A Performance Analysis of the Gamma Database Machine

David J. DeWitt
Shahram Ghandeharizadeh
Donovan Schneider

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

This paper presents the results of an initial performance evaluation of the Gamma database machine based on an expanded version of the single-user Wisconsin benchmark. In our experiments we measured the effect of relation size and indices on response time for selection, join, and aggregation queries, and single-tuple updates. A Teradata DBC/1012 database machine of similar size is used as a basis for interpreting the results obtained. We also analyze the performance of Gamma relative to the number of processors employeed and study the impact of varying the memory size and disk page size on the execution time of a variety of selection and join queries. We analyze and interpret the results of these experiments based on our understanding of the system hardware and software, and conclude with an assessment of the strengths and weaknesses of the two machines.

1. Introduction

This report presents the results of a single-user performance evaluation of the Gamma database machine [DEWI86, GERB86, GERB87]. This evaluation is based on two principal metrics: the absolute performance achieved by Gamma and the performance relative to the number of processors used. As a basis for determining the absolute performance of Gamma, we have used results obtained from a similar study [DEWI87] of the Teradata DBC/1012 database machine [TERA83, TERA85a, TERA85b]. When interpreting the results presented below, the reader should remember that Gamma is not a commercial product and, as such, its results may look slightly better for some queries. In particular, a full recovery mechanism has not yet been implemented in Gamma although distributed concurrency control is provided. Thus, the results presented below should be used as basis for comparing the storage organizations and multiprocessor algorithms of the two database machines and **not** as the basis for drawing any conclusions as to which is the "better" machine.

The objective of the second part of our evaluation was to determine the performance of Gamma relative to the number of processors used. Simply increasing the number of processors, however, has the side effect of increasing the amount of buffer space available for processing join operations. Thus, a join that does not cause a join hash table overflow with 8 processors may result in 7 overflows when the query is executed using a single processor. While one could change the size of the test relations to avoid this problem, we decided instead to keep the total (summed across all processors) amount of buffer space constant when varying the number of processors. Then, in a separate set of tests, we kept the number of processors constant while varying the total amount of buffer space available. In the final suite of tests, we kept the number of buffer pages and processors constant while varying the disk page size.

In Sections 2 and 3, respectively, we describe the Gamma and Teradata configurations that were evaluated. Section 4 presents an overview of database used for the benchmark. Four types of tests were conducted: selections, joins, aggregates, and updates. A description of the exact queries used and the results obtained for each query are contained in Sections 5 through 8. Our conclusions are presented in Section 9.

2. Overview of the Gamma Database Machine

In this section we present an overview of the Gamma database machine including a description of the current hardware configuration and the software techniques used in the implementation. Detailed descriptions of the algorithms used for implementing the various relational operations are presented in Sections 5 through 8 along with

the performance results obtained during the benchmarking process. For a complete description of Gamma see [DEWI86, GERB86].

2.1. Hardware Configuration

Presently, Gamma consists of 17 VAX 11/750 processors, each[1] with two megabytes of memory. An 80 megabit/second token ring [PROT85] is used to connect the processors to each other and to another VAX 11/750 running Berkeley UNIX. This processor acts as the host machine for Gamma. Attached to eight of the processors are 333 megabyte Fujitsu disk drives (8") which are used for database storage. One of the diskless processors is currently reserved for query scheduling and global deadlock detection. The remaining diskless processors are used to execute join, projection, and aggregate operations. Selection and update operations are executed only on the processors with disk drives attached.

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. This is the default strategy in Gamma for relations created as the result of a query. If the hashed strategy is selected, a randomizing function is applied to the key attribute of each tuple to select a storage unit. Since the Teradata database machine uses this technique, all the tests we conducted used this tuple distribution strategy. 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.

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

[1] Several processors have more than 2 megabytes of memory so that the join query speedup tests could be conducted without causing hash table overflow to occur when only 1 or 2 processors are used.

being parsed, optimized, and compiled, the query is sent by the host software to an idle scheduler process through a dispatcher process. The dispatcher process, by controlling the number of active schedulers, implements a simple load control mechanism based on information about the degree of CPU and memory utilization at each processor. The scheduler process, in turn, activates operator processes at each query 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. Results of a query can either be returned to the host or stored as a new relation in the database.

In Gamma, the algorithms for all operators are written as if they were to be run on a single processor. As shown in Figure 1, 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**. After being initiated, a query process waits for a control message to arrive on a global, well-known control port. Upon receiving an operator control packet, the process replies with a message that identifies itself to the scheduler. 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

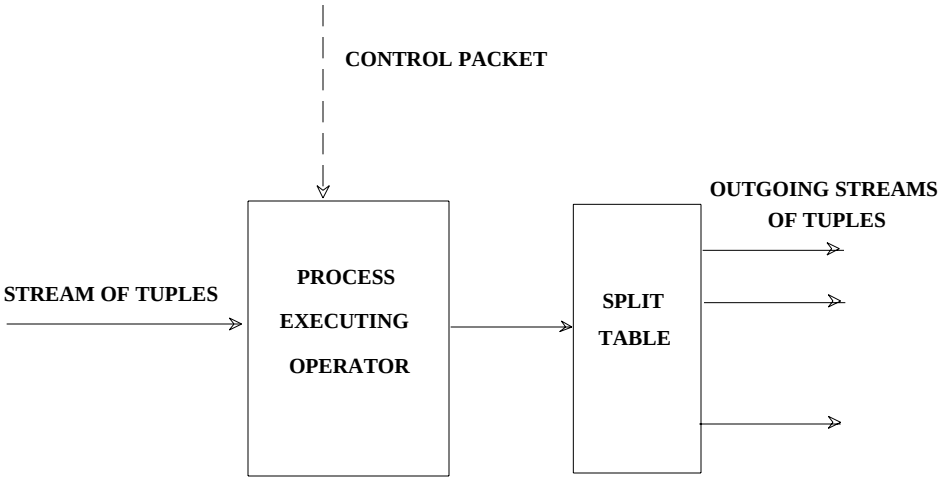

Figure 1

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 1 and N. 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. The store operators assume the responsibility for writing the result tuples to disk.

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, DEWI85, DEWI84, 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

Gamma is built on top of an operating system developed specifically for supporting database management systems. NOSE provides lightweight processes with shared memory and reliable, datagram communication services between NOSE processes on Gamma processors and to UNIX processes on the host machine 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 NOSE 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. A sequential file is a sequence of records. Records may vary in length (up to one page in length), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each sequential file may have one or more associated indices. The index maps key values to the records of the sequential file that contain a

matching value. Furthermore, one indexed attribute may be used as a clustering attribute for the file. The scan mechanism is similar to that provided by System R's RSS [ASTR76] except that predicates are compiled into machine language.

3. Teradata Hardware and Software Configuration

The Teradata machine tested consists of 4 Interface Processors (IFPs), 20 Access Module Processors (AMPs), and 40 Disk Storage Units (DSUs). The IFPs communicate with the host, and parse, optimize, and direct the execution of user requests. The AMPs perform the actual storage and retrieval of data on the DSUs. IFPs and AMPs are interconnected by a dual redundant, tree-shaped interconnect called the Y-net [TERA83, NECH83]. The Y-net has an aggregate bandwidth of 12 megabytes/second. Intel 80286 processors are used in all IFPs and AMPs. Like the Gamma processors, each AMP has 2 megabytes of memory and two[2] 8.8", 525 megabyte (unformatted) Hitachi disk drives (model DK 8155). The host processor was an AMDAHL V570 running the MVS operating system. Software release 2.3 was used for the tests conducted.

All relations on the Teradata machine are horizontally partitioned [RIES78] across multiple AMPs. While it is possible to limit the number of AMPs over which relations are partitioned, all 20 AMPs were used for the tests presented below. Whenever a tuple is to be inserted into a relation, a hash function is applied to the primary key[3] of the relation to select an AMP for storage. Hash maps in the Y-net nodes and AMPs are used to indicate which hash buckets reside on each AMP.

Once a tuple arrives at a site, that AMP applies a hash function to the key attribute in order to place the tuple in its "fragment" (several tuples may hash to the same value) of the appropriate relation. The hash value and a sequence number are concatenated to form a unique tuple id [TERA85a, MC[2]86]. Once an entire relation has been loaded, the tuples in each horizontal fragment are in what is termed "hash-key order." Thus, given a value for the key attribute, it is possible to locate the tuple in a single disk access (assuming no buffer pool hits). This is the only physical file organization supported at the present time. It is **important** to note that given this organization, the only kind of indices one can construct are dense, secondary indices. The index is termed "dense" as it must contain one entry for each tuple in the indexed relation. It is termed "secondary" as the index order is different than the key

---

[2] The software actually treats the drives as a single logical unit.

[3] The primary key is specified when the relation is created.

order of the file. Furthermore, the rows in the index are themselves hashed on the key field and are **NOT** sorted in key order. Consequently, whenever a range query over an indexed attribute is performed, the **entire** index must be scanned.

4. Description of Benchmark Relations

The benchmark relations used are based on the standard Wisconsin Benchmark relations [BITT83]. Each relation consists of thirteen 4-byte integer attributes and three 52-byte string attributes. Thus, each tuple is 208 bytes long. In order to more meaningfully stress the two database machines, we constructed 100,000 and 1,000,000 tuple versions of the original 1,000 and 10,000 tuple benchmark relations. The unique1 and unique2 attributes of the relations are generated in a way to guarantee that each tuple has a unique value for each of the two attributes and that there is no correlation between values of unique1 and unique2 within a single tuple. Two copies of each relation were created and loaded using Unique1 as the key (partitioning) attribute in all cases. The total database size is approximately 464 megabytes (not including indices).

For the Teradata machine all test relations were loaded in the NO FALLBACK mode. The FALLBACK option provides a mechanism to continue processing in the face of disk and AMP failures by automatically replicating each tuple at two different sites. Since we did not want to measure the cost of keeping both copies of a tuple consistent, we elected not to use the FALLBACK feature.

Except where otherwise noted, the results of all queries were stored in the database. We avoided returning data to the host because we were afraid that we would end up measuring the speed of the communications link between the host and the database machine or the host processor itself. By storing all results in the database, these factors were minimized in our measurements. Of course, on the other hand, we ended up measuring the cost of storing the result relations.

Storing the result of a query in a relation incurs two costs not incurred if the resulting tuples are returned to the host processor. First, the tuples of each result relation must be distributed across all processors with disks. In the case of the Teradata database machine, unique1 was used as the primary key of both the source and result relations. While we had expected that no communications overhead would be incurred in storing the result tuples, since the low-level communications software does not recognize this situation, the execution times presented below include the cost of redistributing the result tuples. Since, the current version of Gamma redistributes result tuples in a round-robin fashion, both machines incur the same redistribution overhead while storing the result of a query in a

relation.

The second cost associated with storing the result of a query in a relation is the impact of the recovery software on the rate at which tuples are inserted in a relation. In this case, there are substantial differences between the two systems. Gamma, which provides an extended version of the query language QUEL [STON76], uses the construct "*retrieve into result_relation ...* " to specify that the result of a query is to be stored in a relation. The semantics of this construct are that the relation name specified must not exist when the query is executed. If for some reason the transaction running the query is aborted, the only action that the recovery manager must take is to delete all files associated with the result relation.

The query language for the Teradata database machine is based on an extended version of SQL. In order to execute an SQL query that stores the result tuples in a relation, one must first explicitly create the result relation. After the result relation has been created one uses the syntax:

insert into result_relation
*select * from source_relation where ...*

Since in some cases the result relation may already contain tuples (in which case the insert acts more like a union), the code for *insert into* must log all inserted tuples carefully so that if the transaction is aborted, the relation can be restored to its original state. Since the Teradata insert code is currently optimized for single tuple and not bulk updates, at least 3 I/Os are incurred for each tuple inserted (see [DEWI87] for a more complete description of the problem). A straightforward optimization would be for the the "insert into" code to recognize when it was operating on an empty relation. This would enable the code to process bulk updates much more efficiently (by, for example, simply releasing all pages in the result relation if the "insert into" is aborted).

5. Selection

In this section, we first explore Gamma's performance for a variety of selection queries as the size of the input relations is increased. The results obtained are compared with the results of running the same set of queries on the Teradata database machine. For a subset of these queries, we then varied the number of processors and the disk page size to determine how these factors affect performance.

5.1. Performance Relative to Relation Size

The selection queries were designed with two objectives in mind. First, we wanted to know how the Teradata and Gamma machines would respond as the size of the source relations was increased. Ideally, given constant

machine configurations, the response time should grow as a linear function of the size of input and result relations. Second, we were interested in exploring the effect of indices on the execution time of a selection on each machine while holding the selectivity factor constant.

Our tests used two sets of selection queries: first with 1% selectivity and second with 10% selectivity. On Gamma, the two sets of queries were tested with three different storage organizations: a heap (no index), a clustered index on the key attribute (index order = key order), and a non-clustered index on a non-key attribute (index order ≠ key order). On the Teradata machine, since tuples in a relation are organized in hash-key order, it is not possible to construct a clustered index. Therefore, all indices, whether on the key or any other attribute, are dense, non-clustered indices.

In Table 1, we have tabulated the results of testing the different types of selection queries on three sizes of relations (10,000, 100,000, and 1,000,000 tuples). Two main conclusions can be drawn from this table. First, for both machines the execution time of each query scales in a linear fashion as the size of the input and output relations are increased. Second, as expected, the clustered B-tree organization provides a significant improvement in performance.

As discussed in [DEWI87], the results for the 1% and 10% selection using a non-clustered index (rows three and four of Table 1) for the Teradata machine look puzzling. Both of these queries selected tuples using a

Table 1
Selection Queries
(All Execution Times in Seconds)

	Number of Tuples in Source Relation					
Query Description	**10,000 Teradata**	**10,000 Gamma**	**100,000 Teradata**	**100,000 Gamma**	**1,000,000 Teradata**	**1,000,000 Gamma**
1% nonindexed selection	6.86	1.63	28.22	13.83	213.13	134.86
10% nonindexed selection	15.97	2.11	110.96	17.44	1106.86	181.72
1% selection using non-clustered index	7.81	1.03	29.94	5.32	222.65	53.86
10% selection using non-clustered index	16.82	2.16	111.40	17.65	1107.59	182.00
1% selection using clustered index	-	0.59	-	1.25	-	7.50
10% selection using clustered index	-	1.26	-	7.27	-	69.60
single tuple select	-	0.15	1.08	0.15	-	0.20

predicate on the unique2 attribute, an attribute on which we had constructed a non-clustered index. In the case of the 10% selection, the optimizer decided (correctly) not to use the index. In the 1% case, the observed execution time is almost identical to the result obtained for the nonindexed case. While these results seems to contradict the query plan produced by the optimizer, which states that the non-clustered index on unique2 is to be used to execute the query, the storage organization used for indices on the Teradata machine provides a partial explanation. Since the index entries are hash-based and not in sorted order, the **entire** index must be scanned sequentially instead of scanning only the portion corresponding to the range of the query. Thus, exactly the same number of attribute value comparisons is done for both index scans and sequential scans. However, it is expected that the number of I/Os required to scan the index is only a fraction of the number of I/Os required to scan the relation. Apparently, the response time is not reduced significantly because, while the index can be scanned sequentially, each access to the relation requires a random seek.

Gamma supports the notion of non-clustered indices through a B-tree structure on top of the actual data file. As can be seen from Table 1, in the case of the 10% selection, the Gamma optimizer also decides not to use the index. In the 1% case, the index is used. Consider, for example, a scan with a 1% selectivity factor on a 10,000 tuple relation: if the non-clustered index is used, in the worst case 100(+/- 4) I/Os will be required (assuming each tuple causes a page fault). On the other hand, if a segment scan is chosen to access the data, with 17 tuples per data page, all 588 pages of data would be read. The difference between the number of I/Os is significant and is confirmed by the difference in response time between the entries for Gamma in rows 3 and 4 of Table 1.

Gamma also provides clustered indices (the underlying relation is sorted according to the key attribute and a B-tree search structure is built on top of the data). The response time for the 1% and 10% selections through a clustered index are presented in rows five and six of Table 1. Since the tuples are sorted (key order = index order), only that portion of the relation corresponding to the range of the query is scanned. This results in a further reduction of the number of I/Os compared to the corresponding search through a file scan or a non-clustered index. This saving is confirmed by the lower response times shown in Table 1.

One important observation to be made from Table 1 is the relative consistency of the cost of selection using a clustered index in Gamma. Notice that the response time for both the 10% selection from the 10,000 tuple relation and the 1% selection from the 100,000 tuple relation using a clustered index is 1.25 seconds. The reason is that in both cases 1,000 tuples are retrieved and stored, resulting in the same I/O and CPU costs.

The selection results reveal an important limitation of the Teradata design. Since there are no clustered

indices, and since non-clustered indices can only be used when a relatively small number of tuples are retrieved, the system must resort to scanning entire files for most range selections. While hash files are certainly the optimal file organization for exact-match queries, for certain types of applications, range queries are important. In such cases, it should be possible for the database administrator to specify the storage organization that is best suited for the application.

In the final row of Table 1 we have presented the times required by both machines to select a single tuple and return it to the host. For the Teradata machine, the key attribute is used in the selection condition. After hashing on the constant to select an AMP, the AMP software will hash again on the constant to select the hash bucket that holds the tuple. In the case of Gamma, a clustered index on the key attribute was used. While we only ran this test for the 100,000 tuple relation on the Teradata machine, we would expect comparable times for the 10,000 and 1,000,000 tuple tables. These results indicate that clustered indices have comparable performance with hash files for single tuple retrieves while providing superior performance for range queries.

As discussed in Section 4, since the semantics of QUEL and SQL are different, the results presented in Table 1 are slightly misleading and the times for the two machines are not directly comparable. Teradata treats each insertion as a separate operation [DEWI87]. Thus, the time required to insert tuples into a result relation sometimes accounts for a significant fraction of the execution time of the query. Gamma, on the other hand, pipelines the output of the selection, handling it as a bulk update. Thus, we decided to separate the processing time for the queries from the time to insert tuples into the result relations. We calculated the rate[4] at which tuples can be redistributed and inserted on each machine by dividing the difference in the number of tuples selected by the difference in the time to select 10% and 1% of the tuples (without an index). Then, to get a measure of the processing time alone, we subtracted the approximate time to redistribute and store the result relation (number of tuples retrieved multiplied by the average cost of insertion per tuple) for each entry in Table 1 to come up with Table 2.

5.2. An Analysis of Selection Performance in Gamma

In this section we study how the response time for both the nonindexed and indexed selection queries on the 100,000 tuple relations is affected by the number of processors used and the disk page size. Ideally one would

[4] Actually, an average insertion rate was computed by averaging the value obtained for each of the three relation sizes.

Table 2
Adjusted Selection Queries
(All Execution Times in Seconds)

	Number of Tuples in Source Relation					
Query Description	**10,000 Teradata**	**10,000 Gamma**	**100,000 Teradata**	**100,000 Gamma**	**1,000,000 Teradata**	**1,000,000 Gamma**
1% nonindexed selection	5.90	1.58	18.61	13.43	117.71	130.08
10% nonindexed selection	6.36	1.63	14.88	12.67	152.66	134.02
1% selection using non-clustered index	6.85	0.98	20.33	4.84	127.23	49.10
10% selection using non-clustered index	7.21	1.68	15.32	12.88	153.39	134.30
1% selection using clustered index	-	0.54	-	0.78	-	2.73
10% selection using clustered index	-	0.78	-	2.50	-	21.90

like to see linear improvement in performance relative to the number of processors used.

5.2.1. Constant Page Size, Varying Configuration

In the first set of experiments, the disk page size was kept at 4 Kbytes while the number of processors with disks was increased from 1 to 8. Thus, as the number of processors is increased, the number of tuples stored at each site is reduced proportionally.

NonIndexed Selections

Without an index, all the data pages in the relation must be read from disk and processed. Increasing the number of processors used to process a non-indexed selection increases both the aggregate CPU power and I/O bandwidth available while reducing the number of tuples that must be processed at each processor.

In Figure 2, the average response time for 0%, 1%, and 10% selections on the 100,000 tuple relation is presented as a function of the number of processors with disks. As expected, the response time for each of the queries decreases as the number of sites is increased. The response time for the queries with 1% and 10% selectivity factors is worse than the 0% query due to the cost of transmitting and storing the result tuples. While for selection-only queries one might store all result tuples locally, by partitioning all result relations in a round-robin (or hashed) fashion one can insure that each fragment of every result relation will contain approximately the same number of tuples.

The speedup curve corresponding to Figure 2 is presented in Figure 3. As shown in Figure 3, almost linear

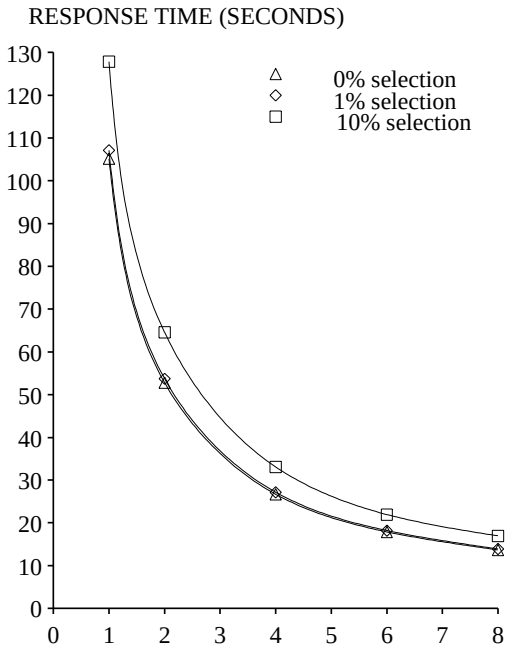

FIGURE 2

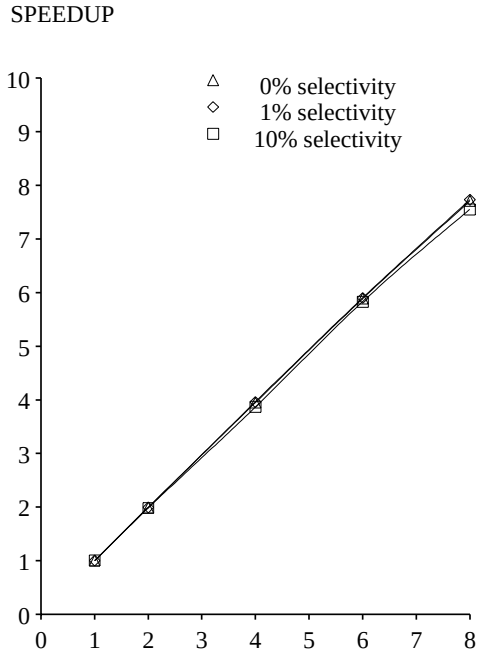

FIGURE 3

speedup is obtained for all three queries. The reason that the 0% selection query does not achieve perfect speedup is that the number of *end of stream* messages (see Section 2) each processor must send increases as processors are added to the system. The 10% selectivity speedup curve is not as close to linear as the 0% or 1% curves due to the effects of *short-circuiting* (again see Section 2). When a single processor is used, all result tuples are "short-circuited" by the low-level communications software. As more processors are used, the fraction of tuples short-circuited decreases (with n processors, $\frac{1}{n}$th the result tuples will be short-circuited). While the actual network is never a bottleneck [GERB86, GERB87], the bandwidth from memory to the communications network is limited by the speed (4 megabits/second) of the Unibus on the VAX 11/750. As the selectivity factor of a query is increased and the number of short circuited tuples decreases, the path to the network becomes a bottleneck. This fact is illustrated by the differences among the curves in Figure 3.

Indexed Selections

For this suite of tests, we constructed, respectively, clustered and non-clustered indices on the Unique1 and Unique2 attributes of the 100,000 tuple relations. In Figure 4, the average response time is plotted as a function of the number of processors with disks for the following three queries: 1% selection using a clustered index, 10%

selection using a clustered index, and 1% selection using a non-clustered index. The corresponding speedup curves are presented in Figure 5 along with the speedup curve obtained for a 0% selection through a non-clustered index. No results are presented for a 10% non-clustered index selection as our optimizer chooses to use a segment scan for this query.

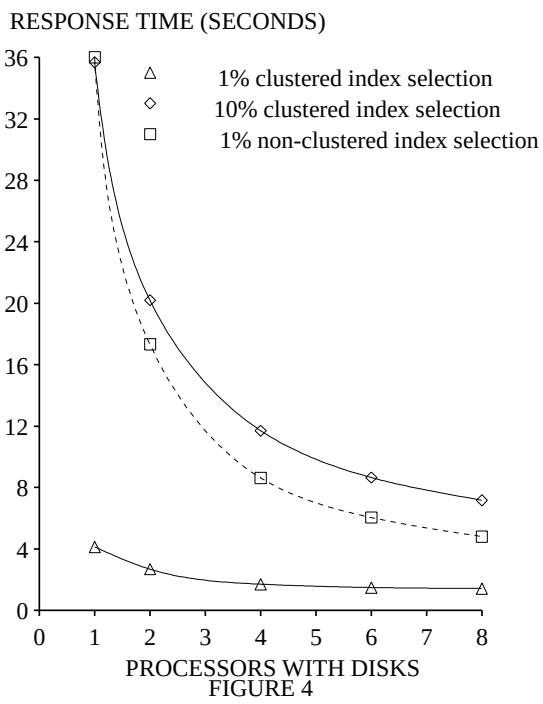

FIGURE 4

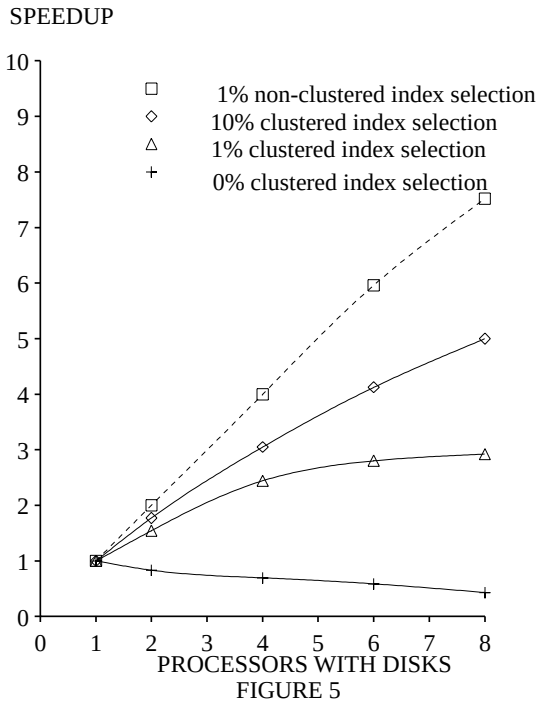

FIGURE 5

The speedup curves presented in Figure 5 reveal a number of interesting insights into the effects of increasing the amount of parallelism when indices are employed. First, in the case of the 0% selection query, the response time for the query actually increases (from 0.25 to 0.58 seconds) as the number of processors is increased. This happens because the cost of initiating a select and store operator at each processor appears to be slightly higher than the cost of performing 1-2 I/O operations to search the index before discovering that no tuples satisfy the predicate. Of the remaining queries, only the 1% selection through a non-clustered index comes close to achieving linear speedup. Why is this, when, without an index, the same queries obtained nearly linear speedups? Consider first the 10% selection query. Without an index, each processor executing this query will produce one network packet (2 Kbytes) of result tuples for approximately every five 4 Kbyte pages it reads from disk. With 4 Kbyte disk pages the system is I/O bound. When the same query is executed using a clustered index, once the first page containing qualifying tuples is found, every subsequent page read from disk will be completely full of result tuples. Thus, for each

data page read, two communication packets must be sent. As the number of processors is increased, the fraction of these packets that are short circuited decreases. Since the disk is producing packets faster than the communications interface can place them on the network, performance degrades. In the case of the 1% selection through a clustered index, the same effect occurs but, as the number of processors is increased, the time to initiate the query and process 2 levels of the index at 8 sites (0.58 seconds) becomes a significant fraction of the total execution time of the query (2 seconds). Finally, the reason that the 1% selection through a non-clustered index achieves very close to linear speedup is that each disk page read requires a random seek, thus significantly reducing the rate at which the disk produces pages.

5.2.2. Effect of Disk Page Size on Selection Performance

In this experiment, the configuration size was kept constant (8 processors with disks), while the disk page size was varied from 2 Kbytes to 32 Kbytes using both sequential and index scans on the 100,000 tuple relations.

NonIndexed Selections

Non-indexed selections with 0%, 1%, 10%, and 100% selectivity factors were executed with disk pages sizes ranging from 2 to 32 Kbytes. The response times for these queries are plotted in Figure 6 and the corresponding speedup curves are presented in Figure 7. These results (most particularly the 0% selection curve which does not generate any network traffic) clearly indicate that with a 2 Kbyte disk page the system is disk bound and that once the page size is increased to 16 Kbytes the system becomes CPU bound. For the VAX 11/750 CPU (0.6 MIP), any increase in the size of the disk page beyond 8K bytes has little or no effect on the response time of the query. Repeating these experiments with a faster CPU would be interesting.

The results presented in Figures 6 and 7 provide further evidence that the network interface can become a bottleneck. With a 2 Kbyte page size, the response time for the 10% selection is 19 percent slower than the response time for the 0% selection. With a 32 Kbyte page size, the 10% selection is 50 percent slower than the 0% selection. It is very clear that as one increases the rate at which result tuples are produced, (either by increasing the size of the disk page or through the use of a clustered index), the network interface increasingly becomes a bottleneck.

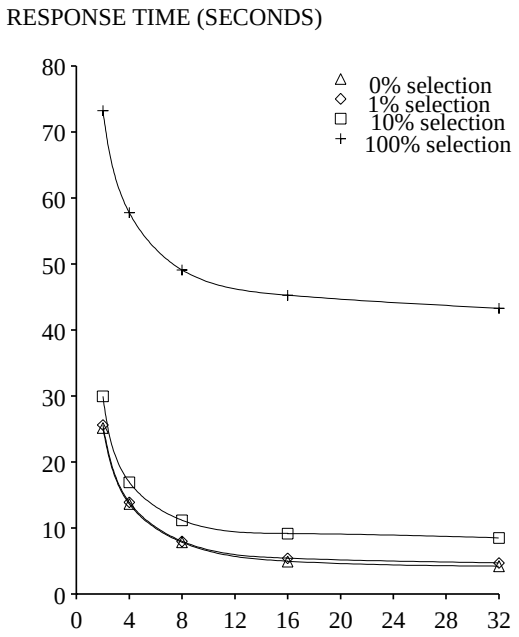

FIGURE 6

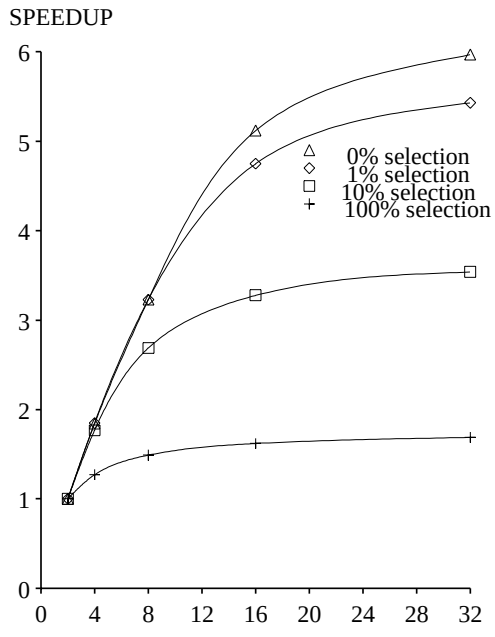

FIGURE 7

Indexed Selections

We repeated the same set of experiments after constructing a clustered index on the Unique1 attribute and a non-clustered index on the Unique2 attribute. In these tests, however, increasing the size of the disk page also increases the fan out of the nodes of the B-tree index.

The average response time and corresponding speedup curves for the queries tested are presented in Figure 8 and 9. The most interesting results are those obtained for the 1% selection through a non-clustered index. As indicated in both figures, any increase in disk page size degrades the performance of this query. Since each tuple retrieved requires fetching two index pages plus one data page, the longer transfer time for the larger pages dominates any advantage provided in terms of fan-out. (For a 32 Kbyte disk page, the transfer time is 13 milliseconds - which is very close to the time required to perform a random disk seek. The disk used has a 40 Kbyte track size).

When a clustered index is employed, this degradation in performance does not occur because once the proper leaf page of the index is located all subsequent tuples on that page and all subsequent leaf pages will satisfy the query. While the 10% selection continues to show improvement with larger disk pages, the response time for the 1% selection actually increases slightly when the page size is increased from 16 to 32 Kbytes. The longer transfer time is again the source of the problem. With 8 processors, each site will produce approximately 125 tuples. With 32 Kbyte pages, each page will hold approximately 150 tuples. If the 125 tuples satisfying the query

span two pages (the expected case), more than 50% of the tuples read will not satisfy the query.

6. Join Queries

We had two goals in mind when we designed our experiments for testing join performance in Gamma. First, we wanted to see how a "typical" Gamma configuration performs on a fixed set of join queries as the size of the input relations is increased. As a reference point, the results from running the same set of queries on a 20 processor Teradata database machine are also presented. Second, we wanted to explore how Gamma's performance is affected as the number of processors with disks is increased, as the disk page size is increased, and as the amount of available memory is reduced.

Join Algorithms

The Teradata machine uses four alternative join algorithms [TERA85a, MC^{2}86]. One computes an outer-join, while two others are used only in special cases (e.g., when the inner relation contains a single tuple). The fourth and most commonly used join method involves first redistributing the two source relations by hashing on the join attribute. As each AMP receives tuples, it stores them in temporary files sorted in hash-key order. After the

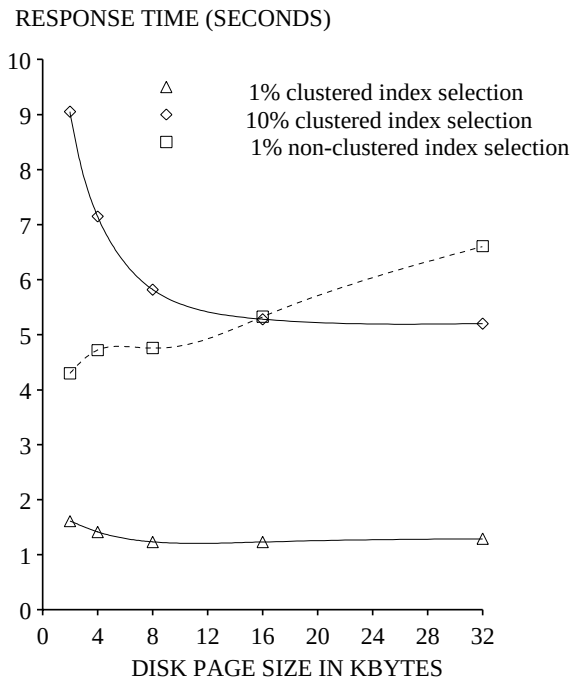

FIGURE 8

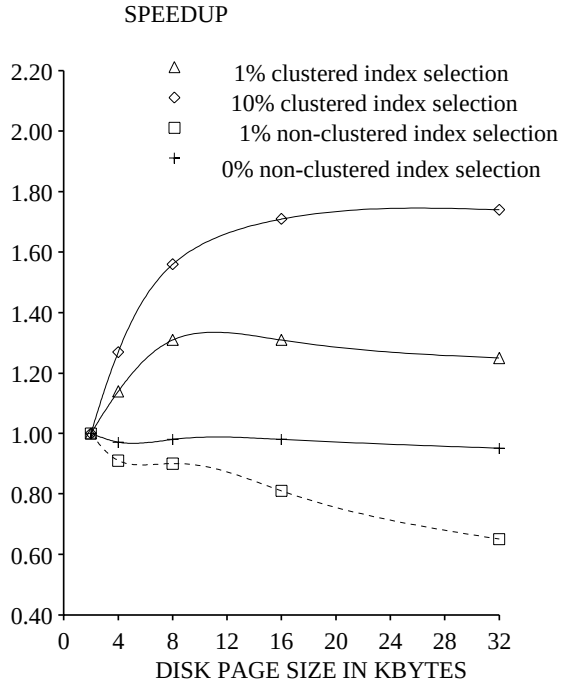

FIGURE 9

redistribution phase completes, each AMP uses a conventional sort-merge join algorithm to complete the join. For our test queries, this was the only algorithm used.

Gamma also partitions its source relations by hashing on the join attributes but, instead of using sort-merge to effect the join, Gamma employs an algorithm based on hashing (see [KITS83, DEWI85, DEWI86, GERB86]). During the first phase of the algorithm, Gamma partitions the smaller source relation by hashing on the joining attribute and builds main-memory hash tables. During phase two, Gamma partitions the larger source relation and uses the corresponding tuples to immediately probe the hash tables built during phase one.

Of course, whenever main-memory hashing is used there is a danger of hash-table overflow. Gamma currently uses a distributed version of the Simple hash-partitioned join algorithm described in [DEWI85] to handle this phenomenon. Basically, whenever a processor detects hash-table overflow it spools tuples to a temporary file based on a second hash function until the hash table is successfully built. The query scheduler then passes the function used to subpartition the hash table to the select operators producing the probing tuples. Probing tuples corresponding to tuples in the overflow partition are then spooled to a corresponding temporary file; all other tuples probe the hash table as normal. The overflow partitions are recursively joined using this same procedure until no more overflow partitions are created and the join has been fully computed.

Gamma can actually run joins in a variety of modes. The selection operators will, of course, run on all disk sites but the hash tables may be built on the processors with disks, the diskless processors, or both sets of processors. These alternatives are referred to as **Local, Remote, and Allnodes,** respectively.

Queries

Three join queries formed the basis of our join tests. The first join query, joinABprime, is a simple join of two relations: A and Bprime. The A relation contains either 10,000, 100,000 or 1,000,000 tuples. The Bprime relation contains, respectively, 1,000, 10,000, or 100,000 tuples. The second query, joinAselB, performs one join and one selection. A and B have the same number of tuples and the selection on B reduces the size of B to the size of the Bprime relation in the corresponding joinABprime query. For example, if A has 100,000 tuples, then joinABprime joins A with a Bprime relation that contains 10,000 tuples, while in joinAselB the selection on B restricts it from 100,000 to 10,000 tuples and then joins the result with A.

The third join query, joinCselAselB contains two joins and two restricts. First, A and B are restricted to 10% of their original size (10,000, 100,000, or 1,000,000 tuples) and then joined with each other. Since each tuple

joins with exactly one other tuple, this join yields an intermediate relation equal in size to the two input relations. This intermediate relation is then joined with relation C, which contains 1/10 the number of tuples in A. The result relation contains as many tuples as there are in C. For example, assume A and B contain 100,000 tuples. The relations resulting from selections on A and B will each contain 10,000 tuples. Their join results in an intermediate relation of 10,000 tuples. This relation will be joined with a C relation containing 10,000 tuples and the result of the query will contain 10,000 tuples.

6.1. Performance Relative to Relation Size

The first variation of the three queries tested involved no indices and used a non-key (non-partitioning, non-indexed) attribute (unique2D or unique2E) as both the join and selection attributes. Since all the source relations were distributed using the key attribute, the join algorithms of both machines required redistribution phases. The results from these tests are contained in the first 3 rows of Table 3. For this series of tests (and also for the results presented in Table 4), Gamma used 4 Kbyte disk pages and all join queries were performed in the **Remote** mode in which the joins are done only on the diskless processors.

The second variation of the three join queries used the key attribute (unique1D or unique1E) as the join attribute. (Rows 4 through 6 of Table 3 contain these results.) Since, in this case, the relations are already distributed on the join attribute, the Teradata machine demonstrated substantial performance improvement (25-50%) because the redistribution step of the join algorithm could be skipped. In the case of Gamma, however, both relations still had to be redistributed since only diskless processors were used for the joins.

From the results in Table 3, one can conclude that the execution time of each of the queries increases in a fairly linear fashion as the size of the input relations are increased. Gamma does not exhibit linearity in the million tuple queries because the size of the building relation (20 megabytes) far exceeds the total memory available for hash tables (4.8 megabytes) and the Simple hash-partition overflow algorithm deteriorates exponentially with multiple overflows. In fact, the computation of the million tuple join queries required six partition overflow resolutions on each of the diskless processors. In Section 6.2.2, we explore in more detail the impact of limited memory on the performance of join queries in Gamma.

The observant reader may have noticed that the Teradata can always do joinABprime faster than joinAselB but that just the opposite is true for Gamma. We will explain the difference by analyzing Table 3 with the 100,000

Table 3
Join Queries
(All Execution Times in Seconds)

Query Description	Number of Tuples in Source Relation					
	10,000 Teradata	10,000 Gamma	100,000 Teradata	100,000 Gamma	1,000,000 Teradata	1,000,000 Gamma
joinABprime with non-key attributes of A and B used as join attribute	34.9	6.5	321.8	47.6	3,419.4	2,938.2
joinAselB with non-key attributes of A and B used as join attribute	35.6	5.1	331.7	34.9	3,534.5	703.1
joinCselAselB with non-key attributes of A and B used as join attribute	27.8	7.0	191.8	38.0	2,032.7	731.2
joinABprime with key attributes of A and B used as join attribute	22.2	5.7	131.3	45.6	1,265.1	2,926.7
joinASelB with key attributes of A and B used as join attribute	25.0	5.0	170.3	34.1	1,584.3	737.7
joinCselAselB with key attributes of A and B used as join attribute	23.8	7.2	156.7	37.4	1,509.6	712.8

Table 4
Adjusted Join Queries
(All Execution Times in Seconds)

Query Description	Number of Tuples in Source Relation					
	10,000 Teradata	10,000 Gamma	100,000 Teradata	100,000 Gamma	1,000,000 Teradata	1,000,000 Gamma
joinABprime with non-key attributes of A and B used as join attribute	25.3	6.0	225.7	41.7	2,458.6	2,890.5
joinAselB with non-key attributes of A and B used as join attribute	25.9	4.6	235.6	35.8	2,573.7	655.4
joinCselAselB with non-key attributes of A and B used as join attribute	18.2	6.5	95.7	37.9	1,071.9	683.5
joinABprime with key attributes of A and B used as join attribute	12.6	5.2	35.2	45.1	304.3	2,878.9
joinASelB with key attributes of A and B used as join attribute	15.4	4.5	74.2	32.1	623.5	689.9
joinCselAselB with key attributes of A and B used as join attribute	14.2	6.7	60.6	33.1	548.8	665.1

tuple joins. Selection propagation by the Gamma optimizer reduces joinAselB to joinselAselB. This means that although both 100,000 tuple relations will be read in their entirety only 10% of each of the relations will be sent over the network and participate in the join. Although joinABprime only reads a 100,000 and a 10,000 tuple relation it must send all 100,000 tuples to the diskless processors to effect the join. Thus, the cost to distribute and probe the 100,000 tuples outweigh the difference in reading a 100,000 and a 10,000 tuple file. On the other hand, the Teradata database machine will compute joinABprime by reading and sorting a 10,000 tuple relation and a 100,000 tuple relation and then merging them. JoinAselB will read two 100,000 tuple relations and then sort and merge a 10,000 and a 100,000 tuple relation. Thus joinAselB will be slower by the difference in reading the 100,000 and 10,000 tuple relations.

6.2. An Analysis of Join Performance in Gamma

In this section, we explore the effects of changing the size of the disk page, reducing the amount of buffer space available for join hash tables, and the performance of join queries relative to the number of processors available. While we would have preferred to use the million tuple relations for these experiments, we do not have enough aggregate memory to execute the million tuple join queries without experiencing partition overflow. Thus, since we did not want the cost of processing the overflows to impact every test conducted, we chose to run the experiments using the 100,000 tuple relations.

6.2.1. Constant Memory, Constant Page Size, Varying Configuration

In the first series of tests we wanted to explore how joins performed when we increased the number of processors with disks attached.[5] In order to concentrate on the effects of changing Gamma's configuration we kept the disk page size constant at 4K bytes and kept the amount of memory available for join hash tables large enough to insure that no partition overflow would occur.

Figures 10 and 11 present, respectively, the response time for the joinABprime query when the joining attributes are also the key (partitioning) attributes and when they are not the partitioning attributes. From the shape of these graphs it is obvious that Gamma significantly reduces response time as additional processors are added. One, though, might expect Remote joins to be twice as fast as Local joins because Remote joins use twice as many

[5] Remember when we add a processor with a disk we also add a processor without a disk. These diskless processors are exploited by the Remote and Allnodes joins.

processors. As was pointed out in [DEWI86] this is not the case because the building and probing phases of the join operator are not overlapped and hence the response time of the query is bounded below by the sum of the elapsed time of these two phases. In a multiuser environment, though, it is expected that offloading the join operators to remote processors will allow the processors with disks to effectively support more concurrent selection and store operators. The validity of this expectation will be determined in future multiuser benchmarks of the Gamma database machine.

An interesting feature of Figures 10 and 11 is that, for larger configurations, the relative performance of Local and Allnodes joins is mirrored with respect to Remote joins (which remain constant). For joins on partitioning attributes, the Local configuration is fastest, followed by Allnodes and Remote joins. When an attribute other than the partitioning attribute is used as the joining attribute, the Remote configuration is the fastest followed by Allnodes and then finally Local. Both graphs are identical for the single processor configuration because the relations are stored entirely on the single disk and hence no "partitioning" of the data occurs.

This mirror-like performance function occurs because Gamma uses the same hash function to partition relations when they are being loaded and when they are being joined. Hence, when the joining and partitioning

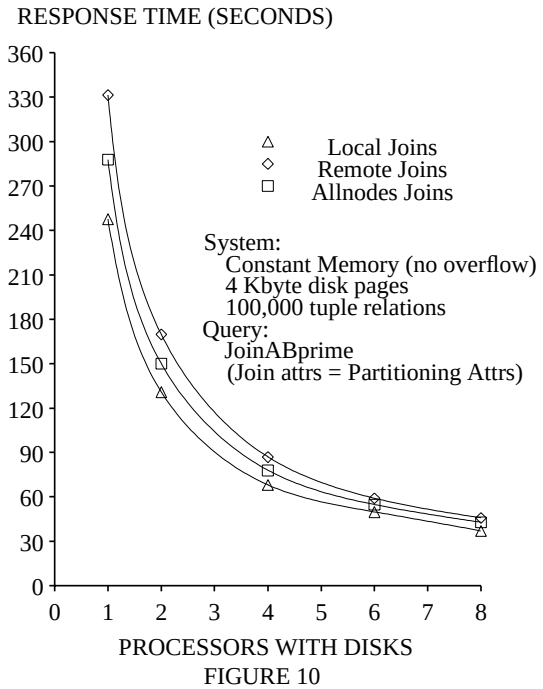

FIGURE 10

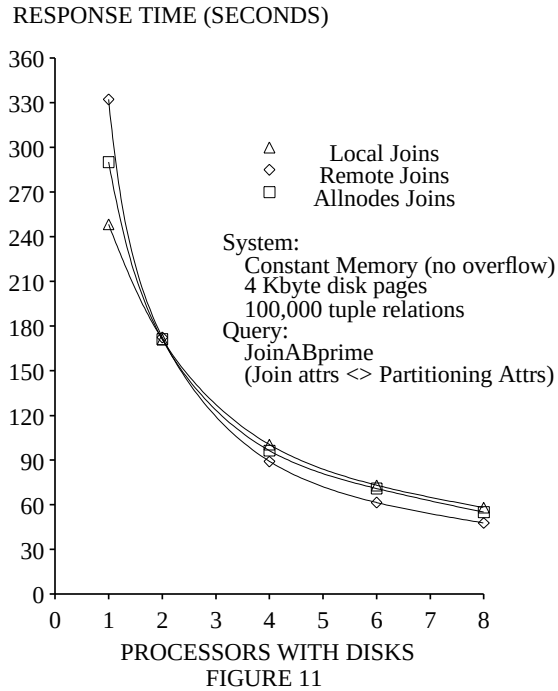

FIGURE 11

attributes are the same, Local joins will short-circuit all input tuples and gain a corresponding performance advantage. Conversely, when joins are performed on non-partitioning attributes, Local joins perform worst because short-circuiting provides no benefit and we have substantially increased contention for the CPUs with disks since the building/probing of the hash tables competes with the selection and store operators. The performance of the Allnodes configuration falls between the Remote and Local configurations as it shares the benefits and drawbacks of both.

The associated speedup curves for the joinABprime queries are shown in Figures 12 and 13. Notice that near linear speedups are obtained. Both speedup curves were plotted using the response time for two processors as a reference point in order to reduce skewing the curves due to short-circuiting. This can be best explained by the following example.

Consider a single-processor configuration with joins being done on their non-partitioning attributes. For Local joins, **all** tuples will short-circuit the network. With two processors, approximately half the tuples will be short-circuited. In general, as the number of processors is increased, the number of short-circuited packets is reduced proportionally. Because these intra-node packets are much less expensive than their corresponding inter-node packets, smaller configurations will benefit more from short-circuiting. Since one intent of plotting these

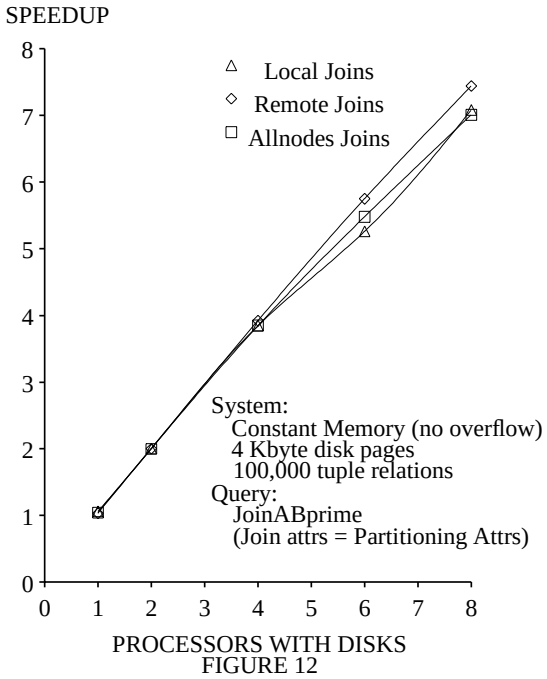

FIGURE 12

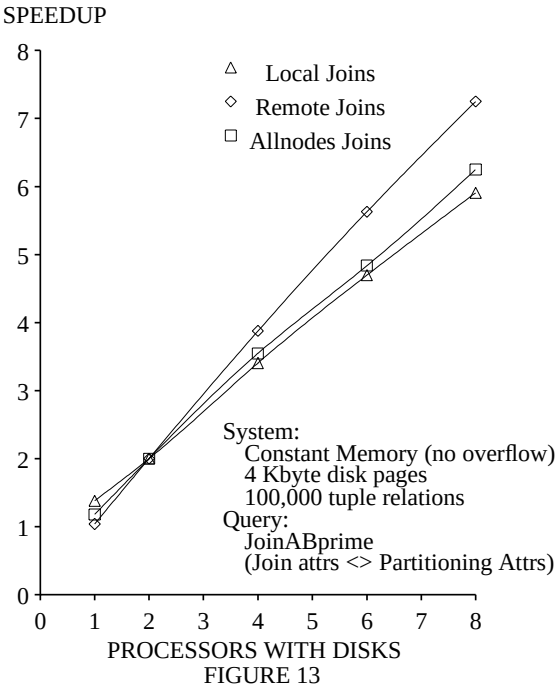

FIGURE 13

speedup curves is to project Gamma's performance as additional processors are employed, using the response time obtained with a single processor as their basis will give artificially low expected performance estimates for larger configurations. A similar argument can be made for Allnodes joins although the degree of short-circuiting will be approximately half that of Local joins. Remote joins are basically unaffected by the change in reference point. Since the two processor configuration still short-circuits half its tuples, the speedup results still underestimate the scalability of Gamma. As an example, the speedup from four to eight processors for non-hash partitioned joins done locally is approximately 1.75.

While these experiments only tested Gamma when the source relations are hash partitioned, the joins performed using non-partitioning attributes (Figures 11 and 13) have the same performance as joins on equivalent size relations partitioned in a round-robin fashion.[6] Additionally, recall that joins done on non hash-partitioned attributes can be performed faster remotely than locally. This shows that join operators can indeed be off-loaded to remote processors even for large relations. This substantiates results obtained in [DEWI86] for smaller relations.

6.2.2. Join Overflow

In this set of experiments, we kept both the configuration size (16 query processors) and disk page size (4 Kbytes) constant but varied the total amount of memory. Available memory was initially set to be sufficient to hold the total number of tuples required in the building phase of the 100,000 tuple join queries, i.e. sufficient to build 10,000 tuples across the available processors. The total amount of available memory was then incrementally lowered by evenly reducing memory from each of the processors.

From the shape of the curves in Figure 14 it is obvious that performance deteriorates rapidly as memory becomes more limited due to our use of a distributed version of the Simple Hash join algorithm to resolve hash-partition overflow (as predicted analytically in [DEWI85]). When viewing the graphs it should be kept in mind that the number of overflows represents the number of overflows detected at **each** of the eight joining sites. Thus, the total number of occurrences of partition overflow is the labeled number times eight.

A few very interesting points can be discovered by careful examination of the curves in Figure 14. First, why do the response times for the Local and Remote join curvers crossover? Recall, from the previous subsection, that joins on partitioning attributes can be done faster locally than remotely, but that just the opposite is true for the

[6]Round-robin partitioning is the default strategy for relations created as the result of a query.

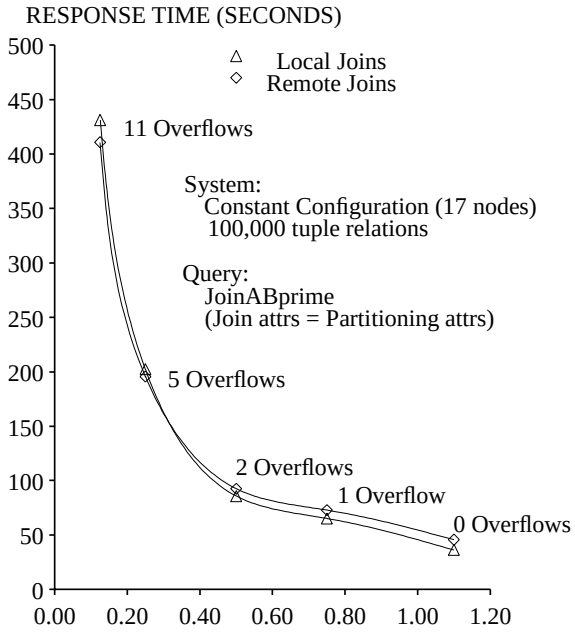

FIGURE 14

same joins done on their non-partitioning attributes. The crossover can be explained because, after the initial overflow, Gamma switches hash functions. This has the effect of changing the joining attributes to non-partitioning attributes. This change in hash functions is necessary in order to ensure that all joining processors are used in the case when only a subset of sites overflow. If the same function was used to distribute both overflow tuples and the original tuples, the same sets of tuples would continuously re-map to the same processors. Thus, processors that do not experience overflow would not be used for subsequent overflow processing.

Also the relative flatness of the response time curves from zero to two overflows indicates that Simple hash-join is effective when only a small number of overflows occur. This result is important because it means that the optimizer can be off by a factor of two in estimating either the amount of memory available or the selectivity factor of an operator without significantly affecting the response time of the query.

6.2.3. Effect of Disk Page Size on Join Performance

In the next set of experiments we wanted to explore the effect of alternative disk page sizes on join execution time. A constant Gamma configuration consisting of 16 query processors (8 with disks) and a scheduling processor was used. Memory was also kept constant and large enough so that no hash-table overflows would occur.

Figure 15 shows the results of the joinAselB query as the disk page size is varied from 2 to 32 Kbytes. As

can be seen, increasing the disk page size significantly reduces join response time although the performance improvement levels off at 16 Kbyte pages. The associated speedup curves are presented in Figure 16.

One may wonder why the speedup curves level off in the observed manner. Recall that, in Gamma, joins are bounded by the time to select tuples from the joining relations. Since Figure 16 presents the results obtained using the joinAselB query, 10% selections were performed on both source relations. Hence, the results obtained are similar to those presented for the 10% non-indexed selection in Figure 7.

One result we have not been able to explain to our satisfaction is the performance of the Allnodes configuration. Intuitively, one would expect Allnodes to always fall between Remote and Local because it shares the benefits and drawbacks of both. One possible explanation is the increased cost of scheduling Allnodes joins when the relations being joined are not large enough to fully exploit the additional processing power. Since Gamma requires four messages to schedule a query operator per node and since a join is logically composed of two operators (build and join) Allnodes will require 64 additional scheduling messages. Assuming seven milliseconds for a small inter-node message about a half of a second of additional scheduling overhead is incurred. This explanation appears to be a possibility because Allnodes joins do fall between Local and Remote joins when performing

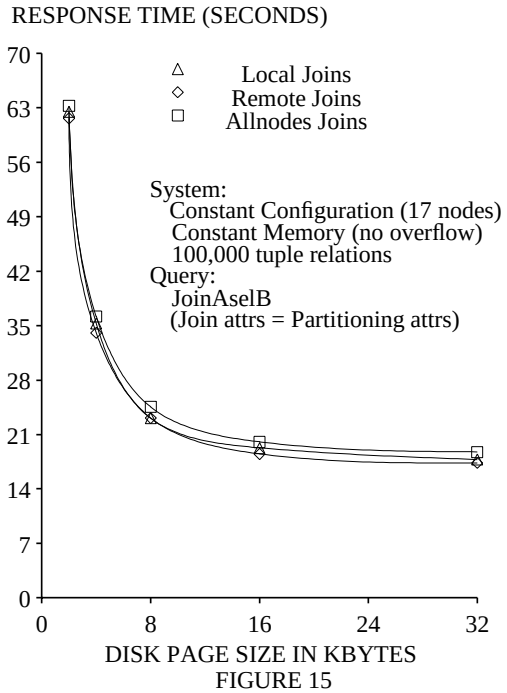

FIGURE 15

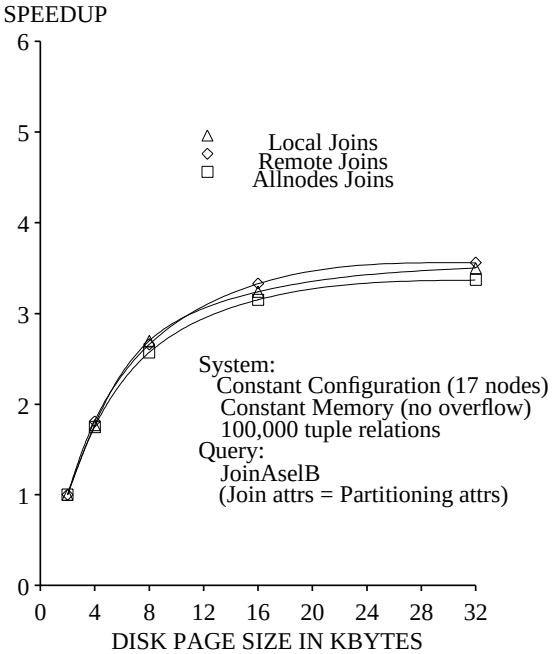

FIGURE 16

joinABprime queries.

7. Aggregate Queries

Our aggregate tests included a mix of scalar aggregate and aggregate function queries. The first query computes the minimum of a non-indexed attribute. The next two queries compute, respectively, the sum and minimum of an attribute after partitioning the relation into 100 subsets. The results from these tests are contained in Table 7. Since each query produces only either a single result tuple or 100 result tuples, we have not bothered to display the query times adjusted by the time to store the result relation.

By treating a scalar aggregate query as an aggregate function query with a single partition, the Teradata machine uses the same algorithm to handle both types of queries. Each AMP first computes a piece of the result by calculating a value for each of the partitions. Next, the AMPs redistribute the partial results by hashing on the partitioning attribute. The result of this step is to collect the partial results for each partition at a single site so that the final result can be computed.

Gamma implements scalar aggregates in a similar manner although the hashed redistribution step discussed above can be skipped. Each disk-based processor computes its piece of the result and then sends it to a process on the scheduler processor which combines these partial results into the final answer. Aggregate functions are implemented almost exactly like in the Teradata machine. As with scalar aggregates, the disk-based processors compute their piece of the result but now the partial results must be redistributed by hashing on the partitioning attribute.

Table 7
Aggregate Queries
(All Execution Times in Seconds)

	Number of Tuples in Source Relation					
Query Description	**10,000 Teradata**	**10,000 Gamma**	**100,000 Teradata**	**100,000 Gamma**	**1,000,000 Teradata**	**1,000,000 Gamma**
MIN scalar aggregate	4.21	1.89	18.29	15.53	127.86	151.10
MIN aggregate function (100 partitions)	8.66	2.86	27.06	19.43	175.95	184.92
SUM aggregate function (100 partitions)	8.94	2.89	24.79	19.54	175.78	185.05

8. Update Queries

The last set of tests included a mix of append, delete, and modify queries. The Teradata machine was executing with full concurrency control and recovery, whereas Gamma used full concurrency control and partial recovery for some of the operators; hence the performance results from the two machines are not directly comparable. The results of these tests are presented in Table 8.

The first query appends a single tuple to a relation on which no indices exist. The second appends a tuple to a relation on which one index exists. The third query deletes a single tuple from a relation, using an index to locate the tuple to be deleted (in the case of Teradata, it is a hash-based index, whereas in the case of Gamma, it is a clustered B-tree index, for both the second and third queries). In the first query no indices exist and hence no indices need to be updated, whereas in the second and third queries, one index needs to be updated.

The fourth through sixth queries test the cost of modifying a tuple in three different ways. In all three tests, a non-clustered index exists on the unique2 attribute on both machines, and in addition, in the case of Gamma, a

Table 8
Update Queries
(All Execution Times in Seconds)

	Number of Tuples in Source Relation					
	10,000 Teradata	**10,000 Gamma**	**100,000 Teradata**	**100,000 Gamma**	**1,000,000 Teradata**	**1,000,000 Gamma**
Append 1 Tuple (No indices exist)	0.87	0.18	1.29	0.18	1.47	0.20
Append 1 Tuple (One index exists)	0.94	0.60	1.62	0.63	1.73	0.66
Delete 1 tuple. Using the Key attribute	0.71	0.44	0.42	0.56	0.71	0.61
Modify 1 tuple using the Key attribute	2.62	1.01	2.99	0.86	4.82	1.13
Modify 1 tuple Modified attribute is odd100 - a non-indexed attribute. The key attribute is used to locate the tuple to be modified.	0.49	0.36	0.90	0.36	1.12	0.36
Modify 1 tuple using a non-key attribute with non-clustered index	0.84	0.50	1.16	0.46	3.72	0.52

clustered index exists on the unique1 attribute. In the first case, the modified attribute is the key attribute, thus requiring that the tuple be relocated. Furthermore, since the tuple is relocated, the secondary index must also be updated. The fifth set of queries modify a non-key, nonindexed attribute. The final set of queries modify an attribute on which a non-clustered index has been constructed, using the index to locate the tuple to be modified.

As can be seen from Table 8, for the fourth and sixth queries, both machines use the index to locate the tuple to be modified. Since modifying the indexed attribute value will cause the tuple to move position within the index, some systems avoid using the index to locate the tuple(s) to be modified and instead do a file scan. While one must indeed handle this case carefully, a file scan is not a reasonable solution. Gamma uses deferred update files for indices to handle this problem[7]. We do not know what solution the Teradata machine uses for this problem.

Although Gamma does not provide logging, it does provide deferred update files for updates using index structures. The deferred update file corresponds only to the index structure and not the data file. The overhead of maintaining this functionality is shown by the difference in response times between the first and second rows of Table 8.

9. Conclusions & Future Directions

In this report we presented the results of an initial evaluation of the Gamma database machine by both comparing its performance to that of a Teradata DBC/1012 database machine of similar size and by examining the performance of Gamma relative to the number of processors used. From this comparison, one can draw a number of conclusions regarding both machines. With regard to Gamma, its most glaring deficiencies are the lack of full recovery features and the extremely poor performance of the distributed Simple hash-join algorithm when a large number of overflow operations must be processed as illustrated by the very high response times for the million tuple joins in Table 3 and from the in-depth investigation of join overflow in Section 6.2.2. The solution we are in the process of adopting is to replace the current algorithm with a parallel version of the Hybrid hash-join algorithm [DEWI84, DEWI85]. This algorithm outperforms the Simple hash-join algorithm because it recognizes memory limitations and repartitions the source relations such that no partition of the smaller relation should exceed available memory. Thus the join is broken up into a collection of smaller joins each of which Gamma then computes as a regular distributed join. The Simple hash-join algorithm will be retained as our overflow resolution method because, as

[7] This problem is known as the Halloween problem in DB folklore.

we have just seen, it performs well when the degree of partition overflow is small; which is expected to be the case when using the Hybrid algorithm. We also intend on implementing a recovery server that will collect log records from each processor.

Based on the experiments in which we varied the disk page size used by Gamma, one can conclude that we should increase the default page size from 4 to 8 Kbytes. While increasing the page size beyond 8 Kbytes provides slight improvement for some queries, the impact on queries that use indices (in particular, non-clustered indices) is very negative. While these results may not be generally applicable, they seem to indicate that adopting track-size pages (as a number of experimental systems are talking about doing) may not be a wise decision.

Finally, it is very clear that the network interfaces used in Gamma present a serious bottleneck. We have almost completed the implementation of a co-processor board for our VAX 11/750 that can transfer packets from memory onto the 80 megabits/second token ring at a rate of 40 megabits/second. Once available, this board should eliminate the network interface bottleneck.

Based on these results a number of conclusions can also be drawn about the Teradata database machine. First, the significantly superior performance of Gamma when using clustered indices indicates that this search structure should be implemented. Second, Teradata should incorporate a hash-based join algorithm into their software. While the current sort-merge join algorithms always provide predictable response times, our results indicate that there are situations (ie. no overflows) when hash-join algorithms can provide significantly superior performance.

We are planning a number of projects based on the Gamma prototype during the next year. First, we intend to thoroughly compare the performance of parallel sort-merge and hash join algorithms in the context of Gamma. While the results presented in this paper indicate what sort of results we expect to see, doing the evaluation on one database machine will provide a sounder basis for comparison. Second, since Gamma provides four alternative ways of partitioning relations across the processors with disks, we intend to explore the effect of these different partitioning strategies on the performance of selection and join queries in a multiuser environment. While, for any one query there will always be a preferred partitioning of the relations referenced, we are interested in determining the tradeoff between response time and throughput in a multiuser environment as a function of the different partitioning strategies.

10. Acknowledgements

Like all large systems projects, a large number of people beyond those listed as authors made this paper possible. Bob Gerber deserves special recognition for his work on the design of Gamma plus his leadership on the implementation effort. The query optimizer was implemented by M. Muralikrishna. Rajiv Jauhari implemented a read-ahead mechanism to improve the performance of sequential scans. Anoop Sharma implemented both the aggregate algorithms and the embedded query interface. Goetz Graefe and Joanna Chen implemented a predicate compiler. They deserve special credit for being willing to debug the machine code produced by the compiler. Finally, we would like to thank the Microelectronics and Computer Technology Corporation for their support in funding the study of the Teradata machine described in [DEWI87].

11. References

[ASTR76] Astrahan, M. M., et. al., "System R: A Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 2, June, 1976.

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

[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.

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

[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.

[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.

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

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

[JARK84] Jarke, M. and J. Koch, "Query Optimization in Database System," 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.

[MC^{2}86] Measurement Concepts Corp., "C^{3}I Teradata Study," Technical Report RADC-TR-85-273, Rome Air Development Center, Griffiss Air Force Base, Rome, NY, March 1986.

[NECH83] Neches, P.M., et al., U.S. Patent No. 4,412,285, October 25, 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.

[STON76] Stonebraker, Michael, Eugene Wong, and Peter Kreps, "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, September, 1976.

[TANE81] Tanenbaum, A. S., **Computer Networks**, Prentice-Hall, 1981.

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

[TERA85a] Teradata Corp., *DBC/1012 Data Base Computer System Manual, Rel. 2.0*, Teradata Corp. Document No. C10-0001-02, November 1985.

[TERA85b] Teradata Corp., *DBC/1012 Data Base Computer Reference Manual, Rel. 2.0*, Teradata Corp. Document No. C03-0001-02, November 1985.

[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.