maximize the extent to which tuples are mapped to hash-join buckets at the same processor.

Figures 5 and 6 display the execution times of the joinABprime query as a function of the amount of memory available relative to the size of the inner relation when the join attributes are not the partitioning attributes (Figure 6) and when they are (Figure 5). These tests were conducted using the "local" configuration of processors in which the joins are executed only on the 8 processors with disks. Several points should be made about these graphs. First, when the smaller relation fits entirely in memory, 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 joins sites for processing (where it will turn out that 1/2 of the tuples belong to the overflow partition - see Section 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 the percentage of available memory relative to the size of the joining

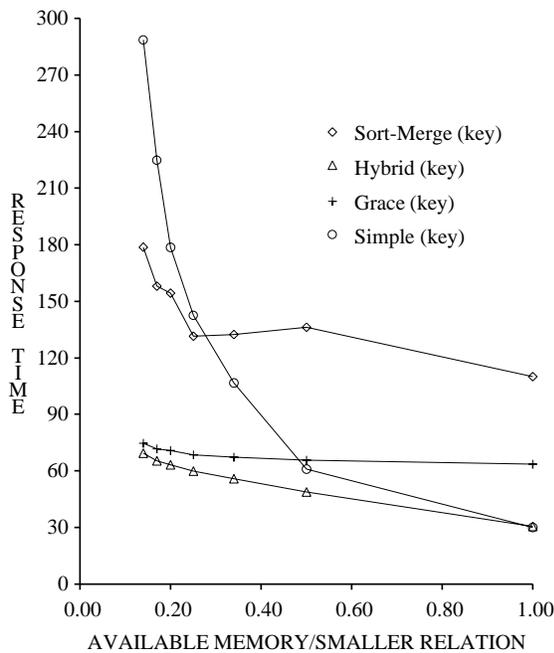


Figure 5
Partitioning Attributes Used as Join Attributes
8 Processors with Disks

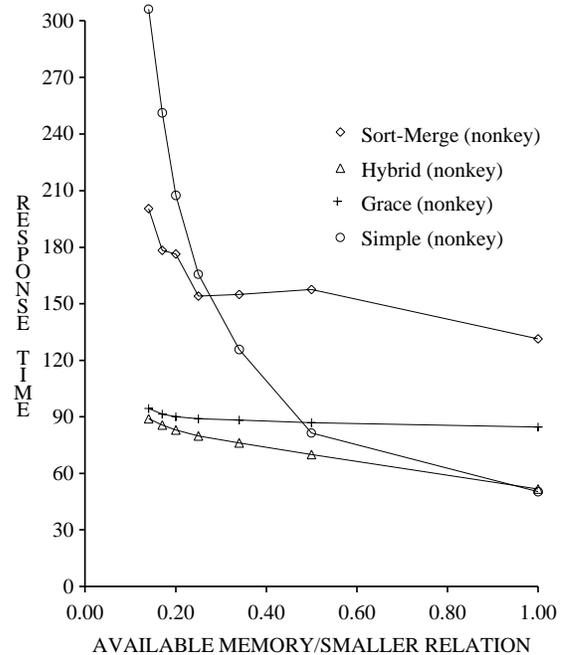


Figure 6
Partitioning Attributes **Not** Used as Join Attributes
8 Processors with Disks

relations is large. 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 extra rise in the curves that occurs when memory is most scarce happens because the partitioning split table exceeds the network packet size (2K) and hence must be sent in pieces.

For both Figures 5 and 6, 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 dominated by 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. An interesting feature of the sort-merge curves is the drop in response time as the memory ratio is decreased from 0.5 to 0.25. Although we are entirely not sure why this phenomenon occurs, we hypothesize that, while the number of merge passes required for sorting the larger joining relation is constant over this memory range, adding additional sort buffers really just adds processing overhead.

It is important to point out that the trends observed in these graphs and their general shape are very similar 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 is interesting to note that the corresponding curves in Figures 5 and 6 differ by a constant factor over all memory availabilities tested. This response time difference turns out to be the difference in shortcircuiting the entire distribution of R and S for partitioning attribute joins as opposed to only shortcircuiting 1/8th of these tuples for non-partitioning attribute joins. For all algorithms, the time to store the final join result is identical because a round-robin tuple distribution strategy is used to store the result tuples from the query. Thus, each algorithm will shortcircuit 1/8th of the result tuples.

With sort-merge, since the attribute values are identical regardless of whether they are partitioning attributes or not, the time to sort and merge will be constant. However, when relations R and S are partitioned across the joining sites, partitioning attribute joins will shortcircuit all tuples in both R and S. Non-partitioning attribute joins, however, will only shortcircuit 1/8th of the tuples. This shortcircuiting accounts for the constant response time difference. It should be noted, however, that the sort-merge times are unnecessarily high when the smaller relation fits entirely in memory. The current implementation doesn't recognize this fact and hence the smaller relation is written back to disk. (We have measured the time to write and then later read this relation as approximately 3 seconds.)

Simple hash will experience the same effects during the joining phase as the set of joining processors is the same as with sort-merge. However, one might have expected that partitioning attribute joins would increasingly reap the benefits of shortcircuiting when processing join overflows. These savings never materialize because the hash function is changed after each overflow, thus converting partitioning attribute joins into non-partitioning attribute joins. This hash function modification is necessary in order to efficiently process non-uniform data distributions. Consider the case where only a subset of the joining sites overflow. If the same joining hash function is used when the tuples from the overflow partition are read from disk and re-split, all the overflow tuples will re-map to the same overflowing processors. Unfortunately, this will leave the non-overflowing processors idle and perhaps more importantly, their associated local memories unused.

The explanation for Grace joins is slightly more complicated. Independent of the amount of memory available, partitioning of R and S into buckets will be completely shortcircuited for partitioning attribute joins while only 1/8th of the tuples will be shortcircuited for non-partitioning attribute joins. Since this accounts for the total response time difference between partitioning and non-partitioning attribute joins, this implies that the joining phases for the partitioning and non-partitioning attribute joins must be identical, and furthermore that all joining tuples will shortcircuit the network. Why is this the case? We can argue intuitively that this should be the case in a good implementation. The important feature of using hashing to execute join operations is that the join is broken into buckets, where tuples within each bucket of the outer relation can only possibly join with tuples in the corresponding bucket of the inner relation. Obviously, this is what Grace joins do; tuples in bucket j of R are only compared with tuples in bucket j of S. But this technique can be applied to a finer level than buckets. In Figure 3, one saw that each bucket is composed of a number of individual fragments. Since these fragments are created by hashing (via the partitioning split table), we are only required to join tuples in fragment i of bucket j of R with tuples

from fragment i of bucket j of S , where i ranges from 1 to the number of disk sites and j ranges from 1 to N . Since the corresponding fragments of buckets j of R and S reside on the same disk (since the same partitioning split table is used for R and S), there is no need to redistribute these tuples during a join when only processors with disks are used to perform the join operation. Thus, non-partitioning attribute joins will perform exactly as partitioning attribute joins after the initial bucket-forming partitioning phase has been completed.

The performance difference between partitioning and non-partitioning attribute Hybrid joins is easy to explain given their similarity to Grace joins. With the "Local" configuration, a N -bucket Hybrid join will have a partitioning split table identical to that of a N -bucket Grace join with the exception that tuples in the first bucket will be joined immediately instead of being stored in temporary files. Since the joining split table is identical for both algorithms, all tuples will shortcircuit the network during the processing of all N buckets (as was the case with Grace joins). Thus, the difference in execution times is due to the cost of partitioning R and S into buckets when the join attributes are the partitioning attributes and when they are not.

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 Figures 5 and 6, the curves should actually be step functions. Alternatively, we could have chosen to let the algorithms overflow at non-integral memory availabilities and use the Simple hash-join algorithm to process the overflow. Basically 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 7 presents a more detailed examination of the performance of the Hybrid join algorithm between the memory ratios of 0.5 and 1.0³. 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

³Only the results for the partitioning attributes being equal to the joining attributes are presented because the trends are identical for the non-partitioning attributes. This snapshot of performance will also show the expected behavior between the other plotted points on the graphs.

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 is used to process 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 ranges of possible 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 in Section 3). We currently try to 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

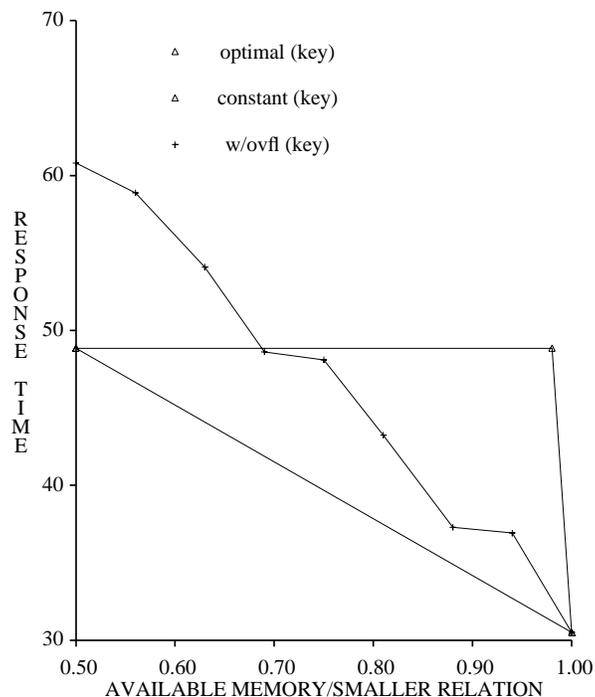


Figure 7
8 Processors with Disks

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 actually 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 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 7 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 because the heuristic forces more than 50% of the tuples to be written to the overflow file (hence incurring additional disk I/Os). Finally, there are the network protocol costs associated with sending these tuples to the join site and returning all tuples that will not fit to the overflow files (for these "Local" joins the transmission of the overflow tuples are all shortcircuited but the protocol cost cannot be ignored). The step-like function occurs between points where the number of passes through the hash table was identical. Obviously these results demonstrate that a tradeoff exists between being pessimistic and increasing the number of buckets and being optimistic and counting on Simple hash-join to resolve any memory overflow.

4.2. Multiprocessor Bit Vector Filtering

In the next set of experiments we ran the joinABprime tests using bit filters [BABB79, VALD84]. In each of the parallel join algorithms, bit filtering of tuples is only applied during the joining phase. With sort-merge joins, 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 for the inner relation are being inserted into the hash tables. With the Simple hash-join algorithm this means that as the number of overflows increases, the opportunities for filtering out non-joining tuples increases (because each overflow is treated as a separate join). With Grace and Hybrid joins, each bucket is treated as a separate join. Thus, separate bit filters 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.

Figures 8 and 9 show the results, respectively, for partitioning and non-partitioning attribute joins with bit filtering applied. Again, these tests were conducted using only the 8 CPUs with disks. Notice that the relative

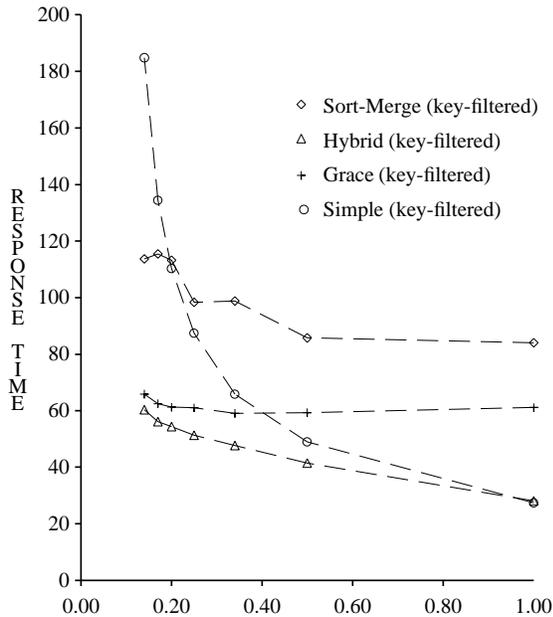


Figure 8
Partitioning Attributes Used as Join Attributes
8 Processors with Disks

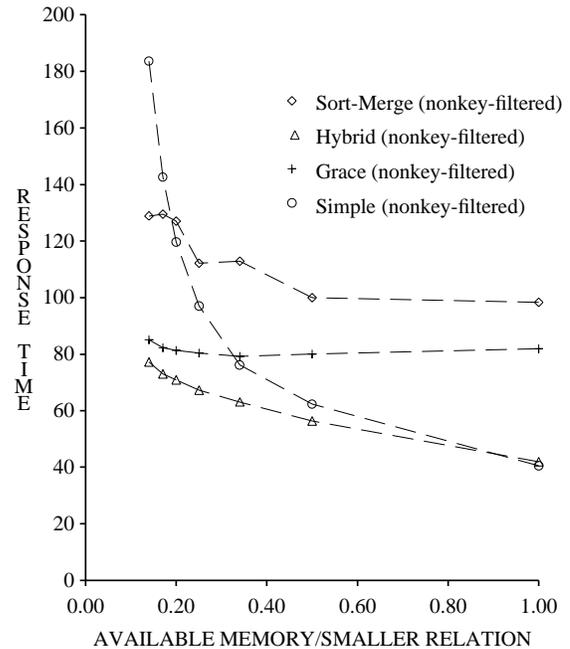


Figure 9
Partitioning Attributes **Not** Used as Join Attributes
8 Processors with Disks

positions of the algorithms have not changed, only the execution times have dropped in comparison to the results shown in Figures 5 and 6. The performance improvements from bit filtering for each individual algorithm are shown in Figures 10 through 13. An interesting effect to note is the shape of the curves for the Grace join algorithm as memory is reduced. Execution time actually **falls** until the available memory ratio reaches 25% (4 buckets) and then begins to rise. This is explained by our implementation of bit filtering. With 100% available memory, all 10,000 tuples (approximately 1,250 at each site) will attempt to set a bit in the filter. Since, Gamma currently uses only a single 2Kbyte packet for a filter (which is shared across all 8 joining sites - yielding 1,973 bits/site after overhead), most bits are set and hence the effectiveness of the filter is low. Thus, only a small percentage of the probing tuples are eliminated by the filter. However, as memory is decreased and the number of buckets increases a separate 2K filter is used for the join of each bucket. With two buckets, the gains from eliminating additional probing tuples exceeds the cost of scheduling the extra bucket and response time drops. This effect keeps occurring until four buckets are used and all non-joining tuples have been eliminated by filtering.

The Hybrid join algorithm experiences the same bit filtering effects but the result is not as obvious because as memory is reduced the amount of disk I/O increases. Thus it is harder to isolate the effects of bit filtering than

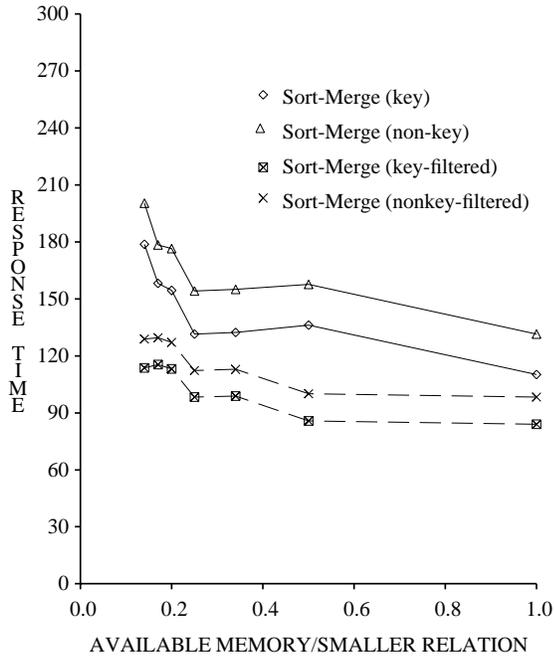


Figure 10
8 Processors with Disks

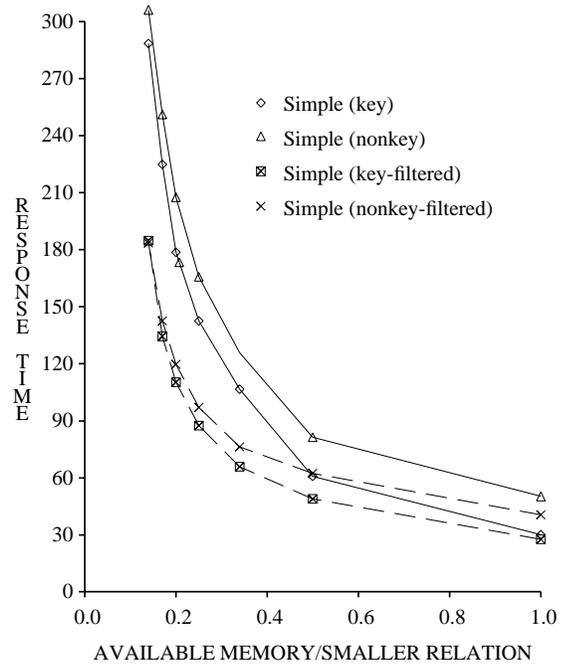


Figure 11
8 Processors with Disks

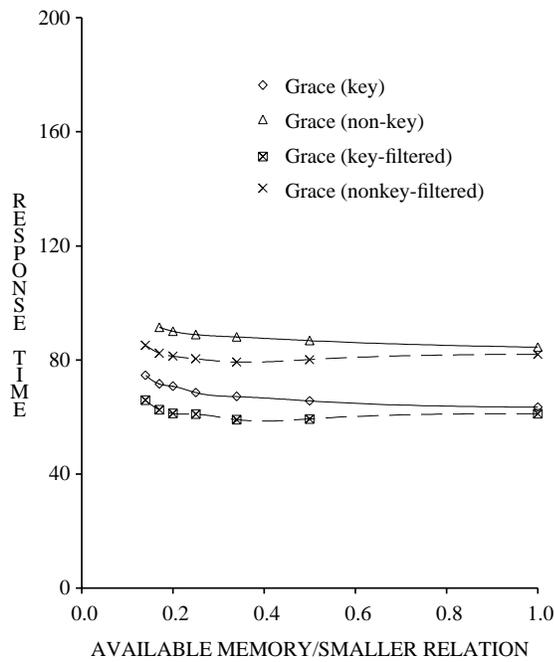


Figure 12
8 Processors with Disks

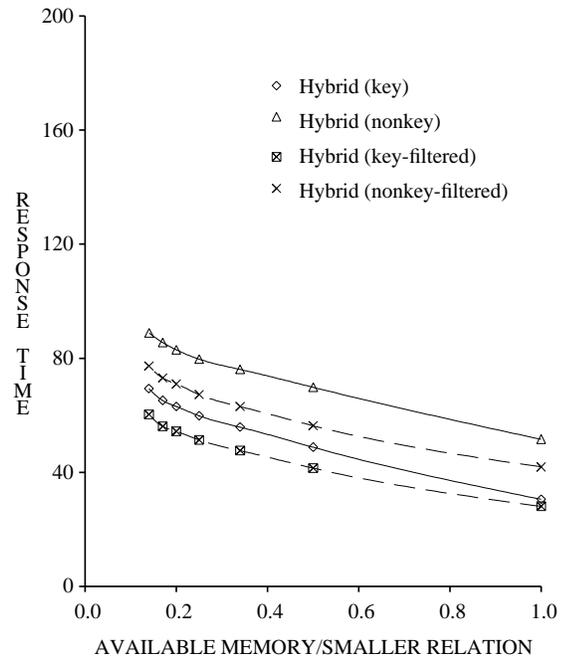


Figure 13
8 Processors with Disks

with the Grace algorithm.

A similar effect occurs for Simple hash-join. As memory is reduced (and overflows occur more frequently) more tuples are eliminated by the bit filters. As you can see, large bit filters are necessary for low response times for Simple hash-join. Eliminating non-joining tuples early will avoid writing these tuples to disk and later reading them multiple times.

The performance of the sort-merge join algorithm also benefits significantly from the use of bit filters, even though only a single 2K filter is used. Tuples of the outer relation (S in our case), 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 each of these join algorithms.

4.3. Remote Joins : Partitioning vs. Non-Partitioning Attributes

Since Gamma can use diskless processors for performing join and aggregate operations, the goal of the tests described in this section was to explore the effect of using diskless processors on the performance of the different join algorithms. The sort-merge algorithm has been excluded from this section because our current implementation of this algorithm cannot utilize diskless processors.

Although Gamma is capable of executing a join operation on a mix of processors with and without disks, earlier tests for the simple hash-join algorithm [DEWI88] indicated the performance of such a configuration was almost always 1/2 way between that of the "local" and "remote" configurations. Thus, for the following experiments, 8 processors with disks were used for storing the relations and 8 diskless processors performed the actual join computation. The Hybrid, Simple, and Grace algorithms were tested using this "remote" configuration and the results for the joinABprime query are displayed in Figure 14.

For the Grace algorithm, the difference in execution time between when the join attribute is also the partitioning attribute and when it is a different attribute is constant over the entire range of memory availability. This occurs because the execution time of the bucket-forming phase is constant regardless of the number of buckets, i.e., both source relations are completely written back to disk. The response time difference between the partitioning and non-partitioning attribute cases reflects the savings gained by shortcircuiting the network during the bucket-forming phases for partitioning attribute joins. Shortcircuiting also explains the spreading difference shown for Hybrid joins. With 100% available memory, all tuples will be shipped remotely during the processing of the join, for both types of joins. However, when only half the required memory is available, partitioning attribute joins will write half the building and probing tuples to disk **locally** during the partitioning phase of the algorithm, thus

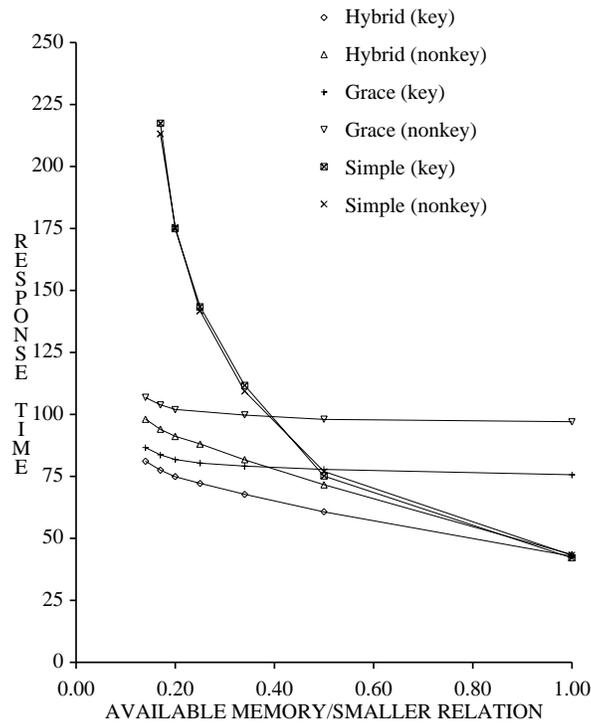


Figure 14
Joins Executed on 8 Diskless Processors

shortcircuiting the network. Non-partitioning attribute joins will write only 1/16th of these same tuples locally. Table 1 presents the difference in the percentage of local writes for partitioning and non-partitioning attribute joins. Analyzing this table shows that as the number of buckets increases (ie. memory is reduced), the relative savings of local writes for partitioning attribute joins increases over that for non-partitioning attribute joins.

The fact that non-partitioning attribute joins perform the same as partitioning attribute joins for Simple

Join Type	Number of Buckets						
	1	2	3	4	5	6	7
Partitioning Attr	0	1/2	2/3	3/4	4/5	5/6	6/7
Non-partitioning Attr	0	1/16	2/24	3/32	4/40	5/56	6/64
Difference	0	.4375	.5827	.6563	.7167	.7440	.7634

Percentage of local writes during redistribution
for partitioning and non-partitioning attribute joins
Table 1

hash-join may seem puzzling at first. At the memory ratio of 1.0 they are identical for exactly the same reasons as with Hybrid (recall Hybrid and Simple are identical at this point). As memory is reduced the curves do not spread as with Hybrid because the hash function is changed after the first overflow, hence turning all joins into non-partitioning attribute joins⁴.

Local vs. Remote Joins: Partitioning Attributes

A comparison of the performance of "local" and "remote" joins for partitioning attributes is shown in Figure 15. With Grace joins, the cost of partitioning is constant regardless of where the join processing is performed and, since these are partitioning attribute joins, partitioning will shortcircuit all tuples. However, during the joining phase, tuples will be distributed to the join processors for building and probing. When joins are done locally, all these tuples will shortcircuit the network (in addition to 1/8th of the result tuples). With remote-join processing, none of these shortcircuiting benefits will be realized during the joining phase. The same explanation describes the observed Hybrid join performance.

The crossover in the response time graphs for the Simple hash-join algorithm as the available memory ratio is decreased is the result of switching the hash functions after each overflow. Recall that the Hybrid and Simple join algorithms are equivalent when enough memory is available to hold the smaller relation. As discussed in the previous section, Hybrid joins can be done faster locally than remotely when the join attributes are also the partitioning attributes. In the following section we show, however, that just the opposite is true for these same joins done on their non-partitioning attributes when 100% memory is available. The crossover occurs because as more overflows occur, the Simple hash-join becomes more and more like a non-partitioning attribute join and hence performance degrades. These results support those reported in [DEWI88] for Gamma using 4 kbyte disk pages.

Local vs. Remote Joins: Non-Partitioning Attributes

When non-partitioning attribute joins are used as the joining attribute, the results presented in the previous sub-section no longer hold. Figure 16 shows the performance of the three hash-join algorithms on non-partitioning attributes for both the "local" and "remote" configurations. We consider Grace joins first. As Figure 16 demonstrates, local joins have superior performance to remote joins by a constant margin over the entire range of available

⁴See Section 4.1 for the motivation behind this change in hash functions.

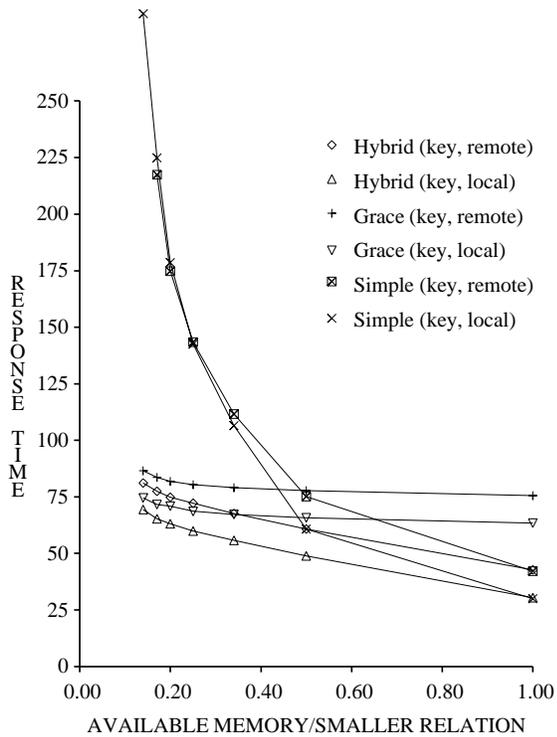


Figure 15
Partitioning Attributes Used as Join Attributes

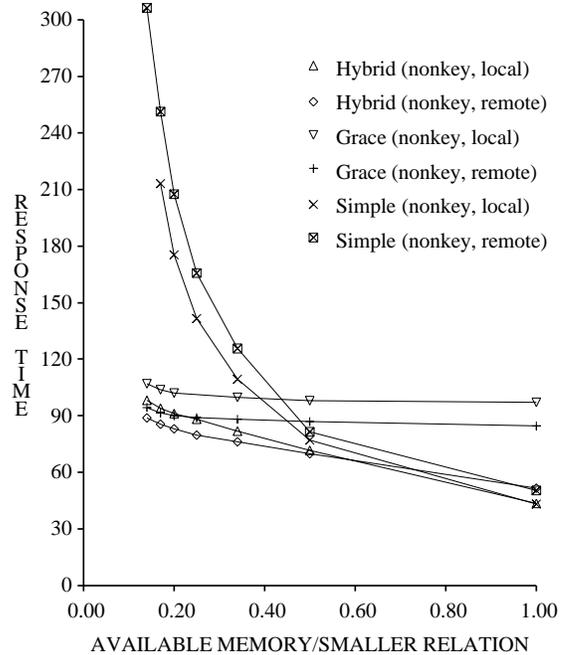


Figure 16
Partitioning Attributes **Not** Used as Join Attributes

memory. Since the bucket-forming phase of the Grace algorithm involves only the processors with disks, the execution time of this phase is constant regardless of what processors are used for the actual join operation. At a memory ratio of 1.0, one would then expect the join phase of the Grace algorithm to behave exactly like that of a one-bucket Hybrid join. But, as you can see, Hybrid executes joins on diskless processors faster at this memory ratio. What is the cause of this inconsistency? Recall from Section 4.1 that with the Grace algorithm non-partitioning attribute joins effectively become partitioning attribute joins after the bucket-forming phase. Thus, the results obtained for partitioning attribute joins that were described in the previous section and displayed in Figure 15 can be directly applied in this case. As one can see, the same difference between local and remote response time exists, although the non-partitioning attribute join times are slower than the partitioning attribute join times because of the increased costs of bucket-forming.

The discussion contained in Section 4.1 also explains the performance of the Hybrid algorithm. At 100% memory availability, the bucket-forming and joining phases are completely overlapped. Since these experiments involve non-partitioning attribute joins, tuples will be distributed across all the joining processors resulting in sub-

stantial network traffic. Remote processing wins in this case because, since the joining relations tuples need to be distributed across the processors anyway, the CPU cycles necessary for building and probing the hash tables can be successfully offloaded to the diskless processors. But, as memory is reduced "Local" joins become better and better. The partitioning phenomena discussed for Hybrid joins in Section 4.2 explains this shift in performance. At a memory availability of 50%, half of the tuples being joined will be written to disk. When this disk bucket is subsequently joined it will effectively become a partitioning attribute join. As memory is further reduced a larger fraction of the join becomes like a partitioning attribute join and, as illustrated by Figure 15, partitioning attribute joins can be executed faster locally than remotely. This explains why the curves crossover and the difference widens as memory is reduced.

Performance of the Simple hash-join algorithm is easy to explain. It is fastest with remote processors at 100% memory availability for the same reasons given for Hybrid. However, it doesn't crossover like Hybrid because its writing and subsequent reading of overflow files never benefits from shortcircuiting as the modification of the hash function after each hash table overflow eliminates the possibility of ever achieving the performance observed with partitioning attribute joins.

4.4. 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 the non-uniform data distributions we varied the distribution of the two join attribute values independently. Figure 17 shows these 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 building (probing) relation. U = Uniform distribution and N = Non-uniform distribution.

The response time of the joinABprime query is again used as the performance metric for this set of experiments. Recall that this query joins a 100,000 tuple relation (20 megabytes) with a 10,000 tuple relation (2 megabytes). The 10,000 tuple relation is the inner (building) relation and the 100,000 tuple relation is the outer (probing) relation. For the non-uniform distribution we chose the **normal** distribution with a mean of 50,000 and a standard deviation of 750. These parameters resulted in a highly skewed distribution of values. In fact, in the 100,000 tuple relation one attribute value was duplicated in 77 tuples. Furthermore, 12,500 tuples had join attribute values in the range of 50,000 to 50,243. The 10,000 tuple relation was created by randomly selecting 10,000 tuples from the

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

Figure 17

100,000 tuple relation. Thus the 10,000 tuple relation's primary key had values uniformly distributed from 0 to 99,999. Also, the normal attribute has the same characteristics for its distribution for the 10,000 tuple relation as it does for the 100,000 tuple relation.

In an attempt to make the initial scan of the two relations being joined as balanced as possible, we distributed each of the relations on their join attribute by using the range partitioning strategy provided by Gamma. This resulted in an equal number of tuples on each of the eight disks.

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 is sufficient **only** if the join tuples are distributed evenly across the joining processors. Eight processors with disks were used for these experiments.

Table 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 10,000 tuples as did the NU joins. The UN joins produced 10,036 result tuples. Results for the NN joins are not presented as the result cardinality for this query was 368,474 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 will analyze the performance of each of the join algorithms with the addition of bit vector filters in the next subsection.

Algorithm	100% memory			17% memory		
	UU	UN	NU	UU	UN	NU
Hybrid w/filter	51.48	51.37	57.27	85.74	88.87	111.36
	36.84	37.74	30.04	73.60	78.20	89.39
Grace w/filter	84.13	84.83	84.53	92.90	95.66	97.89
	79.37	79.93	76.07	83.20	87.28	87.89
Sort-Merge w/filter	167.44	163.81	149.45	210.91	211.18	195.78
	101.27	101.45	66.24	125.06	127.07	70.91
Simple ⁵ w/filter	51.13	51.63	57.76	251.37	246.84	259.30
	36.57	38.00	30.53	167.08	164.41	121.65

Join results with non-uniform join attribute value distributions.

(All response times are in seconds)

Table 2

We begin by analyzing the effects of using normally 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 buckets. 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 building relations tuples **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.

Table 2 shows that NU is indeed slower than UU for these 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 one additional bucket so that no memory overflow would occur. At 100% memory availability, the Hybrid algorithm processes the overflow fairly efficiently. However, as memory is reduced, each

⁵Recall that the Simple and Hybrid hash-join algorithms are identical at 100% memory availability.

bucket-join in the Hybrid algorithm experiences overflow. As shown by the 17% memory availability results, the cost of overflow resolution becomes significant.

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 would 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 53,071) the merge phase does not need to read all of the outer (100K) 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 as significant because only part of the inner (10K) relation can be skipped from reading; all the 100K 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 no noticeable effects are shown (except for sort-merge as discussed above). However, as memory is reduced small differences appear with the Hybrid and Grace join algorithms. 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.

Bit Vector Filtering

As expected, Table 2 shows that the application of bit vector filters improves the performance of each of the parallel join algorithms. Table 3 presents the percentage improvement gained by using bit filters for each of the join algorithms. As shown, the sort-merge and simple join algorithms experience the greatest improvement from bit filtering. This occurs because filtering tuples eliminates a large number of disk I/Os. The Grace join algorithm experiences the worst speedups because filtering is only applied during the joining phase and not the bucket-forming phase. Thus, no disk I/O is eliminated by using bit filters with this algorithm. As was stated in Section 4.2, bit filtering techniques should be extended to the bucket-forming phase.

Within each join algorithm the NU join type experienced the greatest improvements from filtering because the normally distributed attribute values resulted in more collisions when setting bits in the bit filter. Thus, fewer bits were set in the filter. This has the effect of screening out more outer (probing) tuples. Sort-merge has the best filtering improvement for NU because the better filtering allowed one less sorting pass of the outer relation.

The Grace join algorithm at 17% memory availability is the only exception to NU joins experiencing a significant performance improvement. The cause of this reduced speedup can be traced to the addition of the extra bucket for this join. Recall from Section 4 that increasing to seven buckets from six buckets caused extra overhead to be incurred because the network packet size was exceeded. The extra scheduling overhead necessary to send query packets offsets any bit filter benefits gained by the non-uniform join attribute values.

Algorithm	100% memory			17% memory		
	UU	UN	NU	UU	UN	NU
Hybrid	28.4%	26.5%	47.6%	14.2%	12.0%	19.7%
Grace	5.7%	5.8%	10.0%	10.4%	8.8%	10.2%
Sort-Merge	39.5%	38.1%	55.7%	40.7%	39.8%	63.8%
Simple	28.5%	26.4%	47.1%	33.5%	33.4%	53.1%

Percentage improvement using bit vector filters.
Table 3

5. Conclusions and Future Work

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 provide a twist in the relative performance of the join algorithms. Performance of the Hybrid join algorithm degrades when the join attribute values of the inner relation are non-uniformly distributed. In this case, the optimizer should choose to run the Hybrid algorithm with many extra buckets to ensure that no bucket will exceed memory capacity or it should switch to the Grace join algorithm. The sort-merge join algorithm, being a conservative algorithm, provides very stable performance with regard to non-uniform join attribute value distributions. We find it very encouraging that the Hybrid join algorithm still performs best when the joining attribute for 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.

From the results presented, one might be tempted to conclude that using remote processors for executing join operators is not a good idea except when the Hybrid algorithm is used with sufficient available memory and the join is over two non-partitioning attributes. This may not be as restrictive as it seems as a relation can only have one partitioning attribute. Hence, non-partitioning attribute joins may be fairly likely. In addition, when Gamma processes joins "locally", the processors are at 100% CPU utilization. However, when the "remote" configuration is used, CPU utilization at the processors with disk drops to approximately 60%. Thus, in a multiuser environment, offloading joins to remote processors may permit higher throughput by reducing the load at the processors with disks. We intend on studying the multiuser tradeoffs in the near future.

Several opportunities exist for expanding on the work done. For instance, currently only a single join process operates on a processor at a time. We would like to look at increasing the amount of intra-query parallelism. An important point where this would immediately be useful is when the Optimizer Bucket Analyzer detects a potential distribution problem. Instead of increasing the number of buckets, which is potentially very expensive under Hybrid joins, we would also consider increasing the number of join processes. Dynamically adjusting the number of join processes depending on load information would also be very interesting.

Another opportunity for intra-query parallelism is to send buckets to remote disks in large systems. Several buckets could be joined in parallel then.

6. References

- [BABB79] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware" ACM Transactions on Database Systems, Vol. 4, No. 1, March, 1979.
- [BARU87] Baru, C., O. Frieder, D. Kandlur, and M. Segal, "Join on a Cube: Analysis, Simulation, and Implementation", **Database Machines and Knowledge Base Machines**, M. Kitsuregawa and H. Tanaka (eds), Kluwer Academic Publishers, 1987.
- [BITT83] Bitton, D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, October, 1983.
- [BORA88] Boral, H. "Parallelism in Bubba," Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, December, 1988.
- [BRAT84] Bratbergsengen, Kjell, "Hashing Methods and Relational Algebra Operations" Proceedings of the 1984 Very Large Database 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, Chicago, IL, June 1988.
- [DEWI84] DeWitt, D. J., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and D. Wood, "Implementation Techniques for Main Memory Database Systems," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.
- [DEWI85] DeWitt, D., and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," Proceedings of the 1985 VLDB Conference, Stockholm, Sweden, August, 1985.
- [DEWI86] DeWitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K. and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," Proceedings of the 1986 VLDB Conference, Japan, August 1986.
- [DEWI87] DeWitt, D., Smith, M., and H. Boral, "A Single-User Performance Evaluation of the Teradata Database Machine," MCC Technical Report Number DB-081-87, March 5, 1987.
- [DEWI88] DeWitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine", Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.
- [GERB86] Gerber, R., "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," PhD Thesis and Computer Sciences Technical Report #672, University of Wisconsin-Madison, October 1986.
- [JARK84] Jarke, M. and J. Koch, "Query Optimization in Database System," ACM Computing Surveys, Vol. 16, No. 2, June, 1984.
- [KITS88] Kitsuregawa, M. "Query Execution for Large Relation On Functional Disk System," to appear, 1989 Data Engineering Conference.
- [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.

- [PROT85] Proteon Associates, Operation and Maintenance Manual for the ProNet Model p8000, Waltham, Mass, 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.
- [SELI79] Selinger, P. G., et. al., "Access Path Selection in a Relational Database Management System," Proceedings of the 1979 SIGMOD Conference, Boston, MA., May 1979.
- [TANE81] Tanenbaum, A. S., **Computer Networks**, Prentice-Hall, 1981.
- [TAND88] Tandem Performance Group, *A Benchmark of Non-Stop SQL on the Debit Credit Transaction*, Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.
- [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.