The Gamma Database Machine Project

David J. DeWitt
Shahram Ghandeharizadeh
Donovan Schneider
Allan Bricker
Hui-I Hsiao
Rick Rasmussen

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 grant DCR-8512862, by a DARPA/NASA sponsored Graduate Research Assistantship in Parallel Processing, and by research grants from Intel Scientific Computers, Tandem Computers, and Digital Equipment Corporation.

ABSTRACT

This paper describes the design of the Gamma database machine and the techniques employed in its implementation. Gamma is a relational database machine currently operating on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives. Gamma employs three key technical ideas which enable the architecture to be scaled to 100s of processors. First, all relations are horizontally partitioned across multiple disk drives enabling relations to be scanned in parallel. Second, novel parallel algorithms based on hashing are used to implement the complex relational operators such as join and aggregate functions. Third, dataflow scheduling techniques are used to coordinate multioperator queries. By using these techniques it is possible to control the execution of very complex queries with minimal coordination - a necessity for configurations involving a very large number of processors.

In addition to describing the design of the Gamma software, a thorough performance evaluation of the iPSC/2 hypercube version of Gamma is also presented. In addition to measuring the effect of relation size and indices on the response time for selection, join, aggregation, and update queries, we also analyze the performance of Gamma relative to the number of processors employed when the sizes of the input relations are kept constant (speedup) and when the sizes of the input relations are increased proportionally to the number of processors (scaleup). The speedup results obtained for both selection and join queries are linear; thus, doubling the number of processors halves the response time for a query. The scaleup results obtained are also quite encouraging. They reveal that a nearly constant response time can be maintained for both selection and join queries as the workload is increased by adding a proportional number of processors and disks.

1. Introduction

For the last 5 years, the Gamma database machine project has focused on issues associated with the design and implementation of highly parallel database machines. In a number of ways, the design of Gamma is based on what we learned from our earlier database machine DIRECT [DEWI79]. While DIRECT demonstrated that parallelism could be successfully applied to processing database operations, it had a number of serious design deficiencies that made scaling of the architecture to 100s of processors impossible; primarily the use of shared memory and centralized control for the execution of its parallel algorithms [BITT83].

As a solution to the problems encountered with DIRECT, Gamma employs what appear today to be relatively straightforward solutions. Architecturally, Gamma is based on a shared-nothing [STON86] architecture consisting of a number of processors interconnected by a communications network such as a hypercube or a ring, with disks directly connected to the individual processors. It is generally accepted that such architectures can be scaled to incorporate 1000s of processors. In fact, Teradata database machines [TERA85] incorporating a shared-nothing architecture with over 200 processors are already in use. The second key idea employed by Gamma is the use of hash-based parallel algorithms. Unlike the algorithms employed by DIRECT, these algorithms require no centralized control and can thus, like the hardware architecture, be scaled almost indefinitely. Finally, to make the best of the limited I/O bandwidth provided by the current generation of disk drives, Gamma employs the concept of **horizontal partitioning** [RIES78] (also termed **declustering** [LIVN87]) to distribute the tuples of a relation among multiple disk drives. This design enables large relations to be processed by multiple processors concurrently without incurring any communications overhead.

After the design of the Gamma software was completed in the fall of 1984, work began on the first prototype which was operational by the fall of 1985. This version of Gamma was implemented on top of an existing multicomputer consisting of 20 VAX 11/750 processors [DEWI84b]. In the period of 1986-1988, the prototype was enhanced through the addition of a number of new operators (e.g. aggregate and update operators), new parallel join methods (Hybrid, Grace, and Sort-Merge [SCHN89a]), and a complete concurrency control mechanism. In addition, we also conducted a number of performance studies of the system during this period [DEWI86, DEWI88, GHAN89, GHAN90]. In the spring of 1989, Gamma was ported to a 32 processor Intel iPSC/2 hypercube and the VAX-based prototype was retired.

Gamma is similar to a number of other active parallel database machine efforts. In addition to Teradata [TERA85], Bubba [COPE88] and Tandem [TAND88] also utilize a shared-nothing architecture and employ the concept of horizontal partitioning. While Teradata and Tandem also rely on hashing to decentralize the execution of their parallel algorithms, both systems tend to rely on relatively conventional join algorithms such as sort-merge for processing the fragments of the relation at each site. Gamma, XPRS [STON88], and Volcano [GRAE89] each utilize parallel versions of the Hybrid join algorithm [DEWI84a].

The remainder of this paper is organized as follows. In Section 2 we describe the hardware used by each of the Gamma prototypes and our experiences with each. Section 3 discusses the organization of the Gamma software and describes how multioperator queries are controlled. The parallel algorithms employed by Gamma are described in Section 4 and the techniques we employ for transaction and failure management are contained in Section 5. Section 6 contains a performance study of the 32 processor Intel hypercube prototype. Our conclusions and future research directions are described in Section 7.

2. Hardware Architecture of Gamma

2.1. Overview

Gamma is based on the concept of a shared-nothing architecture [STON86] in which processors do not share disk drives or random access memory and can only communicate with one another by sending messages through an interconnection network. Mass storage in such an architecture is generally distributed among the processors by connecting one or more disk drives to each processor as shown in Figure 1. There are a number of reasons why the shared-nothing approach has become the architecture of choice. First, there is nothing to prevent the architecture from scaling to 1000s of processors unlike shared-memory machines for which scaling beyond 30-40 processors may be impossible. Second, as demonstrated in [DEWI88, COPE88, TAND88], by associating a small number of

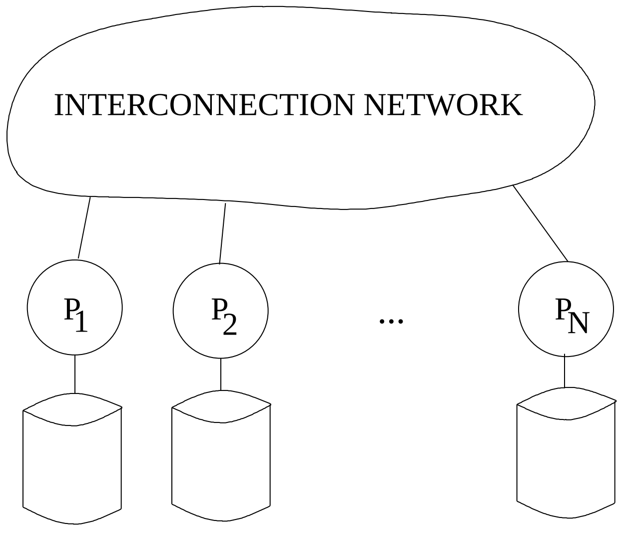


Figure 1

disks with each processor and distributing the tuples of each relation across the disk drives, it is possible to achieve very high aggregate I/O bandwidths without using custom disk controllers [KIM86, PATT88]. Furthermore, by employing off-the-shelf mass storage technology one can employ the latest technology in small 3 1/2" disk drives with embedded disk controllers. Another advantage of the shared nothing approach is that there is no longer any need to "roll your own" hardware. Recently, both Intel and Ncube have added mass storage to their hypercube-based multiprocessor products.

2.2. Gamma Version 1.0

The initial version of Gamma consisted of 17 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring [PROT85] was used to connect the processors to each other and to another VAX running Unix. This processor acted as the host machine for Gamma. Attached to eight of the processors were 333 megabyte Fujitsu disk drives that were used for storing the database. The diskless processors were used along with the processors with disks to execute join and aggregate function operators in order to explore whether diskless processors could be exploited effectively.

We encountered a number of problems with this prototype. First, the token ring had a maximum network packet size of 2K bytes. In the first version of the prototype the size of a disk page was set to 2K bytes in order to be able to transfer an "intact" disk page from one processor to another without a copy. This required, for example, that each disk page also contain space for the protocol header used by the interprocessor communication software. While this initially appeared to be a good idea, we quickly realized that the benefits of a larger disk page size more than offset the cost of having to copy tuples from a disk page into a network packet.

The second problem we encountered was that the network interface and the Unibus on the 11/750 were both bottlenecks [GERB87, DEWI88]. While the bandwidth of the token ring itself was 80 megabits/second, the Unibus on the 11/750 (to which the network interface was attached) has a bandwidth of only 4 megabits/second. When processing a join query without a selection predicate on either of the input relations, the Unibus became a bottleneck because the transfer rate of pages from the disk was higher than the speed of the Unibus [DEWI88]. The network interface was a bottleneck because it could only buffer two incoming packets at a time. Until one packet was transferred into the VAX's memory, other incoming packets were rejected and had to be retransmitted by the communications protocol. While we eventually constructed an interface to the token ring that plugged directly into the backplane of the VAX, by the time the board was operational the VAX's were obsolete and we elected not to spend additional funds to upgrade the entire system.

The other serious problem we encountered with this prototype was having only 2 megabytes of memory on each processor. This was especially a problem since the operating system used by Gamma does not provide virtual memory. The problem was exacerbated by the fact that space for join hash tables, stack space for processes, and the buffer pool were managed separately in order to avoid flushing hot pages from the buffer pool. While there are advantages to having these spaces managed separately by the software, in a configuration where memory is already tight, balancing the sizes of these three pools of memory proved difficult.

2.3. Gamma Version 2.0

In the fall of 1988, we replaced the VAX-based prototype with a 32 processor iPSC/2 hypercube from Intel. Each processor is configured with a 386 CPU, 8 megabytes of memory, and a 330-megabyte MAXTOR 4380 (5 1/4") disk drive. Each disk drive has an embedded SCSI controller which provides a 45 Kbyte RAM buffer that acts as a disk cache on read operations.

The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight[1] full-duplex, serial, reliable communication channels operating at 2.8 megabytes/second. Small messages (<= 100 bytes) are sent as datagrams. For large messages, the hardware builds a communications circuit between the two nodes over which the entire message is transmitted without any software overhead or copying. After the message has been completely transmitted, the circuit is released. The length of a message is limited only by the size of the physical memory on each processor. Table 1 summarizes the transmission times from one Gamma process to another (on two different hypercube nodes) for a variety of message sizes.

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

Table 1

The conversion of the Gamma software to the hypercube began in early December 1988. Because most users of the Intel hypercube tend to run a single process at a time while crunching numerical data, the operating system provided by Intel supports only a limited number of heavy weight processes. Thus, we began the conversion process by porting Gamma's operating system, NOSE (see Section 3.5). In order to simplify the conversion, we elected to run NOSE as a thread package inside a single NX/2 process in order to avoid having to port NOSE to run on the bare hardware directly.

[1] On configurations with a mix of compute and I/O nodes, one of the 8 channels is dedicated for communication to the I/O subsystem.

Once NOSE was running, we began converting the Gamma software. This process took 4-6 man months but lasted about 6 months as, in the process of the conversion, we discovered that the interface between the SCSI disk controller and memory was not able to transfer disk blocks larger than 1024 bytes (the pitfall of being a beta test site). For the most part the conversion of the Gamma software was almost trivial as, by porting NOSE first, the differences between the two systems in initiating disk and message transfers were completely hidden from the Gamma software. In porting the code to the 386, we did discover a number of hidden bugs in the VAX version of the code as the VAX does not trap when a null pointer is dereferenced. The biggest problem we encountered was that nodes on the VAX multicomputer were numbered beginning with 1 while the hypercube uses 0 as the logical address of the first node. While we thought that making the necessary changes would be tedious but straightforward, we were about half way through the port before we realized that we would have to find and change every "for" loop in the system in which the loop index was also used as the address of the machine to which a message was to be set. While this sounds silly now, it took us several weeks to find all the places that had to be changed. In retrospect, we should have made NOSE mask the differences between the two addressing schemes.

From a database system perspective, however, there are a number of areas in which Intel could improve the design of the iPSC/2. First, a light-weight process mechanism should be provided as an alternative to NX/2. While this would have almost certainly increased the time required to do the port, in the long run we could have avoided maintaining NOSE. A much more serious problem with the current version of the system is that the disk controller does not perform DMA transfers directly into memory. Rather, as a block is read from the disk, the disk controller does a DMA transfer into a 4K byte FIFO. When the FIFO is half full, the CPU is interrupted and the contents of the FIFO are copied into the appropriate location in memory.[2] While a block instruction is used for the copy operation, we have measured that about 10% of the available CPU cycles are being wasted doing the copy operation. In addition, the CPU is interrupted 13 times during the transfer of one 8 Kbyte block partially because a SCSI disk controller is used and partially because of the FIFO between the disk controller and memory.

3. Software Architecture of Gamma

In this section, we present an overview of Gamma's software architecture and describe the techniques that Gamma employs for executing queries in a dataflow fashion. We begin by describing the alternative storage structures provided by the Gamma software. Next, the overall system architecture is described from the top down. After describing the overall process structure, we illustrate the operation of the system by describing the interaction of the

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

processes during the execution of several different queries. A detailed presentation of the techniques used to control the execution of complex queries is presented in Section 3.4. This is followed by an example which illustrates the execution of a multioperator query. Finally, we briefly describe WiSS, the storage system used to provide low level database services, and NOSE, the underlying operating system.

3.1. Gamma Storage Organizations

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

The query language of Gamma provides the user with three alternative declustering strategies: *round robin*, *hashed*, and *range partitioned*. With the first strategy, tuples are distributed in a round-robin fashion among the disk drives. This is the default strategy and is used for all relations created as the result of a query. If the hashed partitioning strategy is selected, a randomizing function is applied to the key attribute of each tuple (as specified in the partition command for the relation) to select a storage unit. 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 distribution of tuples shown in Table 2. The partitioning information for each relation is stored in the database catalog. For range and hash-partitioned relations, the name of the partitioning attribute is also kept and, in the case of range-partitioned relations, the range of values of the partitioning attribute for each site (termed a *range table*).

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

An Example Range Table
Table 2

Once a relation has been partitioned, Gamma provides the normal collection of relational database system access methods including both clustered and non-clustered indices. When the user requests that an index be created on a relation, the system automatically creates an index on each fragment of the relation. Unlike VSAM [WAGN73] and the Tandem file system [ENSC85], Gamma does not require the clustered index for a relation to be constructed on

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

the partitioning attribute.

As a query is being optimized, the partitioning information for each source relation in the query is incorporated into the query plan produced by the query optimizer. In the case of hash and range-partitioned relations, this partitioning information is used by the query scheduler (discussed below) to restrict the number of processors involved in the execution of selection queries on the partitioning attribute. For example, if relation X is hash partitioned on attribute y, it is possible to direct selection operations with predicates of the form "X.y = Constant" to a single site; avoiding the participation of any other sites in the execution of the query. In the case of range-partitioned relations, the query scheduler can restrict the execution of the query to only those processors whose ranges overlap the range of the selection predicate (which may be either an equality or range predicate).

In retrospect, we made a serious mistake in choosing to decluster all relations across all nodes with disks. A much better approach, as proposed in [COPE88], is to use the "heat" of a relation to determine the degree to which the relation is declustered. Unfortunately, to add such a capability to the Gamma software at this point in time would require a fairly major effort - one we are not likely to undertake.

3.2. Gamma Process Structure

The overall structure of the various processes that form the Gamma software is shown in Figure 2. The role of each process is described briefly below. The operation of the distributed deadlock detection and recovery mechanism are presented in Sections 5.1 and 5.2. 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.

Catalog Manager

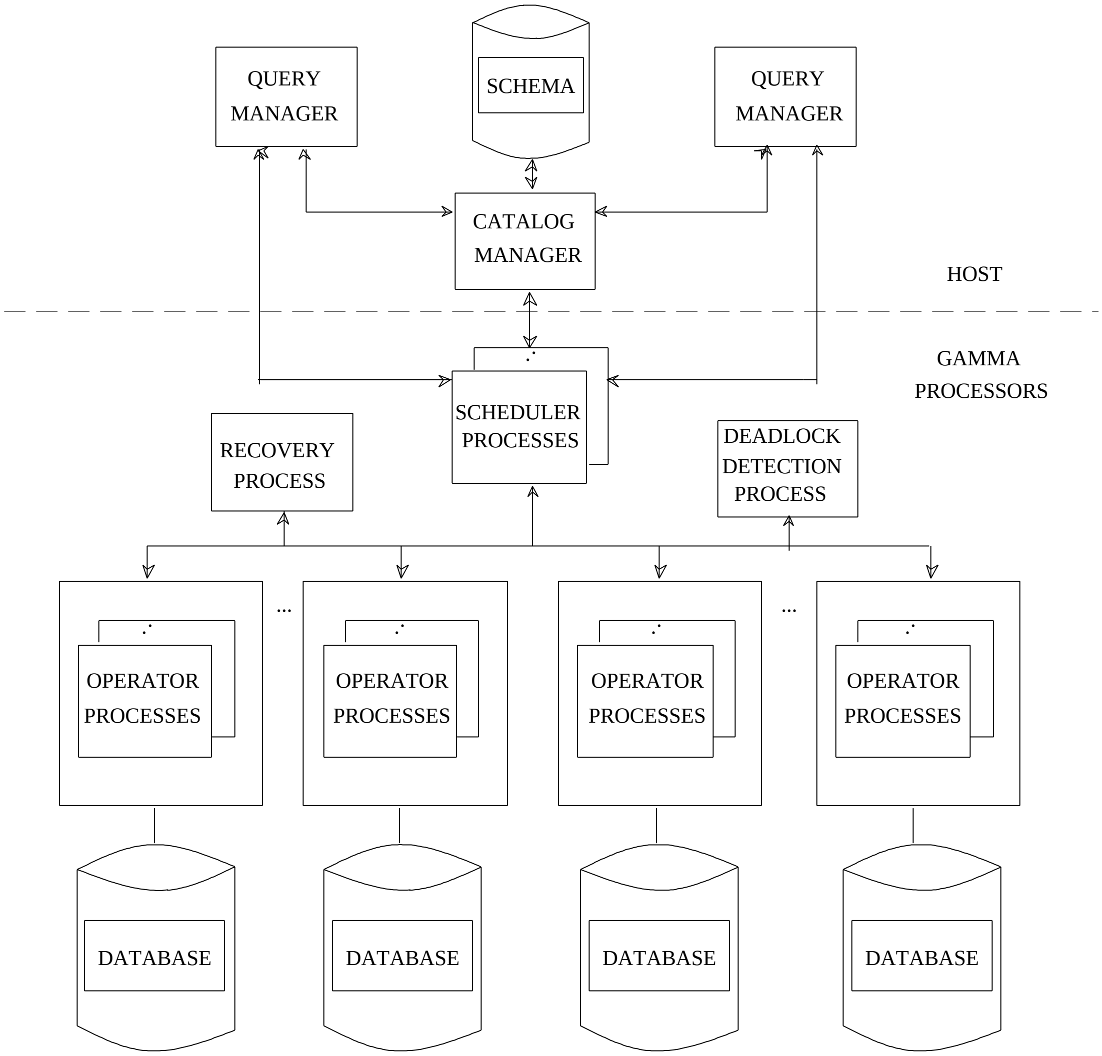
The function of the Catalog Manager is to act as a central repository of all conceptual and internal schema information for each database. The schema information is loaded into memory when a database is first opened. Since multiple users may have the same database open at once and since each user may 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

One query manager process is 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 (our variant of Quel [STON76]), query parsing, optimization, and compilation.

Scheduler Processes

While executing, each 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. Scheduler processes can be run on any processor, insuring that no processor becomes a bottleneck. In practice, however, scheduler processes consume almost no resources and it is possible to run a large number of them on a single processor. A centralized dispatching process is used to assign scheduler processes to queries. Those queries that the optimizer can detect to be single-site queries are sent directly to the appropriate node for execution, by-passing the scheduling process.



Gamma Process Structure
Figure 2

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. These operators are primed at system initialization time in order to avoid the overhead of starting processes at query execution time (additional processes can be forked as needed). The structure of an operator process and the mapping of relational operators to operator processes is discussed in more detail below. When a scheduler wishes to start a new operator on a node, it sends a request to a special communications port known as the "new task" port. When a request is received on this port, an idle operator process is assigned to the request and the communications port of this operator process is returned to the requesting scheduler process.

3.3. An Overview of Query Execution

Ad-hoc and Embedded Query Interfaces

Two interfaces to Gamma are available: an ad-hoc query language and an embedded query language interface in which queries can be embedded in a C program. When a user invokes the ad-hoc query interface, a Query Manager (QM) process is started which immediately connects itself to the CM process through the UNIX Internet socket mechanism. When the compiled query interface is used, the preprocessor translates each embedded query into a compiled query plan which is invoked at run-time by the program. A mechanism for passing parameters from the C program to the compiled query plans at run time is also provided.

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. Queries are compiled into a left-deep tree of operators. At execution time, each operator is executed by one or more operator processes at each participating site.

In designing the optimizer for the VAX version of Gamma, the set of possible query plans considered by the optimizer was restricted to only left-deep trees because we felt that there was not enough memory to support right-deep or bushy plans. By using a combination of left-deep query trees and hash-based join algorithms, we were able to insure that no more than two join operations were ever active simultaneously and hence were able to maximize the amount of physical memory which could be allocated to each join operator. Since this memory limitation was really only an artifact of the VAX prototype, we have recently begun to examine the performance implications of right deep and bushy query plans [SCHN89b].

As discussed in Section 3.1, in the process of optimizing a query, the query optimizer recognizes that certain queries can be directed to only a subset of the nodes in the system. 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 centralized dispatcher process. The dispatcher process, by controlling the number of active schedulers, implements a simple load control mechanism. 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.

3.4. Operator and Process Structure

The algorithms for all the relational operators are written as if they were to be run on a single processor. As shown in Figure 3, the input to an Operator Process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a **split table**. Once the process begins execution, it continuously reads tuples from its input stream, operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table.[4] 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 process 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.

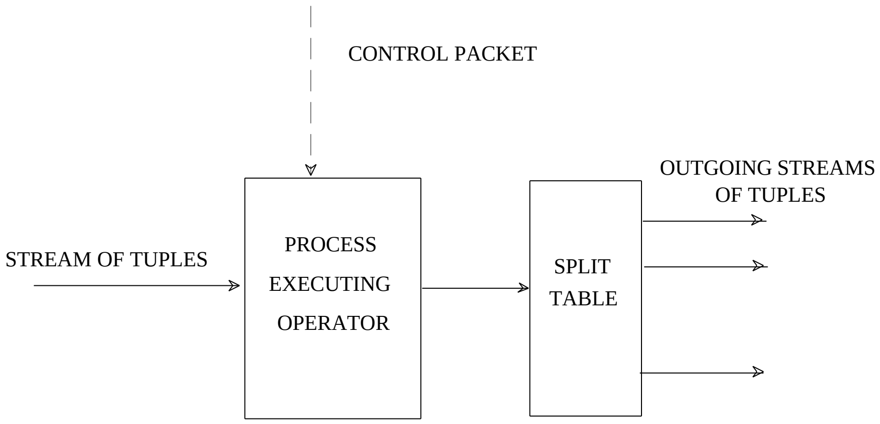


Figure 3

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 [DEWI86]. As an example of one form of split table, consider the use of the split table shown in Figure 4 in conjunction with the execution of a join operation using 4 processors. Each process producing 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.

[4] Tuples are actually sent as 8K byte batches, except for the last batch.

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 4

An Example

As an example of how queries are executed, consider the query shown in Figure 5. In Figure 6, the processes used to execute the query are shown along with the flow of data between the various processes for a Gamma configuration consisting of two processors with disks and two processors without disks. Since the two input relations A and B are partitioned across the disks attached to processors P1 and P2, selection and scan operators are initiated on both processors P1 and P2. The split tables for both the select and scan operators each contain two entries since two processors are being used for the join operation. The split tables for each selection and scan are identical - routing tuples whose join attribute values hash to 0 (dashed lines) to P3 and those which hash to 1 (solid lines) to P4. The join operator executes in two phases. During the first phase, termed the *Building* phase, tuples from the inner relation (A in this example) are inserted into a memory-resident hash table by hashing on the join attribute value. After the first phase has completed, the *probing* phase of the join is initiated in which tuples from the outer relation are used to probe the hash table for matching tuples.[5] Since the result relation is partitioned across two disks, the split table for each join operator contains two entries and tuples of C are distributed in a round-robin fashion among P1 and P2.

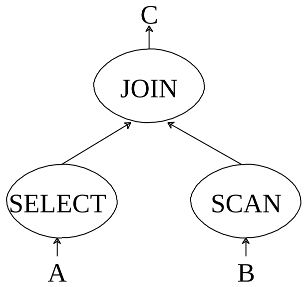


Figure 5

[5] This is actually a description of the simple hash join algorithm. The operation of the hybrid hash join algorithm is contained in Section 4.

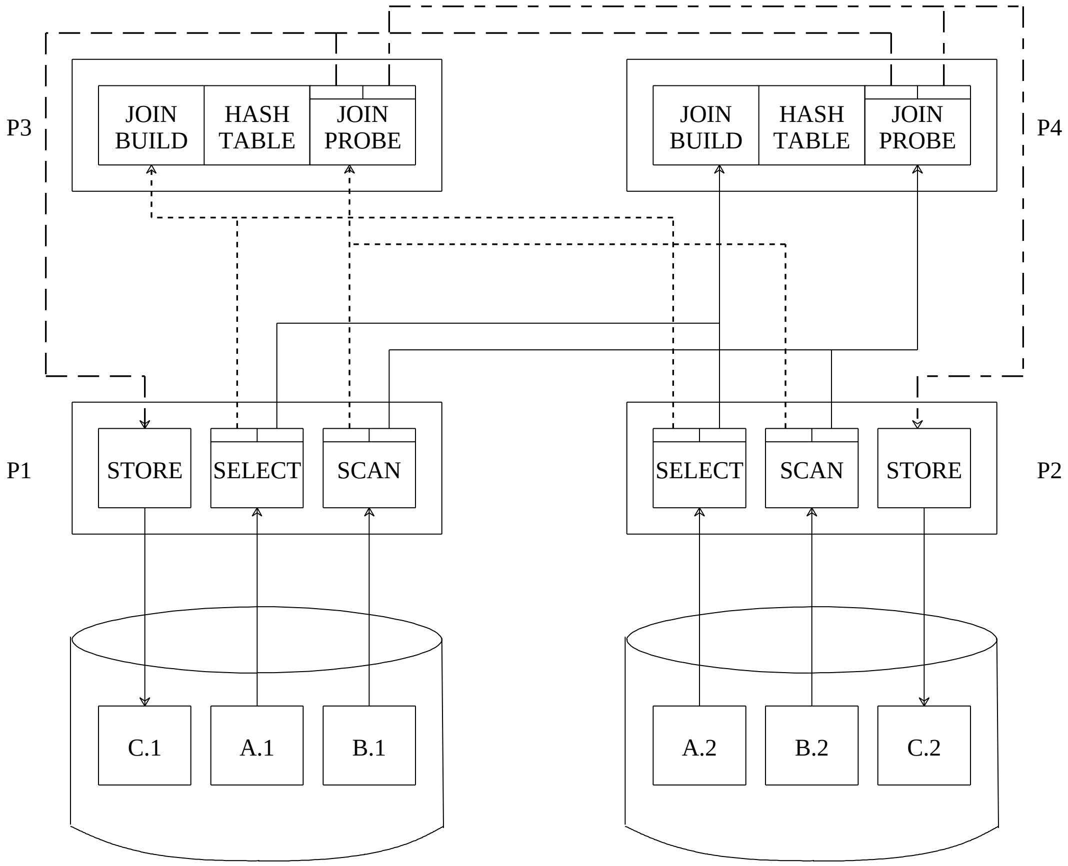


Figure 6

One of the main problems with the DIRECT prototype was that every data page processed required at least one control message to a centralized scheduler. In Gamma this bottleneck is completely avoided. In fact, the number of control messages required to execute a query is approximately equal to three times the number of operators in the query times the number of processors used to execute each operator. As an example, consider Figure 7 which depicts the flow of control messages[6] from a scheduler process to the processes on processors P1 and P3 in Figure 6 (an identical set of messages would flow from the scheduler to P2 and P4). The scheduler begins by initiating the building phase of the join and the selection operator on relation A. When both these operators have completed, the scheduler next initiates the store operator, the probing phase of the join, and the scan of relation B. When each of these operators has completed, a result message is returned to the user.

---

[6] The "Initiate" message is sent to a "new operator" port on each processor. A dispatching processes accepts incoming messages on this port and assigns the operator to a process. The process which is assigned, replies to the scheduler with an "ID" message which indicates the private port number of the operator process. Future communications to the operator by the scheduler use this private port number.

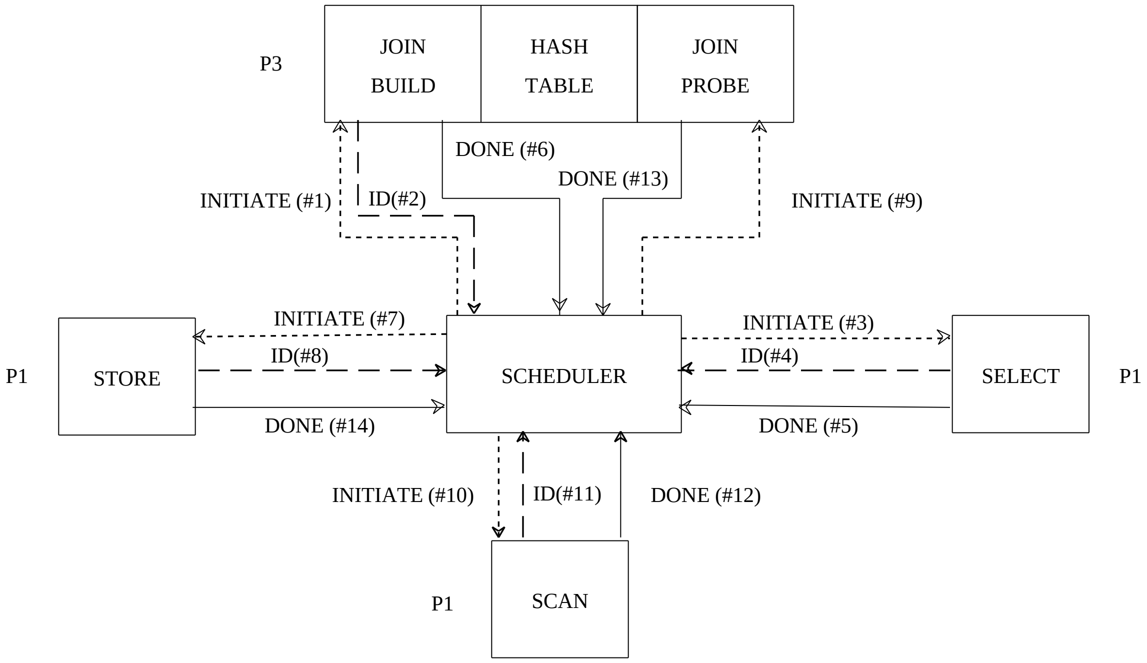


Figure 7

3.5. Operating and Storage System

Gamma is built on top of an operating system designed 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 communications between NOSE processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. 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 file may have one or more associated indices which map key values to the record identifiers of the records in the file that contain a matching value. One indexed attribute may be designated as a clustering attribute for the file. The scan mechanism is similar to that provided by System R's RSS [ASTR76] except that the predicates are compiled by the query optimizer into 386 machine language to maximize performance.

4. Query Processing Algorithms

4.1. Selection Operator

Since all relations are declustered over multiple disk drives, parallelizing the selection operation involves simply initiating a selection operator on the set of relevant nodes with disks. When the predicate in the selection clause is on the partitioning attribute of the relation and the relation is hash or range partitioned, the scheduler can direct the selection operator to a subset of the nodes. If either the relation is round-robin partitioned or the selection predicate is not on the partitioning attribute, a selection operator must be initiated on all nodes over which the relation is declustered. To enhance performance, Gamma employs a one page read-ahead mechanism when scanning the pages of a file sequentially or through a clustered index. This mechanism enables the processing of one page to be overlapped with the I/O for the subsequent page.

4.2. Join Operator

The multiprocessor join algorithms provided by Gamma are based on concept of partitioning the two relations to be joined into disjoint subsets called **buckets** [GOOD81, KITS83, BRAT84]. by applying a hash function to the join attribute of each tuple. The partitioned buckets represent disjoint subsets of the original relations and have the important characteristic that all tuples with the same join attribute value are in the same bucket. We have implemented parallel versions of four join algorithms on the Gamma prototype: sort-merge, Grace [KITS83], Simple [DEWI84], and Hybrid [DEWI84]. While all four algorithms employ this concept of hash-based partitioning, the actual join computation depends on the algorithm. The parallel hybrid join algorithm is described in the following section. Additional information on all four parallel algorithms and their relative performance can be found in [SCHN89a]. Since this study found that the Hybrid hash join almost always provides the best performance, it is now the default algorithm in Gamma and is described in more detail in the following section. Since these hash-based join algorithms cannot be used to execute non-equijoin operations, such operations are not currently supported. To remedy this situation, we are in the process of designing a parallel non-equijoin algorithm for Gamma.

Hybrid Hash-Join

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

Our parallel version of the Hybrid hash join algorithm is similar to the centralized algorithm described above. A **partitioning split table** first separates the joining relations into N logical buckets. The number of buckets is chosen such that the tuples corresponding to each logical bucket will fit in the **aggregate** memory of the joining processors. The N-1 buckets intended for temporary storage on disk are each partitioned across all available disk sites. Likewise, a **joining split table** will be used to route tuples to their respective joining processor (these processors do not necessarily have attached disks), thus parallelizing the joining phase. Furthermore, the partitioning of the inner relation, R, into buckets is overlapped with the insertion of tuples from the first bucket of R into memory-resident hash tables at each of the join nodes. In addition, the partitioning of the outer relation, S, into buckets is overlapped with the joining of the first bucket of S with the first bucket of R. This requires that the partitioning split table for R and S be enhanced with the joining split table as tuples in the first bucket must be sent to those processors being used to effect the join. Of course, when the remaining N-1 buckets are joined, only the joining split table will be needed. Figure 8 depicts relation R being partitioned into N buckets across k disk sites where the first bucket is to be joined on m processors (m may be less than, equal to, or greater than k).

4.3. Aggregate Operations

Gamma implements scalar aggregates by having each processor compute its piece of the result in parallel. The partial results are then sent to a single process which combines these partial results into the final answer. Aggregate functions are computed in two steps. First, each processor computes a piece of the result by calculating a value for each of the partitions. Next, the processors redistribute the partial results by hashing on the "group by" attribute. The result of this step is to collect the partial results for each partition at a single site so that the final result for each partition can be computed.

4.4. Update Operators

For the most part, the update operators (replace, delete, and append) are implemented using standard techniques. The only exception occurs when a replace operator modifies the partitioning attribute of a tuple. 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 which site should contain the tuple.

5. Transaction and Failure Management

In this section we describe the mechanisms that Gamma uses for transaction and failure management. While the locking mechanisms are fully operational, the recovery system is currently being implemented. We expect to begin the implementation of the failure management mechanism in early 1990.

5.1. Concurrency Control in Gamma

Concurrency control in Gamma is based on two-phase locking [GRAY78]. Currently, two lock granularities, file, and page, and five lock modes, S, X, IS, IX, and SIX are provided. Each site in Gamma has its own local lock manager and deadlock detector. The lock manager maintains a lock table and a transaction wait-for-graph. The cost of setting a lock varies from approximately 100 instructions, if there is no conflict, to 250 instructions if the lock request conflicts with the granted group. In this case, the wait-for-graph must be checked for deadlock and the transaction that requested the lock must be suspended via a semaphore mechanism.

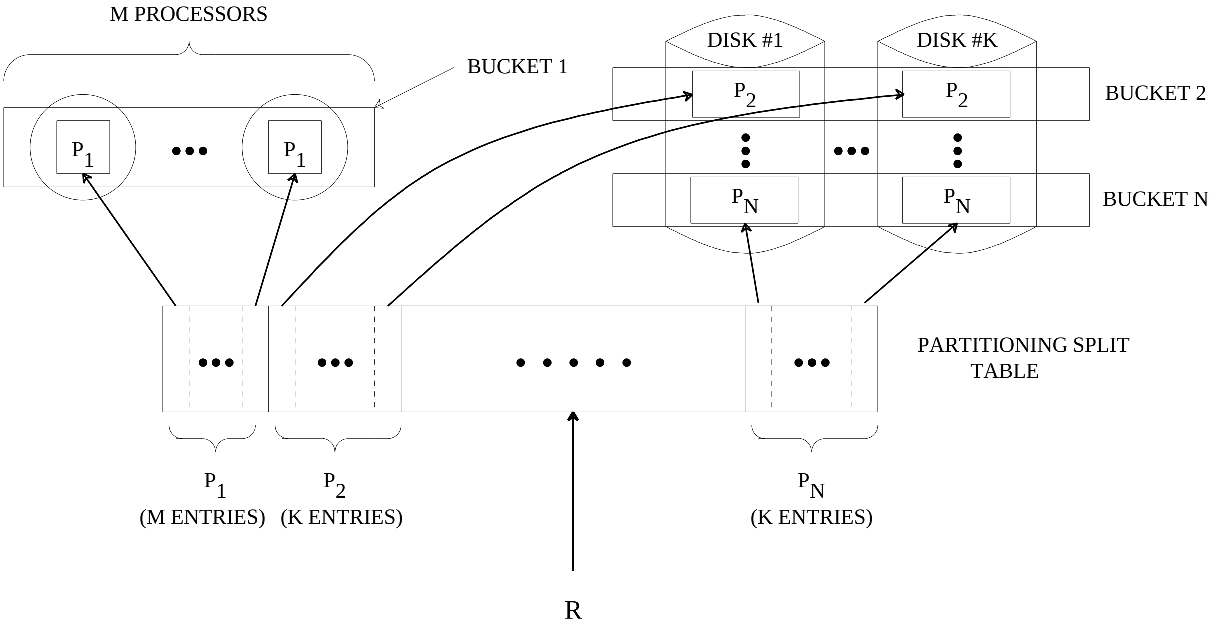
In order to detect multisite deadlocks, Gamma uses a centralized deadlock detection algorithm. Periodically, the centralized deadlock detector sends a message to each node in the configuration, requesting the local transaction wait-for-graph of that node. Initially, the period for running the centralized deadlock detector is set at one second. Each time the deadlock detector fails to find a global deadlock, this interval is doubled and each time a deadlock is found the current value of the interval is halved. The upper bound of the interval is limited to 60 seconds and the lower bound is 1 second. After collecting the wait-for-graph from each site, the centralized deadlock detector creates a global transaction wait-for-graph. Whenever a cycle is detected in the global wait-for-graph, the centralized deadlock manager chooses to abort the transaction holding the fewest number of locks.

5.2. Recovery Architecture and Log Manager

The algorithms currently being implemented for coordinating transaction commit, abort, and rollback operate as follows. When an operator process updates a record, it also generates a log record which records the change of the database state. Associated with every log record is a log sequence number (LSN) which is composed of a node number and a local sequence number. The node number is statically determined at the system configuration time whereas the local sequence number, termed **current LSN**, is a monotonically increasing value.

Log records are sent by the query processors to one or more Log Managers (each running on a separate processor) which merges the log records it receives to form a single log stream. If M is the number of log processors being used, query processor i will direct its log records to the (i mod M) log processor [AGRA85]. Because this algorithm selects the log processor statically and a query processor always sends its log records to the same log processor, the recovery process at a query processing node can easily determine where to request the log records for processing a transaction abort.

When a page of log records is filled, it is written to disk. The Log Manager maintains a table, called the **Flushed Log Table**, which contains, for each node, the LSN of the last log record from that node that has been flushed to disk. These values are returned to the nodes either upon request or when they can be piggybacked on



Partitioning of R into N logical buckets for Hybrid hash-join.
Figure 8

another message. Query processing nodes save this information in a local variable, termed the **Flushed LSN**.

The buffer managers at the query processing nodes observe the WAL protocol [GRAY78]. When a dirty page needs to be forced to disk, the buffer manager first compares the page's LSN with the local value of Flushed LSN. If the page LSN of a page is smaller or equal to the Flushed LSN, that page can be safely written to disk. Otherwise, either a different dirty page must be selected, or a message must be sent to the Log Manager to flush the corresponding log record(s) of the dirty page. Only after the Log Manager acknowledges that the log record has been written to the log disk will the dirty data page be written back to disk. In order to reduce the time spent waiting for a reply from the Log Manager, the buffer manager always keeps T (a pre-selected threshold) clean and unfixed buffer pages available. When buffer manager notices that the number of clean, unfixed buffer pages has fallen below T, a process, termed **local log manager**, is activated. This process sends a message to the Log Manager to flush one or more log records so that the number of clean and unfixed pages plus the number of dirty pages that can be safely written to to disk is greater than T.

The scheduler process for a query is responsible for sending commit or abort records to the appropriate Log Managers. If a transaction completes successfully, a commit record for the transaction is generated by its scheduler and sent to each relevant Log Manager which employs a group commit protocol. On the other hand, if a transaction is aborted by either the system or the user, its scheduler will send an abort message to all query processors that participated in its execution. The recovery process at each of the participating nodes responds by requesting the log records generated by the node from its Log Manager (the LSN of each log record contains the originating node number). As the log records are received, the recovery process undoes the log records in reverse chronological order using the ARIES undo algorithm [MOHA89]. The ARIES algorithms are also used as the basis for checkpointing and restart recovery.

5.3. Failure Management

To help insure availability of the system in the event of processor and/or disk failures, Gamma employs a new availability technique termed **chained declustering** [HSIA90]. Like Tandem's mirrored disk mechanism [BORR81] and Teradata's interleaved declustering mechanism [TERA85, COPE89], chained declustering employs both a primary and backup copy of each relation. All three systems can sustain the failure of a single processor or disk without suffering any loss in data availability. In [HSIA90], we show that chained declustering provides a higher degree of availability than interleaved declustering and, in the event of a processor or disk failure, does a better job of distributing the workload of the broken node. The mirrored disk mechanism, while providing the highest level of availability, does a very poor job of distributing the load of a failed processor.

Data Placement with Chained Declustering

With chained declustering, nodes (a processor with one or more disks) are divided into disjoint groups called **relation-clusters** and tuples of each relation are declustered among the drives that form one of the relation clusters. Two physical copies of each relation, termed the **primary copy** and the **backup copy**, are maintained. As an example, consider Figure 9 where M, the number of disks in the relation cluster, is equal to 8. The tuples in the primary copy of relation R are declustered using one of Gamma's three partitioning strategies with tuples in the i-th primary fragment (designated **Ri**) stored on the {i mod M}-th disk drive. The backup copy is declustered using the same partitioning strategy but the i-th backup fragment (designated **ri**) is stored on {(i + 1) mod M}-th disk. We term this data replication method **chained declustering** because the disks are linked together, by the fragments of a relation, like a chain.

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r7	r0	r1	r2	r3	r4	r5	r6

Chained Declustering (Relation Cluster Size = 8)
Figure 9

The difference between the chained and interleaved declustering mechanisms [TERA85, COPE89] is illustrated by Figure 10. In Figure 10, the fragments from the primary copy of R are declustered across all 8 disk drives by hashing on a "key" attribute. With the interleaved declustering mechanism the set of disks are divided into units of size N called **clusters**. As illustrated by Figure 10, where N=4, each backup fragment is subdivided into N-1 subfragments and each subfragment is placed on a different disk within the same cluster other than the disk containing the primary fragment.

	cluster 0				cluster 1			
Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy		r0.0	r0.1	r0.2		r4.0	r4.1	r4.2
	r1.2		r1.0	r1.1	r5.2		r5.0	r5.1
	r2.1	r2.2		r2.0	r6.1	r6.2		r6.0
	r3.0	r3.1	r3.2		r7.0	r7.1	r7.2	

Interleaved Declustering (Cluster Size = 4)
Figure 10

Since interleaved and chained declustering can both sustain the failure of a single disk or processor, what then is the difference between the two mechanisms? In the case of a single node (processor or disk) failure both the chained and interleaved declustering strategies are able to uniformly distribute the workload of the cluster among the remaining operational nodes. For example, with a cluster size of 8, when a processor or disk fails, the load on each remaining node will increase by 1/7th. One might conclude then that the cluster size should be made as large as possible; until, of course, the overhead of the parallelism starts to overshadow the benefits obtained. While this is true for chained declustering, the availability of the interleaved strategy is inversely proportional to the cluster size. since the failure of any two processors or disk will render data unavailable. Thus, doubling the cluster size in order to halve (approximately) the increase in the load on the remaining nodes when a failure occurs has the (quite negative) side effect of doubling the probability that data will actually be unavailable. For this reason, Teradata recommends a cluster size of 4 or 8 processors.

Figure 11 illustrates how the workload is balanced in the event of a node failure (node 1 in this example) with the chained declustering mechanism. During the normal mode of operation, read requests are directed to the fragments of the primary copy and write operations update both copies. When a failure occurs, pieces of both the primary and backup fragments are used for read operations. For example, with the failure of node 1, primary fragment R1 can no longer be accessed and thus its backup fragment r1 on node 2 must be used for processing queries that would normally have been directed to R1. However, instead of requiring node 2 to process all accesses to both R2 and r1, chained declustering offloads 6/7-ths of the accesses to R2 by redirecting them to r2 at node 3. In turn, 5/7-ths of access to R3 at node 3 are sent to R4 instead. This dynamic reassignment of the workload results in an increase of 1/7-th in the workload of each remaining node in the cluster. Since the relation cluster size can be increased without penalty, it is possible to make this load increase as small as is desired.

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	---	$\frac{1}{7}$R2	$\frac{2}{7}$R3	$\frac{3}{7}$R4	$\frac{4}{7}$R5	$\frac{5}{7}$R6	$\frac{6}{7}$R7
Backup Copy	$\frac{1}{7}$r7	---	r1	$\frac{6}{7}$r2	$\frac{5}{7}$r3	$\frac{4}{7}$r4	$\frac{3}{7}$r5	$\frac{2}{7}$r6

Fragment Utilization with Chained Declustering
After the Failure of Node 1 (Relation Cluster Size = 8)
Figure 11

What makes this scheme even more attractive is that the reassignment of active fragments incurs neither disk I/O nor data movement. Only some of the bound values and pointers/indices in a memory resident control table must be changed and these modifications can be done very quickly and efficiently.

The example shown in Figure 11 provides a very simplified view of how the chained declustering mechanism actually balances the workload in the event of a node failure. In reality, queries cannot simply access an arbitrary fraction of a data fragment, especially given the variety of partitioning and index mechanisms provided by the Gamma software. In [HSIA90], we describe how all combinations of query types, access methods, and partitioning mechanisms can be handled.

6. Performance Studies

6.1. Introduction and Experiment Overview

To evaluate the performance of the hypercube version of Gamma three different metrics were used. First, the set of Wisconsin [BITT83] benchmark queries were run on a 30 processor configuration using three different sizes of relations: 100,000, 1 million, and 10 million tuples. While absolute performance is one measure of a database system, speedup and scaleup are also useful metrics for multiprocessor database machines [ENGL89]. Speedup is an interesting metric because it indicates whether additional processors and disks results in a corresponding decrease in the response time for a query. For a subset of the Wisconsin benchmark queries, we conducted speedup experiments by varying the number of processors from 1 to 30 while the size of the test relations was fixed at 1 million tuples. For the same set of queries, we also conducted scaleup experiments by varying the number of processors from 5 to 30 while the size of the test relations was increased from 1 to 6 million tuples, respectively. Scaleup is a valuable metric as it indicates whether a constant response time can be maintained as the workload is increased by adding a proportional number of processors and disks. [ENGL89] describes a similar set of tests on Release 2 of Tandem's NonStop SQL system.

The benchmark relations used for the experiments were based on the standard Wisconsin Benchmark relations [BITT83]. Each relation consists of tuples that are 208 bytes wide. We constructed 100,000, 1 million, and 10 million tuple versions of the benchmark relations. Two copies of each relation were created and loaded. Except where noted otherwise, tuples were declustered by hash partitioning on the Unique1 attribute. In all cases, the results presented represent the average response time of a number of equivalent queries. Gamma was configured to use a disk page size of 8K bytes and a buffer pool of 2 megabytes.

The results of all queries were stored in the database. We avoided returning data to the host in order to avoid having the speed of the communications link between the host and the database machine or the host processor itself affect the results. By storing the result relations in the database, the impact of these factors was minimized - at the expense of incurring the cost of declustering and storing the result relations.

6.2. Selection Queries

Performance Relative to Relation Size

The first set of selection tests were designed to determine how Gamma would respond as the size of the source relations was increased while the machine configuration was kept at 30 processors with disks. Ideally, the response time of a query should grow as a linear function of the size of input and result relations. For these tests six different selection queries were run on three sets of relations containing, respectively, 100,000, 1 million, and 10 million tuples. The first two queries have a selectivity factor of 1% and 10% and do not employ any indices. The third and fourth queries have the same selectivity factors but use a clustered index to locate the qualifying tuples. The fifth query has a selectivity factor of 1% and employs a non-clustered index to locate the desired tuples. There is no 10% selection through a non-clustered index query as the Gamma query optimizer chooses to use a sequential scan for this query. The last query uses a clustered index to retrieve a single tuple. Except for the last query, the predicate of each query specifies a range of values and, thus, since the input relations were declustered by hashing, the query must be sent to all the nodes.

The results from these tests are tabulated in Table 3. For the most part, the execution time for each query scales as a fairly linear function of the size of the input and output relations. There are, however, several cases where the scaling is not perfectly linear. Consider, first the 1% non-indexed selection. While the increase in response time as the size of the input relation is increased from 1 to 10 million tuples is almost perfectly linear (8.16 secs. to 81.15 secs.), the increase from 100,000 tuples to 1 million tuples (0.45 sec. to 8.16 sec) is actually sub-linear. The 10% selection using a clustered index is another example where increasing the size of the input relation by a factor of ten results in more than a ten-fold increase in the response time for the query. This query takes 5.02 seconds on the 1 million tuple relation and 61.86 seconds on the 10 million tuple relation. To understand why this happens one must consider the impact of seek time on the execution time of the query. Since two copies of each relation were loaded, when two one million tuple relations are declustered over 30 disk drives, the fragments occupy approximately 53 cylinders (out of 1224) on each disk drive. Two ten million tuple relations fill about 530 cylinders on each drive. As each page of the result relation is written to disk, the disk heads must be moved from their current position over the input relation to a free block on the disk. Thus, with the 10 million tuple relation, the cost of writing each output page is much higher.

As expected, the use of a clustered B-tree index always provides a significant improvement in performance. One observation to be made from Table 3 is the relative consistency of the execution time of the selection queries through a clustered index. Notice that the execution time for a 10% selection on the 1 million tuple relation is almost identical to the execution time of the 1% selection on the 10 million tuple relation. In both cases, 100,000 tuples are retrieved and stored, resulting in identical I/O and CPU costs.

The final row of Table 3 presents the time required to select a single tuple using a clustered index and return it to the host. Since the selection predicate is on the partitioning attribute, the query is directed to a single node, avoiding the overhead of starting the query on all 30 processors. The response for this query increases significantly as the

Table 3 - Selection Queries
30 Processors With Disks
(All Execution Times in Seconds)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

relation size is increased from 1 million to 10 million tuples because the height of the B-tree increases from two to three levels.

Speedup Experiments

In this section we examine how the response time for both the nonindexed and indexed selection queries on the 1 million tuple relation[7] is affected by the number of processors used to execute the query. Ideally, one would like to see a linear improvement in performance as the number of processors is increased from 1 to 30. Increasing the number of processors increases both the aggregate CPU power and I/O bandwidth available, while reducing the number of tuples that must be processed by each processor.

In Figure 12, the average response times for the non-indexed 1% and 10% selection queries on the one million tuple relation are presented. As expected, the response time for each query decreases as the number of nodes is increased. The response time is higher for the 10% selection due to the cost of declustering and storing the result relation. While one could always store result tuples locally, by partitioning all result relations in a round-robin (or hashed) fashion one can ensure that the fragments of every result relation each contain approximately the same number of tuples. The speedup curves corresponding to Figure 12 are presented in Figure 13. In Figure 14, the average response time is presented as a function of the number of processors for the following three queries: a 1% selection through a clustered index, a 10% selection through a clustered index, and a 1% selection through a non-

[7] The 1 million tuple relation was used for these experiments because the 10 million tuple relation would not fit on 1 disk drive.

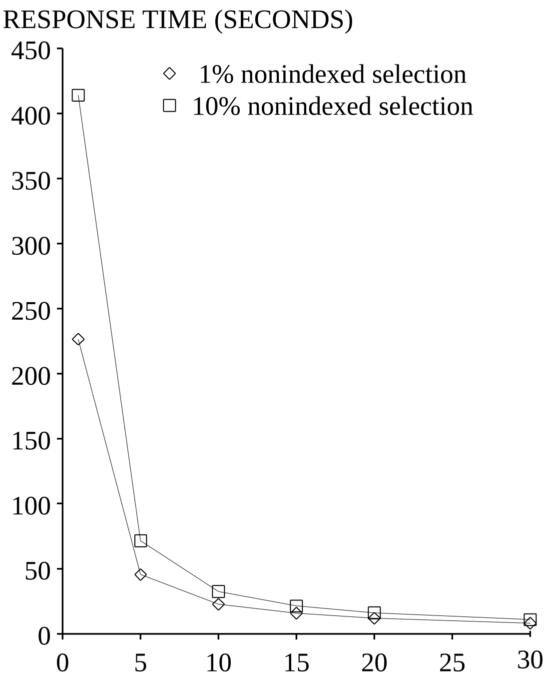


Figure 12

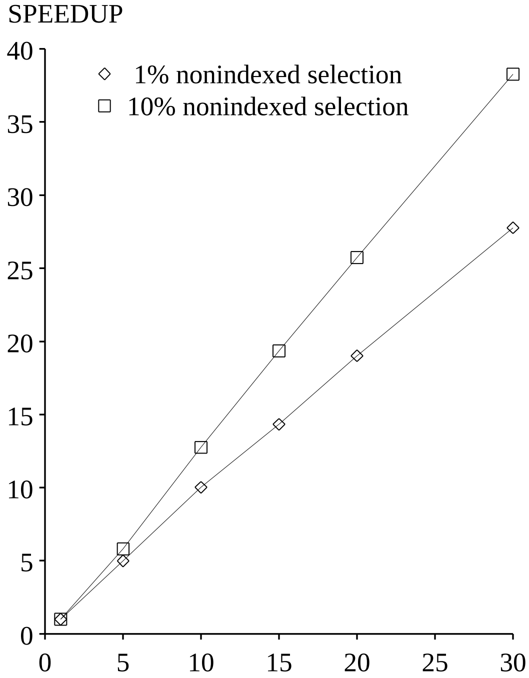


Figure 13

clustered index, all accessing the 1 million tuple relation. The corresponding speedup curves are presented in Figure 15.

Of the speedup curves presented in Figures 13 and 14, three queries are superlinear, one is slightly sublinear, and one is significantly sublinear. Consider first the 10% selection via a relation scan, the 1% selection through a non-clustered index, and the 10% selection through a clustered index. As discussed above, the source of the superlinear speedups exhibited by these queries is due to significant differences in the time the various configurations spend seeking. With one processor, the 1 million tuple relation occupies approximately 66% of the disk. When the same relation is declustered over 30 disk drives, it occupies about 2% of each disk. In the case of the 1% nonclustered index selection, each tuple selected requires a random seek. With one processor, the range of the each random seek is approximately 800 cylinders while with 30 processors the range of the seek is limited to about 27 cylinders. Since the seek time is proportional to the square root of the distance traveled by the disk head [GRAY88], reducing the size of the relation fragment on each disk significantly reduces the amount of time that the query spends seeking.

A similar effect also happens with the 10% clustered index selection. In this case, once the index has been used to locate the tuples satisfying the query, each input page will produce one output page and at some point the buffer pool will be filled with dirty output pages. In order to write an output page, the disk head must be moved

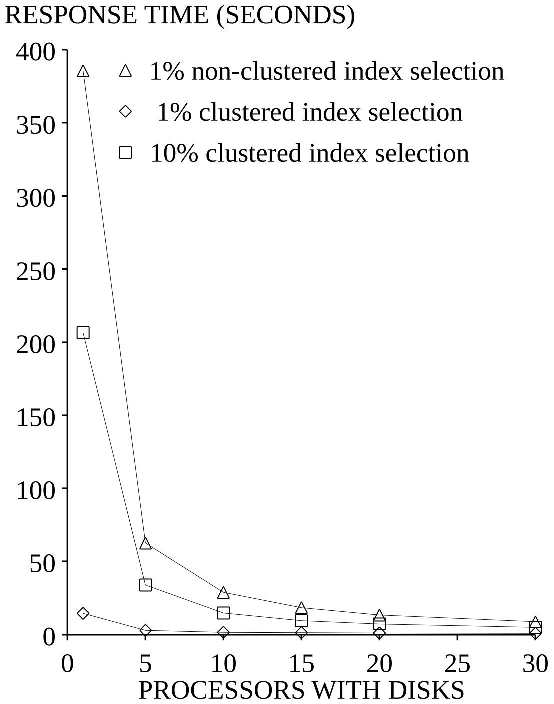


Figure 14

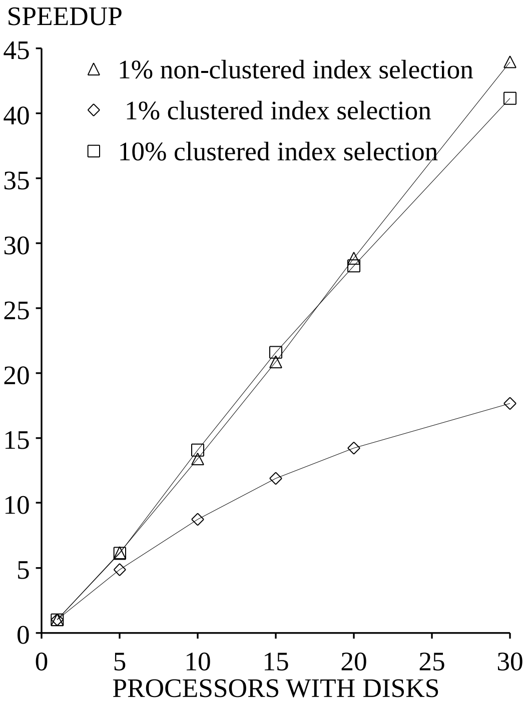


Figure 15

from its position over the input relation to the position on the disk where the output pages are to be placed. The relative cost of this seek decreases proportionally as the number of processors increases, resulting in a superlinear speedup for the query. The 10% non-indexed selection shown in Figure 13 is also superlinear for similar reasons. The reason that this query is not affected to the same degree is that, without an index, the seek time is a smaller fraction of the overall execution time of the query.

The 1% selection through a clustered index exhibits sublinear speedups because the cost of initiating a select and store operator on each processor (a total of 0.24 seconds for 30 processors) becomes a significant fraction of the total execution as the number of processors is increased.

Scaleup Experiments

In the final set of selection experiments the number of processors was varied from 5 to 30 while the size of the input relations was increased from 1 million to 6 million tuples, respectively. As shown in Figure 16, the response time for each of the five selection queries remains almost constant. The slight increase in response time is due to the overhead of initiating a selection and store operator at each site. Since a single process is used to initiate the execution of a query, as the number of processors employed is increased, the load on this process is increased proportionally. Switching to a tree-based, query initiation scheme [GERB87] would distribute this overhead among all the processors.

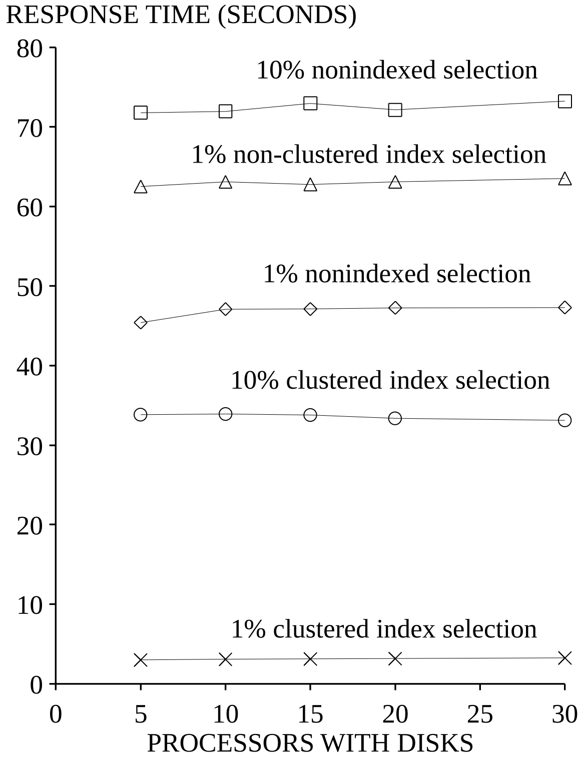


Figure 16

6.3. Join Queries

Like the selection queries in the previous section, we conducted three sets of join experiments. First, for two different join queries, we varied the size of the input relations while the configuration of processors was kept constant. Next, for one join query a series of speedup and scaleup experiments were conducted. For each of these tests, two different partitionings of the input relations were used. In the first case, the input relations were declustered by hashing on the join attribute. In the second case, the input relations were declustered using a different attribute. The hybrid join algorithm was used for all queries.

Performance Relative to Relation Size

The first join query [BITT83], joinABprime, is a simple join of two relations: A and Bprime. The A relation contains either 100,000, 1 million, or 10 million tuples. The Bprime relation contains, respectively, 10,000, 100,000, or 1 million tuples. The result relation has the same number of tuples as the Bprime relation.[8] The second query, joinAselB, is composed of one join and one selection. A and B have the same number of tuples and the

[8] For each join operation, the result relation contains all the fields of both input relations and thus the result tuples are 416 bytes wide.

selection on B reduces the size of B to the size of the Bprime relation in the corresponding joinABprime query. The result relation for this query has the same number of tuples as in the corresponding joinABprime query. As an example, if A has 10 million tuples, then joinABprime joins A with a Bprime relation that contains 1 million tuples, while in joinAselB the selection on B restricts B from 10 million tuples to 1 million tuples and then joins the result with A.

The first variation of the join queries tested involved no indices and used a non-partitioning attribute for both the join and selection attributes. Thus, before the join can be performed, the two input relations must be redistributed by hashing on the join attribute value of each tuple. The results from these tests are contained in the first 2 rows of Table 4. The second variation of the join queries also did not employ any indices but, in this case, the relations were hash partitioned on the joining attribute; enabling the redistribution phase of the join to be skipped. The results for these tests are contained in last 2 rows of Table 4.

The results in Table 4 indicate that the execution time of each join query increases in a fairly linear fashion as the size of the input relations are increased. Gamma does not exhibit linearity for the 10 million tuple queries because the size of the inner relation (208 megabytes) is twice as large as the total available space for hash tables. Hence, the Hybrid join algorithm needs two buckets to process these queries. While the tuples in the first bucket can be placed directly into memory-resident hash tables, the second bucket must be written to disk (see Section 4.2).

As expected, the version of each query in which the partitioning attribute was used as the join attribute ran faster. From these results one can estimate a lower bound on the aggregate rate at which data can be redistributed by the Intel iPSC/2 hypercube. Consider the version of the joinABprime query in which a million tuple relation is

Table 4 - Join Queries
30 Processors With Disks
(All Execution Times in Seconds)

Query Description	**Number of Tuples in Relation A** 100,000	1,000,000	10,000,000
JoinABprime with non-partitioning attributes of A and B used as join attributes	3.52	28.69	438.90
JoinAselB with non-partitioning attributes of A and B used as join attributes	2.69	25.13	373.98
JoinABprime with partitioning attributes of A and B used as join attributes	3.34	25.95	426.25
JoinAselB with partitioning attributes of A and B used as join attributes	2.74	23.77	362.89

joined with a 100,000 tuple relation. This query requires 28.69 seconds when the join is not on the partitioning attribute. During the execution of this query, 1.1 million 208 byte tuples must be redistributed by hashing on the join attribute, yielding an aggregate total transfer rate of 7.9 megabytes/second during the processing of this query. This should not be construed, however, as an accurate estimate of the maximum obtainable interprocessor communications bandwidth as the CPUs may be the limiting factor (the disks are not likely to be the limiting factor as from Table 3 one can estimate that the aggregate bandwidth of the 30 disks to be about 25 megabytes/second).

Speedup Experiments

For the join speedup experiments, we used the joinABprime query with a 1 million tuple A relation and a 100,000 tuple Bprime relation. The number of processors was varied from five to thirty. Since with fewer than five processors two or more buckets are needed, including the execution time for one processor (which needs 5 buckets) would have made the response times for five or more processors appear artificially fast; resulting in superlinear speedup curves.

The resulting response times are plotted in Figure 17 and the corresponding speedup curves are presented in Figure 18. From the shape of these graphs it is obvious that the execution time for the query is significantly reduced as additional processors are employed. Several factors prevent the system from achieving perfectly linear speedups.

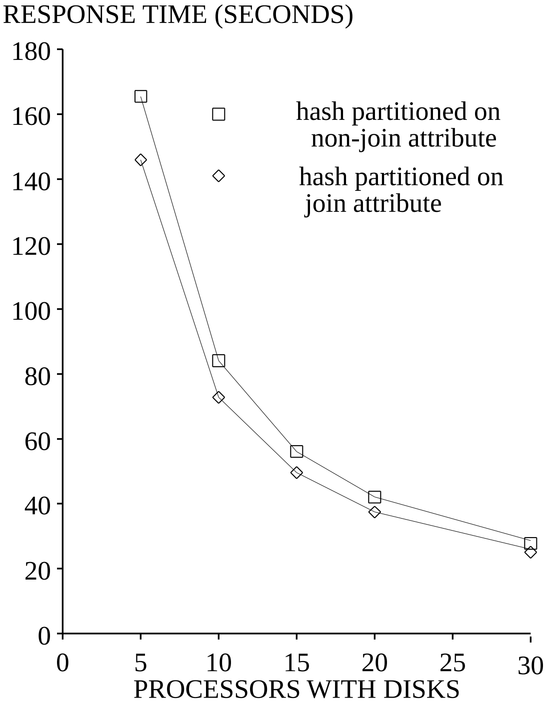


Figure 17

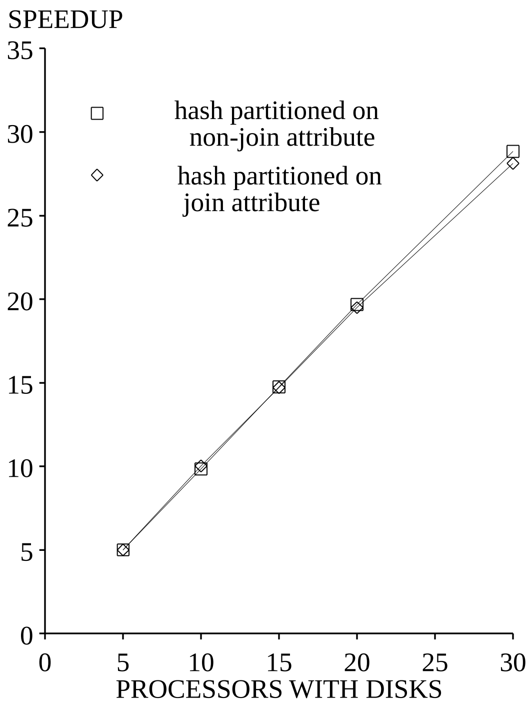


Figure 18

First, the cost of starting four operator tasks (two scans, one join, and one store) on each processor increases as a function of the number of processors used. Second, the effect of short-circuiting local messages diminishes as the number of processors is increased. For example, consider a five processor configuration and the non-partitioning attribute version of the JoinABprime query. As each processor repartitions tuples by hashing on the join attribute, 1/5th of the input tuples it processes are destined for itself and will be short-circuited by the communications software. In addition, as the query produces tuples of the result relation (which is partitioned in a round-robin manner), they too will be short circuited. As the number of processors is increased, the number of short-circuited packets decreases to the point where, with 30 processors, only 1/30th of the packets will be short-circuited. Because these intra-node packets are less expensive than their corresponding inter-node packets, smaller configurations will benefit more from short-circuiting. In the case of a partitioning-attribute joins, **all** input tuples will short-circuit the network along with a fraction of the output tuples.

Scaleup Experiments

The JoinABprime query was also used for the join scaleup experiments. For these tests, the number of processors was varied from 5 to 30 while the size of the A relation was varied from 1 million to 6 million tuples in increments of 1 million tuples and the size of Bprime relation was varied from 100,000 to 600,000 tuples in increments of 100,000. For each configuration, only one join bucket was needed. The results of these tests are presented in Figure 19. Three factors contribute to the slight increase in response times. First, the task of initiating 4 processes at each site is performed by a single processor. Second, as the number of processors increases, the effects of short-circuiting messages during the execution of these queries diminishes - especially in the case when the join attribute is not the partitioning attribute. Finally, the response time may be being limited by the speed of the communications network.

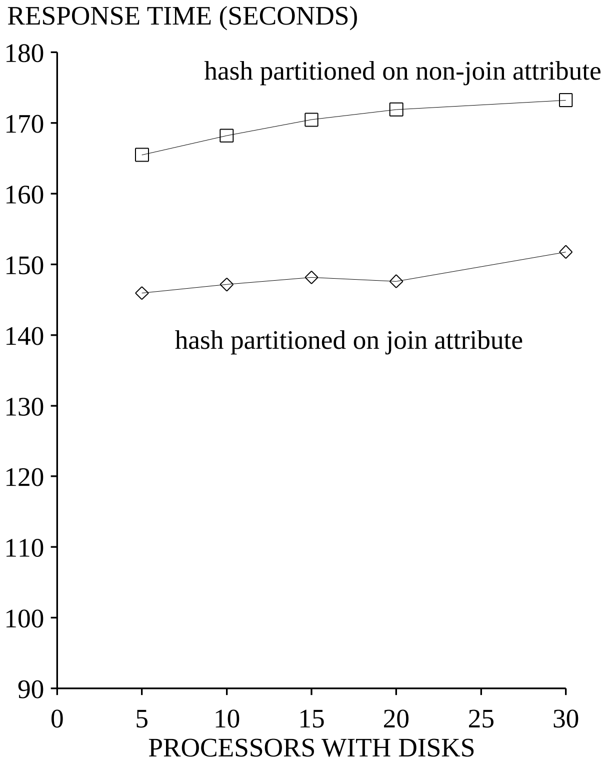


Figure 19

6.4. Aggregate Queries

Our aggregate tests included a mix of scalar aggregate and aggregate function queries run on the 30 processor configuration. 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 20 subsets. Three sizes of input relations were used: 100,000, 1 million, and 10 million tuples. The results from these tests are contained in Table 5. Since the scalar aggregates and aggregate function operators are executed using algorithms that are similar to those used by the selection and join operators, respectively, no speedup or scaleup experiments were conducted.

Table 5 - Aggregate Queries
30 Processors with Disks
(All Execution Times in Seconds)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
Scalar aggregate	1.10	10.36	106.42
Min aggregate function (20 Partitions)	2.03	12.48	120.03
Sum aggregate function (20 Partitions)	2.03	12.39	120.22

6.5. Update Queries

The next set of tests included a mix of append, delete, and modify queries on three different sizes of relations: 100,000, 1 million, and 10 million tuples. The results of these tests are presented in Table 6. Since Gamma's recovery mechanism is not yet operational, these results should be viewed accordingly.

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 a clustered B-tree index to locate the tuple to be deleted. 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, and, in addition, a clustered index exists on the Unique1 attribute. In the first case, the modified attribute is the partitioning attribute, thus requiring that the modified tuple be relocated. Furthermore, since the tuple is relocated, the secondary index must also be updated. The second modify query modifies a non-partitioning, nonindexed attribute. The third modify query modifies an attribute on which a non-clustered index has been constructed, using the index to locate the tuple to be modified.

Table 6 - Update Queries
30 Processors With Disks
(All Execution Times in Seconds)

	Number of Tuples in Source Relation		
	100,000	**1,000,000**	**10,000,000**
Append 1 Tuple (No indices exist)	0.07	0.08	0.10
Append 1 Tuple (One index exists)	0.18	0.21	0.22
Delete 1 tuple	0.34	0.28	0.49
Modify 1 tuple (#1)	0.72	0.73	0.93
Modify 1 tuple (#2)	0.18	0.20	0.24
Modify 1 tuple (#3)	0.33	0.38	0.52

7. Conclusions and Future Research Directions

In this paper we have described the design and implementation of the Gamma database machine. Gamma employs a shared-nothing architecture in which each processor has one or more disks and the processors can communicate with each other only by sending messages via an interconnection network. While a previous version of the Gamma software ran on a collection of VAX 11/750s interconnected via a 80 mbit/second token ring, currently the system runs on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives.

Gamma employs three key ideas which enable the architecture to be scaled to 100s of processors. First, all relations are horizontally partitioned across multiple disk drives which are attached to separate processors; enabling relations to be scanned in parallel without any specialized hardware. In addition, in order to enable the database design to be tuned to the needs of the application, three alternative partitioning strategies are provided. The second major contribution of the Gamma software is its extensive use of hash-based parallel algorithms for processing complex relational operators such as joins and aggregate functions. Finally, the system employs unique dataflow scheduling techniques to coordinate the execution of multioperator queries. These techniques make it possible to control the execution of very complex queries with minimal coordination - a necessity for configurations involving a large number of processors

In addition to describing the design of the Gamma software, we have also presented a thorough performance evaluation of the iPSC/2 hypercube version of Gamma. Three sets of experiments were performed. First, with a constant machine configuration of 30 processors, the response for the standard set of Wisconsin benchmark queries was measured for 3 different sizes of relations. For a subset of these queries we also measured the performance of the system relative to the number of processors employeed when the sizes of the input relations are kept constant (speedup) and when the sizes of the input relations are increased proportionally to the number of processors (scaleup). The speedup results obtained for both selection and join queries are almost perfectly linear; thus doubling the number of processors halves the response time for a query. The scaleup results obtained are also quite encouraging. They reveal that a constant response time can be maintained for both selection and join queries as the workload is increased by adding a proportional number of processors and disks.

We currently have a number of new projects underway. First, we plan on implementing the chained declustering mechanism and evaluating its effectiveness. With respect to processing queries, we have designed [SCHN89b] and are currently evaluating alternative strategies for processing queries involving multiple join operations. For example, consider a query involving 10 joins on a machine with 100 processors. Is it better to use all 100 processors for each join (allocating 1/10 of the memory on each processor to each join), or to use 10 processors for each join (in which case each join operator will have full use of the memory at each processor)? Finally, we are studying several new partitioning mechanisms that combine the best features of the hash and range partitioning strategies.

8. 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 of the first prototype. The query optimizer was implemented by M. Muralikrishna. Rajiv Jauhari implemented the read-ahead mechanism to improve the performance of sequential scans. Anoop Sharma implemented both the aggregate algorithms and the embedded query interface. Goetz Graefe and Joanna Chen implemented a predicate compiler. They deserve special credit for being willing to debug the machine code produced by the compiler.

We would also like to thank Jim Gray and Susan Englert of Tandem Computers for the use of their Wisconsin benchmark relation generator. Without this generator the tests we conducted would simply not have been possible as previously we had no way of generating relations larger than 1 million tuples.

9. References

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

[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.

[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.

[BORR81] Borr, A., "Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing," Proceedings of VLDB, 1981.

[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.

[COPE88] Copeland, G., Alexander, W., Boughter, E., and T. Keller, "Data Placement in Bubba," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[COPE89] Copeland, G. and T. Keller, "A Comparison of High-Availability Media Recovery Techniques," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon June 1989.

[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," *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 8, 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, R., 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.

[DEWI88] DeWitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[ENGL89] Englert, S, J. Gray, T. Kocher, and P. Shah, "A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases," Tandem Computers, Technical Report 89.4, Tandem Part No. 27469, May 1989.

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

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

[GHAN89] Ghandeharizadeh, S. and D. J. DeWitt, "A Multiuser Performance Evaluation of Selection Queries in a Single Processor Database Machine", July 1989, submitted for publication.

[GHAN90] Ghandeharizadeh, S., and D.J. DeWitt, “Performance Analysis of Alternative Declustering Strategies”, Proceedings of the 6th International Conference on Data Engineering, Los Angeles, CA, February 1990.

[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.

[GRAE89] Graefe, G., “Volcano: A Compact, Extensible, Dynamic, and Parallel Dataflow Query Evaluation System”, Working Paper, Oregon Graduate Center, Portland, OR, February 1989.

[GRAY78] Gray, J., "Notes on Database Operating Systems", RJ 2188, IBM Research Laboratory, San Jose, California, February 1978.

[GRAY88] Gray, J., H. Sammer, and S. Whitford, "Shortest Seek vs Shortest Service Time Scheduling of Mirrored Disks," Tandem Computers, December 1988.

[HSIA90] Hsiao, H. I. and D. J. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", Proceedings of the 6th International Conference on Data Engineering, Los Angeles, CA, February 1990.

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

[KIM86] Kim, M., "Synchronized Disk Interleaving," IEEE Transactions on Computers, Vol. C-35, No. 11, November 1986.

[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.

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

[MOHA89] Mohan, C., D. Haderle, B. Linsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", RJ 6649, IBM Almaden Research Center, San Jose, California, January 1989.

[PATT88] Patterson, D. A., G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[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.

[SCHN89a] Schneider, D. and D. DeWitt, “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”, Proceedings of the 1989 SIGMOD Conference, Portland, OR, June 1989.

[SCHN89b] Schneider, D. and D. DeWitt, "Design Tradeoffs of Alternative Query Tree Representations for Multiprocessor Database Machines", Computer Sciences Technical Report #869, University of Wisconsin-Madison, August 1989, submitted for publication.

[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.

[STON86] Stonebraker, M., "The Case for Shared Nothing," Database Engineering, Vol. 9, No. 1, 1986.

[STON88] Stonebraker, M., R. Katz, D. Patterson, and J. Ousterhout, “The Design of XPRS”, Proceedings of the Fourteenth International Conference on Very Large Data Bases", Los Angeles, CA, August, 1988.

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

[TERA85] Teradata, "DBC/1012 Database Computer System Manual Release 2.0," Document No. C10-0001-02, Teradata Corp., NOV 1985.

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