GAMMA - A High Performance Dataflow Database Machine

David J. DeWitt Robert H. Gerber
Goetz Graefe Michael L. Heytens
Krishna B. Kumar M. Muralikrishna

Computer Sciences Department
University of Wisconsin

Abstract

In this paper, we present the design, implementation techniques, and initial performance evaluation of Gamma. Gamma is a new relational database machine that exploits dataflow query processing techniques. Gamma is an operational prototype consisting of 20 VAX 11/750 computers. In addition to demonstrating that parallelism can really be made to work in a database machine context, the Gamma prototype shows how parallelism can be controlled with minimal control overhead through a combination of the use of algorithms based on hashing and the pipelining of data between processes.

1. Introduction

While the database machine field has been a very active area of research for the last 10 years, only a handful of research prototypes [OZKA75, LEIL78, DEWI79, STON79, HELL81, SU82, GARD83, FISH84, KAKU85, DEMU86] and three commercial products [TERA83, UBEL85, IDM85] have ever been built. None have demonstrated that a highly parallel relational database machine can actually be constructed.

In this paper, we present the design of Gamma, a new relational database machine that exploits dataflow query processing techniques. Gamma is a fully operational prototype whose design is based on what we learned from building our earlier multiprocessor database machine prototype (DIRECT) and several years of subsequent research on the problems raised by the DIRECT prototype. Our evaluation of DIRECT [BITT83] showed a number of major flaws in its design. First, for certain types of queries, DIRECT's performance was severely constrained by its limited I/O bandwidth. This problem was exaggerated by the fact that DIRECT attempted to use parallelism as a substitute for indexing. When one looks at indices from the viewpoint of I/O bandwidth and CPU resources, what an index provides is a mechanism to avoid searching a large piece of the database to answer certain types of queries. With I/O bandwidth a critical resource in any database machine [BORA83], the approach used by DIRECT, while conceptually appealing, leads to disastrous performance [BITT83]. The other major problem with DIRECT was that the number of control actions (messages) required to control the execution of the parallel algorithms used for complex relational operations (e.g. join) was proportional to the product of the sizes of the two input relations. Even with message passing implemented via shared memory, the time spent passing and handling messages dominated the processing and I/O time for this type of query.

The remainder of this paper is organized as follows. The architecture of Gamma and the rationale behind this design is presented in Section 2. In Section 3, we describe the process structure of the Gamma software and discuss how these processes cooperate to execute queries. In Section 4 we describe the algorithms and techniques used to implement each of the relational algebra operations. In Section 5, we present the results of our preliminary performance evaluation of Gamma. Our conclusions and future research directions are described in Section 6.

2. Hardware Architecture of GAMMA

2.1. Solutions to the I/O Bottleneck Problem

Soon after conducting our evaluation of the DIRECT prototype, we realized that limited I/O bandwidth was not just a problem with DIRECT. As discussed in [BORA83], changes in processor and mass storage technology have affected all database machine designs. During the past decade, while the CPU performance of single chip microprocessors has improved by at least two orders of magnitude (e.g. Intel 4040 to the Motorola 68020), there has been only a factor of three improvement in I/O bandwidth from commercially available disk drives (e.g. IBM 3330 to IBM 3380). These changes in technology have rendered a number of database machine designs useless and have made it much more difficult to exploit massive amounts of parallelism in any database machine design.

In [BORA83], we suggested two strategies for improving I/O bandwidth. One idea was to use a very large main memory as a disk cache [DEWI84a]. The second was the use of a number of small disk drives in novel configurations as a replacement for large disk drives and to mimic the characteristics of parallel read-out disk drives. A number of researchers have already begun to look at these ideas [SALE84, KIM85, BROW85, LIVN85] and Tandem has a product based on this concept [TAND85].

Although this concept looks interesting, we feel that it suffers from the following drawback. Assume that the approach can indeed be used to construct a mass storage subsystem with an effective bandwidth of, for example, 100 megabytes/second. As illustrated by Figure 1, before the data can be processed it must be routed through an interconnection network (e.g. banyan switch, cross-bar) which must have a bandwidth of at least[1] 100 megabytes/second. If one believes the fabled "90-10" rule, most of the data moved is not needed in the first place.

Figure 2 illustrates one alternative design. In this design, conventional disk drives are used and associated with each disk drive is a processor. With enough disk drives (50 drives at 2 megabytes/second

[1] 100 megabytes/second are needed to handle the disk traffic. Additional bandwidth would be needed to handle processor to processor communications.

each) the I/O bandwidth of the two alternatives will be equivalent. However, the second design has a number of what we consider to be significant advantages. First, the design reduces the bandwidth that must be provided by the interconnection network by 100 megabytes/second. By associating a processor with each disk drive and employing algorithms that maximize the amount of processing done locally, the results in [DEWI85] demonstrate that one can significantly cut the communications overhead. A second advantage is that the design permits the I/O bandwidth to be expanded incrementally. Finally, the design may simplify exploiting improvements in disk technology.

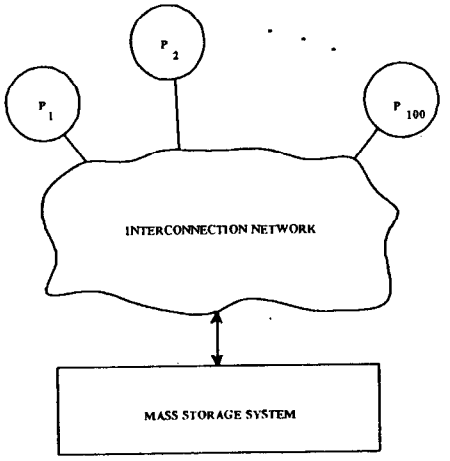


Figure 1

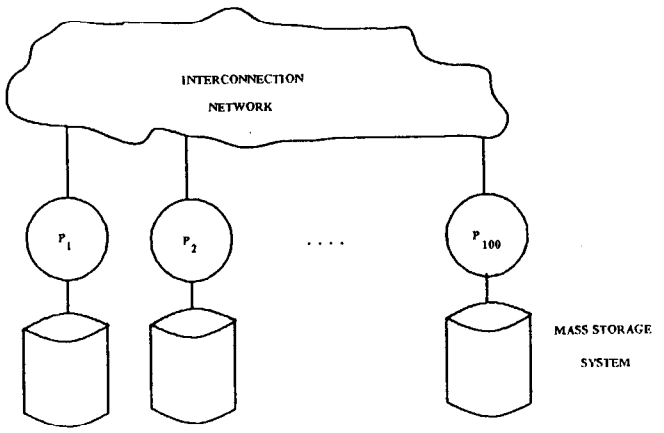


Figure 2

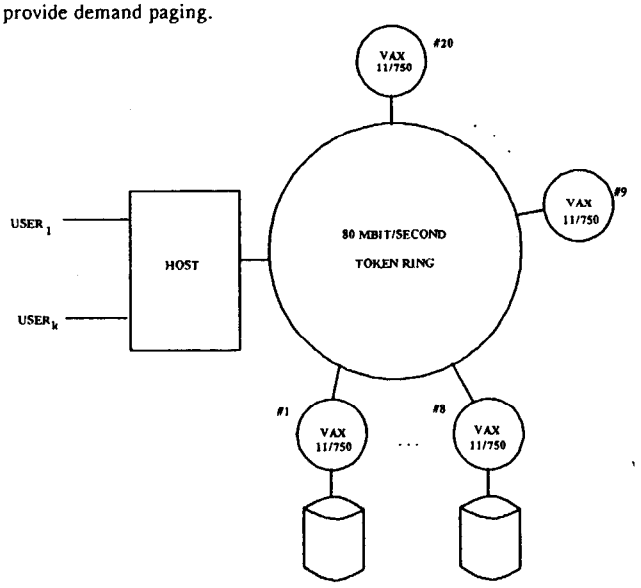
This alternative, on which Gamma is based, seems to have been pioneered by Goodman [GOOD81] in his thesis work on the use of the X-tree multiprocessor for database applications. It is also the basis of several other active database machine projects. In the case of the MBDS database machine [DEMU86], the interconnection network is a 10 megabit/second Ethernet. In the SM3 project [BARU84], the interconnection network is implemented as a bus with switchable shared memory modules. In the Teradata product [TERA83], a tree structured interconnection network termed the Y-net is employed.

2.2. Gamma Hardware

The architecture of the current prototype of the Gamma database machine is shown in Figure 3. Presently, Gamma consists of 20 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring developed for us [DEWI84b] by Proteon Associates is used to connect the processors to each other and to another VAX running Berkeley UNIX. This processor acts as the host machine for Gamma. Attached to eight of the processors are 160 megabyte Fujitsu disk drives (8") which are used for database storage.

2.3. Discussion

One may wonder how Gamma (or MBDS [DEMU86]) is different from a distributed database system running on a local area network. As will become obvious in the next section, Gamma has no notion of site autonomy, has a centralized schema, and a single point for initiating the execution of all queries. Furthermore, the operating system used by Gamma has no capability to dynamically load new programs, has lightweight processes with shared memory, and does not provide demand paging.



Gamma Hardware Configuration
Figure 3

3. Design of the Gamma System Software

3.1. Storage Organization

All relations in Gamma are **horizontally partitioned** [RIES78] across all disk drives in the system. The Gamma query language (gdl - a extension of QUEL [STON76]) 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
- 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 strategy employed in MBDS [DEMU86] and 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. This technique is used by the Terradata database machine [TERA83]. In the third strategy the user specifies a range of key values for each site. For example, with a 4 disk system, the command **partition employee on emp_id (100, 300, 1000)** would result in the following distribution of tuples:

Distribution Condition	Processor #
emp_id ≤ 100	1
100 < emp_id ≤ 300	2
300 < emp_id ≤ 1000	3
emp_id > 1000	4

At first glance, this distribution is similar to the partitioning mechanism supported by VSAM [WAGN73] and the TANDEM file system [ENSC85]. There is, however, a significant difference. In VSAM and in the Tandem file system, if a file is partitioned on a key, then at each site the file must be kept in sorted order on that key. This is not the case in Gamma. In Gamma, there is no relationship between the partitioning attribute of a file and the order of the tuples at a site. To understand the motivation for this capability consider the following banking example. Each tuple contains three attributes: account_#, balance, and branch_#. 90% of the queries fetch a single tuple using account_#. The other 10% of the queries find the current balance for each branch. To maximize throughput, the file would be partitioned on account_#. However, rather than building a clustered index on account_# as would be required with VSAM and the Tandem file system, in Gamma, a clustered index would be built on branch_# and a non-clustered index would be built on account_#. This physical design will provide the same response time for the single tuple queries and a much lower response time for the other queries.

If a 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.

Once a relation has been partitioned, Gamma provides the normal mechanisms for creating clustered (primary) and non-clustered (secondary) indices on each fragment of the relation. However, a special multiprocessor index is constructed when a relation is horizontally partitioned using either of the two range techniques. As shown in Figure 4, the disks, and their associated processors, can be viewed as nodes in a primary, clustered index.[2] The root page of the index is maintained as part of the schema information associated with the index on the host machine. As will be described below, this root page is used by the query optimizer to direct selection queries on the key attribute to the appropriate sites for execution.

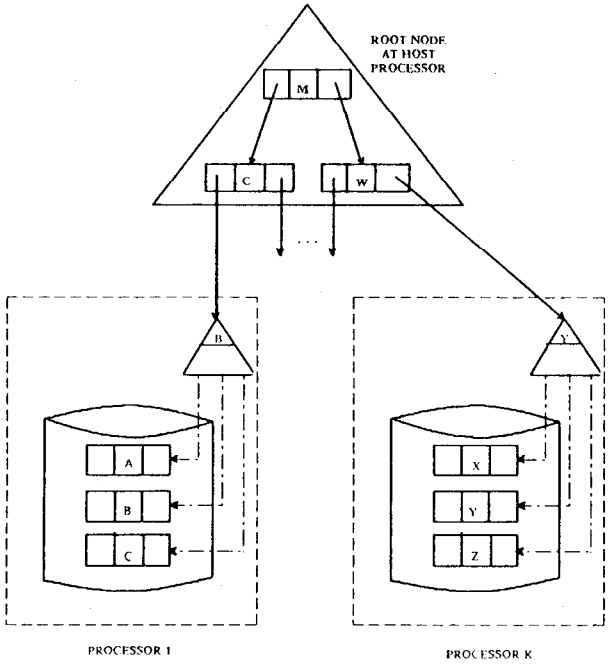
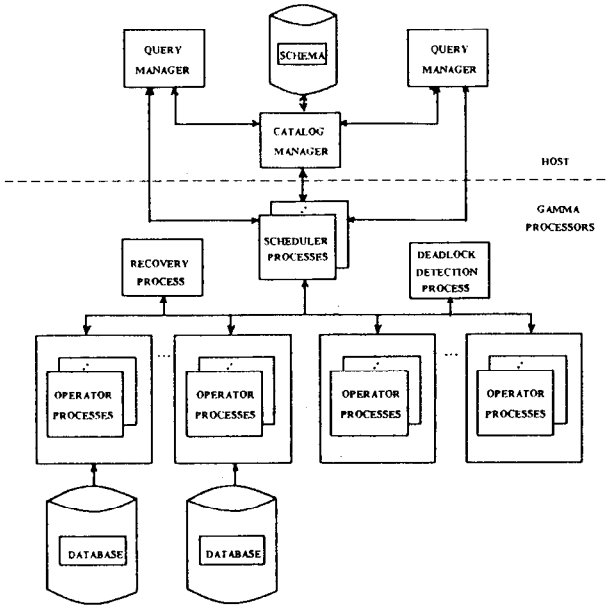


Figure 4

[2] A multiprocessor index may consist of only 1 level if indices have not been created at the disks.

3.2. Gamma Process Structure

In Figure 5, the structure of the various processes that form the software of Gamma is specified. Along with indicating the relationships among the processes, Figure 5 specifies one possible mapping of processes to processors. In discussing the role each process plays in Gamma, we will indicate other alternative ways of mapping Gamma processes to machines. The role of each process is described briefly below. Their interaction is described in more detail in the following section.



Gamma Process Structure
Figure 5

Catalog Manager — The function of the Catalog Manager is to act as a central repository of all conceptual and internal schema information for each database. The schema information is permanently stored in a set of UNIX files on the host and is loaded into memory when a database is first opened. Since multiple users may have the same database open at once and since each user may reside on a machine other than the one on which the Catalog Manager is executing, the Catalog Manager is responsible for insuring consistency among the copies cached by each user.

Query Manager — There is one query manager process associated with each active Gamma user. The query manager is responsible for caching schema information locally, providing an interface for ad-hoc queries using gdl, query parsing, optimization, and compilation.

Scheduler Processes — While executing, each "complex" (i.e. multisite) query is controlled by a scheduler process. This process is responsible for activating the Operator Processes used to execute the nodes of a compiled query tree. Since a message between two query processors is twice as fast as a message between a query processor and the host machine (due to the cost of getting a packet through the UNIX operating system), we elected to run the scheduler processes in the database machine instead of the host. At the present time, all are run on a single processor. Distributing the scheduler processes on multiple machines would be relatively straightforward as the only information shared among them is a summary of available memory for each query processor. This information is centralized to facilitate load balancing. If the schedulers were distributed, access would be accomplished via remote procedure calls.

Operator Process — For each operator in a query tree, at least one Operator Process is employed at each processor participating in the execution of the operator. The structure of an operator process and the mapping of relational operators to operator processes is discussed in more detail below.

Deadlock Detection Process — Gamma employs a centralized deadlock detection mechanism. This process is responsible for collecting fragments of the "wait-for" graph from each lock manager, for locating cycles, and selecting a victim to abort.

Log Manager — The Log Manager process is responsible for collecting log fragments from the query processors and writing them on the log. The algorithms described in [AGRA85] are used for coordinating transaction commit, abort, and rollback.

3.3. An Overview of Query Execution

System Initialization and Gamma Invocation

At system initialization time, a UNIX daemon process for the Catalog Manager (CM) is initiated along with a set of Scheduler Processes, a set of Operator Processes, the Deadlock Detection Process and the Recovery Process. To invoke Gamma, a user executes the command "gdl" from the UNIX shell. Executing this command starts a Query Manager (QM) process which immediately connects itself to the CM process through the UNIX IPC mechanism and then presents a command interpreter interface to the user.

Execution of Database Utility Commands

After parsing a **create database** or **destroy database** command, the QM passes it to the CM for execution. A **create database** command causes the CM to create and initialize the proper schema entries and create the necessary files to hold information on the relations when the database is closed. Although the catalog manager uses UNIX files instead of relations to hold schema information, the catalog structure it employs is that of a typical relational database system. When a **destroy database** command is executed, its actual execution is delayed until all current users of the database have exited. The first step in executing an **open database** command is for the QM to request the schema from the CM. If no other user currently has the requested database open, the CM first reads the schema into memory from disk and then returns a copy of the schema to the requesting QM. The QM caches its copy of the schema locally until the database is closed.

When a user attempts to execute any command that changes the schema of a database (e.g create/destroy relation, build/drop index, partition, etc), the QM first asks the CM for permission. If permission is granted, the QM executes the command, and then informs the CM of the outcome. If the command was executed successfully, the CM records the changes in its copy of the schema and then propagates them to all query managers with the same database open [HEYT85]. A lock manager within the CM ensures catalog consistency.

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.

In the process of optimizing a query, the query optimizer recognizes that certain queries can be executed at a single site. For example, consider a query containing only a selection operation on the relation shown in Figure 4 (assume that q is the name of the attribute on which the relation has been partitioned). If, for example, the selection condition is "$q \geq A$ and $q \leq C$" then the optimizer can use the root page of the multiprocessor index on q to determine that the query only has to be sent to Processor #1.

In the case of a single site query, the query is sent directly by the QM to the appropriate processor for execution. In the case of a multiple site query, the optimizer establishes a connection 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. Once it has established a connection with a scheduler process, the QM sends the compiled query to the scheduler process and waits for the query to complete execution. The scheduler process, in turn, activates operator processes at each query processor selected to execute the operator. Finally, the QM reads the results of the query and returns them through the ad-hoc query interface to the user 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.

3.4. Operator and Process Structure

In Gamma, the algorithms for all operators are written as if they were to be run on a single processor. As shown in Figure 6, 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,

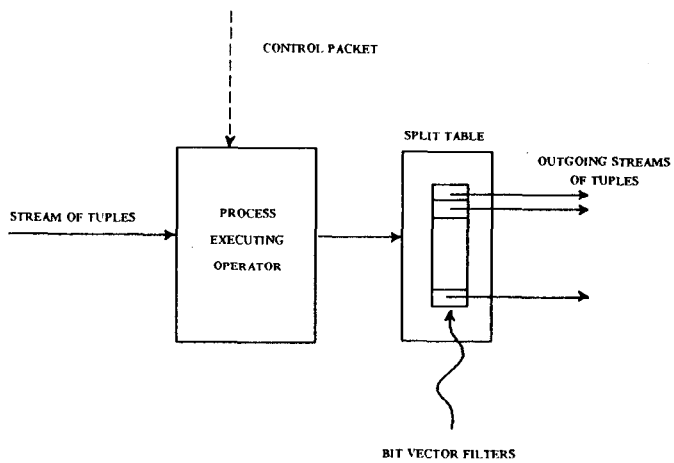


Figure 6

[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.

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. 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.

The split table defines a mapping of values to a set of destination processes. Gamma uses three different types of split tables depending on the type of operation being performed. For example, consider the use of a split table shown in Figure 7 in conjunction with the execution of a join operation using 4 processors. Each process producing source tuples for the join will apply a hash function to the join attribute of each output tuple to produce a value between 0 and 3. This value is then used as an index into the split table to obtain the address of the destination process that should receive the tuple.

Value	Destination Process
0	(Processor #3, Port #5)
1	(Processor #2, Port #13)
2	(Processor #7, Port #6)
3	(Processor #9, Port #15)

An Example Split Table
Figure 7

The second type of split table used by Gamma produces tuple streams that are partitioned on discrete ranges of non-hashed attribute values. In this case, the upper bound of each partition range serves as a key value for each entry in the split table. These range partitioned split tables are used when permanent relations are fragmented using either of the range partitioning strategies described in Section 3.1. These split tables are also applicable when the split attribute targeted by an operation at the leaf of a query tree is the horizontal partitioning attribute (HPA) for a relation. In this case, the split table is initialized with the boundary values defined for the source relation's HPA with the effect that each fragment of the source relation is processed locally. For join operations, if the outer relation is horizontally partitioned on the join attribute, then the relation will generally not be transmitted across the network. In this case, the inner relation of the join would be partitioned and distributed according to the HPA ranges of the fragments of the outer relation at each site.

Gamma uses a third form of split tables when tuples are distributed in a round robin fashion among destination processes. For this distribution strategy, tuples are routed to destination processes represented in the split table, independently of the table's key values.

To enhance the performance of certain operations, an array of bit vector filters [BABB79, VALD84] is inserted into the split table as shown in Figure 6. 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.

3.5. 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 acknowledgement protocol [TANE81]. A delta-T mechanism is used to re-establish sequence numbers [WATS81]. File, record, index, and scan 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. To enhance performance, the page format used by WiSS includes the message format required for interprocessor communications by NOSE. Thus, a page can be read from disk and sent to another processor without requiring that tuples be copied from the page in the buffer pool into an outgoing message template.

4. Query Processing Algorithms

4.1. Selection Operator

The performance of the selection operator is a critical element of the overall performance of any query plan. If a selection operator provides insufficient throughput, then the amount of parallelism that can be effectively applied by subsequent operators is limited. Gamma's use of horizontally partitioned relations and closely coupled processor/disk pairs addresses the I/O bottleneck from a macro-system perspective. However, the efficiency of individual selection operator processes on distinct processors is still important. For a given set of resources, a well-tuned selection operator should provide the necessary throughput to ensure that the rest of the system is effectively utilized.

To achieve the maximum possible throughput, Gamma uses three complementary techniques. First, indices are used whenever possible. Second, selection predicates are compiled into machine language procedures to minimize their execution time. Finally, a limited form of read-ahead is used in order to overlap the processing of one page with the I/O to get the "next" page of the relation from disk.

4.2. Join

The multiprocessor hash-join algorithm used by Gamma is based on a partitioning of source relations into disjoint subsets called **buckets** [GOOD81, KITS83a,b, BRAT84, DEWI84a, VALD84, DEWI85]. The partitioned buckets represent disjoint subsets of the original relations. These partitions have the important characteristic that all tuples with the same join attribute value share the same bucket. The potential power of this partitioning lies in the fact that a join of two large relations can be reduced to the separate joins of many smaller relation buckets [KITS83a,b].

Gamma's Hash Join Algorithm

In Gamma, the tuple streams that are consumed by a hash-join operation are produced by operators using hash-based split tables. These tuple streams partition the tuples based on their join attribute values. As identical split tables are applied to both source relations of a join, a join of two large relations is reduced to the separate joins of many smaller relation buckets. In Gamma, these separate, independent joins provide a natural basis for parallelism. The following discussion considers the details of how this parallelism is achieved and exploited.

Hash-join operators are activated in a manner that is uniform with all other operators. There is an additional control interaction, however, that is unique to the hash-join operator. This control message is required because there are two distinct phases to the hash-partitioned join algorithm. In the first phase, termed the **building phase**, the join operator accepts tuples from the first source relation and uses the tuples to build in-memory hash tables and bit vector filters. At the end of this building phase, a hash-join operator sends a message to the scheduler indicating that the building phase has been completed. Once the scheduler determines that all hash-join operators have finished the building phase, the scheduler sends a message to

each join operator directing the operators to begin the second phase of the operation, the **probing phase**. In this phase, individual join operator processes accept tuples from the second source relation. These tuples are used to probe the previously built hash-tables for tuples with matching join attribute values. At the end of the probing phase, each of the join operators sends a message to the scheduler indicating that the join operation has been completed.

An important characteristic of this algorithm is the simplicity of the interactions between the scheduler and participating operator processes. The net cost of activating and controlling a hash-join operator is five messages per site. (In effect, the building and probing phases of a join operation are considered separate operators for purposes of control.) All other data transfers can proceed without further control intervention by the scheduler.

In addition to controlling individual join operators, the scheduler must also synchronize the join with the operators of adjacent nodes in the query tree. In particular, the scheduler must initiate the operators which will be producing the input tuple streams for the join. The production of these tuple streams must coincide with the activation of the building and probing phases of the hash-join operator.

Hash Table Overflow

During the building phase of the multiprocessor hash-join algorithm, if buckets grow unacceptably large, the in-memory hash tables may overflow. The choice of an appropriate hash function will tend to randomize the distribution of tuples across buckets and, as such, will minimize the occurrence of hash table overflow. Additionally, in order to decrease the likelihood of hash table overflow, the optimizer attempts to build query trees that minimize the size of the relations that are accessed during the building phase of the hash-join algorithm. For joins in the interior of a query tree, this is a difficult task.

When hash-table overflow occurs, a local join operator narrows the dimensions of the tuple partition that is used for the construction of the hash table, in effect creating two subpartitions. One subpartition is used for hash table construction and the other is dumped to an overflow file on disk (possibly remote). Tuples that have already been added to the hash table, but now belong to the overflow subpartition, are removed from the table. As subsequent tuples are read from the input stream, they are either added to the hash table or appended to the overflow file.

When the local join operator notifies the scheduler of the completion of the building phase, it identifies the repartitioning scheme that was used for the handling of any overflow condition. With this knowledge the scheduler can alter the split tables of the second, probing source relation in such a manner that the overflow subpartitions are directly spooled to disk, bypassing the join operators. After the initial non-overflow subpartitions have been joined, the scheduler recursively applies the join operation to the spooled, overflow subpartitions. This method will fail in the case that the combined sizes of tuples having identical join attribute values exceeds the size of available memory. In such a case, a hash-based variation of the nested loops join algorithm is applied [BRAT84, DEWI85].

4.3. Update Operators

For the most part, the update operators (replace, delete, and append) are implemented using standard techniques. The only exception is a replace operation that modifies the partitioning attribute. In this case, rather than writing the modified tuple back into the local fragment of the relation, the modified tuple is passed through a split table to determine where the modified tuple should reside.

5. Performance Evaluation

In this section, we present the results of our preliminary performance evaluation of Gamma. This evaluation is neither extensive nor exhaustive. For example, we have not yet conducted any multiuser tests. Rather, these tests only serve to demonstrate the feasibility of the hardware architecture and software design of Gamma. Concerns of correctness rather than absolute speed have necessarily dominated the current phase of development.

All our tests were run with the host in single user mode. Elapsed time at the host was the principal performance metric. This value was measured as the time between the points at which the query was entered by the user and the point at which it completed execution.

5.1. Test Database Design and Results

The database used for these tests is based on the synthetic relations described in [BITT83]. Each relation consists of ten thousand tuples of 208 bytes. Each tuple contains thirteen, four byte integer attributes followed by three, 52 byte character string attributes.

All permanent relations used in the following tests have been horizontally partitioned on attribute Unique1 which is a candidate key with values in the range 0 through 9,999. Range partitioning was used to equally distribute tuples among all sites. For example, in a configuration with four disks, all tuples with Unique1 values less than 2500*i reside on disk i. All result relations are distributed among all sites using round-robin partitioning. The presented response times represent an average for a set of queries designed to ensure that each query i leaves nothing in a buffer pool of use to query i+1.

5.2. Selection Queries

In evaluating the performance of selection queries in a database machine that supports the concept of horizontal partitioning and multiprocessor indices, one must consider a number of factors: the selectivity factor of the query, the number of participating sites, whether or not the qualified attribute is also the horizontal partitioning attribute, which partitioning strategy has been utilized, whether or not an appropriate index exists, and the type of the index (clustered or non-clustered). If the qualified attribute is the horizontal partitioning attribute (HPA) and the relation has been partitioned using one of the range partitioning commands, then the partitioning information can be used to direct selection queries on the HPA to the appropriate sites. If the hashed partitioning strategy has been chosen, then exact match queries (e.g. HPA = value) can be selectively routed to the proper machine. Finally, if the round-robin partitioning strategy has been selected, the query must be sent to all sites. If the qualified attribute is not the HPA, then the query must also go to all sites.

To reduce the number of cases considered in this preliminary evaluation, we restricted our attention to the following four classes of selection queries:

	selection clause on	clustered index on
S1	Unique1 (HPA)	no index
S2	Unique1 (HPA)	Unique1
S3	Unique2 (non-HPA)	no index
S4	Unique2 (non-HPA)	Unique2

We restricted classes S1 and S2 further by designing the test queries such that the operation is always executed on a single site. This was accomplished by having the horizontal partitioning ranges cover the qualifications of the selection queries. Since queries in both classes S3 and S4 reference a non-HPA attribute, they must be sent to every site for execution. Each of the selection tests retrieved 1,000 tuples out of 10,000 (10% selectivity). The result relation of each query was partitioned in a round-robin fashion across all sites (regardless of how many sites participated in the actual execution of the query). Thus, each selection benefits equally from the fact that increasing the number of disks decreases the time required for storing the result relation.

The results from these selection tests are displayed in Figure 8. For each class of queries, the average response time is plotted as a function of the number of processors (with disks) used to execute the query. Figure 8 contains a number of interesting results. First, as

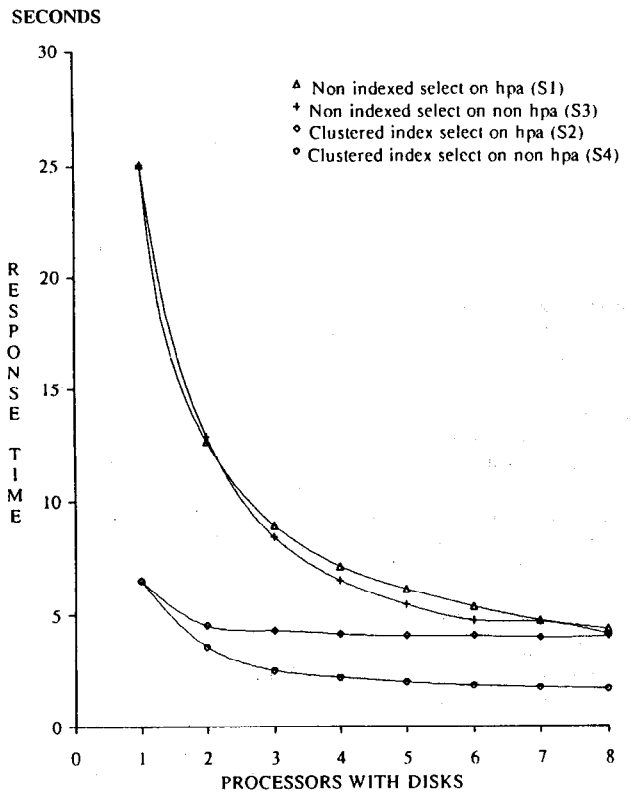


Figure 8

the number of processors is increased, the execution time of S1 and S3 queries drops. This decrease is due to the fact that as the number of processors is increased, each processor scans proportionally less data. Since the entire relation is always scanned in the S3 case, the results for S3 indicate that parallel, non-indexed access can provide acceptable performance for large multiprocessor configurations when there is sufficient I/O bandwidth available.

It is important to understand the difference between the S1 and S3 queries in Figure 8. While both have approximately the same response time, S1 would have a higher throughput rate in a multiuser test since only a single processor is involved in executing the query (assuming, of course, that the queries were uniformly distributed across all processors).

At first, we were puzzled by the fact that S3 was slightly faster than S1. In fact, one might have expected exactly the opposite result due to the overhead (in S3) of initiating the query at multiple sites. In both cases, each processor scans the same number of source tuples. In addition, since the result relation (which has the same size in both cases) is partitioned across all sites, the cost of storing the result relation is the same. The difference seems to be the number of processors used to distribute the tuples in the result relation. In case S1, one processor produces all the result tuples which must be distributed to the other sites. In case S3, all processors produce approximately the same number of result tuples (since Unique2 attribute values are randomly ordered when the file is horizontally partitioned on Unique1). Thus, the cost of distributing the result tuples is spread among all the processors. This explains why the gap between the S1 and S3 curves widens slightly as the number of processors is increased. The anomaly in the curves that occurs when 7 or 8 processors are used is discussed in Section 5.4.

Cases S2 and S4 illustrate different effects of horizontal partitioning and physical database design on response time. In the case of single site, indexed selections on the partitioning attribute (such as S2), increasing the number of disks (and, hence, decreasing the size of the relation fragment at each site) only decreases the cost of the index traversal (by reducing the number of levels in the index) and not the number of leaf (data) pages retrieved from disk. While this effect might be noticeable for single tuple retrievals, the number of levels in the index does not change across the range of sites evaluated. Instead, we attribute the drop in response time as the number of processors is increased from 1 to 2 as a consequence of increasing the number of disks used to store the result relation. Thus, scanning the source relation on site 1 can be partially overlapped with storing half the result relation on site 2. As the number of processors is increased from 2 to 3 one sees a very slight improvement. After three processors, little or no improvement is noticed as the single processor producing the result relation becomes the bottleneck.

In the case of S4 (an indexed selection on a non-partitioning attribute), the query is executed at every site. Since Unique2 attribute values are randomly distributed across all sites, each processor produces approximately the same number of result tuples. Thus, as the number of sites is increased, the response time decreases. The performance of case S4 relative to S3 illustrates how parallelism and indices can be used to complement each other.

An observant reader might have noticed that while the speedup factors for S1 and S3 are fairly close to linear, there is very little improvement in response time for S4 when the number of processors is doubled from 4 to 8. Given the way queries in class S4 are executed, it would be reasonable to expect a linear speedup in performance as the number of processors is increased. The reason this does not occur, while a little difficult to describe, is quite interesting. First, it is not a problem of communications bandwidth. Consider a 4 processor system. Each site produces produces approximately 1/4 of the result relation. Of these 250 tuples, each site will send 63 to each of the other three sites as result tuples are always distributed in a round-robin fashion. Thus, a total of 750 tuples will be sent across the network. At 208 bytes/tuple, this is a total of 1.2 million bits. At 80 million bits/second, approximately 2/100s of a second is required to redistribute the result relation.

The problem, it seems, is one of congestion at the network interfaces. Currently, the round-robin distribution policy is implemented by distributing tuples among the output buffers on a tuple-by-tuple basis. At each site in an 8 processor system, 8 qualifying tuples can change the state of the 8 output buffers from *non-empty* to *full*. Since the selection is through a clustered index, these 8 tuples may very well come from a single disk page or at most two pages. Thus, with 8 processors, 64 output buffers will become full at almost exactly the same time. Since the network interface being used at the current time has buffer space for only two incoming packets, five packets to each site have to be retransmitted (the communications software short-circuits a transmission by a processor to itself). The situation is complicated further by the fact that the acknowledgments for the 2 messages that do make it through, have to compete with retransmitted packets to their originating site (remember, everybody is sending to everybody). Since it is likely that some of the acknowledgements will fail to be received before the transmission timer goes off, the original packets may be retransmitted even though they arrived safely.

One way of at least alleviating this problem is to use a page-by-page round-robin policy. By page-by-page, we mean that the first output buffer is filled before any tuples are added to the second buffer. This strategy, combined with a policy of randomizing to whom a processor sends its first output page, should improve performance significantly as the production of output pages will be more uniformly distributed across the execution of the operation.

Rather than fixing the problem and rerunning the tests, we choose to leave this rather negative result in the paper for a couple of reasons. First, it illustrates how critical communication's issues can be. One of the main objectives in constructing the Gamma prototype is to enable us to study and measure interprocessor communications so that we can develop a better understanding of the problems involved in scaling the design to larger configurations. By sweeping the problem

under the rug, Gamma would have looked better but an important result would have been lost (except to us). Second, the problem illustrates the importance of single user benchmarks. The same problem might not have showed up in a multiuser benchmark as the individual processors would be much less likely to be so tightly synchronized.

As a point of reference, the IDM500 database machine (10 MHz CPU with a database accelerator and an equivalent disk) takes 22.3 seconds for S1 selections. The IDM500 time for S2 selections is 5.2 seconds. Finally, the time in Gamma to retrieve a single tuple using a multiprocessor index such as that used for S2 is 0.14 seconds.

5.3. Join Queries

As with selection queries, there are a variety of factors to consider in evaluating the performance of join operations in Gamma. For the purposes of this preliminary evaluation, we were particularly interested in the relative performance of executing joins on processors with and without disks. We used the following query as the basis for our tests:

```
range of X is tenKtupA
range of Y is tenKtupB
retrieve into temp (X.all, Y.all)
where (X.Unique2A = Y.Unique2B) and (Y.Unique2B < 1000)
```

Each relation was horizontally partitioned on its Unique1 attribute. Execution of this query proceeds in two steps. First, the building phase of the join (see Section 4.2) is initiated. This phase constructs a hash table using the tenKtupA relation on each processor participating in the execution of the join operator. Ordinarily, the optimizer chooses the smallest source relation (measured in bytes) for processing during the building phase. In this query, the source relations can be predicted to be of equal size as the qualification on the tenKtupB relation can be propagated to the tenKtupA relation.

Once the hash tables have been constructed and the bit vector filters have been collected and distributed by the scheduler, the second phase begins. During this phase, the selection on tenKtupB is executed concurrently with the probing phase of the join operation.

Since Unique2 is not the HPA for either source relation, all sites participate in the execution of the selection operations. The relations resulting from the selection and join operations contain 1,000 tuples.

To reduce the number of cases considered, joins were either performed solely on processors with disks attached or solely at processors without disks. For convenience, we refer to these joins, respectively, as local joins and remote joins. We performed four sets of joins with the following characteristics:

	clustered indices on	join performed at processors
J1	no index	without disks (remote)
J2	no index	with disks (local)
J3	Unique2B	without disks (remote)
J4	Unique2B	with disks (local)

The results of these join tests are displayed in Figure 9. For each class of queries the average response time is plotted as a function of the number of processors with disks that are used. For the remote joins, an equal number of processors without disks are also used. Figure 9 demonstrates that there is not a performance penalty for joining tuples on sites remote from the source of the data. In fact, joins on processors without disks are actually slightly faster than those performed on processors with disks. The following discussion addresses this somewhat counterintuitive, but intriguing result.

Two factors contribute to making remote joins slightly faster than local joins (with respect, at least, to a response time metric). First, when joins are performed locally, the join and select operators compete with each other for CPU cycles from the same processor. Second, since Gamma can transfer sequential streams of tuples between processes on two different processors at almost the same rate

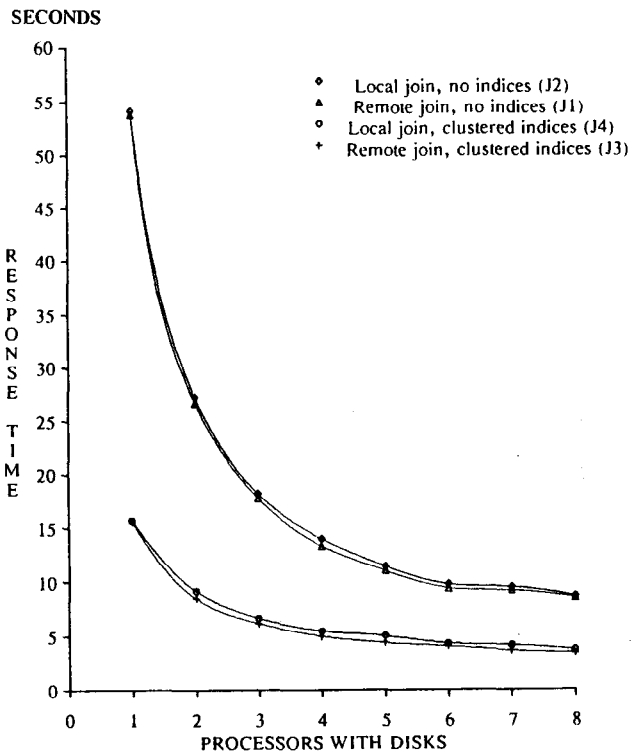


Figure 9

as between processes on the same machine, there is only a very minor response time penalty for executing operations remotely. Additional CPU cycles are, however, consumed while executing the communications protocol. Thus, there is likely to be a loss in throughput in a multiuser environment. We intend to explore the significance of this loss in future benchmark tests.

Since twice as many processors are used by the remote join design, one might wonder why the response times for remote joins were not half those of the corresponding local joins. Since the building and probing phases of the join operator are not overlapped, the response time of the join is bounded by the sum of the elapsed times for the two phases. For the cases tested, it turns out that the execution time for the building and probing phases is dominated by the selections on the source relations. There is, however, another benefit that accrues from offloading the join operator that is not reflected in a response time metric. When the join operation is offloaded, the processors with disks can effectively support a larger number of concurrent selection and store operations.

While remote joins only marginally outperform local joins, we consider the implications significant. Having demonstrated that a complex operation such as a join can be successfully offloaded from processors with disks provides a basis for expanding the design spectrum for multiprocessor database machines.

As a point of reference for the join times of Figure 9, the IDM500 took 84.3 seconds for J2 joins and 14.3 seconds for joins of type J4.

5.4. Speedup of Join Elapsed Times

In Figure 10, response-time speedup curves are presented for the join tests described in the previous section. These results confirm our hopes that multiprocessor, partitioned hash-join algorithms can effectively provide a basis for a highly parallel database machine. The anomalous shape of the speedup curves for systems with 7 or 8 disks can be attributed to two factors. First, the seventh and eighth disks

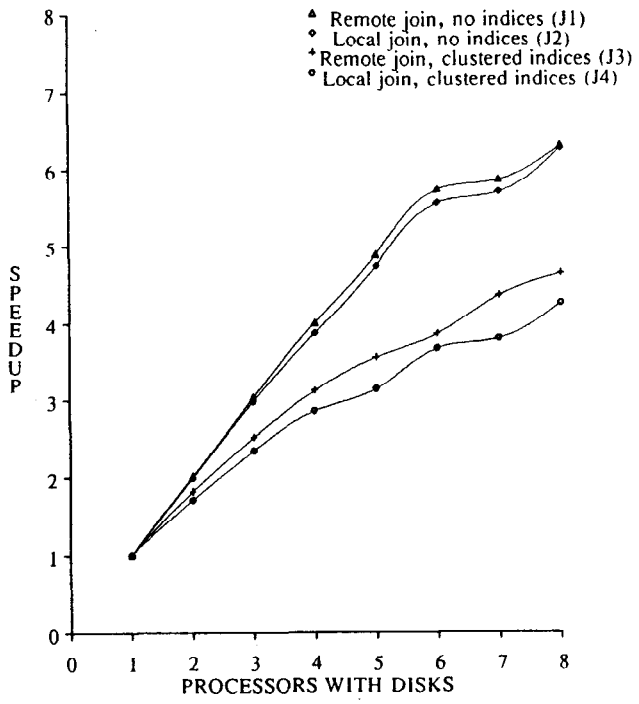


Figure 10

that were added to the system have only 82%[4] the performance of each of the other six disks. With evenly partitioned source relations, these slower disks increased the time required for scanning each of the source relations. All of this additional time is directly reflected in increased response times for join operations because the building and probing phases of the join operation are not overlapped.

A second factor also contributes to the shape of the speedup curve for systems with large numbers of processors. The ratio of control messages to data messages per processor increases as processors are added to the system. This factor only becomes significant once the volume of tuples processed by each processor becomes small. In the join tests presented, this effect may become noticeable when as few as eight disks are used because the Gamma query optimizer recognizes that the qualification on the first source relation (tenKtupB) can be propagated to tenKtupA. Therefore, only 1000 tuples are produced by the selections on each source relation. When join operators are active on eight processors, this means that each join operator will process approximately fourteen data pages from each relation and five control messages.

The reduced (and less impressive) speedup factors for joins J3 and J4 appear to be a consequence of the reduced speedup obtained for selection S4 which is executed as part of join queries J3 and J4 (see Section 5.2). As discussed above, for the join tests conducted, the execution time for the building and probing phases of the join is dominated by the selections on the source relations.

As Gamma enters a more mature stage of development, further speedup results will be obtained from queries that generate more massive amounts of data. For the current time, we present the speedup data for purposes of illustrating the potential that the system promises.

6. Conclusions

In this paper we have presented the design of a new relational database machine, Gamma. Gamma's hardware design is quite simple. Associated with each disk drive is a processor and the processors are interconnected via an interconnection network. The initial prototype consists of 20 VAX 11/750 processors interconnected with an 80 megabit/second token ring. Eight of the processors have a 160 megabyte disk drive. This design, while quite simple, provides high disk bandwidth without requiring the use of unconventional mass storage systems such as parallel read-out disk drives. A second advantage is that the design permits the I/O bandwidth to be expanded incrementally. To utilize the I/O bandwidth available in such a design, all relations in Gamma are horizontally partitioned across all disk drives.

In order to minimize the overhead associated with controlling intraquery parallelism, Gamma exploits dataflow query processing techniques. Each operator in a relational query tree is executed by one or more processes. These processes are placed by the scheduler on a combination of processors with and without disk drives. Except for 3 control messages, 2 at the beginning of the operator and 1 when the operator terminates execution, data flows between between the processes executing the query without any centralized control.

The preliminary performance evaluation of Gamma is very encouraging. The design provides almost linear speedup for both selection and join operations as the number of processors used to execute an operation is increased. Furthermore, the results obtained for a single processor configuration were demonstrated to be very competitive with a commercially available database machine. Once we have completed the prototype, we plan on conducting a thorough evaluation of the single and multiuser performance of the system. This evaluation will include both more complex queries and non-uniform distributions of attribute values.

Acknowledgements

This research was partially supported by the Department of Energy under contract #DE-AC02-81ER10920, by the National Science Foundation under grants, DCR-8512862, MCS82-01870 and MCS81-05904, and by a Digital Equipment Corporation External Research Grant.

7. References

[AGRA85] Agrawal, R., and D.J. DeWitt, "Recovery Architectures for Multiprocessor Database Machines," Proceedings of the 1985 SIGMOD Conference, Austin, TX, May, 1985.

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

[BARU84] Baru, C. K. and S.W. Su, "Performance Evaluation of the Statistical Aggregation by Categorization in the SM3 System," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[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.

[BORA83] Boral H. and D. J. DeWitt, "Database Machines: An Idea Whose Time has Passed," in **Database Machines**, edited by H. Leilich and M. Missikoff, Springer-Verlag, Proceedings of the 1983 International Workshop on Database Machines, Munich, 1983.

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

[BROW85] Browne, J. C., Dale, A. G., Leung, C. and R. Jenevein, "A Parallel Multi-Stage I/O Architecture with Self-Managing Disk Cache for Database Management Applications," in **Database Machines: Proceedings of the 4th International Workshop**, Springer Verlag, edited by D. J. DeWitt and H. Boral, March, 1985.

[4] This value was determined by measuring the elapsed time of scanning a 10,000 tuple relation on the two sets of disk drives. While all the drives are 160 megabyte Fujitsu drives, six are newer 8" drives while the other two are older 14" drives.

[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.

[DEMU86] Demurjian, S. A., Hsiao, D. K., and J. Menon, "A Multi-Backend Database System for Performance Gains, Capacity Growth, and Hardware Upgrade," Proceedings of Second International Conference on Data Engineering, Feb. 1986.

[DEWI79] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June, 1979.

[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. Also University of Wisconsin-Madison Computer Sciences Department Technical Report, September, 1984.

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

[ENSC85] "Enscribe Programming Manual," Tandem Part# 82583-A00, Tandem Computers Inc., March 1985.

[FISH84] Fishman, D.H., Lai, M.Y., and K. Wilkinson, "Overview of the Jasmin Database Machine," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[GARD83] Gardarin, G., et. al., "Design of a Multiprocessor Relational Database System," Proceedings of the 1983 IFIP Conference, Paris, 1983.

[GOOD81] Goodman, J. R., "An Investigation of Multiprocessor Structures and Algorithms for Database Management", University of California at Berkeley, Technical Report UCB/ERL, M81/33, May, 1981.

[HELL81] Hell, W. "RDBM - A Relational Database Machine," Proceedings of the 6th Workshop on Computer Architecture for Non-Numeric Processing, June, 1981.

[HEYT85a] Heytens, M., "The Gamma Query and Catalog Managers," Gamma internal design documentation, December, 1985.

[IDM85] The IDM 310 Database Server, Britton-Lee Inc., 1985.

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

[KAKU85] Kakuta, T., Miyazaki, N., Shibayama, S., Yokota, H., and K. Murakami, "The Design and Implementation of the Relational Database Machine Delta," in **Database Machines: Proceedings of the 4th International Workshop**, Springer Verlag, edited by D. DeWitt and H. Boral, March, 1985.

[KIM85] Kim, M. Y, "Parallel Operation of Magnetic Disk Storage Devices," in **Database Machines: Proceedings of the 4th International Workshop**, Springer Verlag, edited by D. DeWitt and H. Boral, March, 1985.

[KITS83a] 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.

[KITS83b] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, "Architecture and Performance of Relational Algebra Machine Grace", University of Tokyo, Technical Report, 1983.

[LEIL78] Leilich, H.O., G. Stiege, and H.Ch. Zeidler, "A Search Processor for Database Management Systems," Proceedings of the 4th VLDB International Conference, 1978.

[LIVN85] Livny, M., Khoshafian, S., and H. Boral, "Multi-Disk Management Algorithms," Proceedings of the International Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1985.

[OZKA75] Ozkarahan E.A., S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," Proc. 1975 NCC, Vol. 45, AFIPS Press, Montvale N.J.

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

[SALE84] Salem, K., and H. Garcia-Molina, "Disk Striping", Technical Report No. 332, EECS Department, Princeton University, December 1984.

[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.

[STON79] Stonebraker, M. R., "MUFFIN: A Distributed Database Machine," University of California, Electronics Research Laboratory, Memorandum UCB/ERL M79/28, May 1979.

[SU82] Su, S.Y.W and K.P. Mikkilineni, "Parallel Algorithms and their Implementation in MICRONET", Proceedings of the 8th VLDB Conference, Mexico City, September, 1982.

[TAND85] 4120-V8 Disk Storage Facility, Tandem Computers Inc., 1985.

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

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

[UBEL85] Ubell, M., "The Intelligent Database Machine (IDM)," in **Query Processing in Database Systems**, edited by Kim, W., Reiner, D., and D. Batory, Springer-Verlag, 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.

[WAGN73] Wagner, R.E., "Indexing Design Considerations," IBM System Journal, Vol. 12, No. 4, Dec. 1973, pp. 351-367.

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