A Single User Evaluation of the Gamma Database Machine

by

David J. DeWitt
Shahram Ghandeharizadeh
Donovan Schneider
Rajiv Jauhair
M. Muralikrishna
Anoop Sharma

Computer Sciences Technical Report # 712
August 1987

A Single User Evaluation of the Gamma Database Machine

David J. DeWitt
Shahram Ghandeharizadeh
Donovan Schneider
Rajiv Jauhari
M. Muralikrishna
Anoop Sharma

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 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

In this report we present the results of an initial evaluation of the Gamma database machine [DEWI86, GERB86]. As a basis of comparison we have used results from a similar study [DEWI87] of the Teradata DBC/1012 database machine [TERA83, TERA85a, TERA85b]. Our objective in writing this paper was to compare the storage organizations and multiprocessor algorithms of the two database machines and **not** to compare their absolute performance. In interpreting the results presented below, the reader should remember that Gamma is not a commercial product at this time and, as such, its results may look slightly better for some queries. The most obvious deficiency in Gamma is that full recovery capabilities have not yet been implemented although distributed concurrency control is provided.

While we have so far limited our efforts to single user tests, we plan on conducting multiuser experiments in the near future. For the most part this work is based on the benchmarking techniques described in [BITT83] (what is popularly known as the "Wisconsin benchmark"), extended to utilize relations commensurate in size with the capabilities of these database machines.

In Sections 2 and 3, respectively, we describe the Teradata and Gamma configurations that were evaluated. Section 4 presents an overview of the benchmark relations used and a discussion of the types of indices used during the benchmark process. 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. See Appendix I in [DEWI87] for the SQL version of the queries used in the benchmark.

2. 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 were used in all IFPs and AMPs.

Each AMP had 2 megabytes of memory and two[1] 8.8", 525 megabyte (unformatted) Hitachi disk drives

(model DK 8155). The host processor was an AMDAHL V570 running the MVS operating system. Release 2.3 of the database machine software was used for the tests conducted on this configuration. While [DEWI87] also evaluated a similar Teradata configuration but with 4 megabytes of memory per processor, we have used the results obtained with the 2 megabyte/AMP configuration as each processor in Gamma also has 2 megabytes of memory.

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[2] 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 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.

3. Overview of the Gamma Database Machine

In this section we present an overview of the Gamma database machine. After describing the current hardware configuration, we present an overview of the software techniques used in implementing Gamma. Included in this discussion is a description of the alternatives provided by Gamma for partitioning relations plus an overview of query execution in Gamma. More 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

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

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

benchmarking process. For a complete description of Gamma see [DEWI86, GERB86].

3.1. Hardware Configuration

Presently, Gamma consists of 17 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring developed for the Crystal project [DEWI84b] by Proteon Associates [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.

3.2. Software Overview

Physical Database Design

All relations in Gamma are **horizontally partitioned** [RIES78] across all disk drives in the system. The Gamma query language (gdl - an extension of QUEL [STONE76]) provides the user with four alternative ways of distributing the tuples of a relation: 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 (as specified in the partition command of gdl) 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. Finally, if the user does not have enough information about his data file to select key ranges, he may elect the final distribution strategy. In this strategy, if the relation is not already loaded, it is initially loaded in a round robin fashion. Next, the relation is sorted (using a parallel merge sort) on the partitioning attribute and the sorted relation is redistributed in a fashion that attempts to equalize the number of tuples at each site. Finally, the maximum key value at each site is returned to the host processor.

Query Execution

Gamma uses traditional relational techniques for query parsing, optimization [SELI79, JARK84], and code generation. The optimization process is somewhat simplified as Gamma only employs hash-based algorithms for joins and other complex operations [DEWI85]. Queries are compiled into a tree of operators. At execution time, each operator is executed by one or more operator processes at each participating site.

After being parsed 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 result of a query is either returned to the user through the ad-hoc query interface or through the embedded query interface to the program from which the query was initiated.

In the case of a multisite query, 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. The root node of a query tree is either a **store** operator in the case of a "retrieve into" query or a **spool** operator in the case of a retrieve query (ie. results are returned to the host). In the case of a **Store** operator, the optimizer will assign a copy of the query tree node to a process at each processor with a disk. Using the techniques described below, the **store** operator at each site receives result tuples from the processes executing the node which is its child in the query tree and stores them in its fragment of the result relation (recall that **all** permanent relations are horizontally partitioned). In the case of a **spool** node at the root of a query tree, the optimizer assigns it to a single process; generally, on a diskless[3] processor.

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

[3] The communications software provides a back-pressure mechanism so that the host can slow the rate at which tuples are being produced if it cannot keep up.

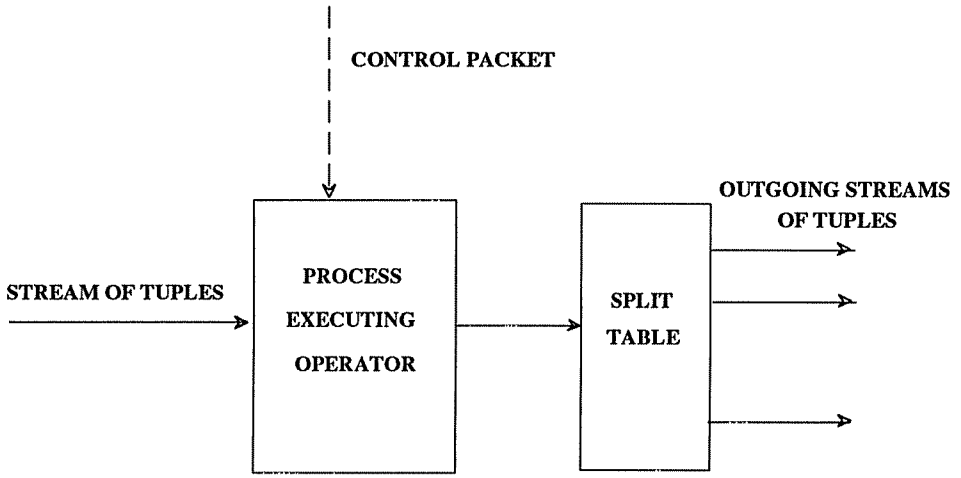

Figure 1

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 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.

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 outer relation [BRAT84, DEWI85, DEWI84a, VALD84]. When the hash table for the outer 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 inner 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 multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys [BLAS79] from occurring. NOSE provides reliable communications between NOSE processes on Gamma processors and to UNIX processes on the host machine. The reliable communications mechanism is a timer-based, one bit stop-and-wait, positive acknowledgment protocol [TANE81]. A delta-T mechanism is used to re-establish sequence numbers [WATS81]. File services in NOSE are based on the Wisconsin Storage System (WiSS) [CHOU85]. Critical sections of WiSS are protected using the semaphore mechanism provided by NOSE.

The file services provided by WiSS include structured sequential files, byte-stream files as in UNIX, B^+ indices, 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.

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. As in the original benchmark, 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. 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 (in which the result relation for a query must be created as a separate step), since unique1 was used as the primary key of both the source and result relations, we had expected that no communications overhead would be incurred in storing the result tuples. However, 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 provides an extended version of the query language QUEL [STON76]. In QUEL, one 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).

Since the Teradata machine provides "more functionality" than Gamma when inserting result tuples from a query into a relation, for those queries which produce a significant number of result tuples, we present the measured execution time of the query plus the estimated execution time of producing the result relation without the cost of either redistributing or actually storing the tuples in the result relation. Our technique for estimating the cost of redistributing and storing the result tuples is described in the following section.

All queries were submitted in scripts. Six query scripts were used: one for each of three relation sizes tested for both indexed and non-indexed cases. For each query submitted, the two machines report the number of tuples affected and the elapsed time. The times reported below represent an average for a set of similar queries using the techniques described in [BITT83]. The benchmark is designed to minimize the effect of the buffer pool on the response time of queries in the same test set in spite of their similarity.

5. Selection

5.1. Overview

The performance of the selection operator is a crucial element of the overall performance of any query plan. If a selection operator provides insufficient throughput, it can become a bottleneck, limiting the amount of parallelism that can effectively be applied by subsequent operators. Both Gamma and Teradata use horizontally partitioned relations and closely coupled processor/disk pairs to achieve parallelism within a selection operator.

Although parallelism improves the performance of the selection operator, it cannot be used as a complete substitute for indices. Gamma provides both clustered and non-clustered B-tree organizations as alternative index structures whereas Teradata offers only dense, secondary indices. As will be shown in the next section, a clustered B-tree organization significantly reduces the execution time for range selection queries even in a parallel database

machine environment.

5.2. Performance

The selection queries were designed with two objectives in mind. First, we wanted to know how the two 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

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

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 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. This seems to contradict the query plan produced by the optimizer which states that non-clustered index on unique2 is to be used to execute the query.

A partial explanation of the above paradox lies in the storage organization used for indices on the Teradata machine. 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 supports the notion of clustered index (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 1% and 10% selection through the clustered index is presented in rows five and six of Table 1. Since the tuples in the actual data 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 amount of 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 that case, 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. Thus, 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. This is largely due to the fact that Teradata provides full recovery for the resulting relation, whereas Gamma does not need to provide this level of functionality. Furthermore, Teradata treats each insertion as a separate operation [DEWI87] (rather than as part of a bulk update), while Gamma pipelines the output of the selection result handling it as a bulk update. Thus, in the Teradata machine the time required to insert tuples into a result relation accounted for a significant fraction of the execution time of the query. We calculated the rate at which tuples can be redistributed and inserted by dividing the difference in the number of tuples selected by the difference between the time to select 10% of the tuples and the time to select 1% of the tuples. For example, on the Teradata machine, the 10% nonindexed selection on a 1,000,000 tuple relation takes 1106.86 seconds and inserts 100,000 tuples into the result relation. The 1% selection takes 213.13 seconds and inserts 10,000 tuples. Since the time to scan the million tuple relation is the

same in both cases, we concluded that the time to redistribute and insert 90,000 tuples is 893.73 seconds. This is an average rate of insertion of 100.7 tuples/second using all 20 disk drives or approximately 5.0 tuples/second/drive. We calculated the insertion rate for relations of different sizes on each machine and then calculated the overall average. The average number of tuples inserted per second for the Teradata machine was 104.08 tuples/sec (5.2 tuples/sec/drive), whereas in the case of Gamma, the average insertion rate was 2096.23 tuples/sec (262.03 tuples/sec/drive).

Since the insertion rate is such a dominant factor in the Teradata results, we decided to separate the processing time for the queries from the time to insert tuples into the result relations. 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. As a result of these calculations, some inconsistencies became apparent (e.g., 10% nonindexed selection on 100,000 tuple relation for Teradata shows a lower response time than 1% nonindexed selection on that same relation). These are largely due to the fact that we used a single constant insertion rate for all our calculations for each machine.

As can be seen from Table 2, for the 100,000 and 1,000,000 tuple relations, the two machines have comparable times for the non-indexed selections. In fact, in the case of 1% nonindexed selections on the million tuple relations the Teradata system is faster than Gamma. The reader should, however, remember that the Teradata

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

machine is running with more than twice as many CPUs and disk drives (Gamma uses 8 disks and 8 CPUs for the selections, whereas the Teradata machine used 20 CPUs and 20 disk drives[4]).

6. Join Queries

The second series of tests consists of a collection of join queries. The Teradata machine uses four alternative join algorithms [TERA85a, MC²86]. One computes an outer-join, while two others are used only in special cases (for example, when the inner relation contains a single tuple). The fourth and most common way in which the Teradata executes join queries involves 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 redistribution phase completes, each AMP uses a conventional sort-merge join algorithm to complete the join. For our test queries, the Teradata used only this fourth join algorithm.

Gamma also partitions its source relations by hashing but instead of using sort-merge to effect the join, it uses a distributed hashing algorithm (see [KITS83, DEWI85, DEWI86, GERB86]). The algorithm works in two phases. During phase one, Gamma partitions the smaller source relation and builds main-memory hash tables. During phase two, Gamma partitions the larger source relation and uses the corresponding tuples to immediately probe within the hash tables built in phase one. Note that as the second source relation is being partitioned its tuples are **not** stored in temporary files but rather are used immediately to probe the hash tables. This data pipelining gives Gamma much of its superior performance.

Of course, whenever main-memory hashing is used there is a danger of hash table overflow. To handle this phenomenon, Gamma currently uses a distributed version of the Simple hash-partitioned join algorithm described in [DEWI85]. 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 this 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 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. As previous analytical models [DEWI85] predicted and the following test results show, this method degrades drastically when the size of the building relation is significantly larger than

[4] Recall that although the Teradata machine actually had 40 disk drives, the 2 drives on each AMP are treated as one logical unit and thus seeks are not overlapped.

the hash table size. For this reason, we will implement the distributed Hybrid hash-join algorithm [DEWI85] in the near future.

Gamma can actually run joins in a variety of configurations. 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. Initial results [DEWI86] showed that offloading the join operators to diskless processors can be done inexpensively freeing the processors with disk for performing operations requiring access to the disk. Future multiuser performance tests will determine the validity of this assumption. The results for all join queries reported below are based on the Remote configuration (8 disk nodes, 8 nondisk nodes and 1 scheduler) in which the joins are done **only** on the diskless processors.

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. As an 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.

The first variation of these three queries that we tested involved no indices and used a non-key attribute (unique2D or unique2E) as the join attribute and selection attributes. Since all the source relations were distributed

using the key attribute (unique1D or unique1E), the join algorithms of both machines required redistribution (partitioning) phases. The results from these tests are contained in the first 3 rows of Tables 3.

The second variation of the three join queries used the key attribute (unique1D or unique1E) as the join attribute. These results are contained in Rows 4 through 6 of Table 3. Since, in this case, the relations are already distributed on the join attribute the Teradata demonstrated substantial performance improvement (25-50%) because the redistribution step of the join algorithm could be skipped. Since Table 3 shows the "Remote" configuration of Gamma all data must still be redistributed to the diskless processors.

Although we are not entirely sure why Gamma showed improvement it is probably due to the fact that, in this second set of queries, the redistribution step maps all tuples from processor 1 (with a disk) to processor 9 (without a disk), and from processor 2 to processor 10, etc. In [GERB86], Bob Gerber showed that, when all Gamma processors send to all other sites in synchronization, the network interfaces can become a temporary bottleneck. The effect of the different Gamma configurations will be discussed in more detail below.

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. Given the quadratic cost of the join operator, this is an impressive result. 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. To demonstrate how costly overflow resolution is, we ran the joinAselB query with 400K of hash table memory per node instead of 600K. The query then required ten partition overflow resolutions and the time rose from the listed 737.7 seconds to 1016.1 seconds.

Bit Vector Filters in Gamma

In [DEWI85], bit vector filtering was analyzed and shown to be extremely effective for a wide spectrum of multiprocessor join algorithms (including distributed sort-merge); the primary benefit being the elimination of non-qualifying probing tuples at their selection sites - saving the costs of sending them over the network and subsequent processing in the join computation. Table 5 show the effects of bit vector filtering for the joinABprime query in Gamma as the configuration is changed. As you can see all three configurations performed substantially better with bit filters.

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	46.5	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	36.3	3,534.5	703.1
joinCselAselB with non-key attributes of A and B used as join attribute	27.8	7.0	191.8	38.4	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	36.9	1,584.3	737.7
joinCselAselB with key attributes of A and B used as join attribute	23.8	7.2	156.7	37.9	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

Table 5
Bit Vector Filtering in Gamma
JoinABprime - 100K Tuple Relations - No Indices - Join on Non-Key Attrs
(All Execution Times in Seconds)

	Local	Remote	Allnodes
without bit filters	56.31	46.53	51.69
with bit filters	42.12	35.26	39.07
%speedup	25%	24%	24%

The joinAselB and joinCselAselB queries show no improvement from bit filtering because the Gamma optimizer is clever enough to propagate the selection predicates for these queries and thus no non-joining tuples will participate in the join (this explains why joinAselB ran significantly faster than joinABprime in Tables 3 and 4). Performance gains similar to those in Table 5 did not occur for the million tuple joinABprime queries because at the present time only a single 2K page is used to hold the bit vectors from all building sites. After deducting the space for the communication protocol header, this corresponds to only 122 bytes (976 bits) for the bit filter for each site. If one assumes that the partitioning phase results in uniformly distributing the 100,000 tuple relation across the 8 building sites, each site will end up with 12,500 tuples; thus overwhelming the bit filters and rendering them useless.

Join Performance of Alternative Gamma Configurations

The choice of Gamma configuration also directly affects the response time of join queries. Gamma's "Local" configuration corresponds most closely to Teradata's configuration. When the join attributes are also the partitioning attributes, no data packets will be sent over the network, i.e. all building relation tuples will be put in their respective local hash tables and all probing tuples will only probe their respective local hash tables. In Table 6, the performance of the joins on the 100K tuple relations with the join attributes being the partitioning attributes is shown. In order to determine the effects of data transmission on the three Gamma configurations we turned off bit vector filtering for the queries. Since the joinABprime query handles the most data its entry will be most interesting. As the joinABprime query results show, shipping large numbers of data packets remotely can be costly. Although the results reported in [DEWI86] showed that remote joins were as cheap as local joins there is no contradiction with these results. In the earlier paper, the relations were range-partitioned on their key attribute instead of hash-partitioned and thus repartitioning was required in every configuration. Its obvious that local joins are very

attractive if the source relations are partitioned on their join attributes.

Table 6
Effect of Alternative Gamma Configurations on Join Execution Time
100K Tuple Relations - No Indices - Join Attributes are Partitioning Attributes
(All Execution Times in Seconds)

	Local	Remote	Allnodes
joinAselB	37.14	35.93	37.24
joinABprime	38.24	45.55	42.19
joinCselAselB	43.01	37.86	42.29

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 tuple joins. Selection propagation by the Gamma optimizer will reduce 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 the entire 100,000 tuples over the network and probe with all these tuples (assuming no bit filters). Recall that Table 5 shows that approximately 24% speedup can be obtained just by reducing the number of probing tuples via bit filters. Thus the costs 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.

Join Summary

As shown by the very high response times for the million tuple joins in Table 3, Gamma must find an alternative for its present use of Simple hash for join overflow resolution. Also, the size of the bit filters should be increased in order to help with very large joins.

Teradata should strongly consider incorporating bit vector filtering techniques and selection propagation. The implementation costs are very low and the potential gains very high. In fact, every page of tuples eliminated via bit filtering saves two disk I/O's in a conventional sort-merge algorithm. Selection propagation definitely improves performance for these join queries by reducing the number of tuples in the join computation. How often selection propagation can be applied in the "real world" is an open question, though.

Finally, after analyzing Tables 3 and 4, and keeping in mind the selection costs reported in Section 5, it is obvious that hash-join outperforms sort-merge when no partition overflow is encountered. This substantiates the analytical results reported in [DEWI85].

7. Aggregate Queries

The third set of queries includes 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

Some relational database systems have special cased the computation of scalar aggregates over indexed attributes by considering just the index as opposed to the base relation. Teradata would benefit little from such a special case operator since its indices are not maintained in key order. On the other hand, Gamma could improve performance but the potential gains were deemed to be less than the implementation costs.

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

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 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 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[5]. 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.

In examining the results in Table 8, one will notice that the cost of performing an update operation in Teradata is sometimes effected by the size of the relation being updated, whereas this is not the case in Gamma. For example in Teradata the time to append a single tuple in a relation with no indices increases as the size of the relation increases.

9. Conclusions & Future Directions

In this report we presented the results of an initial evaluation of the Gamma database machine by comparing its performance to that of a Teradata DBC/1012 database machine of similar size. 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 algo-

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

rithm when a large number of overflow operations must be processed. In the near future we intend on rectifying these deficiencies by implementing a recovery server that will collect log records from each processor and a distributed version of the Hybrid hash join algorithm. Since the Simple hash-join algorithm has superior performance when no overflows occur, the query optimizer will select the appropriate join algorithm based on the expected size of the smaller input relation.

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 bit-vector filtering and a pipelined join strategy 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 provided 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 much more reasonable 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. Goetz Graefe made a number of contributions to the project including the first version of a SARGable predicate compiler. Joanna Chen was responsible for completing the predicate compiler and deserves special thanks 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]. Without this earlier work, this paper would not have been possible.

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.

[BLAS79] Blasgen, M. W., Gray, J., Mitoma, M., and T. Price, "The Convoy Phenomenon," Operating System Review, Vol. 13, No. 2, April, 1979.

[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.

[DEWI84a] 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.

[DEWI84b] DeWitt, D. J., Finkel, R., and Solomon, M., "The Crystal Multicomputer: Design and Implementation Experience," to appear, IEEE Transactions on Software Engineering, August 1987.

[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.

[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.

[WATS81] Watson, R. W., "Timer-based mechanisms in reliable transport protocol connection management" *Computer Networks 5*, pp. 47-56, 1981.