DATAFLOW QUERY PROCESSING USING MULTIPROCESSOR HASH-PARTITIONED ALGORITHMS

by

Robert Howard Gerber

Computer Sciences Technical Report #672
October 1986

DATAFLOW QUERY PROCESSING USING MULTIPROCESSOR HASH-PARTITIONED ALGORITHMS

by

ROBERT HOWARD GERBER

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1986

This research was partially supported by the Department of Energy under contract #DE-AC02-81ER10920, by the National Science Foundation under grants MCS82-01870, MCS81-05904, and DCR-8512862, by DARPA Contract N00039-86-C-0578 and by a Digital Equipment Corporation External Research Grant.

ACKNOWLEDGEMENTS

It has been my privilege and good fortune to work and study with Professor David DeWitt for whom I have the greatest respect. I would like to take this opportunity to thank him for all his encouragement, support, and direction. David has provided extremely generous support, a stimulating research environment, and the opportunity to attend national and international database conferences. His enthusiasm, encouragement and patience have made this thesis possible.

I would also like to thank the other members of my committee: Mike Carey, Miron Livny, Andrew Pleszkun and Yannis Ioannidis. Their efforts and suggestions considerably improved the quality of this thesis.

Special thanks are due to Mike Heytens who during the early days of the Gamma Project was instrumental in developing a reliable catalog manager and command interpreter. In addition, the long hours spent by M. Muralikrishna on the Gamma optimizer have been critical to the success of our research. The many significant contributions and suggestions of Goetz Graefe, Krishna Kumar, and Donovan Schneider are also appreciated and have been invaluable to the Gamma Project. The current work of Rajiv Jauhari, Shahram Ghandeharizadeh and Anoop Sharma on enhancements and extensions to Gamma have also provided valuable insights into the Gamma design.

During the early years of my graduate studies, it was also my pleasure to have the opportunity to work and study with Bob Cook. His direction and encouragement were important factors in my decision to pursue a research career in computer science.

The advice, assistance, and encouragement of Chuck Wise provided the initial impetus and later confidence to begin a career in computer science. I am grateful for all his efforts and counsel.

Our research has benefited greatly from the excellent distributed programming environment provided by the Crystal Project. We wish to thank the entire staff of the Crystal Project for all their help and advice.

We are very grateful to Paul Beebe and the entire staff of the Computing Systems Laboratory for the excellent facilities and support that have consistently been provided.

Sheryl Pomraning has been an invaluable help to our research. We are especially grateful for her skilled assistance in the preparation of technical reports.

I would be amiss, if I did not acknowledge how fortunate I've been to have a family that has given me the strength to follow my dreams. Throughout the years, I've learned from my parents and grandparents to value those achievements and contributions that are made possible through a committment to hard work. Without their support and inspiration these years of graduate study would not have been possible.

The friendship, encouragement and inspiration of Bob Christiaansen have also been critical to the maintenance of my sanity during these hectic years. Finally, I want to acknowledge the encouragement of two young ones who have assured me that being a ''computer doctor'' is a fine career. Thanks, Katie and Kristin!

ABSTRACT

In this thesis, we demonstrate that hash-partitioned query processing algorithms can serve as a basis for a highly parallel, high performance relational database machine. In addition to demonstrating that parallelism can really be made to work in a database machine context, we will show that such parallelism can be controlled with minimal overhead using dataflow query processing techniques that pipeline data between highly autonomous, distributed processes.

For this purpose, we present the design, implementation techniques, and initial performance evaluation of Gamma, a new relational database machine. Gamma is a fully operational prototype consisting of 20 VAX 11/750 computers. The Gamma architecture illustrates that a high performance database machine can be constructed without the assistance of special purpose hardware components.

Finally, a simulation model of Gamma is presented that accurately reflects the measured performance of the actual Gamma prototype. Using this simulation model, we explore the performance of Gamma for large multiprocessor systems with varying hardware capabilities.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

The relational data model has provided a conceptual framework with numerous advantages. The simplicity and regularity of the relation data structure has provided a clean environment for research into the logical foundations of database systems. High levels of physical and logical data independence have been made possible by the separation of the physical, conceptual and external layers of relational systems. Advances in the understanding of the issues of concurrency control, integrity and security have also been made possible by the uniformity of the model.

In contrast, acceptable levels of overall performance have been obtained only through the use of complex, well-tuned access methods and occasionally through the availability of database machines. Database machine research has been motivated by the necessity of improving the performance of the various relational operators. This research has led to the development of database machines that show improved performance for selections and updates. However, high performance database machine support for the complex and often unanticipated join operation has proven to be an elusive goal. In particular, the application of parallelism to the complex relational operations[1] has not provided significant improvements in performance.

In fact, 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, DEWI79a, 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. Of the commercial products the most successful one (the IDM500) does not exploit parallelism in any form.

[1]Joins, aggregate functions, duplicate elimination, etc.

Our current research suggests a solution to these problems. In this dissertation, we will examine the design and implementation of a highly parallel, high performance database machine that performs complex relational operations using hash-partitioned query processing algorithms. Three intrinsic properties of the hash-partitioning algorithms recommend their use. The minimal ordering properties of partitioned sets provide a less expensive, but sufficient method for reducing the processing requirements of complex operations. Secondly, the workload associated with the algorithms can be easily distributed. Finally, the algorithms produce and consume data in a manner that provides a basis for pipelining data directly between the operators in a compiled query tree. In the architectural environment for our database machine, pipelining has been demonstrated to be an important performance determinant [LU85].

1.2. Goals

The goal of this dissertation is to demonstrate that hash-partitioned query processing algorithms can serve as a basis for a highly parallel, high performance relational database machine. In addition to demonstrating that parallelism can really be made to work in a database machine context, we will show that such parallelism can be controlled with minimal overhead using dataflow query processing techniques that pipeline data between highly autonomous, distributed processes representing operators at different levels of a compiled query tree. Finally, a simulation model of a distributed system will be demonstrated to be an extremely valuable tool in the design of a multiprocessor database machine.

1.3. Methodology and Plan

After the publication of the classic join algorithm paper in 1977 by Blasgen and Eswaran [BLAS77], the topic was virtually abandoned as a research area. Everybody "knew" that a nested-loops algorithm provided acceptable performance on small relations or large relations when a suitable index existed and that sort-merge was the algorithm of choice for ad-hoc[2] queries. [DEWI84a, BRAT84] took another look at join algorithms for centralized relational database systems. In particular, both papers compared the performance of the more traditional join algorithms with a variety of algorithms based on hashing. The two papers reached the same conclusion: that while sort-merge is the commonly accepted algorithm for ad-hoc joins, it is, in fact, not nearly as fast as several join

[2] By "ad-hoc" we mean a join for which no suitable index or ordering for the source relations exists.

algorithms based on hashing. Chapter 2 examines the properties of the hash-partitioned algorithms that provide the basis for this attractive performance. Chapter 3 extends this examination by surveying research related to the use of the hash-partitioned algorithms.

In order to verify the performance of the hash-partitioned join algorithms in a centralized environment, we present performance measurements taken from implementations of these algorithms on a single processor running Berkeley Version 4.2 UNIX. Chapter 4 compares these performance measurements with those of a nested loops and a sort merge join algorithm.

In Chapter 5, we discuss design alternatives for extending the hash-partitioned algorithms to a distributed environment.

In [DEWI85], we used a simulation model of a distributed database system to investigate the performance of multiprocessor hash-partitioned join algorithms. Using the lessons learned from these simulation studies, we implemented a working prototype of a new relational database machine, Gamma. Gamma is a highly parallel database machine that employs hash-partitioned algorithms and dataflow query processing techniques. Chapter 6 presents the design, implementation and single user performance benchmarks for Gamma.

Using the measured performance of the Gamma prototype, we verified the accuracy of the simulation model. In Chapter 7, we describe the simulation model and present the results of using the model to explore software enhancements to the Gamma design. The simulation model is also used to investigate the impact of altering the number and capability of the hardware resources employed by Gamma.

The simulation model provided a number of valuable insights into the software design of Gamma. Many of the software design corrections and enhancements that were indicated as valuable by the simulation study have been added to the Gamma design. Chapter 8 presents performance measurements that demonstrate the improved performance that resulted from these modifications.

Finally, in Chapter 9, we discuss our conclusions concerning the use of multiprocessor hash-partitioned algorithms in high performance database machines. Also, in that chapter, we discuss our future research plans with respect to Gamma.

CHAPTER 2

AN OVERVIEW OF HASH-PARTITIONED RELATIONAL OPERATORS

2.1. Partitioning Relations

Hash partitioned algorithms are based on an ability to divide a collection of tuples into discrete, disjoint subsets. These subsets or partitions have the important characteristic that all tuples with the same value for a targeted attribute (or set of attributes) will share the same partition. While these partitions have been referred to as "buckets" [GOOD81, KITS83], this term will not be used in the following discussion in order to avoid confusion with the buckets of a hash table chain.

While the hash partitioning process can be applied for processing many complex relational operators[3], the following discussion will consider it's application in the context of the join operation. Two relations, R and S, will be referenced. Relation R is assumed to be the smaller relation. That is, relation R has fewer pages than relation S.

Tuples of relations R and S are assigned to partitions based upon the value of a tuple's join attribute value. Two functional steps are required in order to determine the assignment of a tuple to a partition. First, a hash-function is applied to a tuple's join attribute value. A **split** function is then applied to the resulting hashed attribute value. The resulting split value defines the partition to which a tuple belongs. **Split** functions map a large continuous range of numeric values into a smaller, discrete range of integer values. Each integer in the smaller range of values identifies a specific partition.

The prior application of a hash function to a targeted attribute value is recommended as a means of randomizing the distribution of values in the determinant of the split function. The modulus function is a good example of an appropriate **split** function. The combined application of a hash function and a split function will subsequently be referred to as a hashed-split function or merely as a split function when the context is unambiguous. Other forms of split functions will later be demonstrated to also have a range of applicability. Specifically, non-

[3] Joins, aggregate functions, duplicate elimination, etc.

hashed range and round-robin split functions can be used to produce partitions of tuples with certain desired characteristics. The immediately following discussion will, however, specifically consider hashed partitions.

When used in conjunction with a join operation, a split function partitions the potential range of hash values into subsets $X_1, ..., X_n$. Every tuple of relation R whose hashed join attribute value falls into the range of values associated with X_i will be put into the partition R_i. Similarly, a tuple of relation S that is split to X_i will be put into the partition S_i. Since the same hashed split function is used with both relations, the tuples in partition R_i will only have to be joined with those tuples in S_i. This follows from the observation that tuples from partition R_i will never have join attribute values equal to those of tuples in S_j where $i \neq j$. 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 the smaller partitions of the relations.

2.2. Phases of the Hash-Partitioned Join Algorithms

The various hash-partitioned join (HP-join) algorithms share three functional phases. The following discussion examines these phases in their most basic, sequential form. Later descriptions of specific HP-join algorithms will demonstrate how these basic phases can be varied and combined to achieve various performance goals.

This first phase of a HP-join algorithm is generally invoked only once. In this initial **partitioning** phase, both source relations are split into disjoint partitions. The second and third phases of the HP-join algorithms are called the **building** and **probing** phases. These phases are invoked once for each pair of partitions that were created during the **partitioning** phase. As the **building** and **probing** phases effect the actual joining of tuples, they are in combination frequently referred to as the joining phase(s) of the HP-join algorithm. The source relation whose partitions are accessed during the building phase is referred to as the building relation. Similarly, the relation accessed during the probing phase is referred to as the probing relation. The following discussion examines each of the three phases of the HP-join algorithms in more detail.

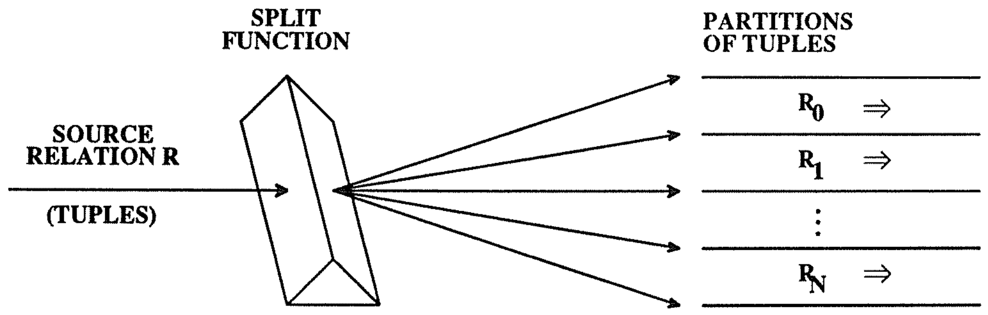
2.2.1. Partitioning Phase

The **partitioning** phase creates disjoint partitions of tuples from each of the source relations to a join. The number of partitions that are created is determined by the size of the smallest source relation which is defined to be the building relation. Smallest is defined in terms of the relation with the fewest pages. The **partitioning** phase must create sufficiently many partitions such that each partition of the building relation will not exceed the size of available main memory. This constraint is necessary because individual partitions of the building relation will be materialized into memory in their entirety in the **building** phase of the algorithm. Unlike the partitions of the building relations, the partitions of the probing relation are never fully materialized in memory. Therefore, the sizes of the partitions of the probing relation are not a concern for the HP-join algorithms.

The logical effect of the partitioning phase is depicted in Figure 2.1. In that figure, a single stream of tuples is split into multiple partitions. As tuples are split among partitions, their hashed values can be appended for later use during the **building** and **probing** phases. The additional bytes required for appending a hashed value to each tuple can be expected to outweigh the costs of recomputing the hashed value at a later stage of processing.

Guaranteeing that a chosen hashed-split function will produce partitions of the building relation that will fit in memory is not necessarily trivial. The problem of partitions growing unacceptably large is termed partition overflow. In general, partition overflow can be detected and resolved either during the partitioning or building phases of the algorithm. Also, preventive measures taken during the partitioning phase can reduce the likelihood of an occurrence of partition overflow.

The following discussion will consider the resolution of partition overflow. The level of attention that is given to this problem reflects the fact that partition overflows may occur during hash-partitioning and must be reasonably handled. However, the attention given to the problem should not be interpreted as indicating that this problem is either particularly prevalent or intractable. On the contrary, a query optimizer should be able to prevent



Splitting a Stream of Tuples into Partitions
Figure 2.1

most occurrences of partition overflow.[4]

Preventing Partition Overflow

A query optimizer can effectively prevent most potential occurrences of partition overflow by estimating the cardinality of the building relation in a join operation.[5] With such an estimate, it is possible to forestall many occurrences of partition overflow by creating a larger than necessary number of partitions. As the number of partitions increases, the tuple cardinality of individual partitions can be expected to decrease. Therefore, the probability of any single partition exceeding the size of available memory will decrease correspondingly. The more numerous, smaller partitions can be selectively coalesced during the building phase into aggregate partitions that are close to the size of available memory. This method of preventing partition overflow has been termed 'bucket tuning' (partition tuning) by the designers of the GRACE database machine [KITS83].

Alternately, it may be prudent to select from a variety of hash-split functions as a means of preventing partition overflow. The choice of an effective hash-split function will tend to randomize the distribution of tuples across partitions and, as such, will minimize the occurrence of partition overflow. A family of hash-split functions can enhance the possibility of finding an effective one and might offer a greater degree of effectiveness across a larger range of attribute domains. A sampling of the effectiveness of these alternative split functions for specific attributes can be determined at runtime or precomputed and stored in the schema.

Resolving Partition Overflow

All attempts to prevent partition overflow will fail under certain circumstances. For example, if most or all of the tuples have identical join attribute values, then partitioning will not be effective. Detecting the resulting partition overflow during the partitioning phase has the potential advantage that a method for handling the overflow can be selected before the larger, probing relation is ever accessed.[6] The techniques described for preventing

[4]Partition overflow is only dependent on the size and characteristics of the building relation of a join operation. This fact simplifies the detection and prevention of partition overflow.

[5]Query optimizers can effectively estimate the cardinality of the result relations of operations on single base relations. Estimating the cardinality of a building relation produced by a previous join operation is a more difficult task and is a topic for future research.

[6] Partition overflow is defined according to the sizes of partitions of the building relation, which is always partitioned prior to the probing relation.

partition overflow can also be used to resolve an occurrence of partition overflow. When overflow is detected, the specific overflowing partition of the building relation can be resplit. If necessary, the entire building relation can be resplit using an alternative hash-split function or the ranges of hashed values associated with partitions can be altered to produce a larger number of partitions. The new hashed-split function governing the repartitioning of the building relation is then be applied to the subsequent partitioning of the probing relation.

This repartitioning will fail in the case that the combined sizes of tuples having identical target attribute values exceeds the size of available memory during the subsequent building phase. This circumstance can be detected prior to repartitioning by collecting a histogram of the target attribute values that are seen during the initial partitioning. A repartitioning can also fail if a hash function is not available that sufficiently randomizes the values in the domain of the target attribute. In either of these cases, an alternative join method would have to be employed.

2.2.2. Building Phase

During the **building** phase, a single partition of the building relation is read into memory. An in-memory hash table is constructed from the partition. The success of this phase is determined by two factors. First, the size of the selected partition must not exceed the limits of available memory. The initial **partitioning** phase is designed to produce individual partitions of the building relation that satisfy this constraint. Second, the constructed in-memory hash table should randomize tuples among the various hash chains. Such randomization will reduce the average cost of searching the hash table for matching tuples during the subsequent **probing** phase.

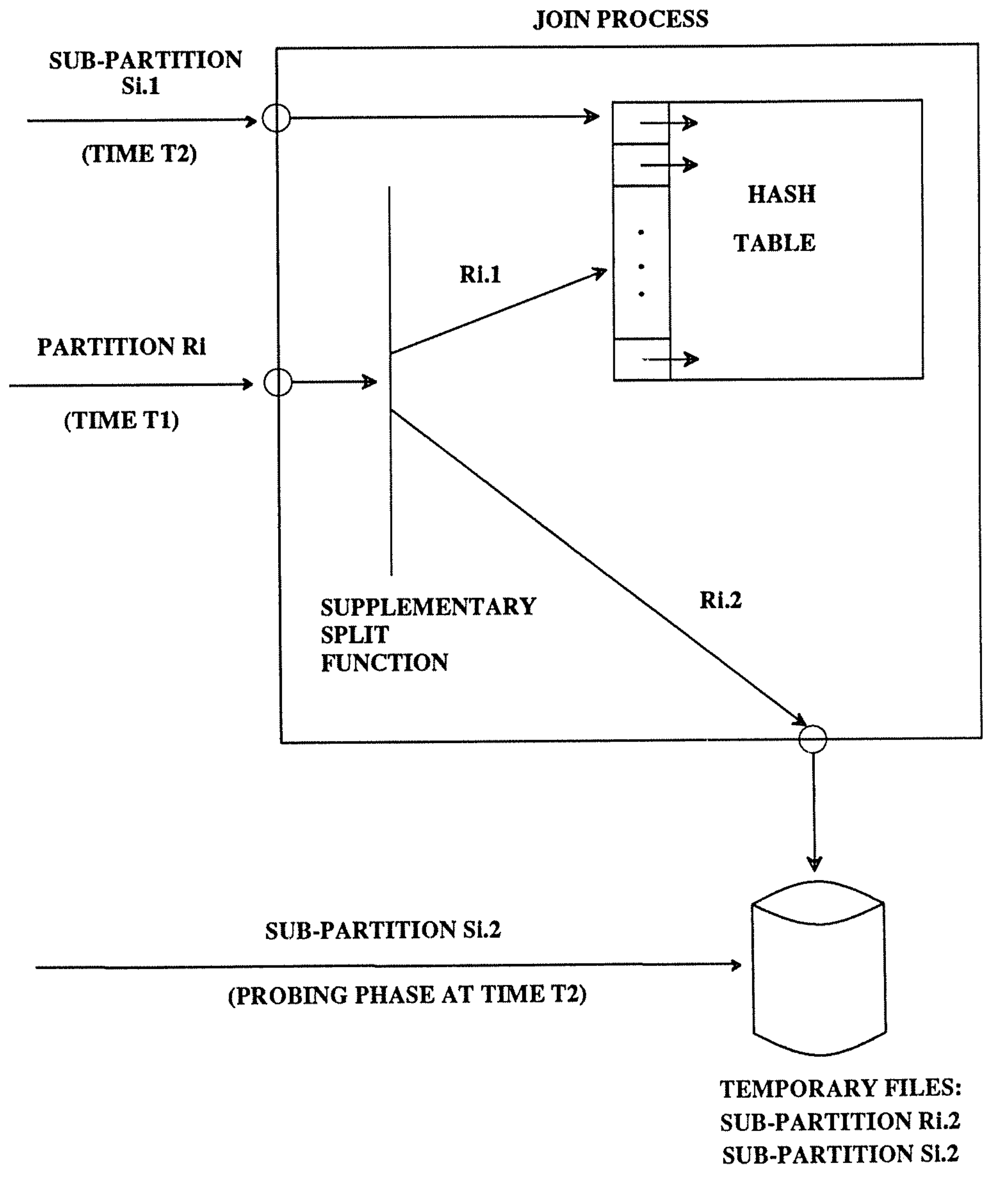
Handling Partition Overflows During the Building Phase

The detection and resolution of partition overflow may be delayed until the building phase. Delaying the resolution of overflow allows complete information on both the building and probing relations to be collected. Also, certain HP-join algorithms overlap the partitioning and building phases or pipeline tuples between the two phases. In these algorithms, partition overflow is most easily handled in the context of the building phase.

The solutions used to handle partition overflow during the partitioning phase can also be applied when partition overflow is detected during the building phase. Repartitioning during the building phase, however, requires modifying the in-memory hash table that is being constructed. This partition overflow resolution narrows the dimensions of the partition associated with the hash table by creating two sub-partitions. One sub-partition remains attached to the hash table and the other is spooled to a temporary file on disk.

The repartitioning or sub-partitioning is accomplished by applying a second split function to the tuples of the overflowing partition. First, this supplementary split function is applied to the tuples that have already been added to the hash table. Only the tuples belonging to one of the new, smaller sub-partitions are kept in memory. The hash table is purged of the other tuples which are written a new sub-partition file on disk. The second, supplementary split function is then applied to all subsequent tuples that are read from the original partition.

Figure 2.2 illustrates this second phase in the resolution of a partition overflow. The original partition R_i has



Partition Overflow Resolution Using Repartitioning
Figure 2.2

been split into two sub-partitions, $R_i.1$ and $R_i.2$. As tuples are read from the original partition R_i, they are separated by the supplementary split function into the two separate sub-partitions.

Note that this repartitioning can handle reoccurring instances of partition overflow. That is, the boundary between the two sub-partitions can be adjusted multiple times to adapt to repeated occurrences of partition overflow that impact the same partition. After the boundaries of the sub-partition associated with the hash table have been narrowed, the newly excluded tuples are assigned to the expanded overflow sub-partition. With each iteration of this partition overflow exception handler, the existing hash table is recleaned and tuples belonging to the overflow sub-partition are removed and appended to the original temporary overflow file.

Also to be considered, is the fact that some partition overflows cannot be resolved by repartitioning. This may occur due to a lack of diversity in the join attribute values of a partition of tuples. Alternately, the employed hash function may fail and map alternate join attribute values to the same hashed value. Specific methods of dealing with this eventuality will be discussed later, in the context of a specific HP-join algorithm.

Hash Table Collisions

An effective hash-split function spreads tuples uniformly among the available partitions. However, the performance of the HP-join algorithms also depends upon being able to randomize access to tuples of partitions once they have been staged into an in-memory hash table. Collisions within the in-memory hash table must be minimized in order to decrease the average cost of accessing the hash table during the **probing** phase.

Two solutions are possible. A different hash function from that used in partitioning phase can be used to assign tuples to the in-memory hash table. Alternately, different bits of a tuple's original hashed value can be used for table assignment than were used for partition assignment. A split function will typically only use a portion of this hashed value, i.e. the low order bits in the case of a modulus function. Access to the hash table can then be based upon those bits of a tuple's hashed value that were not used by the split function. For example, if a modulus-based split function were used, then tuples can be mapped to entries in the hash-table using division.

2.2.3. Probing Phase

In the third phase of a hash-join algorithm, tuples from the corresponding partition of the probing relation are read and matched against the tuples found in the current hash table. Since tuples of the probing phase are accessed only a single time, they can be streamed through memory using a single page frame. Therefore, an entire partition of the probing relation is never fully materialized in memory as each partition of the relation is accessed a page at a time.

If any repartitioning was applied during the building phase, then the adjusted partition boundaries are similarly applied to the probing relation. As illustrated in Figure 2.2, tuples belonging to an overflow sub-partition of the probing relation are directly sent to a temporary, overflow file. Corresponding pairs of overflow sub-partitions from the building and probing relation are processed after the partition retained in the hash table has been processed. Overflow may occur during the join of an overflow sub-partition. In this case, the new overflow condition can also be resolved by repartitioning.

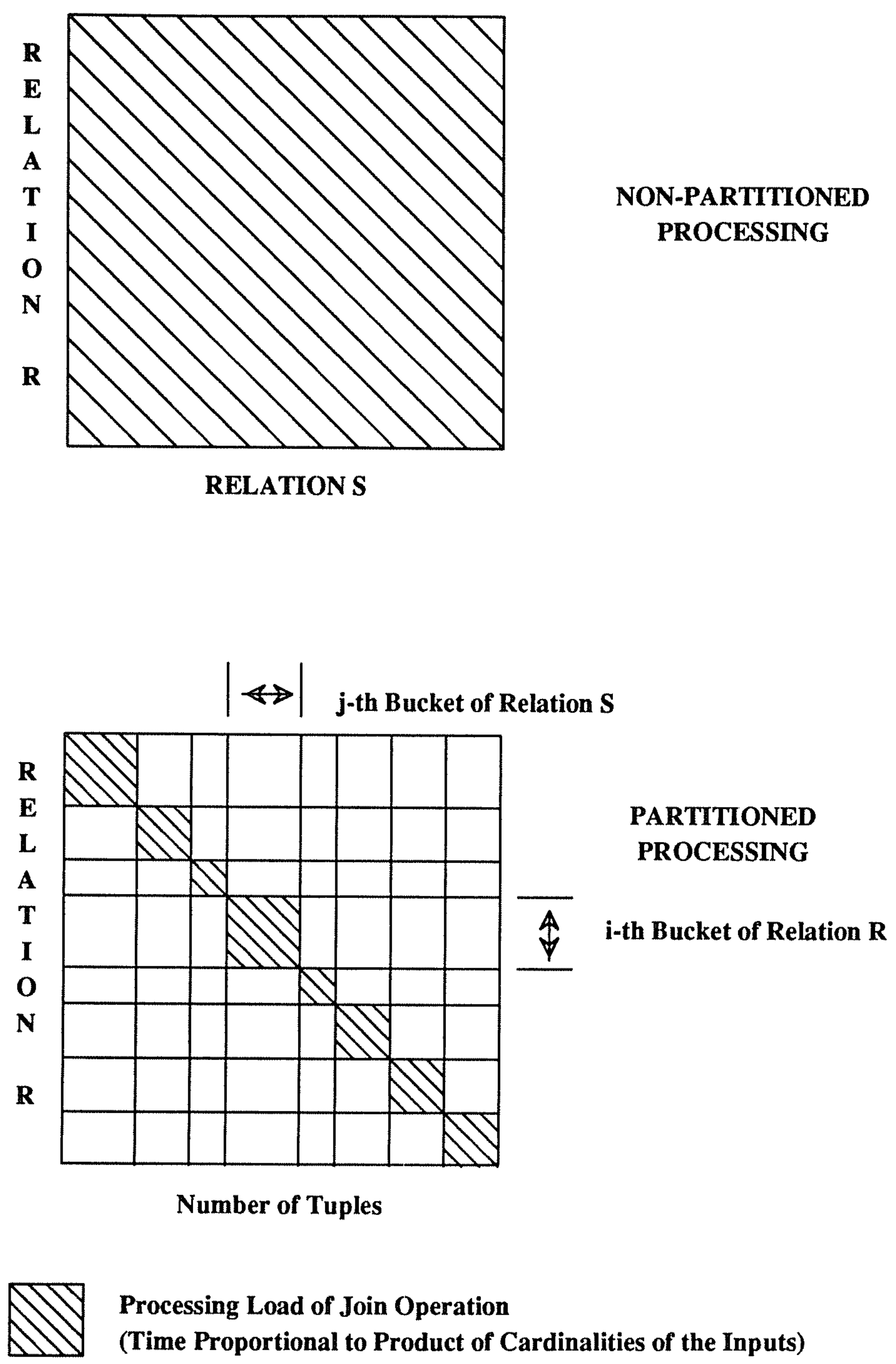
In the discussion, so far, only single join operations have been considered. In a more complex query, the result relation from one join operation may very well become a source relation for a subsequent join operation. In this case, it is interesting to note the potential for combining the probing phase of one join operation with the partitioning phase of the subsequent join operation. Instead of writing out a single result relation, the probing phase can immediately apply the split function that is required by the partitioning phase of the second join operation. Potentially significant I/O savings can be achieved by overlapping these phases.

2.3. Properties of the Hash-Partitioned Algorithms

The hash partitioned algorithms have a number of advantageous properties that recommend their use in a relational database system. The following discussion will identify these properties and will contrast them with the potential disadvantages that must be addressed by specific implementations.

Clustering-based Load Reductions:

Partitioning the source relations of a join operation provides a basis for a significant reduction in the required processing complexity. Figure 2.3 [KITS83] depicts the load reduction that would accompany a nested loops algorithm that used hash-partitioning. With hash partitioning, the performance of the illustrated nested loops algorithm is reduced from O(M*N) to $O(\Sigma(M_i*N_i))$. Significantly, the load reduction provided by the clustering of related tuples into corresponding partitions is independent of the actual join algorithm that is employed. For example, partitioning be used to reduce the processing load associated with a sort-merge join algorithm [GOOD81,KITS83].


RELATION R
RELATION S
NON-PARTITIONED PROCESSING
j-th Bucket of Relation S
RELATION R
PARTITIONED PROCESSING
i-th Bucket of Relation R
Number of Tuples
Processing Load of Join Operation
(Time Proportional to Product of Cardinalities of the Inputs)

Join Processing Load Reduction Via Hash-Partitioning
Figure 2.3

While the quantitative advantages of the clustering effects of hash-partitioning are well illustrated by Figure 2.3, there is an additional, significant qualitative advantage that should be considered. While a join algorithm cannot realistically assume that either of the source relations are small enough to fit into available memory, by creating a sufficient number of partitions, it is reasonable to expect that individual partitions of the building relation will be small enough to fit into available memory. These individual partitions can then be used iteratively as the basis for a memory resident join operation. Therefore, the clustering of tuples provided by hash-partitioning can be used to avoid the I/O costs associated with repetitive scans of the source relations to a join operation.

Total Ordering Avoided:

Significant reductions in the processing requirements of a join operation can be achieved by presorting both source relations. However, the imposition of a total ordering upon both source relations via sorting provides a greater degree of ordering than is necessary. That is, the join operation does not inherently require the total ordering of join attribute values that is provided by sorting.

The HP-join algorithms provide a basis for performing joins and other complex relational operations without imposing an complete ordering on the source relations. Thus, the processing costs associated with the creation of such an ordering can be avoided. Instead, the HP-join algorithms rely upon the clustering properties of hash-partitioning, a computationally much simpler task.

Effective Memory Utilization:

The HP-join algorithms will be shown to be capable of very effectively using main memory to reduce the processing requirements of a join operation. These algorithms use increased quantities of main memory to reduce the level of I/O activity that is required for complex relational operations. This property gains additional significance as the cost of additional increments of main memory become increasingly inexpensive.

Additionally, the memory requirements of the HP-join algorithms are determined by the size and composition of the smaller source relation, the building relation, and are independent of the size of the larger of the source relations being joined. With respect to the larger relation of a join operation, the HP-join algorithms incur only those costs related to scanning the probing relation. That is, the larger relation is never materialized in main memory, but is instead accessed as a sequential stream of tuples.

Partition Overflow:

The effectiveness of a chosen hashed-split function is dependent on the ability to randomize the assignment of tuples to partitions based on the existing values of a targeted attribute. This, in turn, determines whether clusters of tuples in the building relation can be constructed that have manageable sizes. As previously described, certain instances of partition overflow will completely resist this clustering and alternative join methods will be required.

2.4. Extending Hash-Partitioning to Other Relational Operators

There are other relational operators that can benefit from the clustering properties of the hash-partitioned algorithms. Clustering can be used to improve the performance of these operations for reasons similar to those that apply to the join operation.

2.4.1. Duplicate Elimination

The elimination of duplicate tuples following a projection operation is a computationally expensive operation that a relational database system must frequently perform. The clustering properties of hash-partitioning can provide a basis for the elimination of duplicates if a hash-split function is applied to the composite attributes of the tuples resulting from a projection. Used as a basis for partitioning, these hashed values have the effect of clustering duplicate tuples within the same partition.

Following this partitioning phase, hash tables can be built using the tuples from each individual partition. As tuples are added to a hash table, duplicates are eliminated by comparing newly added tuples against the existing members of a specific hash table chain. A new tuple is inserted into the hash table only if a duplicate of the tuple is not found. To reduce the number of required comparisons within a chain of tuples, tuples are inserted in order of their hashed values and are only compared if their hashed values are equal (hashed values are preserved for each tuple). Unique tuples can be written to the result relation immediately after being added to a hash table. Alternately, the production of unique tuples can be delayed until an entire partition has been processed, at which time, the entire hash table can be written out to the result relation.

2.4.2. Aggregate Functions

The aggregate function operator is another example of a complex relational operator that can be processed via hash-partitioning. Commonly, the aggregate function operation is processed through the use of a memory resident B-tree. If the number of groups of an aggregate function is sufficiently small, this method suffices. Otherwise, the source relation is first sorted on the grouping attributes. The aggregate function operation is then performed by scanning the sorted relation. As each of the groups is clustered by the sorting process, the scan of the sorted relation can process each of the groups sequentially.

It has been noted that hash-partitioning can also provide a sufficient degree of clustering to handle the aggregate function operation [JOHN82]. A hash-partitioned aggregate function operator splits a source relation into partitions based on the hashed value of the composite of the grouping attributes. Each of the partitions are then processed either via a B-tree based algorithm or via a memory resident hash table method [LEHM86].

CHAPTER 3

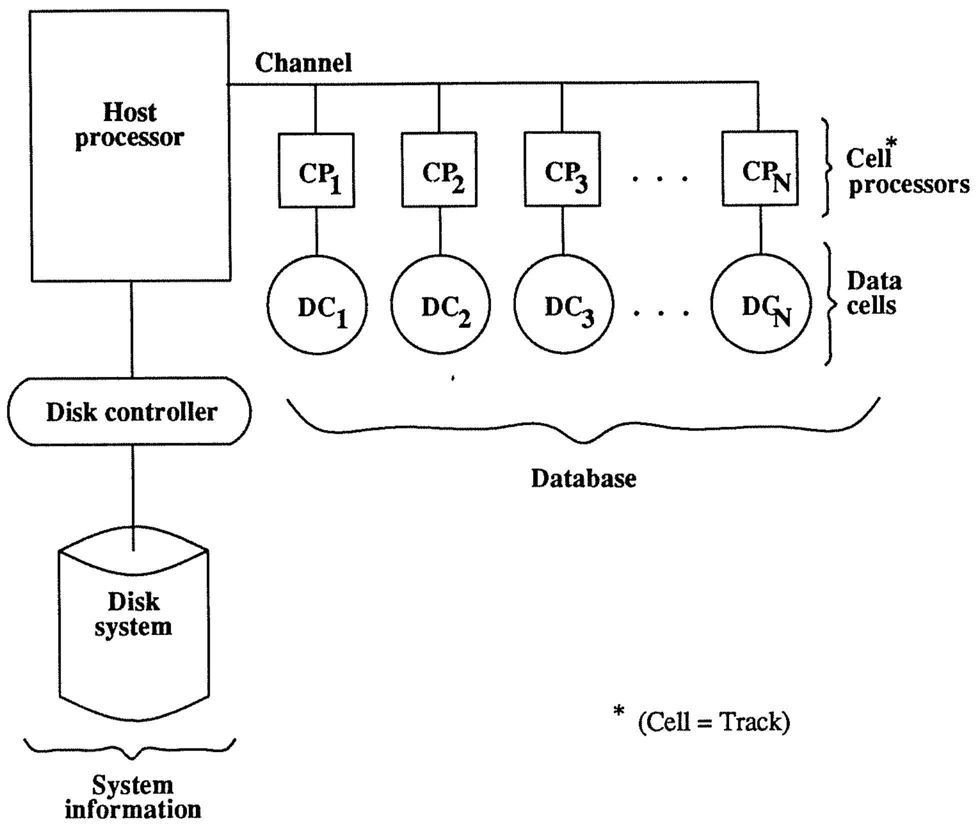
A SURVEY OF RELATED RESEARCH

In the past decade and a half, database machine research has held great promise of producing significant performance enhancements for relational database systems. In more recent years, however, there have been good reasons to suspect that such improvements in performance might prove much more elusive than earlier thought, given the current and projected state of commercially available mass storage technology. The following discussion first considers an analysis of the impediments to advancement in database machine development. This is followed by a survey of research related to the use of hash-partitioned algorithms.

3.1. Database Machine Constraints

The obstacles to continued advancement in database machine development were analyzed by Boral and DeWitt in 1983 [BORA83]. Many of the existing designs for parallel database machines were found to assume the existence of storage technologies that have not proven economically feasible and, therefore, will not become commercially available. Three categories of database machines were identified. The difficulties impeding performance improvements in each of the categories of database machines were identified and examined.

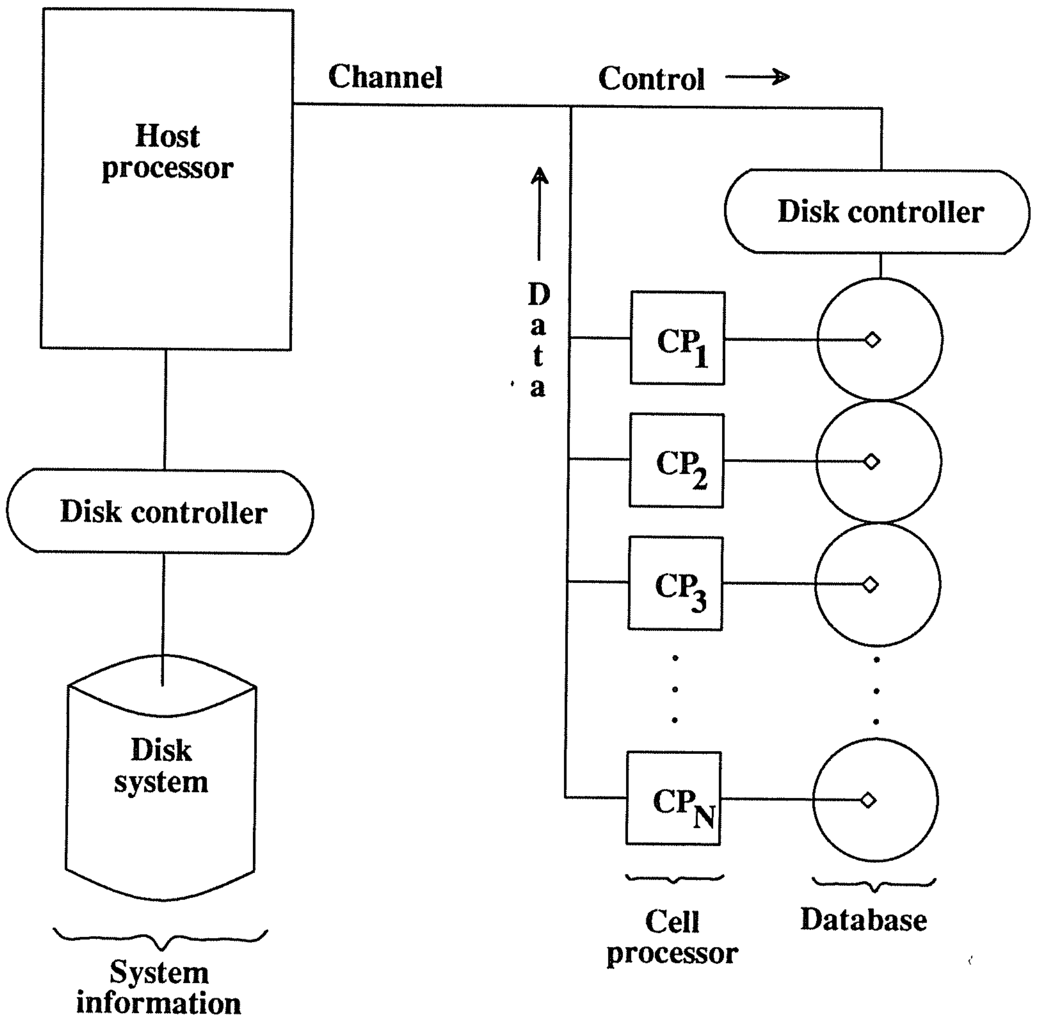
The first class of database machines, the processor per track architectures, assume that a processor can be associated with each head of a fixed head disk. Figure 3.1 illustrates a generic example of this type of architecture.



Processor Per Track Architecture
Figure 3.1

These processors share a global bus and perform restrictions and simple projections at the speed of the disk. Designs for implementing the more complex relational operators are based upon multiple combinations of the simpler search facilities. While these designs potentially suffer from high levels of contention for the global bus, a more serious problem exists. A track of data has a fairly limited size. In order to support a large database on the order of 150 million bytes, a database machine of this design would require approximately 10,000 processor/track pairs (assuming a track size of 15,000 bytes) [BORA83]. Given this constraint, many of these database machines designs have been altered. Magnetic bubble and charge-coupled storage devices have been substituted in the place of fixed head disks. However, neither of these storage technologies appear likely to become economically viable alternatives. The initial 'logic per track' design of Slotnick [SLOT70] has been followed by subsequent designs and implementations such as CASSM [SU75], RAP [OZKA77] and RARES [LIN76]. The CAFS [BABB79] database machine is a processor per track design that utilizes hashing to prefilter the tuples participating in joins.

The second category of database machines that was identified [BORA83] was characterized as the processor per head design. An example of a processor per head database machine is depicted in Figure 3.2.

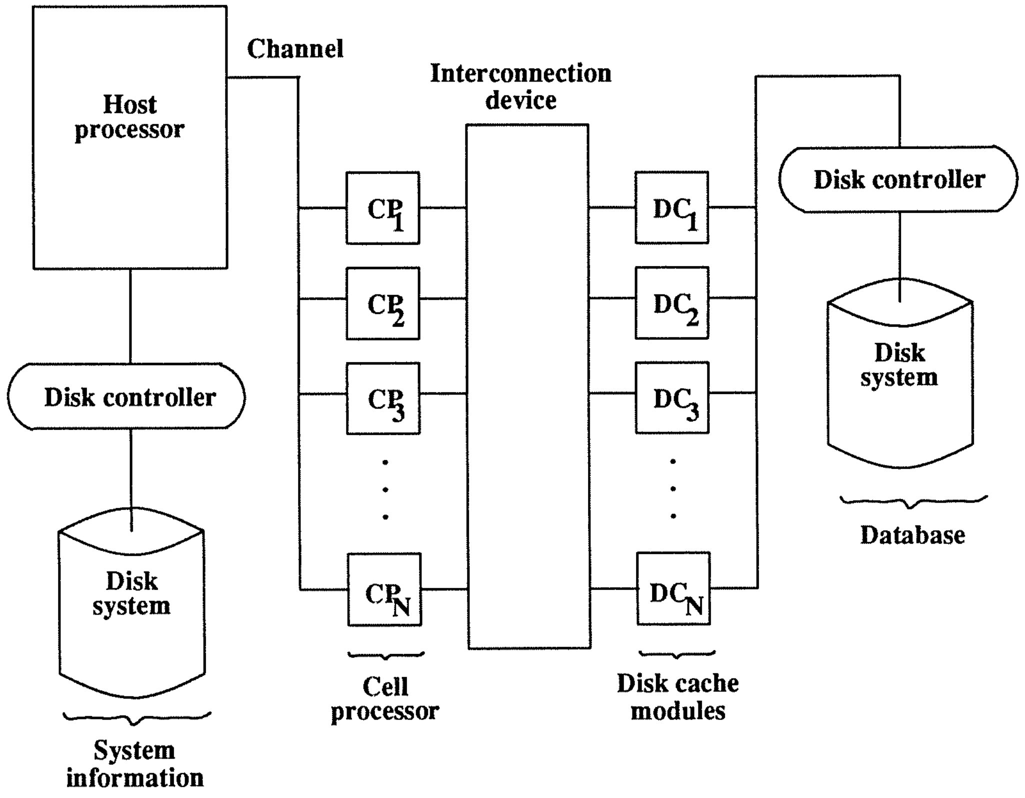


Processor Per Head Architecture
Figure 3.2

This design associates a processor with each head of a moving arm disk. These parallel read-out disks are designed to enable the reading of an entire cylinder of a disk simultaneously. The potential of such a design is enhanced even further by the addition of indexing mechanisms that are used to locate objects at the granularity of cylinders. Such indexing methods [BANE78] serve to limit the number of cylinders that must be searched. If successful, these designs could perform selections by scanning whole cylinders of a disk in parallel. However, there appear to be fundamental economical and technological reasons why machines of this design will not become viable. With all the heads of the disk attached to the same disk arm assembly, it becomes difficult to align all the heads in optimal positions. With non-optimal head positioning, the error rate is likely to increase. The necessary duplication of error detection and correction on a per head basis can significantly affect the cost of such a disk. Boral and DeWitt

[BORA83] concluded that for a large database environment the costs of correcting these problems would overshadow the potential gain in performance. HYPERTREE [GOOD81] and GRACE [KITS83] are hash-based database machines that have proposed using parallel read-out disks.

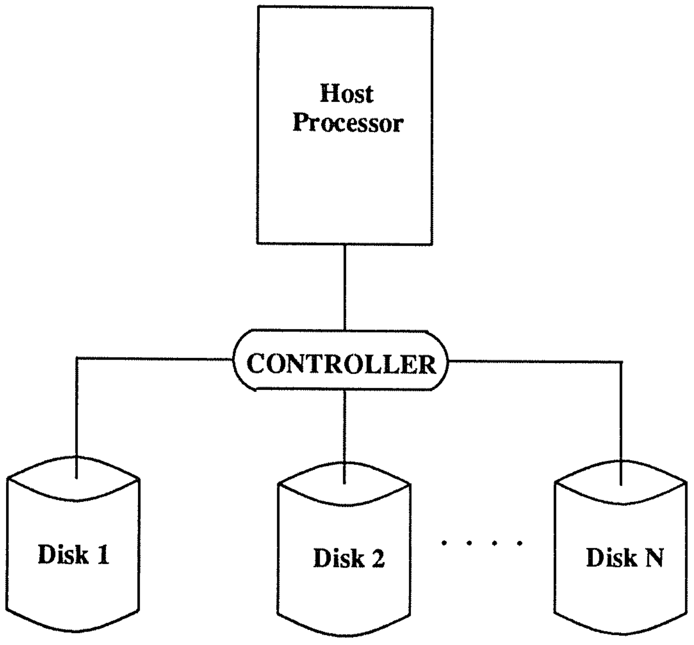
The third category of database machine design is termed the multiprocessor cache architecture (Off-the-Disk design). These designs separate the query processors from secondary storage with large disk caches. An example of a multiprocessor cache architecture is shown in Figure 3.3.



Multiprocessor Cache Architecture
Figure 3.3

The potential advantage of this approach results from the fact that once data has been staged into the disk cache, query processors are able to access it in parallel. The disk cache can also serve as a repository for the intermediate results of a multilevel query. However, a question remains as to how many processors are needed to handle the flow of data from a disk. Recent advancements in mechanical head positioning have and appear likely to continue to increase the storage capacity of disks to a greater extent than advances in the technology of the disk heads will increase the bandwidth of a disk. The increases in disk capacity have resulted from being able to place tracks in closer proximity rather than being able to place more information within a track. At the same time, processor speeds have been increasing. An analysis is presented [BORA83] that addresses the issue of how many processors are needed to handle data at the rate delivered by a disk. The analysis considers an environment where data is accessed a page at a time by a processor comparable to a VAX 11/750 that has an attached Fujitsu Eagle disk. For this environment, it is concluded that a single processor can process a selection at the rate at which data is delivered by the disk. Furthermore, three processors provided sufficient processing power to perform a sort-merge join algorithm at the speed at which data could be delivered by the disk. This analysis indicates that the success of the multiprocessor cache architectures depends upon increasing the bandwidth of the secondary storage devices. DIRECT [DEWI81], GRACE [KITS83], and SABRE [VALD84] are examples of a database machines that have used this approach.

In Figure 3.4, an alternative architecture is illustrated that has the potential to increase the aggregate I/O bandwidth of a database machine.



Alternative Architecture Using Conventional Disks
Figure 3.4

This design is significant as [BORA83] identified disk I/O bandwidth as being a critical bottleneck in the design of database machines. In this design, multiple, conventional disks are associated with a single processor. Data is distributed across related tracks of the multiple disk drives. Whereas the processor per head design accessed all the heads of a single disk in parallel, this design accesses the single heads of multiple disks in parallel.

The aggregate I/O bandwidth of the two approaches should be similar. However, since this design uses conventional off-the-shelf disk drives, it can benefit from improvements in commercially available disk technology. A customized disk controller is, however, required for controlling access to the multiple disk units. More recently, this **disk striping** strategy has been investigated by [GARC84, LIVN85, KIM85, BROW85].

3.2. CAFS

The CAFS database machine [BABB79] introduced a hashed based algorithm that used hashed values as a means of filtering out tuples prior to the processing of a relational operator. In the case of the join operator, a hash function was applied against the join attribute values of participating tuples. Then each hashed value was used as an index into a memory resident bit vector. The net effect is that each join attribute is mapped to a single (potentially non-unique) bit position in the vector.

For the purposes of this discussion, the bit vectors for relation R and S will be named R-bits and S-bits. In the first stage of the algorithm, the bit vector R-bits is created using the hashed values that are calculated during a scan of the smaller relation. Then during a scan of the larger relation, the bit vector R-bits is used to identify those S tuples qualifying for the join operation. A tuple from relation S qualifies if it's hashed value matches an entry in the bit vector R-bits. These tuples are marked and output to a single, remote join processor. The hashed values of qualifying tuples from relation S are also used to build a second bit vector S-bits. Finally, relation R is scanned a second time and the bit vector S-bits is used to select those tuples of relation R that qualify for the join. These qualifying R tuples are marked and output to the remote join processor that contains tuples from the corresponding partition of relation S.

The bit vectors produce subsets of relation R and S that are approximate semijoins of those relations. There are, however, phantom tuples that exist in the output that is sent to the remote join processor. Those phantom tuples result from the collisions of hashed values in the bit vectors. When the remote join processor actualizes the join result, these phantom tuples are eliminated. The net performance gain realized by the filtering of the bit vectors depends on the semijoin selectivity factors and the number of phantom tuples that are generated.

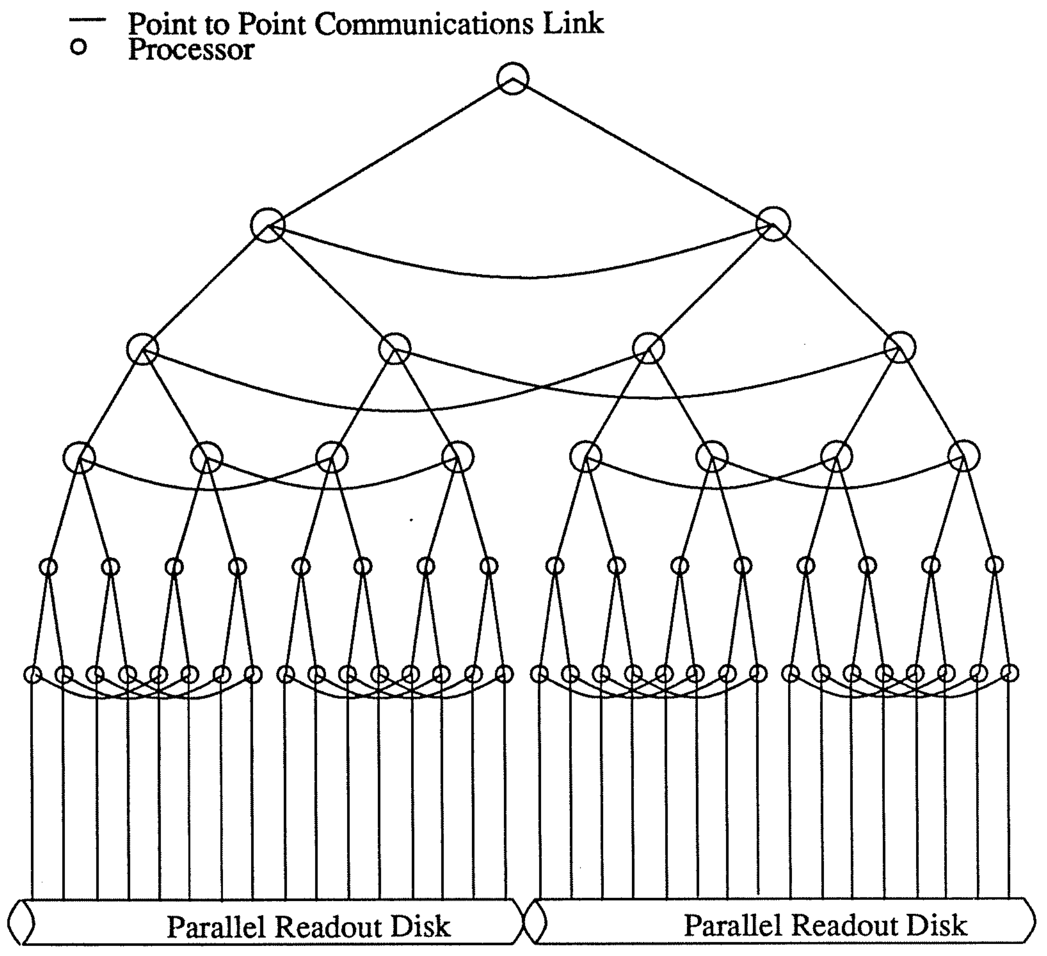
The semijoin selectivity factor for relation R is defined to be the ratio of the number of tuples in the semijoin of a relation R by relation S relative to the total number of tuples in relation R. The semijoin selectivity factors are determined by the relative correlation of join values in the relations being joined.

While the number of phantom tuples is affected by the uniformity of the distributions of join attribute values, it is also significantly determined by the size of the hash table. Too small of a bit vector will result in an increased number of collisions and phantom tuples. Increasing the size of the bit vector relative to the number of unique join attribute values has the expected effect of reducing the number of phantoms. However, another factor that is independent of the size of the bit vector was also found to have a significant effect. Babb [BABB79] concluded that a significant reduction in the number of phantom tuples can be realized by splitting a bit vector into a number of equal-sized smaller vectors. A separate hash function is associated with each of the smaller bit vectors.

3.3. HYPERTREE

In 1981, Goodman investigated the performance of various join algorithms in a multiple processor environment [GOOD81]. Three hash-join algorithms were analyzed in the context of the proposed HYPERTREE architecture. The performance of the hash-join algorithms were compared against the performance of traditional join algorithms as extended to the HYPERTREE environment.

The HYPERTREE architecture illustrated in Figure 3.5 is designed for VLSI implementation.



HYPERTREE Architecture

Figure 3.5

Large numbers of single chip processors were connected via high speed point to point communications links. The communications bandwidth was projected to be in the 10 Mbyte per second range. The processors were connected in a binary tree format that was augmented by additional communications links between pairs of sibling processors. The leaf node processors were also connected to one of their siblings via an interconnection strategy called the perfect shuffle structure [STON71]. Additionally, each of the leaf processors were assumed to have a direct port to the head of a movable arm disk. Given such a parallel read-out disk architecture, the relations in a database were assumed to have been spread uniformly across the tracks of the same group of cylinders. Scans of relations could then be executed a cylinder at a time.

The hash-join algorithms are proposed in the context of the join query presented by Blasgen and Eswaran [BLAS77]. In this query, the two source relations R and S have restrictions applied to them prior to the join. The result of the join is then projected to remove unwanted columns.

In the first hash join algorithm, no join or restriction indices are assumed to exist. The cylinders containing relation R are scanned and the restriction is applied by the leaf nodes. A bit vector is constructed by hashing the join attributes of those tuples surviving the restriction. At the end of this phase, each leaf node contains a bit vector representing the hashed join attributes of those tuples seen by the node that are eligible to participate in the join. The individual bit vectors are then transmitted and merged upward through the HYPERTREE structure. The root node ends up containing the composite bit vector representing the hashed join attributes of all eligible R tuples. This vector will be referred to as the R-bits vector.

The same procedure, followed for the S relation, produces the vector S-bits. The root of the tree then intersects R-bits and S-bits creating a new bit vector RS-bits. Bit vector RS-bits effectively represents the hashed join attribute values of tuples that will participate in the join operation. This new vector RS-bits is then propagated downward to the leaf nodes of the processor tree. Each relation is then scanned a second time. Tuples with hashed join attributes represented in the vector RS-bits are identified. The unwanted attributes of these tuples are projected away and the resulting tuples are sent to a predetermined node.

The selection of a destination node for these tuples is done via a partitioning process which guarantees that all tuples with the same join attribute value will be sent to the same destination node. Once all possible tuples have been collected by a destination node, a sort merge algorithm is used to effect the final joining of matching tuples. The sort merge process eliminates those 'phantom' tuples that will have resulted from collisions in the bit vector which was used to screen the join candidate tuples.

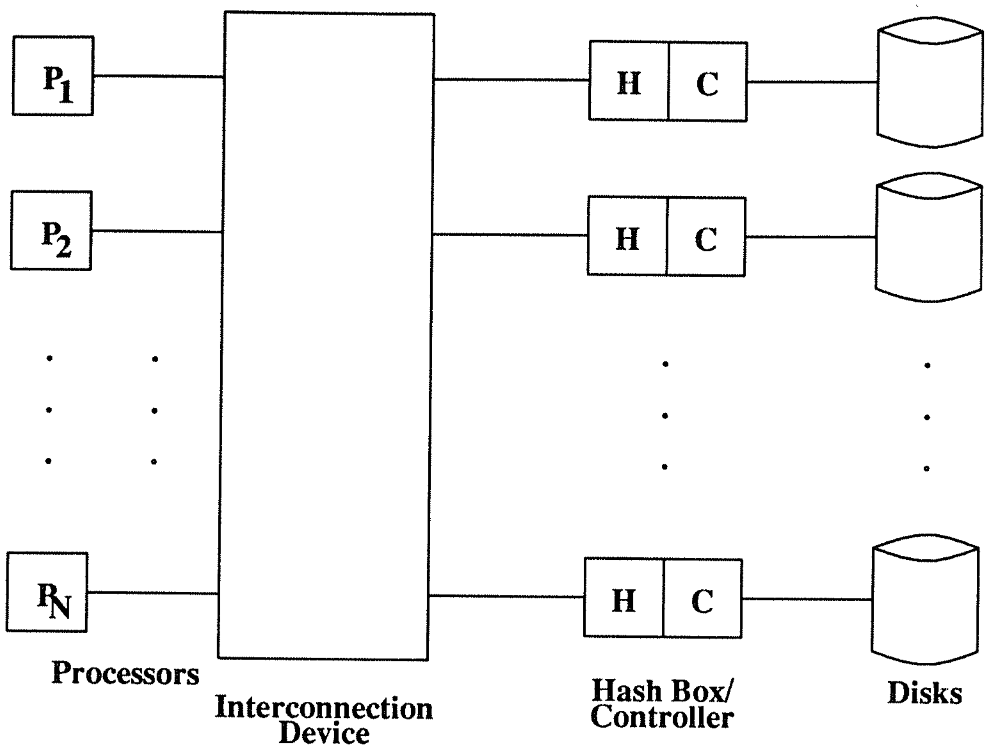
The second hash-join algorithm assumes the existence of join indices on relations R and S. Bit vectors for relation R and S are created in the leaf processors via a scan of the join indices. The join attributes found in the indices are hashed and entered into the appropriate bit vector. The bit vectors R-bits and S-bits are merged upwards in the tree. The root node intersects R-bits and S-bits and creates a new bit vector RS-bits in a similar fashion to that used in first hash-join algorithm. The resulting vector RS-bits is propagated to the leaf nodes and relations R and S are scanned and the restriction predicates are applied. Tuples that pass the restriction become candidates for the join if their hashed join attribute values are represented in the bit vector RS-bits. The join candidate tuples are projected and sent to predetermined nodes. The final phase of the algorithm uses a sort-merge join method to actualize the final result relation.

The third hash-join method assumes that both join and restriction indices exist for both source relations. The restriction index of relation R is scanned and the tuple identifiers (TIDs) of qualifying tuples are hashed and entered into the TID bit vector R'-bits. All the vectors R'-bits are merged upwards from the leaves of the processor tree. The composite vector R'-bits is then propagated back to the leaves. Now, the join index of R is scanned. The individual TIDs from the join index are hashed and compared to the entries in vector R'-bits. If the join tuple TID is present in the bit vector, then the hashed value of the join attribute is transmitted to the parent of the leaf processor where it is entered into join attribute bit vector R-bits. When the scan of the R join index is complete, the bit vectors R-bits are all merged upwards to the root of the tree. The same process is repeated for relation S. The root node then intersects the join attribute bit vectors R-bits and S-bits. The resulting join attribute bit vector RS-bits is sent back to the leaves. The relations R and S are now scanned and the required restriction is applied. The join attribute values of qualifying tuples are hashed and compared against the entries in vector RS-bits. The resulting join candidate tuples are projected and sent to destination nodes and joined via a sort-merge algorithm.

These three hash-join algorithms are compared against HYPERTREE distributed versions of four traditional join methods presented by Blasgen and Eswaran [BLAS77]. Except for small relations, the hash-join algorithms dominated the distributed versions of the traditional join algorithms. In particular, if join indices exist, hash join method number two was always best. Otherwise, hash join method one was the best choice. The performance of the hash-join method using TID bit vectors was always dominated by the performance of the other two hash-join algorithms. The dominance of the hash-join algorithms was attributed to their ability within the HYPERTREE architecture to efficiently access large blocks of data sequentially.

3.4. An Aggregation Database Machine

In [JOHN82], a multiprocessor database machine has been proposed that performs aggregations with a hash-based algorithm. The Multiprocessor Hash (MPH) architecture as depicted in Figure 3.6 was designed to improve the performance of the aggregation operation.



Multiprocessor Hash (MPH) Architecture
Figure 3.6

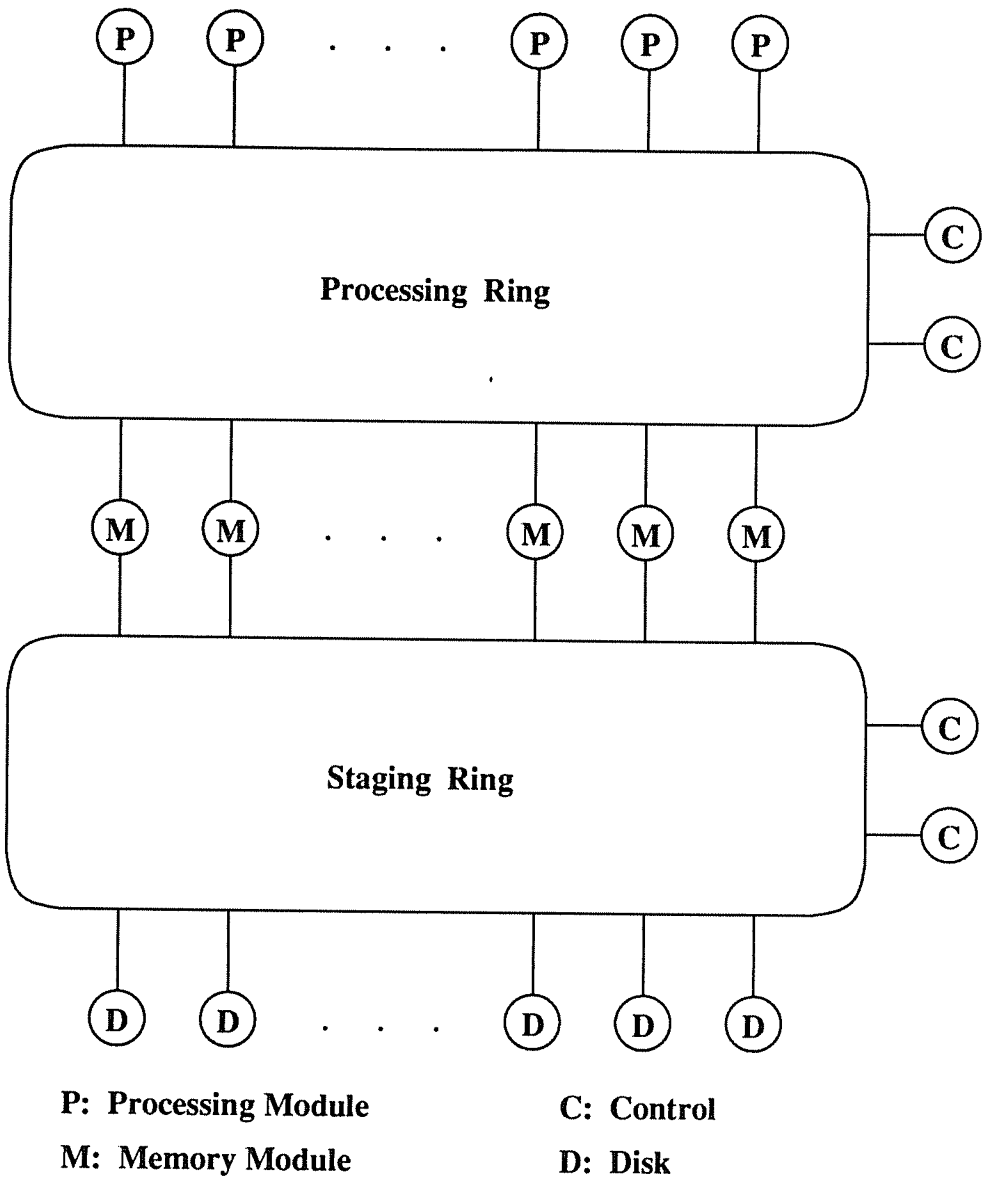
The MPH architecture associates a filtering processor with a conventional disk. The filtering processor performs selections on the tuple stream produced by the disk. The filter processor also is capable of computing a hashed value based on the grouping attributes of the individual tuples. The tuples are partitioned using these hash values. Multiple query processors are connected to the filtering processor via a high speed bus. A single query processor may be assigned the processing task corresponding to one or more partition. Based upon the hashed value computed by the filter processor the tuples are sent to the appropriate query processor. A memory resident B-tree algorithm is used by each query processor to perform the aggregation for the groups it receives. The clustering property of hash-partitioning guarantees that all tuples belonging to a specific group will be placed in the same partition and will therefore be processed by a single query processor.

3.5. GRACE

In 1983, the designers of the relational algebra machine Grace [KITS83] described a hash-join algorithm that used hashing to partition the processing load associated with the join operation. While the Grace database machine is designed to incorporate parallel processing, the Grace algorithm can also be applied in the context of a centralized database management system.

Potentially, the nonuniformity of partition sizes can present a problem for the Grace algorithm. Partition overflow occurs when a nonuniform partitioning of tuples among partitions causes certain individual partitions to grow beyond the supported capacity of the second phase of the join algorithm. In a centralized environment, this problem leads to a longer execution time as the employed join algorithm may no longer be able to assume that one of the partitions of a relation can fit into main memory. The problem is potentially more severe in the distributed Grace environment where partitions must be fit into disk caches of limited size. In the Grace database machine, a preventive measure called bucket (partition) tuning is employed to reduce the occurrence of partition overflow. Initially, many small partitions are created. Later, multiple partitions are integrated into larger partitions that are of optimal size for the processing module that actually performs the join.

The Grace database machine is designed to take advantage of the fact that the partitions can be joined in parallel by multiple processors. In order to actualize this potential, the Grace machine uses a three level memory hierarchy to stage the data between the disks and the join processors. This architecture is illustrated in Figure 3.7.



GRACE Database Machine Architecture
Figure 3.7

At the lowest level of the memory hierarchy are conventional moving head disks. Associated with each disk is a filtering processor that is capable of performing simple selections and projections. The intermediate level of the memory hierarchy consists of multiple disk caches. The top level of the memory hierarchy is comprised by the local memories of the join processors. Data transfers between the levels of the memory hierarchy occur on a multi-channel ring bus.

The intermediate level disk caches are comprised of a combination of semiconductor RAM and bubble memory. These memory modules are designed to generate a stream of tuples belonging to a given partition at the request of a join processor. The disk cache modules associatively retrieve partitions by referencing the hashed value of the join attribute that is stored as a prefix to each tuple in the cache. The partitions of a relation are assumed to have been spread across multiple cache modules as a result of the manner in which a relation is staged from the disk modules. The dispersing of partitions across multiple disk caches is designed to enhance the concurrent gathering of partitions by multiple join processors. At the same time, partition dispersement avoids the potential problem of a particular partition exceeding the capacity of a single disk cache.

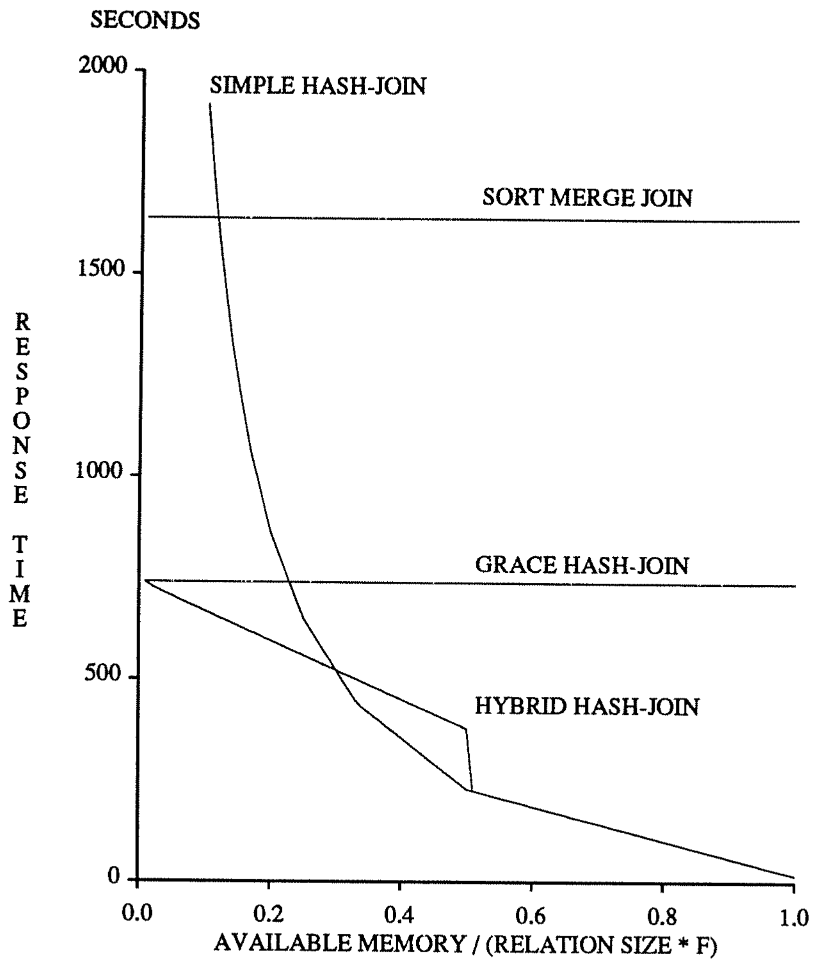
The join processing modules are designed to implement the actual join operation via a sort-merge algorithm. The processing modules initiate the transfer of partitions of tuples from the cache memory modules. These processing modules 'collect' partitions of interest from the different cache memory modules in a serial fashion. Tuples read from the memory cache are directly input to a hardware sorter. The sorted partitions are later merged by the join processor. If necessary, the processing module computes and prepends to result tuples the hashed value of the next targeted join attribute. The result relation is then propagated downward through the memory hierarchy.

The various Grace modules can be allocated independently of one another. This allows Grace to concurrently execute multiple relational operators. In particular, intermediate results of a complex query may sometimes not propagate all the way back to disk. After being written back to a cache module, the intermediate results may be directly consumed by a processing module that is working on the next level of a complex query tree.

3.6. Join Algorithms for Main Memory Databases

Hash-join algorithms have been studied in the context of a centralized database management system whose processing environment contains large amounts of main memory [DEWI84]. This study was motivated by the trend toward increasingly inexpensive, large main memories. The study examined various hash-join algorithms whose performance is enhanced by such an environment. The analysis included the traditional sort-merge join algorithm [BLAS77], the Simple and Grace [KITS83] hash-partitioned join algorithms and a new algorithm, the Hybrid hash-join [DEWI84]. Analytical models for the join algorithms were derived and used as a basis for a comparison of the performance of the algorithms.

The results of the comparison of join methods is presented in Figure 3.8.



Main Memory Databases: Join Performance Comparison
Figure 3.8

The performance of the algorithms was studied as a function of the relative amount of available memory which was defined in terms of that proportion of the smaller relation which fit in main memory. Increasing the amount of available memory beyond the size of the smaller relation was not found to make any further changes in the performances of any of the algorithms. Hybrid hash-join was found to outperform the other join algorithms over a wide range of available memory environments. The authors suggested that additional benefits could also derived from the potential simplification of query optimization afforded by the nature of hash-based relational operators. This is possible because the hash-based algorithms do not require that the tuples of the source relations be ordered.

3.7. A Centralized Hash-Partitioned Join Algorithm

[BRAT84] analytically compared the performance of a centralized hash-partitioned join algorithm with a nested loops and a sort merge algorithm. The hash-partitioned join algorithm completed all partitioning prior to the start of the joining phase of the algorithm. The performance of the join algorithms were studied with respect to a fixed amount of main memory as the relative and absolute sizes of the source relations were varied.

With unordered source relations, the performance of the hash-partitioned join algorithm was found to dominate both the sort merge and nested loops join algorithms over the entire available memory spectrum. Also, with respect to a sort merge join, the hash-partitioned algorithm was better able to take advantage of the situation when source relations were of differing sizes. Additionally, [BRAT84] observed that during the merge join phase of a sort merge algorithm, files must be strictly scanned in sequential order. By contrast, a hash-partitioned join can access a file in any order, potentially optimizing the scan order to minimize the aggregate disk access time for the file.

3.8. SABRE

An analytical evaluation of a hash-partitioned, a sort merge and a nested loops join algorithm was conducted in the context of a multimicroprocessor database machine [VALD84]. Versions of these join algorithms were adapted to the architectural environment of the SABRE database machine. Due to different architectural constraints, however, the hash-partitioned algorithm that is presented in [VALD84] differs significantly from the previously described hash-partitioned algorithms [DEWI84a, DEWI85]. As a result, this study reaches different conclusions concerning the relative performance of a hash-partitioned join algorithm versus that of a sort-merge or nested loops join algorithm. These differences reflect the fact that hash-partitioning is a technique that can be used as a basis for a variety of algorithms and can be adapted for use within different architectural contexts.

In the presented SABRE system, a number of microprocessors communicate and exchange data via a cache memory. An interconnection network provides for parallel transfers of data pages between query processors and the cache. Also, a page from the cache can be broadcast to multiple query processors. In the specific system tested, each query processor was assumed to have a limited amount of local memory, i.e. five 4 kilobyte pages. Permanent relations are stored on conventional disks. A filtering processor is associated with each disk. These filtering processors apply selection and projection predicates and stage the resulting data streams into the global cache

memory. The filtering processors are designed to apply predicates at the rate at which the disks produce data.

The evaluated hash-partitioned join algorithm begins by splitting the smaller source relation into disjoint partitions that are written to the cache. If the size of the smaller source relation exceeds the number of available cache pages, some of the data pages of these partitions may migrate to disk. In the process of partitioning the smaller source relation, a bit vector filter is constructed that represents the join attribute values of the partitioned relation. After this initial partitioning stage, pages from the larger source relation of the join are sequentially distributed among all query processors with each query processor receiving a single page. The join algorithm iteratively processes any remaining pages of the larger source relation in subsequent passes of the algorithm. A copy of the bit vector filter that was produced during the partitioning of the smaller source relation is also distributed to each of the query processors.

For each page that is received from the larger relation, a query processor qualifies tuples as candidates for the join operation by comparing their hashed join attributes against the local bit vector filter. Each qualified tuple of the larger relation is then compared against every tuple of the corresponding partition of the smaller relation. That is, all the pages of a specific partition of the smaller relation are retrieved across the network from the global cache for each qualified tuple of the larger relation. Each tuple contained in these pages is matched against the qualifying tuple of the larger source relation. Matching tuples are written to the local output buffer which is flushed to the cache when full.

The performance of the hash-partitioned join algorithm was found to dominate the distributed nested loops and sort-merge algorithm only when the semijoin selectivity of the larger relation by the smaller relation was less than 0.01. That is, the hash-partitioned algorithm performed best when at least 99% of the tuples of the larger relation were disqualified via bit vector filtering.

Two differences should be noted when comparing these results with those of [BRAT84, DEWI84a, DEWI85]. The hash-partitioned SABRE algorithm utilized only a very limited amount of local memory, i.e. one 4 kilobyte input page per source relation. In contrast, the hash-partitioned algorithms of [BRAT84, DEWI84a, DEWI85], evaluated hash-partitioning in environments where larger amounts of main memory were available with respect to the size of the smaller source relation. In those algorithms, the additional increments of main memory were used to decrease the number of I/O operations that were required for the execution of a join operation. Also, in SABRE, the larger source relation to a join is never partitioned. Instead, bit vector filtering is relied upon to reduce the computational complexity of effecting the final joining of tuples. Because the larger relation is not partitioned, it must be iteratively processed in multiple passes when it exceeds the aggregate memory capacity of the query processors. By contrast, the memory requirements of the HP-join algorithms of [BRAT84, DEWI84a, DEWI85] are determined by the size of the smaller source relation to a join. In an environment with more significant amounts of local memory, [DEWI84a] found that completely partitioning both source relations to a join could significantly reduce the execution time of a join operation with respect to a hash-partitioned algorithm that iterated through one of the source relations doing a minimal amount of partitioning.[7]

3.9. Multiprocessor Hash-Partitioned Join Algorithms

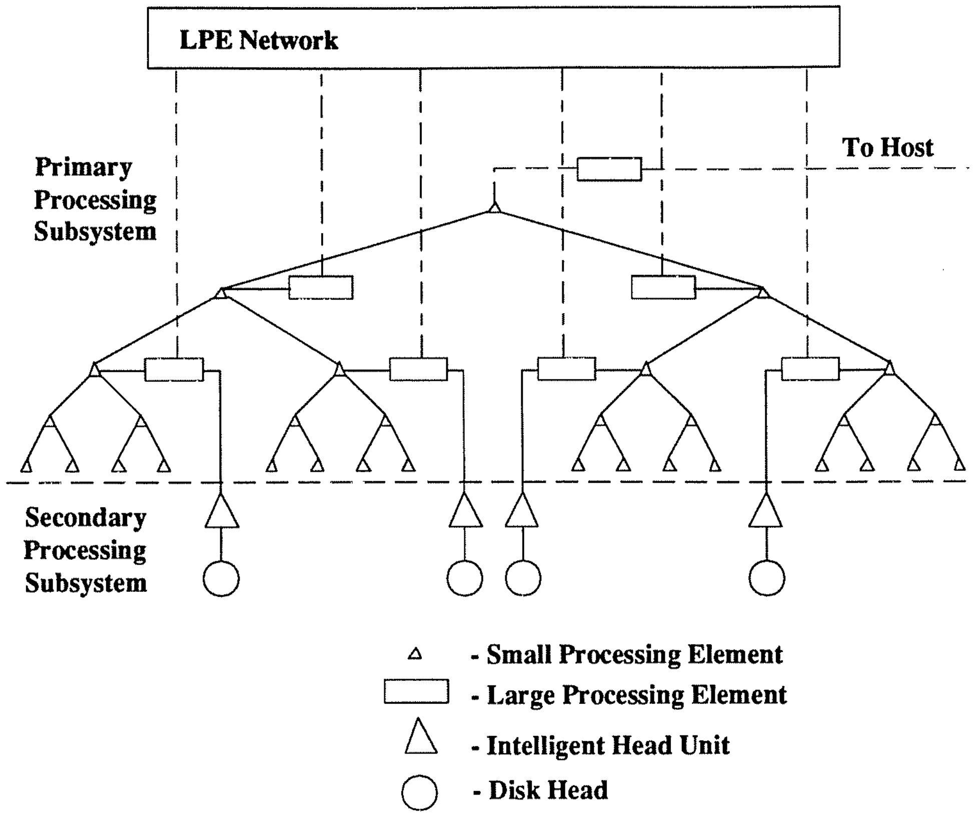
In [DEWI85], we investigated the performance characteristics of multiprocessor hash-partitioned algorithms using a simulation model. The modeled architecture used loosely-coupled processors and conventional disks. By associating a processor with each disk, the communications overhead of the distributed design was minimized.

Permanent source relations were horizontally partitioned across all disks. The processors with attached disks provided parallel access to permanent relations. Processors without disks were effectively used to support the joining phase of a distributed hash-partitioned join algorithm. The results of this study provided the basis for the subsequent design and implementation of Gamma, a new relational database machine [DEWI86].

3.10. NON-VON

Shaw has proposed a relational database machine that uses a two level associative memory architecture [SHAW80,SHAW85]. As illustrated in Figure 3.9,

[7]The Hybrid hash-partitioned join algorithm partitions both source relations. The Simple hash-partitioned join algorithm does a minimal amount of partitioning. (See Section 4.3)


LPE Network
To Host
Primary Processing Subsystem
Secondary Processing Subsystem
- Small Processing Element
- Large Processing Element
- Intelligent Head Unit
- Disk Head

NON-VON Database Machine Architecture

Figure 3.9

relations are stored on a secondary processing system (SPS), a logic per track device designed for implementation via parallel read-out disks. The SPS device processes simple selections and has the capability to compute hash functions from composite attributes. Qualifying tuples are staged into a primary processing system (PPS). The PPS is a smaller associative memory that provides faster associative memory retrievals. The implementation of PPS is proposed as a large collection of microprocessors connected in an augmented binary tree format. Query processors with direct access to the PPS perform the relational operators by associatively retrieving tuples with the requisite attribute values. When the size of a source relation exceeds the capacity of the PPS, NON-VON uses hashing to split the relation into partitions called 'key disjoint partitions'. If necessary, this splitting process is recursively applied until all partitions are less than or equal in size to the PPS. In [SHAW85] this partitioning is completed prior to the start of the join algorithm by scanning the source relation once and creating separate files for each of the partitions. In an earlier design [SHAW80], each partition of tuples was sequentially produced, as needed, by rescanning the original source relation. To effect a join, NON-VON stages corresponding partitions of the source relations of a join into the PPS. A nested loops algorithm then joins matching tuples using the associative retrieval capabilities of the PPS.

3.11. Query Processing Using Fragmentation Techniques

[SACC86] investigated the use of a hash-partitioning join algorithm in the context of a centralized database system. The performance of hash-partitioning was compared analytically with a sort merge join algorithm. The hash-partitioned join algorithm recursively partitioned the source relations until each fragment of the smaller relation fit into main memory. Hash-partitioning performed significantly better than sort merge when both source relations were unordered or when the sort merge algorithm required a substantial amount of sorting, i.e. if the larger source relation had to be sorted.

The hash-partitioned join algorithm also performed better than sort merge when unordered source relations with non-uniform distributions of join attribute values were used. That is, a hash-partitioned join algorithm using a nested loops overflow algorithm was found to dominate sort-merge when join attribute values were replicated and distributed non-uniformly.

CHAPTER 4

CENTRALIZED HASH-PARTITIONED JOIN ALGORITHMS

The following chapter describes the design and implementation of various hash-partitioned algorithms in the context of a single processor, relational database system. The performance of these algorithms is compared with that of a sort-merge and nested loops algorithm for an ad-hoc join query. That is, the source relations will be unordered and suitable indices will not exist.

4.1. Implementation Rationale

To verify the analysis presented in [DEWI84a] and to gather information on CPU and I/O utilizations of the hash-partitioned algorithms, we implemented three hash-partitioned algorithms on a VAX 11/750 running 4.2 Berkeley UNIX. As in the earlier analytical investigation of hash-partitioning [DEWI84a], centralized versions of the Simple, Grace, and Hybrid algorithms were used. In addition to these hash-partitioned join algorithms, two other popular join algorithms were studied. These algorithms, a sort-merge algorithm and a hash-based nested loops algorithm, provide a context for comparing the performance of the hash-partitioned join algorithms.

4.2. Implementation Environment

All the join algorithms used the services of the Wisconsin Storage System (WiSS) [CHOU85]. While originally conceived as a replacement for the UNIX file system (WiSS can run on top of a raw disk under UNIX), WiSS has also been ported to run on the Crystal multicomputer [DEWI84b]. The services provided by WiSS include structured sequential files, byte-stream files as in UNIX, B^+ indices, stretch data items, a sort utility, and a scan mechanism. As demonstrated in [CHOU85], WiSS's performance is comparable to that of commercially available database systems.

4.3. Centralized Hash-Partitioned Join Algorithms

The following sections describe the details of the individual join algorithms that were evaluated. All the algorithms were allocated identical amounts of main memory for buffering pages of the relation. Similarly, all the algorithms accessed relations on disk a page at a time, blocking until disk operations completed. Disk pages contained 4 kilobytes of information.

4.3.1. Simple Hash-Join

The Simple hash-partitioned join algorithm does not complete an entire partitioning phase before initiating the join phases. Instead, the algorithm processes partitions one at a time through all three phases. Two files are associated with relations R and S. There are files R-input (S-input) which contain tuples that are waiting to be processed by the current phase of the algorithm. The files R-output (S-output) contain tuples that have been passed over by the current phase of the algorithm. At the start of the algorithm, R-input and S-input are set to equal the relations R and S. R-output and S-output are initially empty.

A partitioning basis (split function) consisting of a number and range of of hash values is chosen at the start. There are as many iterations to the algorithm as there are partitions of relation R. The partitions of R are sequentially used to build hash tables in main memory. One hash table is built during each iteration. Each iteration begins with a scan of R-input. As each tuple is considered, if it belongs to the targeted memory partition R_i, then the tuple is added to the hash table. Otherwise, the tuple is written to R-output. R-output will contains all the remaining partitions that are not of current interest. Then S-input is scanned sequentially. If a tuple of S-input hashes to partition S_i then it is used to probe the hash table built from partition R_i. If a match is found, the tuples are joined and output. Otherwise (if the tuple does not belong to partition S_i) it is written to S-output.

At the end of each stage of the Simple hash-join, the R-output (S-output) file becomes the R-input (S-input) file that will be used by the next stage. As the algorithm progresses, the R-output (S-output) file becomes progressively smaller as the partitions of interest are consumed. The algorithm finishes when no tuples are passed over, that is, when R-output is empty following a processing stage.

4.3.2. Simple Toss Hash-Join

This algorithm is almost identical to the previously described Simple hash-join algorithm. The difference is that this algorithm does not write out the tuples that are passed over by each stage of processing. That is, there are no R-output or S-output files. At each stage of processing, the entire relations R and S are scanned as the tuples belonging to the current partitions R_i (S_i) are identified and used. At first glance this algorithm would appear to have performance characteristics that are markedly inferior to those of the Simple hash-join. The Simple hash-join scans progressively smaller input files as the algorithm proceeds while Simple Toss always scans both entire relations at each stage. It turns out, however, that these algorithms have very similar performance characteristics. This results from the fact that Simple hash-join incurs a substantial cost overhead when it writes out the pages of passed-over tuples. The cost of these extra writes almost completely counteracts the savings that are accrued during later scans.

4.3.3. Grace Hash-Join

The Grace hash join algorithm is characterized by a complete separation of the partitioning and joining phases. The partitioning of relations R and S is completed prior to the start of the join phase. Ordinarily, this partitioning phase need only create as many partitions from relation R such that each partition R_i can be used to build a hash table that will fit into memory. Since only a single page frame is needed as an output buffer for a partition, it is possible that memory pages will remain unused after the requisite number of partition buffers have been allocated.

In the Grace algorithm, these extra pages can be used to increase the number of partitions that are generated by the partitioning phase. Following the partitioning phase, these smaller partitions can be logically integrated into larger partitions that are of optimal size for building the in-memory hash tables. This strategy is termed 'bucket tuning' or partition tuning [KITS83]. Partition tuning is a useful method for avoiding partition overflow.

4.3.4. Hybrid Hash-Join

The Hybrid hash join was first described in [DEWI84]. All partitioning is finished in the first stage of the algorithm in a fashion similar to the Grace algorithm. However, whereas the Grace algorithm uses any additional available memory during the partitioning phase to split the relations into a smaller number of partitions, Hybrid uses additional memory to begin the joining process.

Hybrid creates the minimum number of partitions such that each partition can be reasonably expected to fit into memory. Allocating one page frame to be used as an output buffer for each partition, the Hybrid algorithm utilizes any remaining pages frames to build a hash table from a single partition of relation R. The split function creates N equal-sized partitions, $R_1 \ldots R_N$, that are written to disk and one independently sized partition, R_0, that is used to build the hash table. The same partitioning range is used for relation S. Tuples of S that hash into the partition range associated with the hash table are immediately used to probe the hash table for matches.

When the partitioning phase completes, the Hybrid hash join has already completed processing part of the work of the join phase. As a result of this immediate processing, the Hybrid hash join realizes a performance advantage based on the fact that the immediately processed tuples do not have to be written to and retrieved from the disk between the partitioning and joining phases. These savings become increasing important in environments with substantial amounts of main memory.

4.3.5. Hashed Loops Join

A modified version of the nested loops algorithm, termed Hashed Loops, was also implemented. The Hashed Loops algorithms is so named because it uses hashing as means of effecting the internal join of tuples in main memory. For each phase of the Hashed Loops algorithm, pages of the outer relation R are staged into memory and a hash table is constructed. Then the entire inner relation S is scanned and each tuple is used as a probe into the hash table. Constructing such a hash table avoids exhaustively scanning all of the R tuples in memory for each tuple in S as is done with the simpler form of the nested loops algorithm.

While both the hashed loops algorithm and the hash-partitioned algorithms use in-memory hash tables, there are significant differences between the algorithms. The hashed loops algorithm does not partition the source relations at all. This means that the entire inner relation must be scanned on each iteration of the algorithm.

4.3.6. Sort-Merge Join

The last algorithm, the sort merge join, employed the sort utility provided by WiSS. The implemented algorithm applied the WiSS sort routine to both source relations prior to the start of the join phase. The join algorithm found matching tuples via sequential scans of the sorted relations. An alternative, but not implemented algorithm, could incorporate a sort utility directly into the join algorithm. Such an algorithm could combine the merge phase of a sort utility with the search for matching tuples. That is, the various runs of the two source relations could be merged in synchrony and matching tuples could be simultaneously identified.

4.4. Performance Comparison

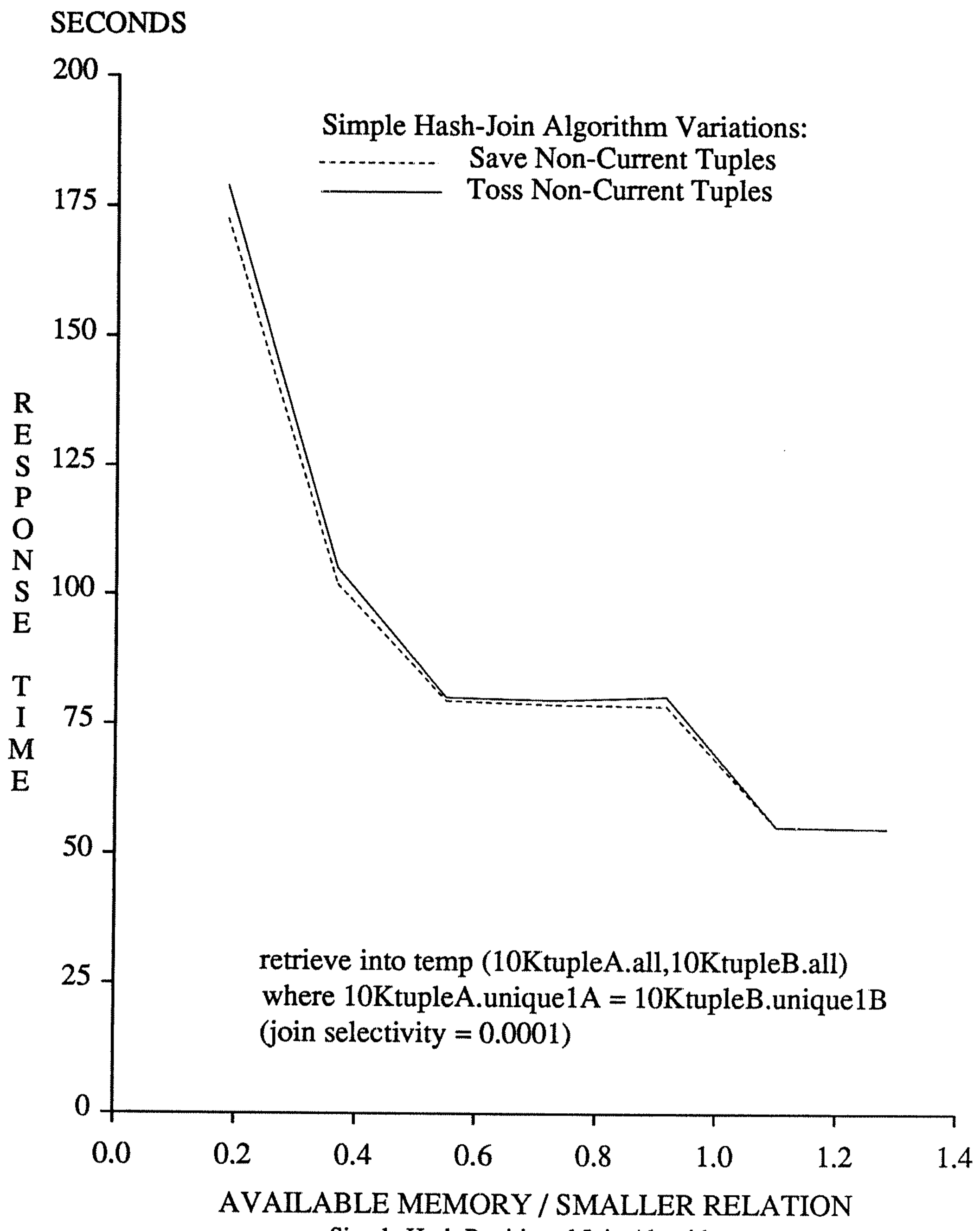
The join algorithms were compared using queries and data from the Wisconsin Benchmark Database [BITT83]. The comparison measures the performance of the algorithms on a join of two 10,000 tuple relations.

The join attributes are randomly ordered, two byte integers. 10,000 tuples are produced in the result relation. Every tuple from both relations participates in the result relation produced by this join query.

The response time of each join algorithm is illustrated as a function of the amount of available memory relative to the size of the 10,000 tuple relation. The elapsed times for all join algorithms include the time required to write the final result relation to disk. All tests were run in single user mode. The test machine had 8 megabytes of memory so no paging occurred. Bucket overflow did not occur in any of the tests of the hash-partitioned algorithms.

4.4.1. Comparison of Simple Hash-Join Algorithms

Prior to comparing performance of all the join algorithms, the performance of the two Simple hash-partitioned algorithms will be considered. These two algorithms are identical except for the manner in which they handle tuples of the remaining, unprocessed partitions. The response times for the Simple hash-join algorithms are shown in Figure 4.1.



Simple Hash-Partitioned Join Algorithms

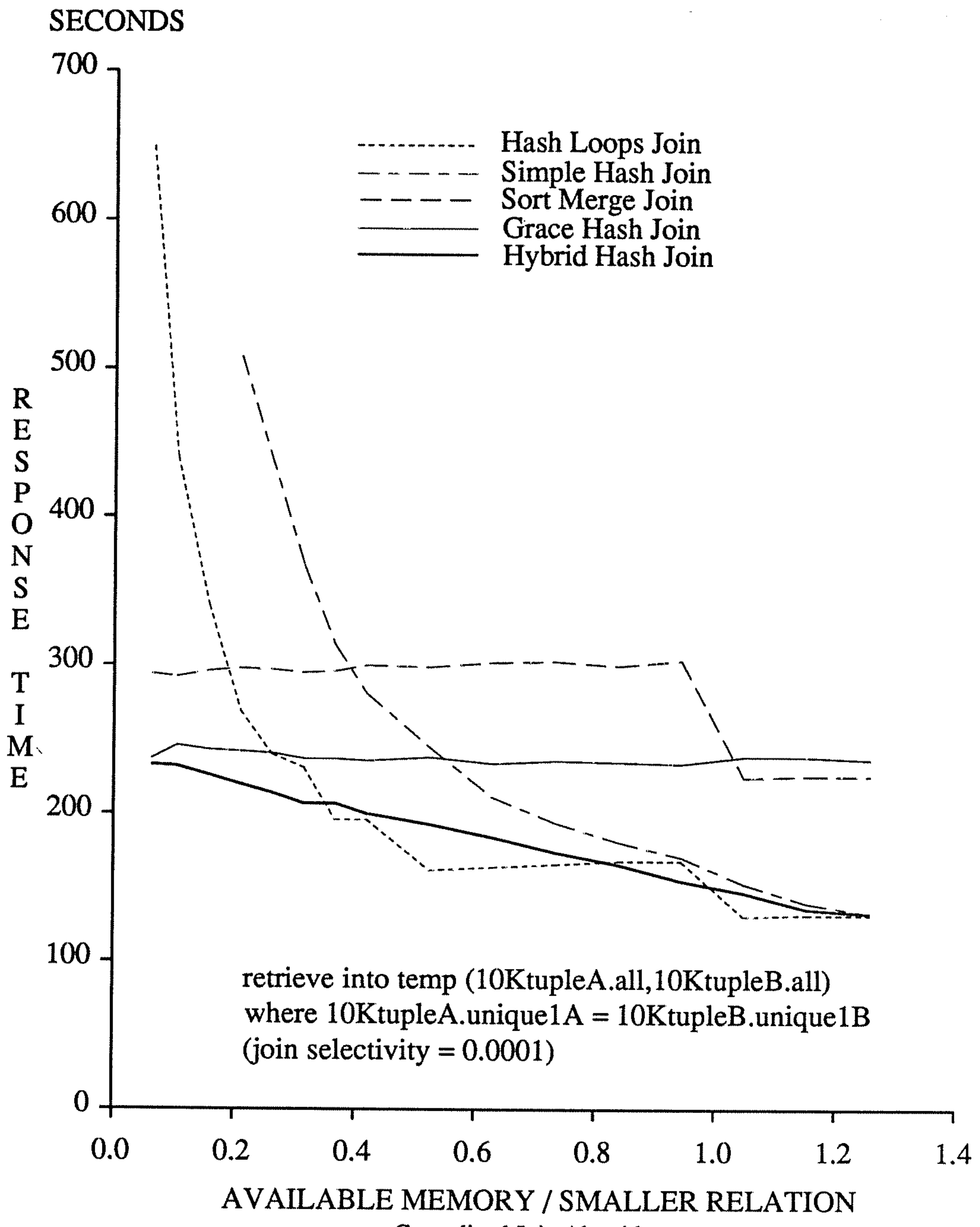
Figure 4.1

Both algorithms perform well only when memory is substantially large relative to relation R. The Simple Toss algorithm might have been expected to yield better, relative performance in view of its avoidance of repetitive, complete scans of the source relations. This potential advantage, however, is mitigated by the cost of writing the passed-over tuples to the temporary files R' and S'. For relative memory values between 0.5 and 1, the equivalent of 2 partitions are required for all the algorithms. In this range of relative memory, the simple hash-join algorithms require two passes to accomplish the join. Once relative memory exceeds a value of 1, the algorithms both process the join by building a memory resident hash table using the entire relation R. The discontinuity in the performance of the algorithms near this relative memory threshold reflects the fact that equal size partitions are being used. With a finer granularity between data points, this step-wise performance transition would be noticed whenever the amount of available memory enabled the algorithms to shift to the use of a smaller number of partitions. If necessary, these transitions could be smoothed by associating a unique size with the partition that is used to build the first hash table.

The Simple-Toss algorithm will not be used in subsequent performance comparisons with the other hash-join algorithms. Given the performance equivalence of the Simple and Simple-Toss algorithms, we will use only the Simple hash-join algorithm in these comparisons.

4.4.2. Response Time Measurements

An earlier study of the hash-partitioned algorithms [DEWI84a] presented performance results that were calculated from analytical models. These results were illustrated by Figure 3.8. The measured performance of actual implementations of the algorithms for a similar query is presented in Figure 4.2. The similarity of these figures is reassuring.



Centralized Join Algorithms
Figure 4.2

In Figure 4.2, the performance of the Grace hash-join algorithm is constant for the given range of available memory. This results from the total separation of the partitioning and join phases in the Grace algorithm. From a performance viewpoint, the Grace algorithm only uses memory optimally during the joining phases. Excess memory during the partitioning phase is used as a means of creating a large number of partitions for the partition tuning process. In contrast, the performance of the Simple hash-join algorithm is significantly affected by the amount of available memory and performs well only when the smaller relation is less than twice the size of available memory.

The performance of the Hybrid algorithm reflects the fact that it combines the best performance features of the Grace and Simple[8] hash-join algorithms. Since Hybrid completes all partitioning in a single pass through both source relations, it's performance is always as least as good as that of the Grace algorithm. The Hybrid algorithm increasingly outperforms Grace as the amount of relative memory increases because the additional memory is used for immediately joining tuples from one partition of each source relation. Such immediate joining eliminates the cost of writing and reading these tuples to disk between the partitioning and joining phases.

The performance of all of the hash-partitioned algorithms remains unchanged once the smaller relation fits in memory. This point occurs at a relative memory value of 1.2 and not when available memory exactly equals the size of the smaller relation. This results from the fact that the hash-join algorithms use some of the available memory during the join phase for the structure of the hash table itself. Also, it must be realized that partitioning is a predictive process and as such, prudence requires that additional memory be used to accommodate fluctuations in the size of the hash tables that are constructed from partitions. In these tests, partitions were constructed with an allowance for a 20% fluctuation in their predicted size.

The performance of the Sort-Merge join algorithm is constant over a wide range of available memory in Figure 4.2. Until a source relation fits into memory, the sorting process completely reads and writes the relation at least twice, once when the sorted runs are produced and a second time when the sorted runs are merged. The Sort-Merge join algorithm then reads the source relations a third time to effect the final joining of tuples. As mentioned earlier, an optimization to the sort-merge algorithm is possible. Given sufficient memory, the sorted runs of both

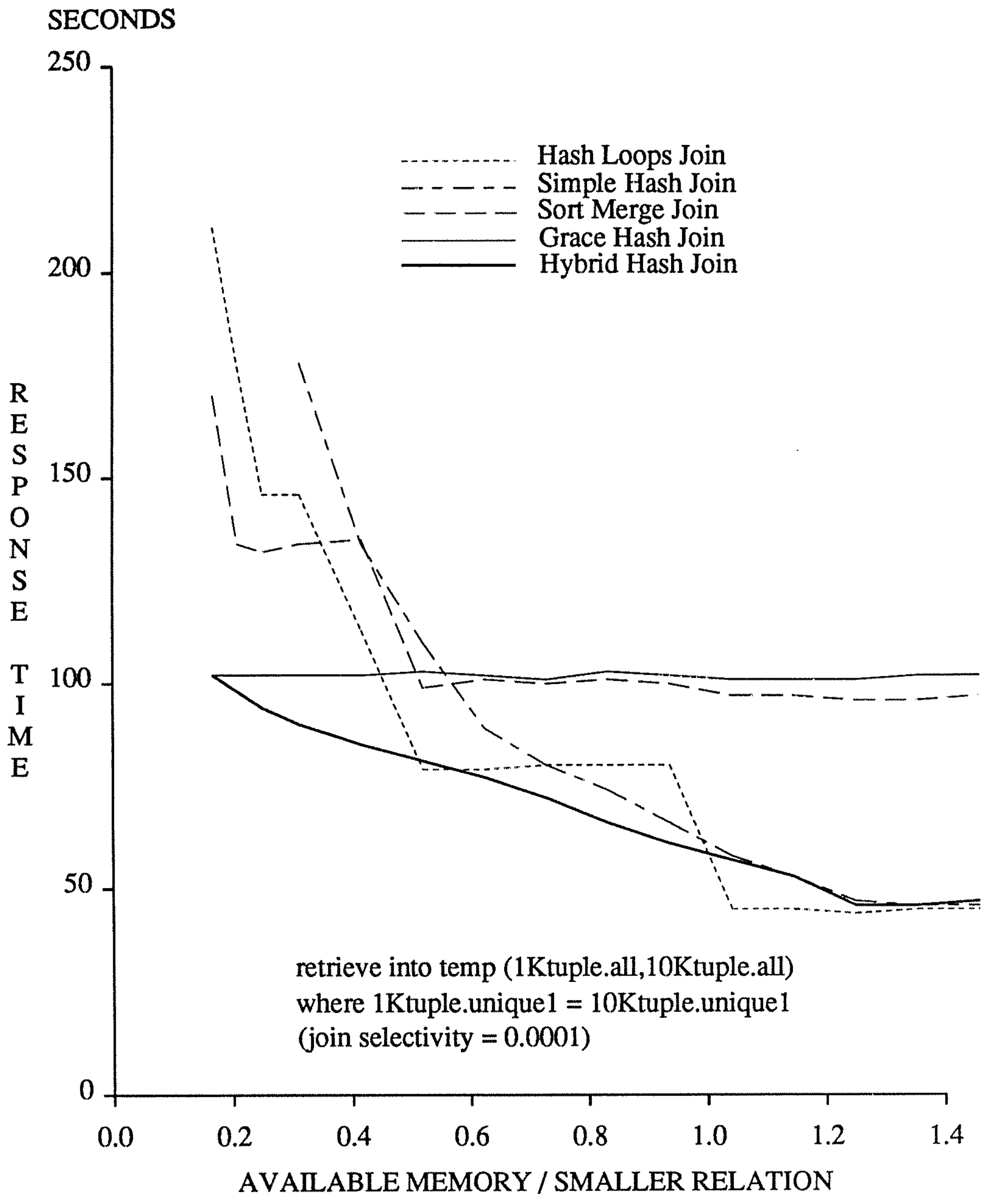
[8] The Simple hash-join algorithm should have performance equal to that of the Hybrid algorithm for relative memory values in excess of approximately 0.5. The difference in the performance of the Hybrid and Simple hash-join algorithms for relative memory values between 0.5 and 1.2 is an artifact of the implementation.

relations can be merged and joined simultaneously. In this case, the performance of the Sort-Merge algorithm could be expected to be similar to the Grace algorithm as each algorithm would access every page of each source relation three times (two reads and a write).

Perhaps, surprisingly, the Hashed Loops algorithm has quite good performance over a wide range of available memory in Figure 4.2. Due to the existence of the hash table, the cost of of probing for matches with tuples from relation S is a relatively inexpensive operation. The algorithm performs especially well when the size of the smaller relation is less than twice the size of available memory. As this is exactly the situation one would expect in the case of partition overflow, the hash-based nested loops algorithm is an attractive strategy for handling partition overflow. For example, when available memory is half the size of the 10,000 tuple relation, the Hashed Loops algorithm scans the outer relation once and the inner relation twice. By comparison, the Hybrid hash-join reads each source relation only once. However, Hybrid writes out and later reads in partitions[9] containing at least half the tuples from each source relation. Therefore, the Hybrid algorithm incurs total I/O costs equivalent to at least four scans of a 10,000 tuple relation while the Hashed Loops algorithm incurs I/O costs equivalent to only three scans of a 10,000 tuple relation.

The performance of the join algorithms for a join of a 1,000 tuple relation with a 10,000 tuple relation is shown in Figure 4.3.

[9]Two partitions are created for later processing. A third partition is processed immediately by the Hybrid algorithm. While only two partitions would be sufficient if tuples were uniformly distributed among partitions, as previously mentioned, the partitions for the current test have been created with 20% additional space to handle variations in the distribution of tuples to partitions.

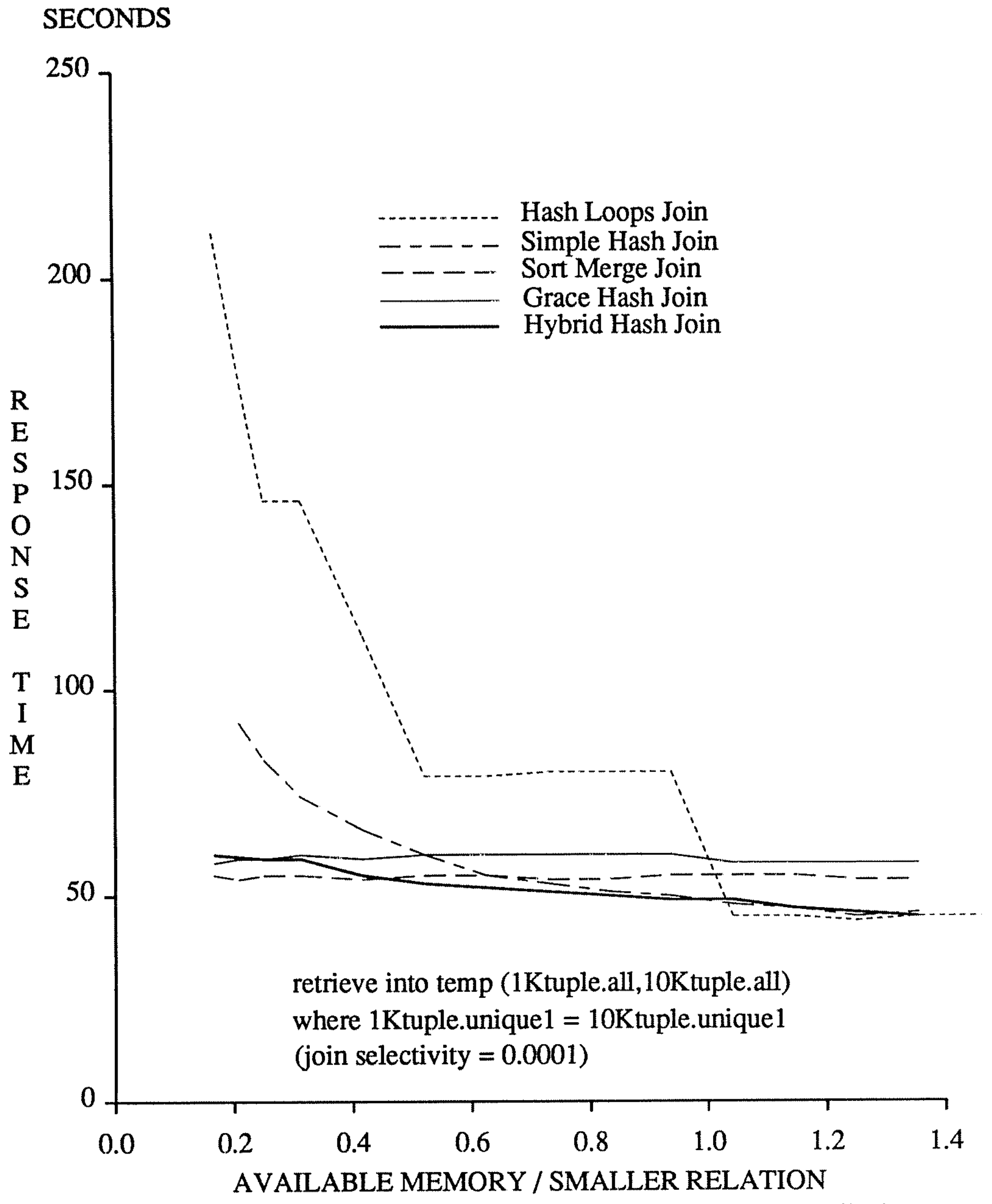


Join Algorithms With Unequal Sized Operands
Figure 4.3

Each tuple from the smaller relation participates in the join. However, only 10% of the tuples from the larger relation contribute to the result relation. The result relation contains 1,000 tuples. The Hybrid algorithm continues to dominate all the other join algorithms over a wide range of relative memory values.

Bit Vector Filtering

Figure 4.4 reflects the performance of the join algorithms on the same query used for Figure 4.3 which joins a 1,000 tuple relation with a 10,000 tuple relation.



Join Algorithms With Unequal Sized Operands & Bit Vector Filtering
Figure 4.4

The difference is that in Figure 4.4 all the algorithms use bit vector filtering techniques [BABB79, BRAT84, VALD84]. The notable performance improvements demonstrated are the result of eliminating, at an earlier stage of processing, those tuples that will not produce any result tuples. The bit vector filtering technique used by the hash-partitioning and Sort-Merge algorithms are very similar.[10]

Prior to the initial scan of relation R, a bit vector is initialized by setting all bits to 0. As each R tuple's join attribute is hashed, the hashed value is used to set a bit in the bit vector. Then as relation S is scanned, the appropriate bit in the bit vector is checked. If the bit is not set, then the tuple from S can be safely discarded. Applying the bit vector from relation R against relation S approximates a semijoin of relation S by relation R.

The net impact of this process depends on the semijoin selectivity factor of relation S by R which is defined to be the ratio of tuples resulting from the semijoin of S by R relative to the cardinality of S. In the example query of Figure 4.4, the semijoin of relation S by R has a semijoin selectivity factor of 0.1. The net effect is that approximately 90% of the tuples of relation S can be eliminated at a very early stage of processing by the hash-partitioned and Sort-Merge algorithms. Significant I/O savings accrue from the fact that these non-participating tuples do not have to be stored on disk between the partitioning and joining phases of the hash-partitioning algorithms. Two disk accesses are saved for every page of tuples that can be eliminated by the bit vector filtering of relation S. Also, it is significant that the bit vector filter is constructed from the smaller source relation and applied against the larger source relation (the probing relation).

Since, the Hashed Loops algorithm does not complete a scan of relation R until the end of the query, it must instead use bit vector filtering to approximate a semijoin of relation R by S. In Figure 4.4 the semijoin selectivity factor for a semijoin of R by S is 1.0. Therefore, in this instance, the Hashed Loops algorithm doesn't derive any benefit from applying bit vector filtering. Since the semijoin selectivity of S by R is 0.1 in the tested query, the performance of the Hashed Loops join could have been improved by reversing the roles of the inner and outer relations. However, in general, such semijoin selectivity factors will not be known with sufficient certainty to justify the reversal of the roles of the inner and outer relations to a Hashed Loops join.

[10] The bit vector filtering technique used by the hash-partitioned and Sort-Merge algorithms is directly extendible to the case of Hashed Loops if the names of the relations in the discussion are reversed.

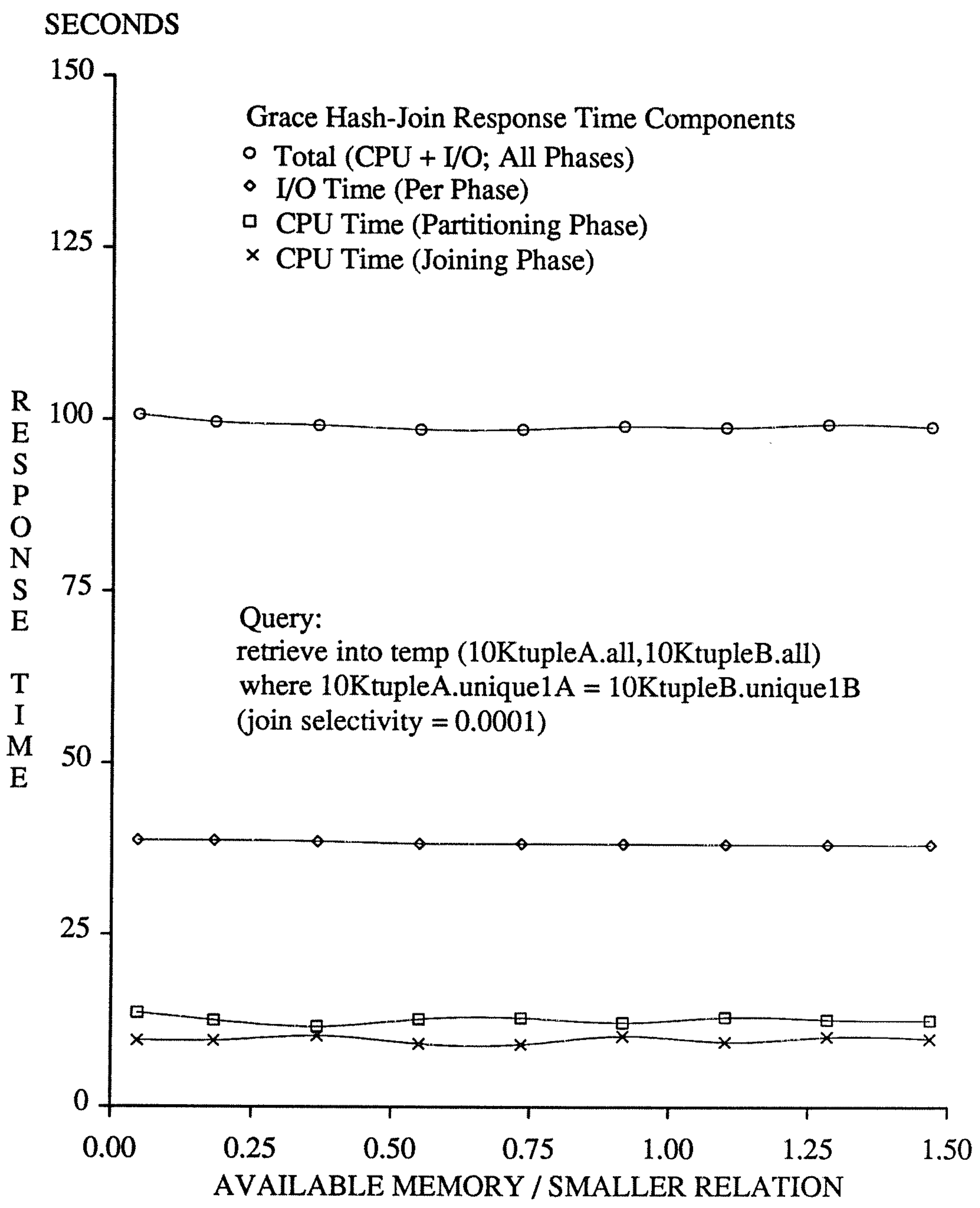
Collisions that occur in the process of accessing bit vectors may result in non-qualified (phantom) tuples being propagated along to the final joining process. The phantom tuples will, however, be eliminated by the final joining process. The number of phantom tuples can be reduced by increasing the size of the bit vector or by splitting the vector into a number of smaller vectors [BABB79]. A separate hash function would be associated with each of the smaller bit vectors. The costs associated with bit vector filtering are modest. For the given test, a single bit vector of length 4K bytes was used. Since the hash-partitioning algorithms already compute the hashed value of each tuple's join attribute, the only additional cost of bit vector filtering for these algorithms is the amount of space required for the bit vector itself.

4.5. Performance Analysis

In this section, we consider the various resource utilizations that accompanied the performance tests of the HP-join algorithms.

4.5.1. Resource Utilizations

The uniformity of the performance of the centralized Grace join algorithm is reflected in the CPU and I/O usages of the algorithm as reflected in Figure 4.5. The CPU usage was obtained through the monitoring facilities provided by UNIX (getrusage). The I/O costs were indirectly calculated by counting the number of disk accesses and multiplying by the estimated, average cost for a sequential disk access (21 milliseconds [FUJI82]).



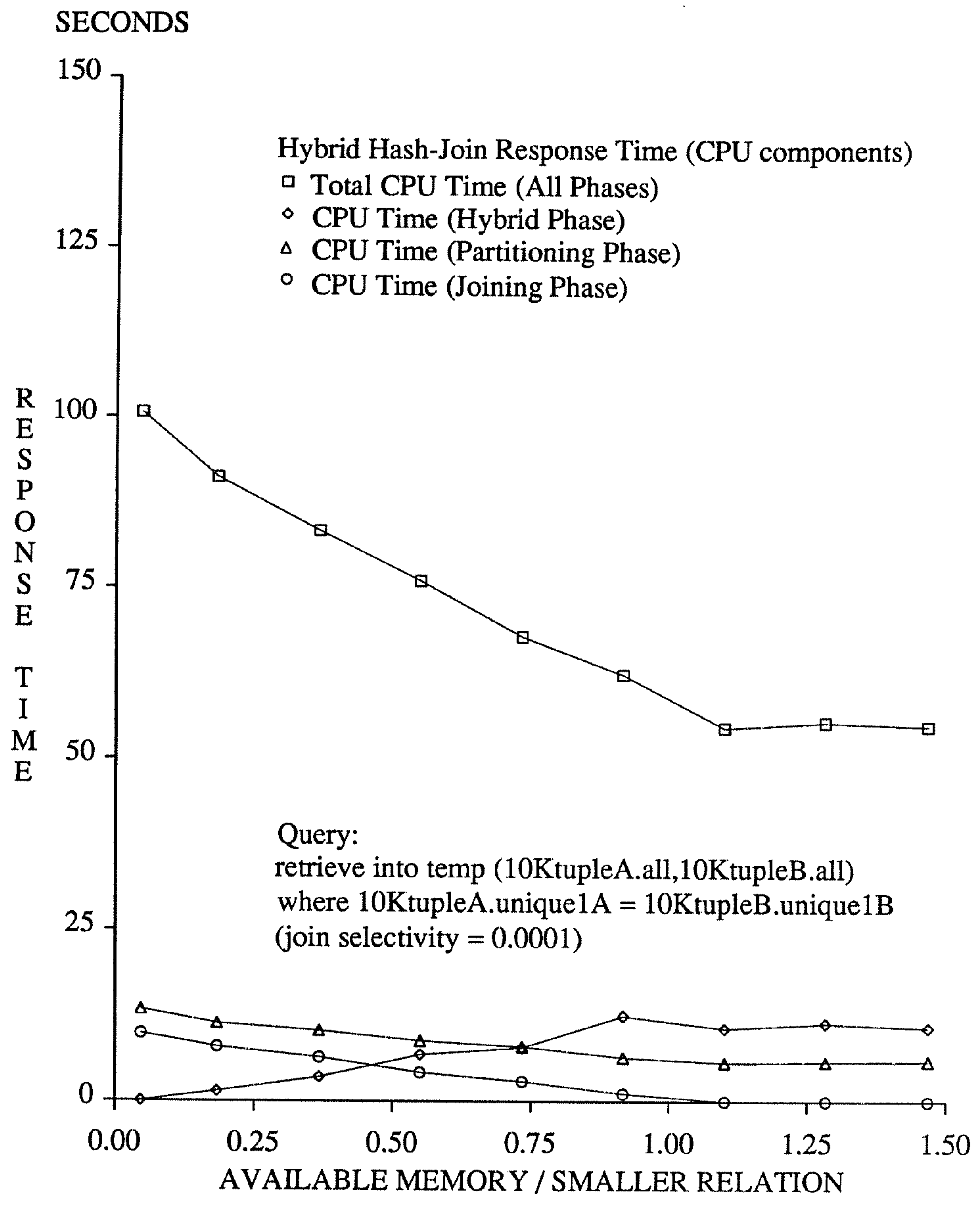
Grace Hash-Join Response Time Components
Figure 4.5

For the Grace algorithm, the I/O costs of the partitioning and join phases are identical and the CPU costs of the two phases are very similar. This similarity is partially due to the fact that the algorithm reads all tuples of both source relations during the partitioning and joining phases. However, the exact equivalence of the resource usages of the two phases is due to the fact that for this particular query there is a one to one correspondence between the number of result pages that are written during the joining phase and the number of partitioned pages that were written during the partitioning phase. That is, all source tuples are represented exactly once in the result relation.

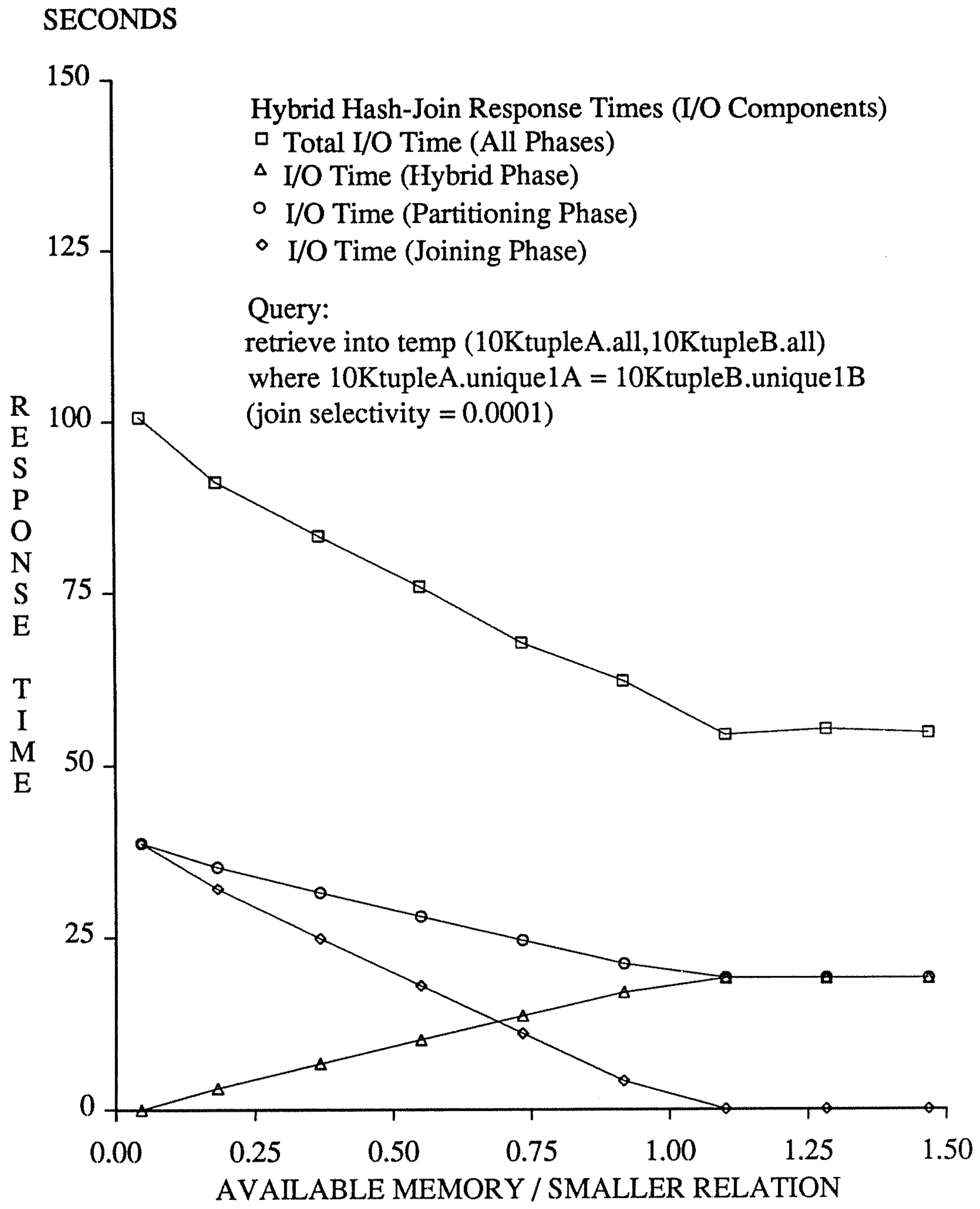
Two other observations about the resource utilizations of the Grace algorithm are more directly a consequence of the characteristics of the algorithm itself. First, the resource utilizations of Grace are demonstrated to be entirely independent of the memory that is available in the system. This results from the fact that the algorithm is not designed to optimize it's performance in response to increased memory availability. Instead, the algorithm uses additional memory as a means of reducing the likelihood of an occurrence of partition overflow. Such preventive measures will not enhance the performance characteristics of the join operation when partition overflow does not occur. Since this particular query did not measure the response of the algorithms to partition overflow, the benefit of the algorithm's preventive mechanism is not reflected. The costs of the prevention are, however, demonstrated by the lack of improvement in response time as increased amounts of memory are made available. Figure 4.5 also illustrates the relative dominance of the I/O costs for the Grace algorithm. The measured I/O costs accounted for 79% of the response time of the Grace algorithm.

Resource cost assignment for the Hybrid HP-join algorithm is complicated by the fact that some of the joining of tuples is done simultaneously with the partitioning phase. Therefore, I/O and CPU costs are identified separately for three phases of the algorithm: partitioning, hybrid and joining. The hybrid phase is invented here only as a convenient entity for purposes of identifying the costs associated with the immediate joining of tuples during the partitioning phase. Specifically, the hybrid phase is assigned the costs of building and probing the hybrid hash table and the cost of writing out tuples of the result relation. The partitioning and joining phases are assigned costs in a manner similar to that used for the Grace algorithm.

For clarity, the CPU and I/O utilizations for the Hybrid algorithm have been separated into two separate illustrations. Figure 4.6 shows the CPU utilizations of the Hybrid algorithm and Figure 4.7 illustrates the corresponding I/O utilizations.



Hybrid Hash-Join Response Times (CPU Components)
Figure 4.6



Hybrid Hash-Join Response Times (I/O Components)
Figure 4.7

At the low end of the relative memory spectrum, the hybrid phase doesn't exist. At that point, all available memory is used for partitioning. That is, when available memory is scarce, the algorithm does not assign a partition for immediate processing. Under those conditions, the Hybrid algorithm performs identically to the Grace algorithm.

As the relative available memory increases, one partition of tuples is assigned for immediate joining. The size of this **hybrid** partition is expanded in response to the increasingly available main memory. Correspondingly, the cost of joining is increasingly shifted from the join phase to the hybrid phase. Finally, once memory is large enough to contain a hash table built from all of relation R, all joining costs have been shifted to the hybrid phase.

As was the case for Grace, the response times of the Hybrid algorithm are dominated by I/O costs. I/O costs as a percent of total response time decrease from a high of 79% to a low of 72% as the relative memory values increase.

Given the fact that the response times of the Grace and Hybrid algorithms are dominated by I/O costs, it is constructive to consider alternative means for handling the I/O requirements of the algorithms. All the hash-partitioned algorithms share a very regular pattern of disk accesses. Relations and partitions are always, predictably scanned sequentially in their entirety. The regularity of these disk requests will probably be recognized by most file systems and read-ahead will be profitably employed. The response times of the algorithms will decrease accordingly. Another, more important performance enhancement, is also possible in some processing environments. Disk requests can be issued for data in blocks that are larger than a page. As the size of the data block is increased, the costs associated with the disk access are amortized over a larger amount of data [HAGM86]. The net result is a lower aggregate I/O cost. For I/O bound algorithms such as Grace and Hybrid, this decrease in I/O costs would have a significant effect on the overall response time.

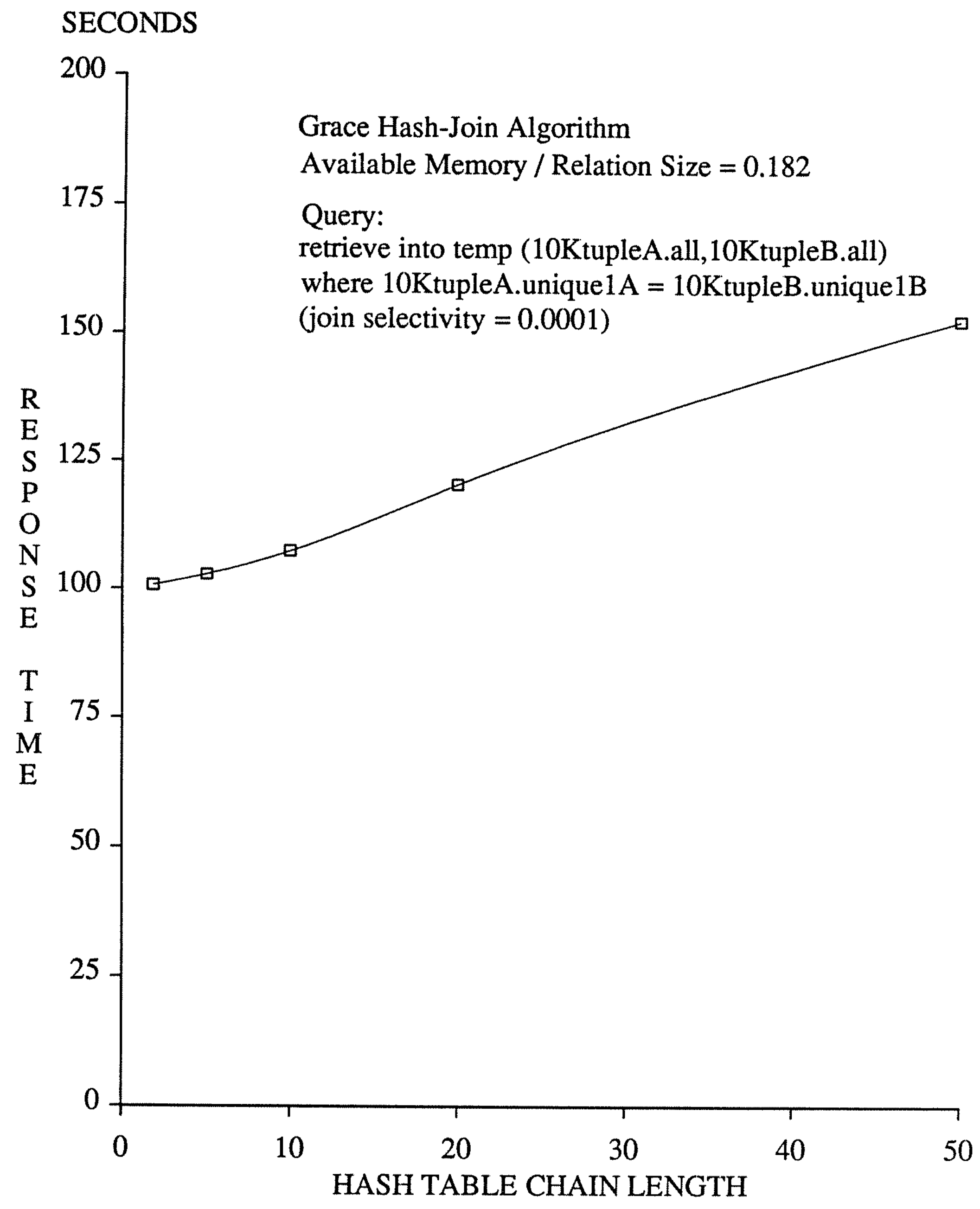
Consider the effect upon the Grace algorithm of an increased blocking factor for disk requests. In processing the join of 10ktup_R.unique1 by 10ktup_S.unique2, Grace reads and writes a total of 3688 pages. Pages are 4 Kbytes long. With an I/O cost of 21 milliseconds [FUJI82] per disk request[11], the Grace algorithm incurs an I/O cost of 77.448 seconds compared to a total CPU cost of 23.28 seconds. Assuming that a track of data contains 28

[11] The hash-partitioned algorithms read and write sequential data files.

Kbytes, the cost associated with reading a track would equal the previously defined time to read a page plus the time to transfer the additional 24 Kbytes. At at transfer rate of 1.8 Mbytes per second [FUJI82], approximately 13 msec of time are required for the additional transfer of 24 Kbytes. The cost of an I/O operation of the size of a track is then approximately 34 msec. Using track-at-a-time I/O, Grace would incur a total I/O cost of approximately 18 seconds. Compared against a CPU time of 23 seconds, Grace now is CPU bound instead of I/O bound.

4.5.2. Hash Table Collisions

The performance of the HP-join algorithms is also dependent on the internal mechanics of the hash function and table. Hashing-algorithms are potentially vulnerable to degradations in performance resulting from collisions in the hash table. The severity of collisions in the hash table can be reduced by increasing the dimensions of the table, by selecting an appropriate hash function and by using different bits of a tuple's hashed value that were used by the partition's split function. The magnitude of the performance effects of hash table collisions is illustrated in Figure 4.8.



Performance Impact of Hash Table Collisions
Figure 4.8

This figure correlates increased response times with changes in the length of the average chain of a hash table. For the purposes of this test, the hash function was altered to produce a controlled level of collisions in the hash table. Increased levels of hash table collisions produce a linear increase in the response time of the algorithm. The performance cost of searching longer hash chains for matching tuples is significant, but the net effect indicates that moderate levels of hash table collisions can be accommodated without severely impacting the performance of the algorithms.

CHAPTER 5

MULTIPROCESSOR HASH-PARTITIONED ALGORITHMS

Multiprocessor versions of the Hybrid and Grace algorithms are attractive for a number of reasons. First, the ability of these algorithms to cluster related tuples together in buckets provides a natural opportunity for applying parallelism. In addition, the number of partitions produced during the partitioning phase (or activated in the building and probing phases) of each algorithm can be adjusted to produce a variable level of parallelism during the joining phase. Second, the independent nature of partitions can serve to minimize communications overhead between processors executing the same operation. Furthermore, just as the centralized form of the Hybrid algorithm made very effective use of main memory in order to minimize disk traffic, one would expect that a multiprocessor version of the Hybrid hash join algorithm should be able to use memory on multiple processors to minimize disk traffic. Finally, and, perhaps, most importantly, it appears that control of these algorithms can be decentralized in a straightforward manner.

Prior to considering the specific properties of the algorithms, a specific hardware environment will be adopted. Next, a horizontal partitioning design for the physical database will be examined in the context of the selected architecture and hash-partitioned algorithms. Finally, having defined architectural and storage constraints, the properties of the multiprocessor hash-partitioned algorithms will be examined.

5.1. Design Constraints

This section will examine the design of a hardware architecture and a physical database that complement the application of the hash-partitioned algorithms.

5.1.1. Hardware Considerations

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], two strategies were suggested 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 5.1, 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[12] 100 megabytes/second. If one believes the fabled "90-10" rule, most of the data moved is not needed in the first place.

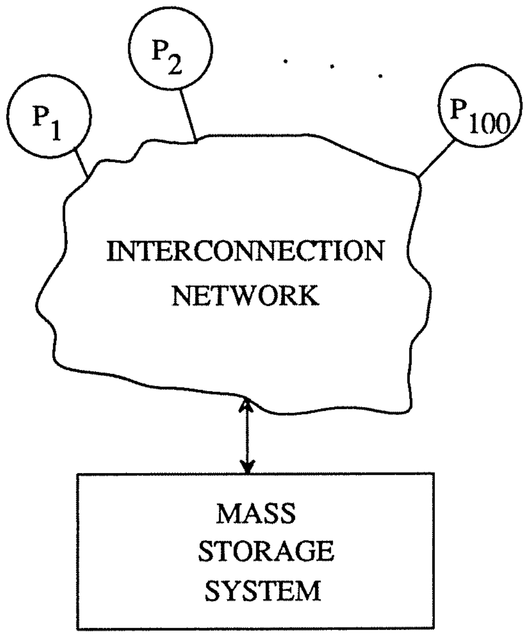


Figure 5.1

[12] 100 megabytes/second are needed to handle the disk traffic. Additional bandwidth would be needed to handle processor to processor communications.

Figure 5.2 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 each) the I/O bandwidth of the two alternatives will be equivalent.

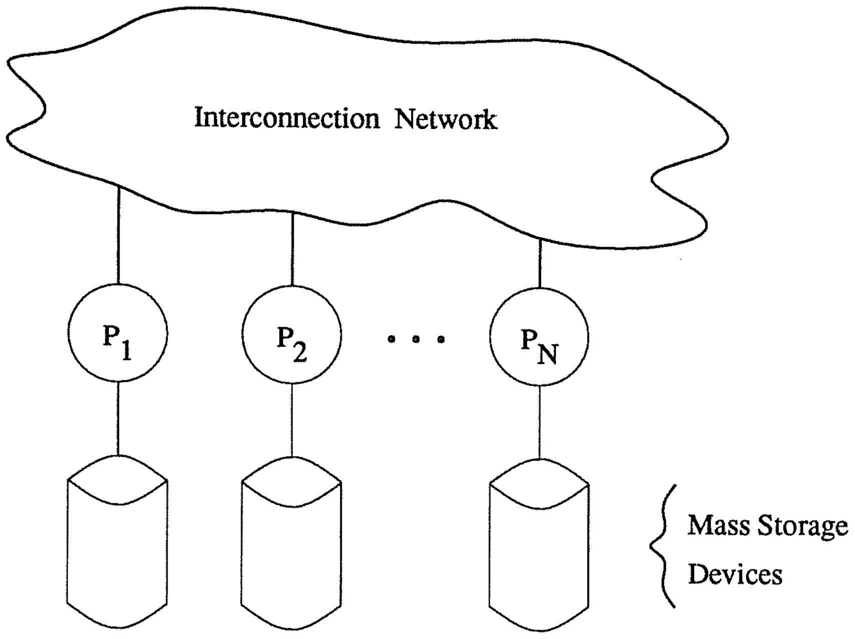


Figure 5.2

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 [DEMU86]. Finally, the design may simplify exploiting improvements in disk technology.

A similar alternative 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. While each of these alternatives is attractive, our feeling is that it is too early to say exactly what form the interconnection network between the processors should take[13]. The answer may very well depend on the nature of the algorithms employed to execute complex database operations. Based on the preliminary results contained in [DEWI85], it is not at all clear that an interconnection network as complicated as the Y-net [TERA83] is justified.

The architecture of Figure 5.2 addresses the problem of the I/O bottleneck that can exist between the storage system and the processing elements of a database machine. The architecture, however, can be further enhanced by the addition of processors without disks. Since complex relational operations such as the join operation do not access base relations, these operations can potentially be supported on processors without attached disks. As processors without disks (or disk controllers) represent a "cheaper" resource than processors with attached disks, their effective use can potentially improve the cost/performance characteristics of a database machine. That is, an architecture containing such processors can increase the level of parallelism that can be applied to complex relational operations in a cost effective manner.

5.1.2. Physical Database Design

In this section, we examine possible strategies for designing a physical database that complements the use of hash-partitioning in a multiprocessor environment. The impact that various types of horizontal fragmentation of a physical database have on the performance of a multiprocessor, hash-partitioned database machine are considered.

Horizontal Partitioning Rationale

The hash-partitioning algorithms are dependent on the production and use of discrete horizontal partitions of tuples. A natural extension of these algorithms is the use of a physical database that is horizontally partitioned across a number of physical disks. In this manner, parallelism can effectively be applied both during the scanning and storing of permanent relations. While permanent relations can potentially be partitioned across only a subset of the disks in the system, the following discussion will assume that all relations are horizontally partitioned [RIES78, EPST78, STON79] across all disk drives in the system. This assumption provides a maximum level of potential parallelism while simplying the physical database design. From the view point of raw bandwidth, this approach has

[13] Perhaps it should be shared memory.

the same aggregate bandwidth as the disk striping strategies [GARC84, KIM85, BROW85] given an equal number of disk drives.

Defining Horizontal Partitions

There are at least two obvious strategies for distributing tuples across the disks drives in the system. One approach is to apply a hash function to each tuple (or the key attribute of the tuple) to select a disk for storing the tuple. The advantage of this approach is that as additions are made to a relation file, the number of tuples on each disk should remain relatively well balanced. A second partitioning mechanism clusters tuples by ranges of key value at each disk in the system.

Since each of these horizontal partitioning policies produce partitions of a relation that are disjoint, each of the processors can maintain independent, local indices on the tuples stored on its disk. In the case of relations partitioned on ranges of attribute values, the disks, and their associated processors, can be viewed as nodes in a primary, clustered index. A controlling processor can act, in effect, as the root page of the index. This approach is similar to, but much simpler than, the clustering approach employed by MDBS [HE83]. In MDBS [HE83], each backend processor must examine every query as the clustering mechanism is implemented by the backends, not the controlling processor.

The real advantage of the range partitioning approach comes when processing queries. With the hashed strategy (ie. tuples distributed randomly), except in the case of exact match queries on the attribute used to distribute tuples, all processors must execute every query. (This is what occurs in the MBDS design.) With the second distribution strategy, the controlling processor (which maintains the root page of each index) can direct each query to the appropriate processors. While overhead is incurred in performing this function, it is certainly less than the cost of sending the query to all the processors. Furthermore, even for a fairly large database, the root pages of all indices should fit in the controlling processor's main memory. While the two distribution strategies should provide approximately the same response time in single user benchmarks, the system throughput might reasonably be expected to be significantly higher with the second distribution strategy in a multiuser environment. When no suitable index is available, all processors are still available to perform the query.

A third strategy for horizontally partitioning a relation distributes tuples uniformly among all partitions. This uniform distribution can be achieved by a round-robin partitioning strategy that assigns consecutive tuples to different partitions. Alternately, fixed numbers of consecutive tuples can be assigned to the same partition before assigning tuples to the next partition.

5.2. Distribution Requirements of the Hash-Partitioned Algorithms

As in the centralized algorithms, distinct partitioning, building and probing phases are used by multiprocessor hash-partitioned algorithms. However, in a multiprocessor environment with horizontally partitioned relations, each of these phases for a single operator may be active at multiple sites in the system.

5.2.1. Multiprocessor Partitioning Phase

In a centralized environment, the creation and management of partitions is simple. A local relation is read from secondary storage and multiple, local partitioned files are written back to a disk. Later, these partitioned files are sequentially accessed and processed. In contrast, creating partitions in a multiprocessor system implies that multiple split processes[14] on distinct processors will be contributing tuples to the same partition.

Placement of Split Processes

The partitioning phase will be active at all sites where source data exists. In the case where the source data is a permanent relation, the partitioning phase will be active only on those processors with attached disks. Having assumed that all relations are partitioned across all disks in the system, then the partitioning phase may be active on all of these processors. However, if the split attribute is also the horizontally partitioning attribute, then it may be possible to activate the partitioning phase only on those sites having qualified tuples. In this case, the schema information associated with a horizontally partitioned attribute can be used to determine the appropriate sites.

A partitioning phase may also be active on processors without disks when the result relation produced by a join operation is partitioned.

Fragmented Production of Horizontal Partitions

As a consequence of the distributed nature of the assumed architecture, partitions of tuples will be created in fragmented form. Therefore, before a partition can be processed, all fragments of the partition have to be collected

[14] An instantiation of a partitioning phase will frequently be referred to as a split process as these processes split a stream of tuples into discrete partitions.

together at a single site. The responsibility for implementing this collection of partition fragments can potentially be implemented by the partitioning phase. This requires a consensus among the distributed split processes as to which destination sites are associated with each partition. Then, as partitions are produced, the tuples can be sent to specific destination sites for storage or immediate processing.

The number of pipelines that exist between two consecutive operators in a query tree is a product of the number of processes that represent the consuming and producing operators. That is, each of the producing operator processes will be splitting it's fragment of a result relation among a number of partitions equal to the number of consuming processes. If the benefits of pipelined data flows are to be maintained as the level of intra-query concurrency is increased, then the mechanism for establishing and controlling the tuple pipelines should be inexpensive and simple. Otherwise, the cost associated with expanding the matrix of pipelines between producers and consumers could very well negate the benefit of the increased parallelism. In particular, the routing tables associated with these pipelines should be either static or the subject of very infrequent updates. In this manner, the number of control messages that are necessary to establish these pipelines can be kept to a low and perhaps fixed number of messages. If static routing tables are used, a single control message can potentially provide this routing information.

5.2.2. Multiprocessor Building Phase

The mechanics of a multiprocessor building phase can be very similar to those of a centralized algorithms. In both environments, the requirements of the building phase are the same. Tuples from a distinct partition are collected into local memory on a single processor and a hash table is built. However, process synchronization and resource allocation become more complicated in a distributed environment.

Synchronizing the Building Phase

A number of process activation and synchronization issues impact the design of the build phase of a hash-partitioned algorithm. First, the processes that are assigned to the building of hash tables, i.e. the building processes, must be coordinated with the processes that produce the source relations. From the perspective of a join operation, the operators producing source tuples represent the partitioning phase of a hash-partitioned algorithm, i.e. the split processes. Depending on whether data is pushed or pulled across the network, the building processes will have to be aware of all the instances of the splitting processes or vice versa.

This routing awareness will impact the order in which the operators are activated. That is, whichever set of operator processes is passive in the transfer of partitions will most likely have to be the first operation to be activated. In this manner, the network addresses of these passive operator processes may be established and provided to those processes that will actively manage the transfer of partitions across the network.

Also, since the hash-partitioned algorithms have two distinct joining phases, it will also be necessary to mediate the transition of the distributed representatives of a join operation from the building to the joining phase. This is required because each of these phases must be coordinated with the production of different source relations, i.e. the building and probing source relations. The times at which particular join processes complete the construction of hash tables and are prepared to accept tuples from the probing relation may vary. Also, variations in the time of these transitions can be accentuated by local occurrences of partition overflow.

A multiprocessor hash-partitioned algorithm also provides opportunities for controlling the level of intra-query concurrency that is utilized. By controlling the number of build processes that are activated in the system, the level of parallelism that is applied to any particular operator can be controlled. One of the factors that must be considered when choosing an appropriate level of intra-query concurrency is the availability of resources on the various candidate processors. One of the most critical resources in this regard for the hash-partitioned algorithms is the availability of local memory on each of the processors in the system.

Memory Availability

The performance of the hash partitioned algorithms can be enhanced by the availability of additional increments of memory. The following discussion considers the question of how the available memory on the various query processors should be allocated to the query processes. The predominant memory requirement for query processes occurs when hash tables are built.

Query processes require a minimum guaranteed amount of memory to ensure that progress can be made on a complex query operation. This requirement is reinforced by the the dependency of an applied split function on the memory available to a HP-join operator process. That is, the number of partitions that will be created depends directly upon the memory which is expected to be available to each operator process. These issues reflect the need for a somewhat static approach to memory allocation.

However, a system with a completely static memory allocation policy would be unable to respond to variations in the memory requirements of query processes at runtime. For example, while one query process might exhaust it's assigned memory allocation, another process could very well be underutilizing it's allocation. In fact, in an unloaded system, many processes would not be using their static memory allocations at all. In such a system, it appears that significant amounts of memory could be idle at any given time. With a static memory allocation policy, it would be difficult to create a system that supported concurrent queries and also provided optimal response times for individual queries. That is, such a system could not easily focus it's memory resources on a single query.

In contrast, a dynamic memory allocation policy is more compatible with the runtime characteristics of hash-partitioned operations. Hash-tables may grow in unpredictable ways. This is especially true for those hash tables which are built in the interior of a query tree. The query processes building these hash tables should, therefore, be able to allocate additional amounts of memory at runtime.

The following proposal suggests a memory allocation policy that has both static and dynamic elements. Each query process is guaranteed an initial, minimum amount of memory. When necessary, additional amounts of memory can be requested. However, there is no requirement that such requests be granted.

This memory allocation policy provides a basis for mapping operators to processors with sufficient available memory. The initial static allocation of memory would enable the design of a split function that would produce partitions of the desired size. The dynamic ability of a processor to request additional increments of memory would also provide support for unanticipated variations in the memory requirements of operator processes at runtime.

5.2.3. Multiprocessor Probing Phase

During the probing phase, a join process collects tuples from a partition of the probing relation corresponding to the partition of the building relation that is represented in a current, local hash table. As in the centralized algorithms, each tuple in the probing partition is matched against the existing tuples of a hash table.

Distributing this phase of a HP-join operation encounters many of the synchronization and control issues previously described in the context of the building phase. As in the building phase, fragments of a partition will be collected from multiple sources. However, this phase is somewhat simpler to distribute due to the fact that the memory requirements of the HP-join operation are predominantly limited to the building phase. That is, once hash tables have been constructed, the HP-join algorithms consume very little additional memory.

5.3. Dimensions of Potential Parallelism

Significant opportunities for applying parallelism exist in a multiprocessor, hash-partitioned database machine. Subsequent sections will consider the nature and control of this parallelism. Before considering these interactions, however, it is useful to consider the sources of parallelism in such a system.

Within the context of a given query, two forms of intra-query concurrency are possible. Multiple operators of a query may be executed in parallel providing **intra-query/inter-operator** concurrency. Also, parallelism can be applied within an individual operator by assigning separate processes to distinct partitions providing **intra-query/intra-operator** concurrency. Significantly, both of these forms of parallelism can be distributed to distinct processors in a multiprocessor system. In addition to such distributed parallelism, each instance of an operator process can be multiprogrammed providing additional, local parallelism.

A third dimension of available parallelism is also possible. In addition to the described sources of intra-query concurrency, multiple queries can be simultaneously active providing **inter-query** concurrency.

The total level of parallelism that is available in a multiprocessor database machine is a product of these three dimensions of query concurrency. Ideally, the extent to which this *cube* of parallelism can be effectively expanded should be determined by the availability of hardware resources on which to distribute the various query processes. The previous sections considered designs of an architecture and physical database that would remove artificial impediments to an effective distribution of these processes. The following sections will consider how scheduling algorithms and policies can materialize this capacity for parallelism.

5.4. Intra-Query Concurrency

One of the design challenges for multiprocessor database machines is to find a synchronization and control mechanism whose cost does not exceed the performance improvements provided by the parallelism of the system. The hash-partitioned algorithms have a potential for significant parallelism within and between operators of the same query. Therefore, a critical task in the design of a database machine that exploits hash-partitioning is the identification of a scheduling mechanism whose cost does not obscure the potential of the system.

5.4.1. Intra-Query/Intra-Operator Parallelism

The maximum level of parallelism that can be applied to a single hash-partitioned operator is bounded only by the number of partitions of tuples that are created and the available hardware resources of the system. That is, arbitrary levels of parallel execution can be achieved for single operators since distinct processes on separate processors can be assigned the independent task of processing discrete partitions of tuples.

The question of how much parallelism should be applied to a single operator therefore becomes as much a policy question as an implementation question. The level of parallelism that is applied to a hash-partitioned operation can be adjusted by producing fewer partitions of source tuples. Alternately, the same number of partitions can be created, but only some of these partitions might be immediately associated with active, consuming operator processes. The unassigned partitions would be spooled to temporary files.

These adjustments to the level of query parallelism can be made in response to a variety of conditions. In a heavily loaded system, the throughput of the system might be improved by reducing the level of intra-operator parallelism. Adjustable intra-operator parallelism can also be used to provide levels of priority service on a per-query basis.

Finally, a scheduler might adjust the parallelism applied to a single operator in coordination with the parallelism that is being applied between multiple operators of the same query. Since the processes representing the various instances of a hash-partitioned operator will compete with processes representing different operators of the same query, there is a strong interaction between these two types of parallelism. In particular, an appropriate policy of allocating scarce resources such as memory to a complex query requires discriminating between the improvements in performance corresponding to alternative levels of these two forms of parallelism.

5.4.2. Intra-Query/Inter-Operator Parallelism

Another opportunity for intra-query concurrency is provided when separate operators of the same query are executed in parallel, i.e. intra-query/inter-operator concurrency. The structure of the HP-join algorithms provide an excellent basis for effecting this parallelism. This results from two factors. First, the split phase for these operators can be executed by the process that is producing the source tuples for the operation, e.g. the previous selection or join operation. Secondly, in the probing phase of a HP-join operator, tuples can be streamed through the join operation producing result tuples that are simultaneously partitioned and sent to a subsequent operator. The combination of these two factors creates an environment where pipelines of at least three consecutive operators[15] can be effectively constructed. Finally, even for operations on very large source relations, inter-operator parallelism and an even flow of tuples through pipelines can be maintained due to the ability of the algorithms to partition the processing requirements of a join operation. It is also significant that tuple pipelines will not be delayed or blocked awaiting the sorting of an entire relation as sorting is never required by the hash-partitioned operators.

Inter-operator parallelism requires coordination by a query scheduler at runtime. One determinant of the effectiveness of inter-operator parallelism is the structure of the compiled query trees that are produced by a query optimizer and traversed by a scheduler. Different formats of query trees (compiled query plans) can significantly affect the parallelism that can easily be applied to a query as the structure of these compiled query trees can simplify or complicate the task of scheduling the execution of a query. More complex query trees potentially offer richer opportunities for realizing the parallelism of a query. However, perhaps more important than the gross level of parallelism within a query is the extent to which parallelism can be combined with a data flow scheduling policy. That is, the savings that result from "as-needed" activation of operators and the pipelining of data between operators are important considerations in the design of a intra-query scheduler.

In the following discussion, we consider the merits of using various query tree structures. All of the alternative query plan structures considered are equally capable of supporting the intra-operator parallelism described in the previous section. Therefore, the following discussion will concentrate on issues involving inter-operator parallelism. The various compiled query tree alternatives will be evaluated according to four criteria:

- **Parallelism.** A compiled query tree should provide a basis for activating multiple operators of a query in parallel.
- **Pipelined Data Flow.** A high degree of parallelism may not increase the performance of a query if the results of relational operations are frequently written to temporary files. Instead, query operators should be activated in a manner that allows data to be pipelined directly between operators.

[15]When a join operation enters the probing phase, it can read tuples from one pipeline and the result tuples from the join can be written to another output pipeline. Associated with these two pipelines will be three simultaneously active operators: the producer of the probing relation, the join operation, and the consumer of the result of the join operation.

- **Resource Utilization.** The level of parallelism that can be supported for any given complex query is dependent both on the pattern of resource utilization of that query and other concurrent queries. In the case of the hash-partitioned algorithms, performance is directly related to the available memory resources in the system. Potentially, fewer parallel operators with adequate memory resources can outperform a more highly parallel set of operators competing for scarce memory resources. For similar reasons, contention for CPU and network resources are also important considerations.
- **As-Needed Activation.** Ideally, operators in a complex query tree should not be activated until their inputs are ready to be materialized. An earlier activation of an operator will waste the resources that are allocated to the idle operator, i.e. memory for stack space. However, a tardy activation of an operator will require that intermediate results be either spooled to temporary files or buffered in memory (wasting a valuable resource). The structure of a query tree can provide useful information about the appropriate times for activating query operators.

Three distinct forms of compiled query trees can serve to illustrate the implications that static query trees have on the scheduling of an individual query. Specifically, a query optimizer can build left-deep, right-deep or bushy query trees. Left-deep query trees are defined to be trees having only leaf nodes as right descendants. Similarly, right-deep trees have only leaf nodes as left descendants. Bushy query trees are unrestricted in their structure and are arbitrarily branched as the result of the query optimizer's attempt to optimize the characteristics of the various intermediate relations, i.e. the minimization of the size of the intermediate relations.

In the following discussion, we consider the impact of representing a complex query in each of these three formats. Each of the query trees follow the convention that the left descendant of a join operator will represent the building relation. That is, the hash tables of the HP-join operations will be constructed from the left-child of a join operation. Right descendents of join operators will represent probing relations. The following example assumes the existence of four relations with the following attributes:

Suppliers (name, supplier#, discount#, address)
Parts (name, part#, weight)
Supplier-Parts (supplier#, part#)
Discounts (discount#, percentage, sales-level)

The first three relations represent an enterprise wherein there is a many-to-many relation between suppliers and the parts supplied. The Discounts relation represents the fact that parts can be bought at various discount rates.

The following query stated in QUEL requests the names of all the suppliers who sell parts weighing less than 7 pounds for at least a 10% discount.

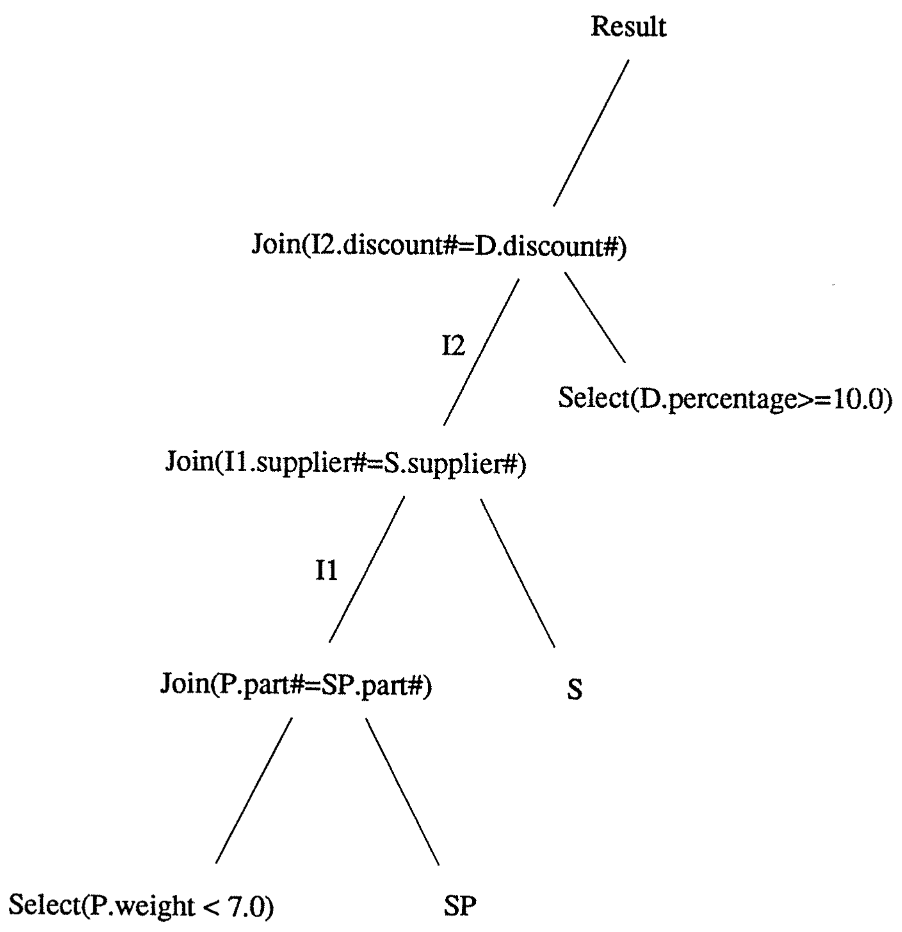
```
range of S is Suppliers
range of P is Parts
range of SP is Supplier-Parts
range of D is Discounts
retrieve into Small-Part-Sources (S.name)
where P.weight < 7.0
and P.part# = SP.part#
and S.supplier# = SP.supplier#
and S.discount# = D.discount#
and D.percentage >= 10.0
```

Figure 5.3

In the following discussion, we describe how inter-operator parallelism can be achieved using alternative representations of the query of Figure 5.3 in left-deep, right-deep and bushy query trees. It is important to remember that when the descriptions refer to the activation or execution of an operator, that such an action will implicitly involve multiple processes distributed throughout the system. For simplicity, explicit reference to this intra-operator parallelism will be omitted whenever possible in order to concentrate on the topic of inter-operator parallelism.

5.4.2.1. Left-Deep Query Trees

One potential left-deep query tree representation for this query is illustrated in Figure 5.4.


Result
Join(I2.discount#=D.discount#)
I2
Select(D.percentage>=10.0)
Join(I1.supplier#=S.supplier#)
I1
Join(P.part#=SP.part#)
S
Select(P.weight < 7.0)
SP

Left-Deep Compiled Query Tree
Figure 5.4

The fact that the HP-join algorithms can be cleanly separated into three distinct phases provides a basis for overlapping the execution of three of the relational operators in this left-deep query tree. Consider the following sequence of operator activation. Initially, the P-join-SP and select-P operators are activated. Tuples from the selection operation on the Parts relation are pipelined to the building phase of the P-join-SP operator. The P-join-SP operator uses those tuples to build hash tables. During this initial stage of processing, two operators are simultaneously active.

After the building phase of the P-join-SP operation completes, the scan on the Suppliers-Parts relation is activated. The tuple production of this scan operation is partitioned and pipelined to the probing phase of the P-join-SP operation. The P-join-SP operation simultaneously produces result tuples that are partitioned and pipelined to the building phase of the I1-join-S[16] operation. At this point in the query, three operators of this left-deep query tree are simultaneously active in various stages of execution, i.e. retrieve-SP, P-join-SP, and I1-join-S.

In similar fashion, after the retrieve-SP and P-join-SP operations complete, the I1-join-S operation enters the probing phase. Then, the scan of the Suppliers relation produces the source partitions that are used by the probing phase of the I1-join-S operation. As matching tuples are produced, they are pipelined to the building phase of the I2-join-D operator.

Finally, after the scan of the Suppliers relation is exhausted, the I1-join-S operation completes. Then, the I2-join-D operator enters the probing phase. The selection on the Discount relation provides the probing relation for the I2-join-D operation. Matching tuples are immediately split to the operator processes responsible for storing the result relation on disk.

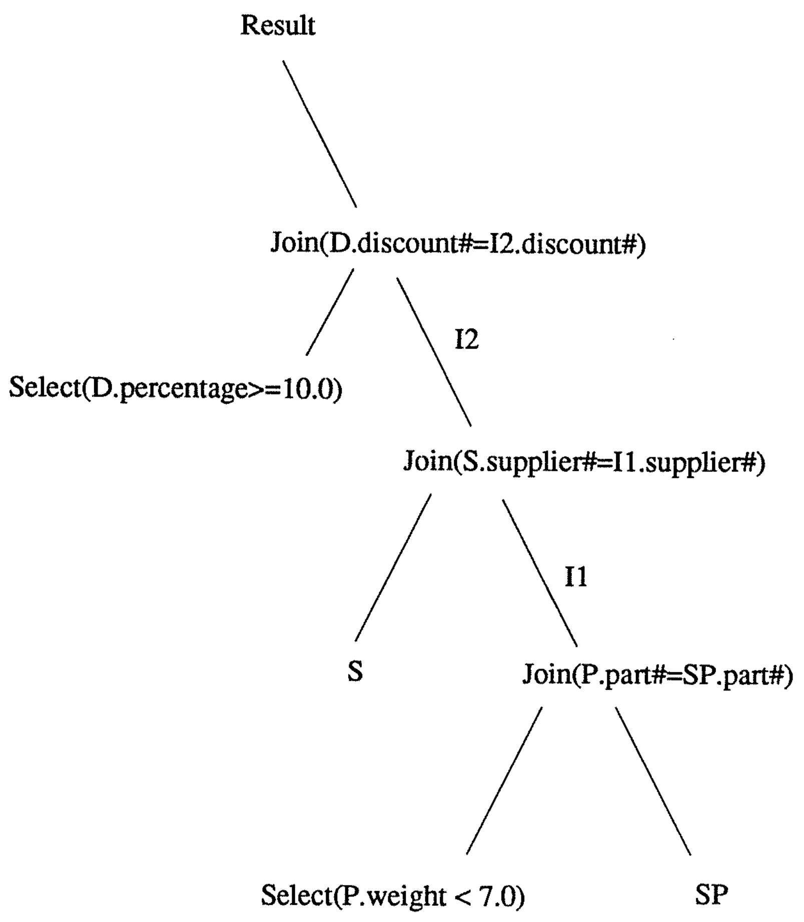
Throughout this activation sequence, pipelined data flows are used. In the absence of partition overflow, the storage of temporary relations on disk is unnecessary. In addition to the pipelining of data, multiple operators within the query are concurrently active. The level of inter-operator parallelism that can be achieved in such a left-deep query tree is three operators (two HP-join operations and a source relation scan). That is, with left-deep query trees the simultaneous execution of three hash-partitioned operators is the maximum level of intra-query/inter-operator concurrency that is possible. However, within these three operators, arbitrary levels of intra-operator parallelism are possible.

With respect to the memory consumption of left-deep based query activations, at most two operators will be using hash-tables at any given time for a single query. That is, one HP-join operator (or other hash-partitioned operator) can be executing a probing phase and producing tuples that are being used to construct hash-tables for a subsequent HP-join operator in the building phase. Since hash-tables are the major consumers of memory, an intra-scheduler will only have to balance the memory requirements of two operators at any given time. Also, it should be remembered that these operators will be competing with the operators of other distinct, concurrent queries. Therefore, the restrictions on the potential parallelism of a query represented by a left-deep query tree make it simpler to enforce a policy of fairness in the sharing of the memory resources of the system by multiple queries.

[16] I1 represents the first intermediate relation that is generated.

5.4.2.2. Right-Deep Query Trees

A right-deep tree representation of the query of Figure 5.3 is depicted in Figure 5.5.


Result
Join(D.discount#=I2.discount#)
Select(D.percentage>=10.0)
I2
Join(S.supplier#=I1.supplier#)
S
I1
Join(P.part#=SP.part#)
Select(P.weight < 7.0)
SP

Right-Deep Compiled Query Tree
Figure 5.5

In this structure, each left descendant in the tree is a leaf node representing an access to a permanent source relation.

This query tree offers an opportunity to execute all of the HP-join operators in parallel. Each of the source relations represented by the left leaves of the query tree can simultaneously be scanned (or restricted) and the tuple production of each can be split into partitions. All of the join processes can also simultaneously be building hash tables from their respective partitions of the building relation. Specifically, the restriction on the Parts relation outputs its partitions to the building phase of the P-join-SP operators. The scan of the Suppliers relation sends output partitions to the building phase of the S-join-I1 operators. At the same time, the selection of the Discount relation produces partitions for the building phase of the D-join-I2 operators.

In this manner, three sets of hash tables are simultaneously built through the efforts of six concurrently active operators[17] Additionally, each of these operators will be represented by multiple processes distributed throughout the system. The net effect will be a **VERY** highly parallel query execution.

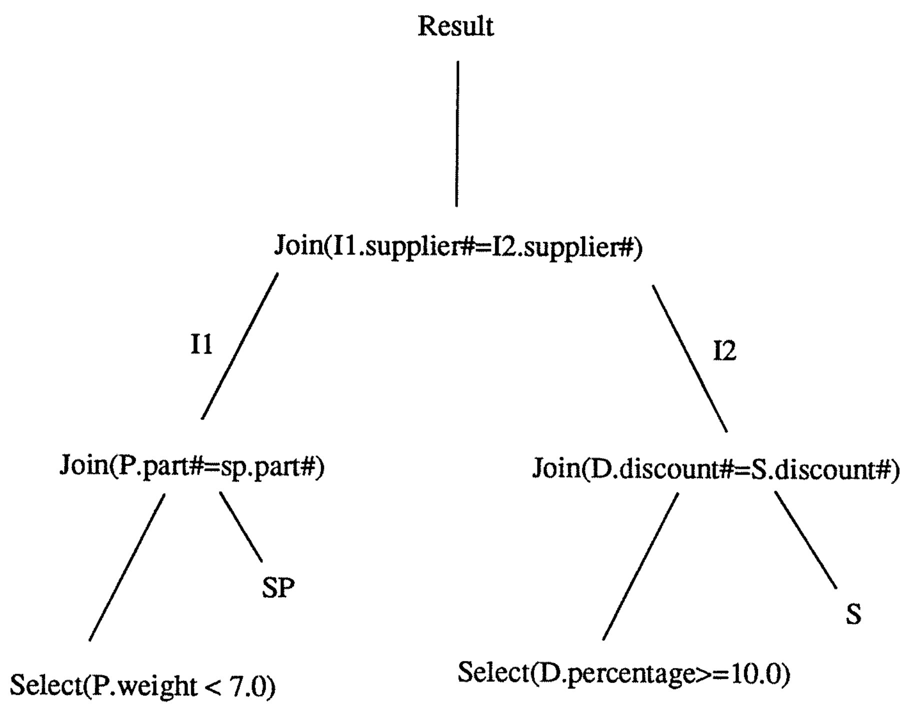
Once all hash tables have been built for the join operators, the probing phases of all the join operators can pipeline their output immediately to the subsequent join operation. First, the scan of the Supplier-Parts relation is initiated. The tuples from this scan are partitioned and sent to the probing phase of the P-join-SP operators. Matching tuples from the P-join-SP operation are partitioned and immediately pipelined to the probing phase of the S-join-I1 operators. In a similar fashion, matching tuples from S-join-I1 operation are simultaneously partitioned and pipelined to the D-join-I2 operation. As illustrated by this example, right-deep query trees using hash-partitioned operators provide the potential for the creation of pipelines that simultaneously contain all the HP-join operators in the query.

Although right-deep query trees are recommended by their unmatched ability to inherently support extensive inter-operator parallelism and pipelined data flows, there are potential problems concerning memory utilization. In order to support this extensive parallelism and the pipelined data flows, right-deep trees require the simultaneous formation of a number of hash tables equal to the number of simultaneously active join operator processes. All of these hash tables compete with one another and with the memory requirements of other concurrent queries. Alternately, if only some of the join operators are simultaneously activated, the last join operator in a pipeline will have to spool its tuple production to a temporary file. Also, if memory resources become scarce during the execution of the building phases of multiple join operations, the consequences of partition overflow may be multiplied by the existence of the multiple competing join operator processes.

5.4.2.3. Bushy Query Trees

The last form of query tree that will be considered is the bushy query tree. A bushy tree corresponding to the example query is shown in Figure 5.6.

[17]2 selections, 1 scan, and 3 joins.


Result
Join(I1.supplier#=I2.supplier#)
I1
I2
Join(P.part#=sp.part#)
Join(D.discount#=S.discount#)
SP
S
Select(P.weight < 7.0)
Select(D.percentage>=10.0)

A Bushy Compiled Query Tree
Figure 5.6

The branches of a bushy query tree present all the benefits and problems associated with left and right-deep query trees, if sub-branches are considered as equivalent to the leaves of the linear trees. However, the junctions of branches present problems that are unique to the bushy query trees. For example, in Figure 5.6, should the branches of the I1-join-I2 operator be activated sequentially or simultaneously?

If the branches are activated sequentially, it would appear advantageous to activate the I1 branch first. In this manner, the hash table for the I1-join-I2 operator can be constructed prior to the production of the probing relation I2. In the current example, this sequence seems reasonable. However, as the depth of the right branch of a bushy query tree increases, the resource allocation problems associated with right-deep query trees begin to appear. That is, either more hash tables must be simultaneously supported or temporary relations must be used. In particular, the idle hash table associated with a junction operation such as the I1-join-I2 operation will consume memory resources while waiting for the probing relation from the right branch to be materialized. As the right branch could be arbitrarily deep, the idle hash table of a join operator at such a junction can represent a significant waste of memory resources.

Alternately, if the right branch of Figure 5.6 were activated first, then the probing relation produced by the D-join-S operator would have to be spooled in a temporary file until the hash tables of the I1-join-I2 operator were completed.

All of the problems associated with the sequential activation of branches would also occur if branches of a query tree were activated simultaneously. However, an additional problem would now exist. A scheduler would have to control the situation where hash tables were still being constructed when a probing relation began to be materialized. How long should the probing relation be delayed, waiting for the completion of a build phase? At what point should the probing relation merely be spooled to a temporary file? Timing decisions of this nature would significantly increase the complexity of the scheduling algorithm. Synchronizing the simultaneous execution of multiple branches would also represent significant complexity.

The bushy query trees are, however, attractive for other reasons. Given the unrestricted format of such a query tree, a query optimizer can structure query trees in a manner that minimizes the sizes of the intermediate relations. The minimization of these intermediate relations can significantly enhance the performance of a query while simultaneously reducing the load that is placed on the system. The linear, compiled query trees provide less flexibility in this regard and might be expected to produce larger intermediate relations as a result. However, [LOHM85] suggests that linear query trees considerably simplify an optimizer and do not miss many optimal join orderings.

5.5. Inter-Query Scheduling

The previous section described how intra-query parallelism can be applied within a database machine that is based upon the use of hash-partitioned algorithms. Inter-query concurrency is also possible. All that is required is a means of associating queries with individual instances of scheduling processes. This functionality can be provided by a centralized dispatching service. In addition to providing a queuing point for incoming queries, a dispatcher can assume responsibility for controlling the load that is imposed on a system.

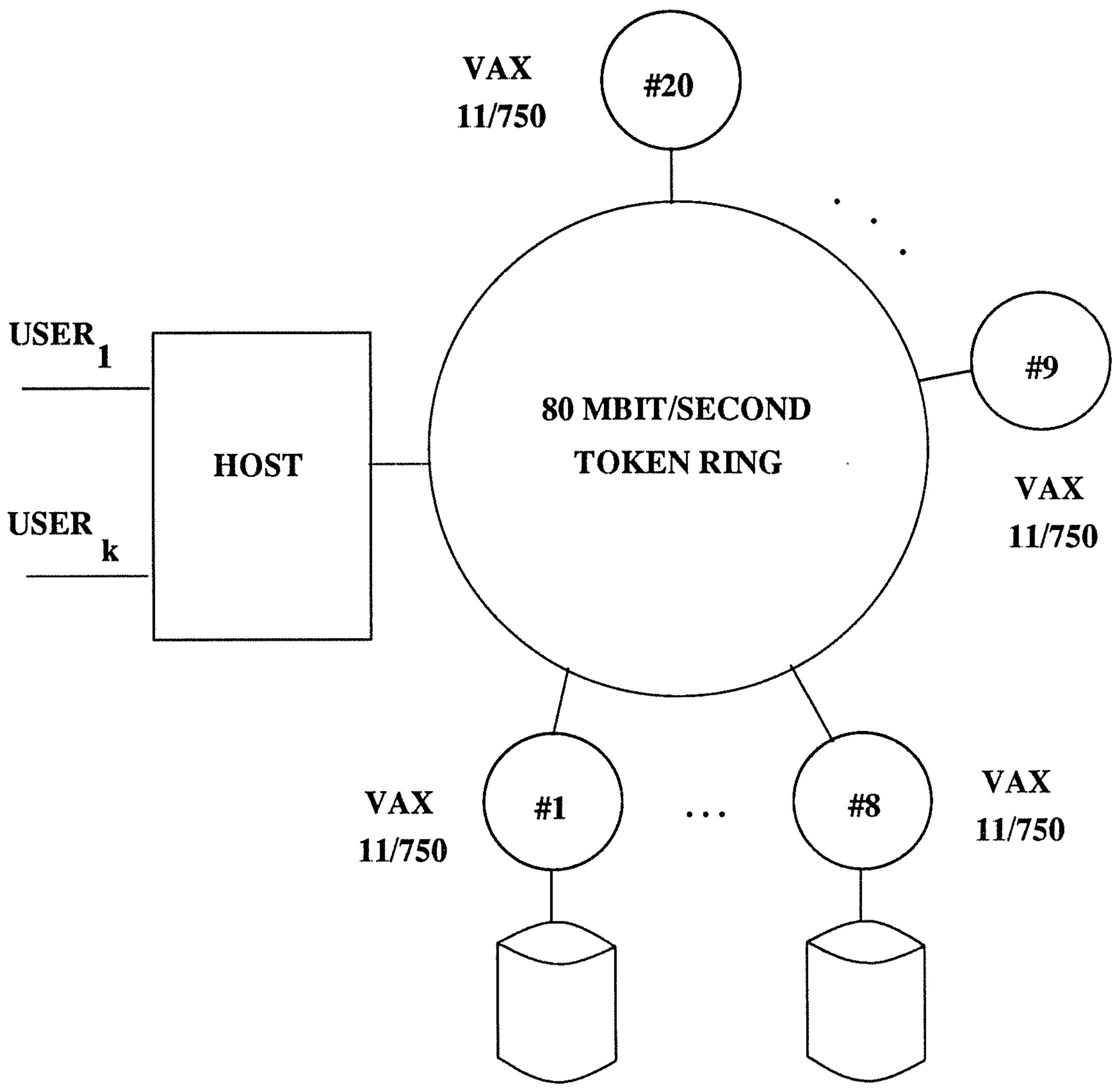
CHAPTER 6

GAMMA DATABASE MACHINE: DESIGN AND IMPLEMENTATION

In this chapter, we present the design and implementation of Gamma, a new relational database machine that reflects one set of design decisions selected from the domain of alternatives described in the previous chapter. Gamma is a fully operational prototype that demonstrates how the hash-partitioned algorithms can provide a basis for utilizing dataflow query processing techniques. Implementing a prototype of Gamma achieves a number of important objectives. First, it demonstrates that parallelism can really be made to work in a database machine context. A second objective is that, although not as flexible as a model, a prototype provides much more reliable information about the performance bottlenecks of our design.

6.1. Hardware Architecture of GAMMA

The architecture of the current prototype of the Gamma database machine is shown in Figure 6.1. Presently, Gamma consists of 20 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring developed for the Crystal project [DEWI84b] by Proteon Associates [PROT85] is used to connect the processors to each other and to another VAX 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.

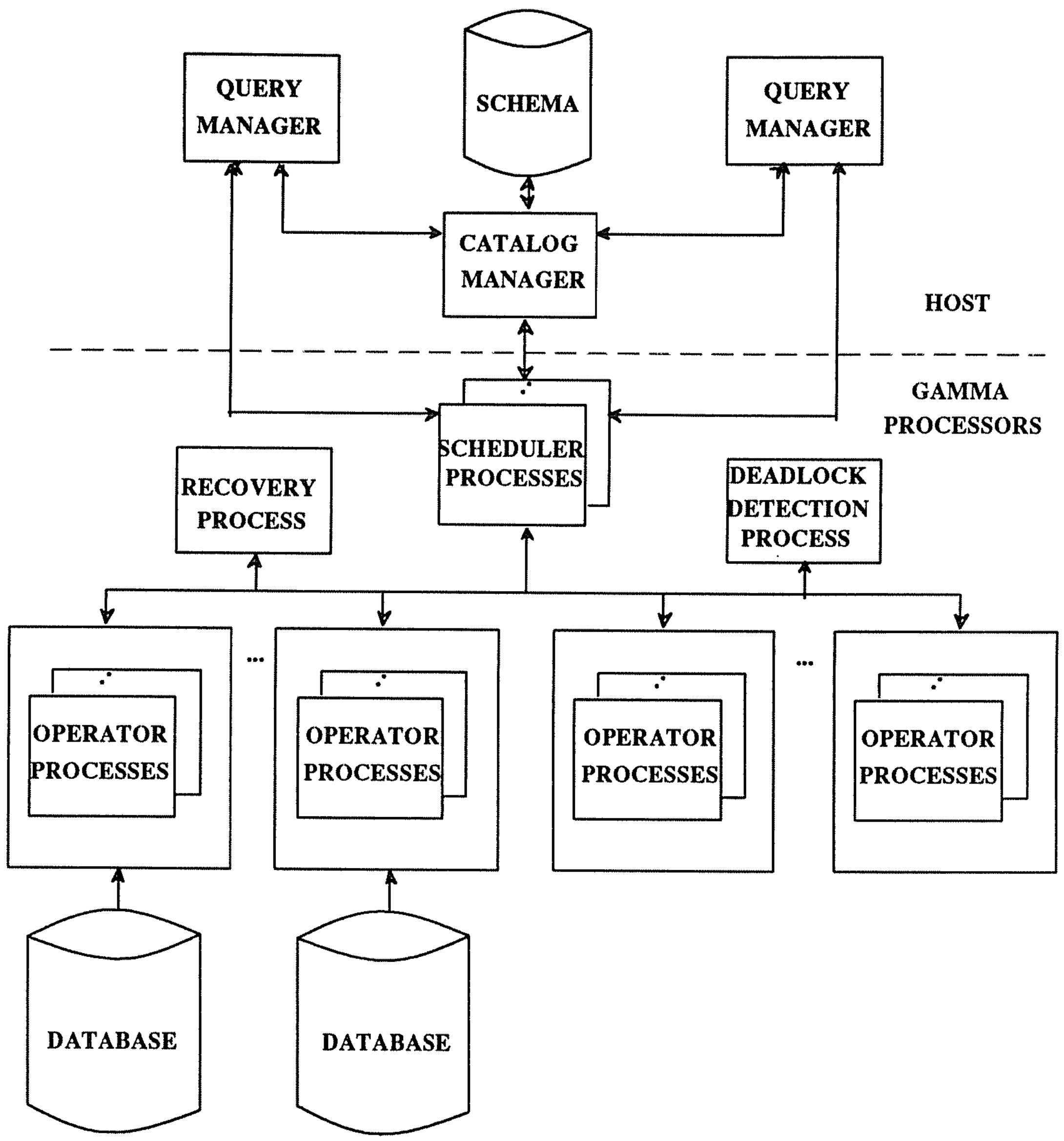


Gamma Hardware Configuration
Figure 6.1

6.2. Software Architecture

6.2.1. Process Topology

In Figure 6.2, the structure of the various processes that form the software of Gamma is specified.



Gamma Process Structure
Figure 6.2

Along with indicating the relationships among the processes, Figure 6.2 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.

Catalog Manager

The function of the Catalog Manager is to act as a central repository for 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

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 "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.

Operator Processes

For each operator in a query tree, at least one Operator Process is usually 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

Rather than use a distributed deadlock detection mechanism, Gamma employs a centralized deadlock detection process. This process is responsible for collecting fragments of the "wait-for" graph from each lock manager[18], 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 and abort.

6.2.2. 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 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 record files, byte-stream files as in UNIX, B^+ tree indices, long data items, a sort utility, and a scan mechanism. Records may vary in length (up to one page in length), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each sequential file may have one or more associated indices. The index maps key values to the records of the sequential file that contain a matching value. Furthermore, one indexed attribute may be used as a clustering attribute for the file. The scan mechanism is similar to that provided by System R's RSS [ASTR76]. When a scan is opened, a boolean expression is specified in disjunctive normal form. The scan returns only those tuples that satisfy the boolean expression.

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 the network and immediately written to disk without requiring that tuples be copied from an incoming message template into a page in the buffer pool.[19]

[18]There is one lock manager process on each processor with a disk.

[19] While initially attractive as a method of avoiding the copying of tuples between network and disk buffers, binding the network and disk page formats together was, in retrospect, a mistake (see Section 6.9).

6.3. Database Streams and Storage

6.3.1. Horizontally Partitioned Physical Database

All relations in Gamma are **horizontally partitioned** [RIES78] across all disk drives in the system. Storing a relation across all the disks in the system improves the response time of both read and write accesses to a relation. This results from the fact that all the disks in Gamma can be accessed in parallel.

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, when tuples are loaded into a relation in the first strategy, 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 Teradata 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:

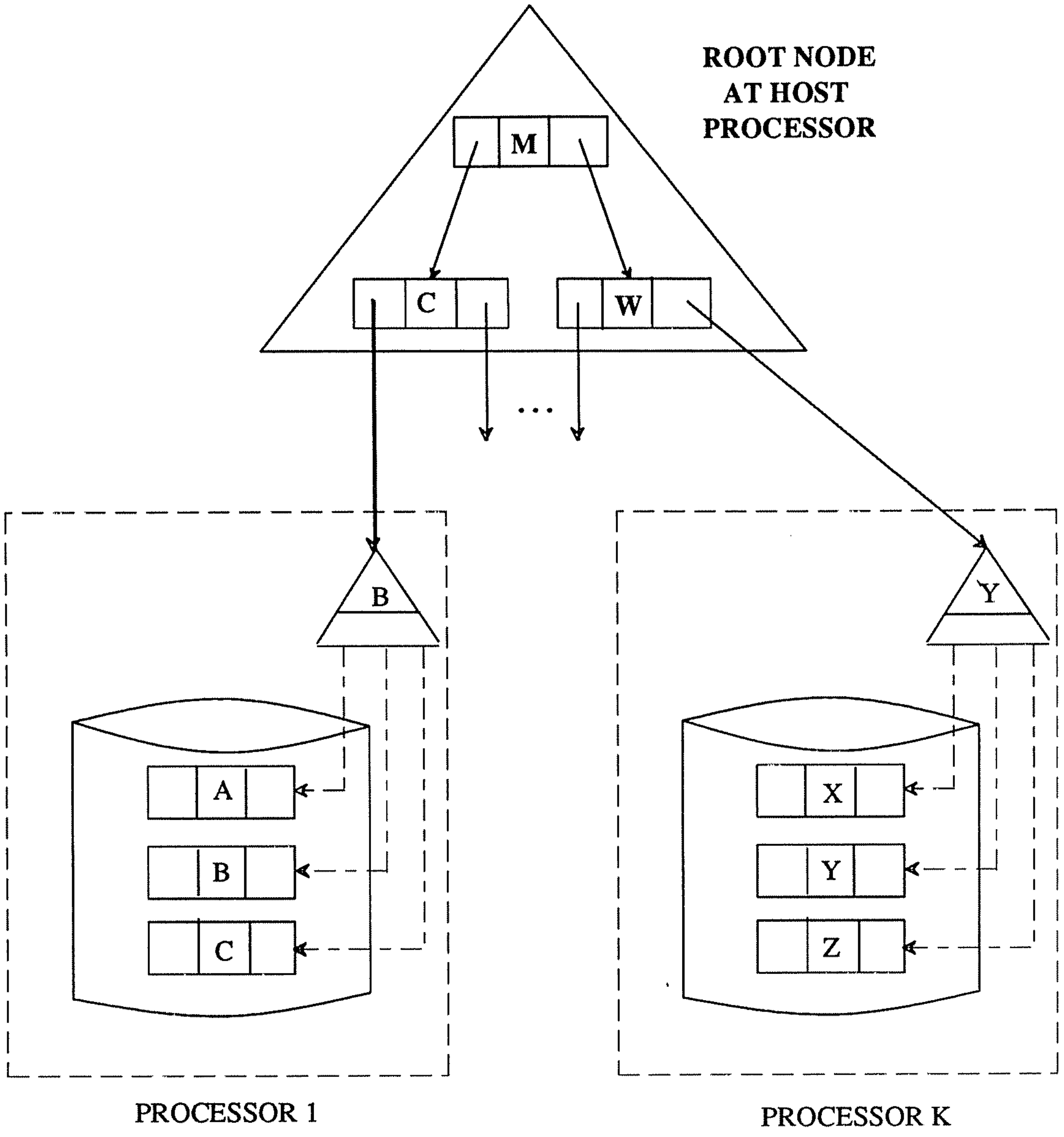
Distribution Condition	Processor #
emp-id $\leq$ 100	1
100 < emp-id $\leq$ 300	2
300 < emp-id $\leq$ 1000	3
emp-id > 1000	4

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 6.3, the disks, and their associated processors,



Multiprocessor Index
Figure 6.3

can be viewed as nodes in a primary, clustered index.[20] 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[21] to direct selection queries on the key attribute to the appropriate sites for execution.

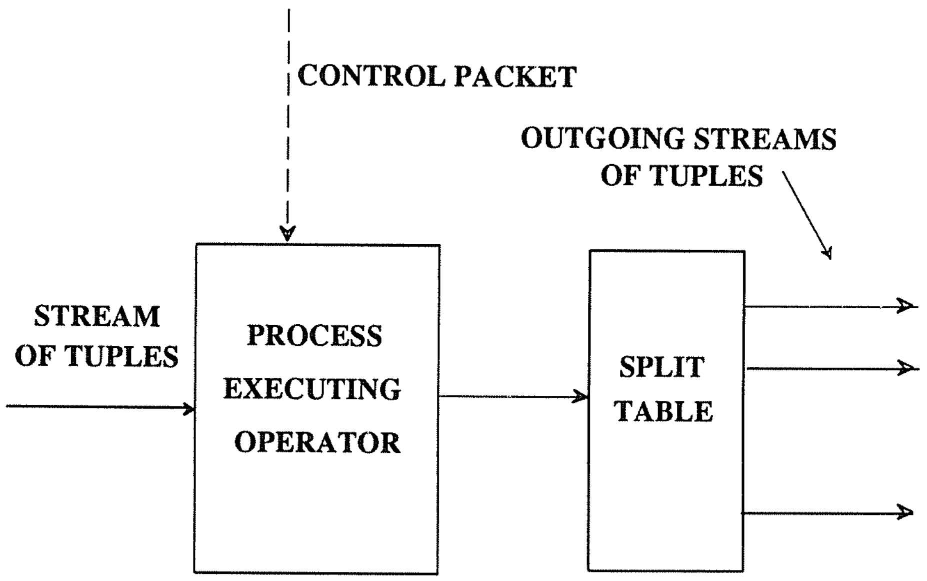
[20] A multiprocessor index may consist of only 1 level if indices have not been created at the disks.

[21]To ensure consistency, before the horizontal partitioning ranges for a relation can be altered, (updating the root page of the multiprocessor index) an exclusive lock on the relation is required at the level of the catalog manager. That is, the root page of a multiprocessor index is part of the schema information that is maintained for a database.

6.3.2. Writing Partitioned Data Streams

Although Gamma is a multiprocessor dataflow system, all operators are written as if they were to be run on a single processor. A process executing a relational operator reads tuples from a single, local input stream, operates on each tuple, and produces a single, local output stream. Hidden from the operators are the facts that output streams are split and routed across the network via multiplexed pipelines and that input streams demultiplex the fragments of partitions. Providing this level of transparency insulates individual operator processes from the details of partitioned data streams. The following discussion considers how these data streams are created and written. A later section describes how the fragments of partitions are collected into a single input stream by an operator process.

As shown in Figure 6.4,



Split Table Demultiplexing a Stream of Tuples
Figure 6.4

the output of an Operator Process is a stream of tuples that is demultiplexed through a structure we term a **split table.** These split tables define the boundaries of the tuple partitions that are produced. Each result tuple is assigned a split value which depends on a specific attribute value called the **split attribute.** The designation of a **split attribute** is determined by the requirements of the partitioning phase of the query operation which will be receiving the result tuples. For example, if the receiving operator process is a join operation, then the join attribute of that operation will be defined to be the split attribute of the current operation.

The query optimizer specifies boundaries that partition the potential range of split values into the subsets X_1, ..., X_n. These boundary assignments are reflected in the split table. Every tuple of a produced relation R whose split value falls into the range of values associated with X_i will be assigned to partition R_i. For example, consider the use of a split table shown in Figure 6.5 in conjunction with the execution of a join operation using 4 processors. Each process producing source tuples for the join will apply a split function to the join attribute of each output tuple to produce a split value between 0 and 3. This value is then used as a key for a lookup on the split table to obtain the address of the destination process that should receive the tuple. (For the most common form of split table, the split value is used directly as an index into the split table. No lookup is performed. This process of computing these index values is described below for Hashed split tables.)

Split Key	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 6.5

Due to the multiprocess nature of operators in the Gamma environment, identical split tables must be used by all processes that are working on the same query operation. Also, for hash-join operations, the split tables for the probing relation (the right descendent of the join node) are inherited from and are identical to those split tables applied to the building relation (left descendent).

Three distinct types of split tables exist: RANDOM, HASHED and RANGE. In the case of HASHED and RANGE split tables, a hash function is applied to the targeted split attribute of the tuples. These hashed values are used to resolve the partition placements for the tuples being produced. Since these same hashed values are also required by and frequently referenced by subsequent hashed-based operators, the values are appended to all tuples during the partitioning process.

Tuples are appended to and read from partitions via a buffered tuple interface. This interface enables query processes to process data a tuple at a time, while network data transfers are always done a page at a time. When used in conjunction with split tables, the buffered tuple interface insulates individual query processes from the distributed nature of Gamma. From the perspective of a query process, an operation merely reads from one stream of tuples and writes to another.

Hashed Split Tables

Hashed split tables are the most common form of split tables used by Gamma. At runtime, as result tuples are produced, a split function is applied to the targeted split attribute of each tuple. Each tuple is then assigned to a partition that corresponds to the tuple's computed split value. That is, the split table entry corresponding to each tuple is directly computed (not found as the result of searching the split table). The tuple is then written to the destination partition.

Round-Robin Split Tables

Round-robin split tables are the simplest form of split tables. The split value that is assigned to each tuple is actually independent of the tuple (an exception to the rule that split values depend on a tuple attribute). That is, the split tables used for round-robin partitioning do not determine the mapping of tuples to partitions. Rather, tuples are uniformly distributed among all partitions. The round-robin split tables do, however, contain the destination addresses associated with each partition. Tuples can be assigned round robin to partitions on a tuple by tuple basis. Alternately, an entire pageful of consecutive tuples can be assigned to individual partitions in turn. Round-robin split tables are used to populate the partitioned, permanent relations that may be produced as the result of a query.

Range Split Tables

The third type of split table used by Gamma produces tuple streams that are partitioned on discrete ranges of non-hashed attribute values. Tuples are assigned to partitions via a binary search of a range split table using a tuple's split attribute value as a search key. The upper bound of each partition range serves as a primary key value for each entry in the split table. These range partitioned split tables are used when permanent relations are fragmented and stored on disk using either of the range partitioning strategies described in Section 6.3.1.

Range split tables are also applicable during query processing in an effort to reduce the network communications costs associated with the processing of permanent source relations. This will be the case 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 using the boundary values defined for the source relation's HPA. These boundaries or range keys ensure that as many tuples as possible are retained at the processors with disks where the source tuples originate. For join operations, if the building relation is horizontally partitioned on the join attribute, then a significant portion[22] of the relation will not be transmitted across the network at all. In this case, the probing relation of the join would be partitioned and distributed according to the HPA ranges of the fragments of the building relation.

When **range** split tables are used for query processing, hashed keys are still used when tuples are split into partitions. The need for these hashed keys is motivated by the need to be able to create a number of partitions that is larger than the number of HPA partitions corresponding to a permanent file. While the number of HPA partitions always equals the number of disks in the system, a larger number of partitions is necessary during query processing in order to utilize the full resources of the system, i.e. the processors without disks. A larger number of partitions are also required when the sizes of the horizontal partitions in a permanent, source relation exceed the size of available memory on local processors. Both of these circumstances may justify transferring data across the network even though entire partitions of a source relation are concentrated at local sites.

The hashed keys in a **range** split table are used as secondary keys, effectively sub-partitioning an existing HPA partition.[23] When relations are split using a range split table, the actual value of a tuple's split attribute is used as a primary key and the hashed value that is computed for the split attribute is used as a secondary key. The partition associated with the tuple is then found via a binary search of the range split table using both of these search key values.

The HPA partitions reflected in a range split table are always split into equal sized sub-partitions. This is accomplished by assigning each of the sub-partitions a hashed key value that represents the upper bound of a subrange of potential hashed values. These discrete subranges equally divide the total range of values that a hash function can produce. For example, consider a permanent relation that has been partitioned according to the conditions stated in Figure 6.6 which assume a system configuration containing four processors with disks.

[22]As will be described, range split tables have the capability of sub-partitioning the tuples that belong to a single horizontal partition of a relation.

[23]Also, recall that for a join operation, the building and probing relations will be partitioned using identical split tables.

Distribution Condition	Processor #
emp-id ≤ 100	1
100 < emp-id ≤ 300	2
300 < emp-id ≤ 1000	3
emp-id > 1000	4

Horizontal Partitioning Criteria
Figure 6.6

Figure 6.7 shows a RANGE split table that could be used to produce eight partitions from the original relation. Such a partitioning would be useful if there were four processors without disks in addition to those with disks. The figure assumes that valid hashed values fall in the range 0..1000. Note that the repetition of the secondary hashed keys is typical of the fact that range tables are ordered on the primary range key values that represent the upper boundary of the corresponding HPA partitions.

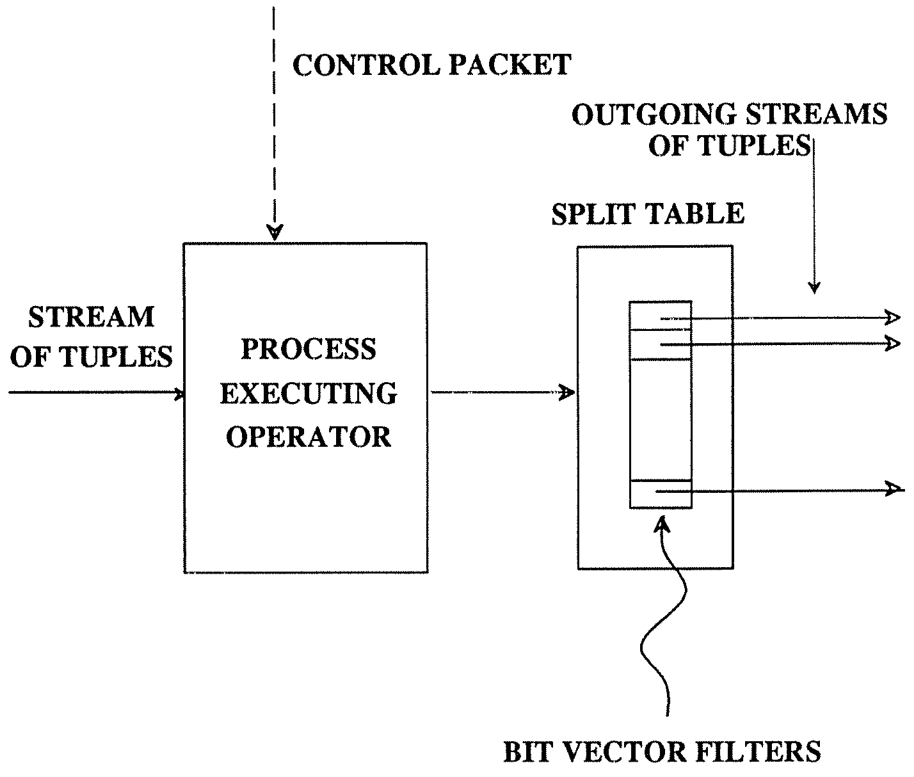
Range Value	Hashed Value	Destination Process
100	500	(Processor #1, Port #5)
100	1000	(Processor #5, Port #7)
300	500	(Processor #2, Port #13)
300	1000	(Processor #6, Port #23)
1000	500	(Processor #3, Port #6)
1000	1000	(Processor #7, Port #17)
MAXINT	500	(Processor #4, Port #15)
MAXINT	1000	(Processor #8, Port #3)

An Example Split Table
Figure 6.7

In this example, partitions 1, 3, 5 and 7 of the building relation would be processed locally at their site of origin. This strategy would be expected to save approximately half the cost of transmitting the building relation across the network, a potentially significant savings.

Application of Bit Vector Filters

To enhance the performance of certain operations, Gamma employs bit vector filtering. This filtering is applied by inserting an array of bit vector filters [BABB79, VALD84] into a split table as shown in Figure 6.8.



A Split Table with a Bit Vector Filter
Figure 6.8

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 building relation [BRAT84, DEWI85, DEWI84a]. A collection of these filters is provided to the processes responsible for producing the inner (probing) relation of the join. Each of these processes uses the set of filters to immediately eliminate those tuples of the probing relation that are not represented in the bit vector filter and will therefore not contribute to the result of the join operation.

This filtering provides an inexpensive approximation of one of the benefits of an index join. That is, nonqualifying tuples are discarded from consideration at a relatively early stage of processing. In a multiprocessor environment, bit vectors have the additional advantage that such nonqualifying tuples can be discarded at the site where they are produced. Filtering at that point reduces the volume of tuples that must be transmitted to the processes that effect the actual join.

The advantages of bit vector filtering do not, however, eliminate the need for an index join method in Gamma. With an existing index on the join attribute of the larger source relation to a join, a nested loops index join will frequently outperform ad-hoc join methods when the second source relation to the join is sufficiently small, i.e.

the result of a selection [LU85].

6.3.3. Reading Partitioned Data Streams

Relational operators in the Gamma environment produce fragmented partitions of tuples. This results from the fact that operators are represented by multiple processes and each process produces only a single fragment of each of the partitions of a relation. Therefore, before a subsequent operator can process any given partition, all the fragments of that partition must be collected together. Furthermore, the collection of these fragments should be done transparently to the consuming process.

The NOSE operating system provides a communication primitive that satisfies these requirements. Ports in the NOSE system are message queuing points that are globally visible throughout the local network. These ports are identified by a processor#, port# pair. A basis for the collection of partition fragments is made possible by associating partitions with ports on a one-to-one basis.

Before an operator begins execution, it is provided with a split table. These split tables provide the mapping of partitions to ports. Given this mapping, a consuming process can acquire an input stream from the local port that is associated with a specific partition. The inherent queuing action of the port provides the multiplexing action that is required for the collection of the fragments of a partition.

The port mechanism of NOSE also provides a flow control mechanism between the producers and consumers of a partition of tuples. Incoming messages are buffered on a queue associated with a port. These buffers can smooth the exchange of data between fast producers and slow consumers. The amount of buffering associated with individual ports is controllable and can be altered by a consuming process in response to the nature of the current operation. Also, once all the buffers for a port have been filled, there is a ''back pressure'' mechanism provided by NOSE which can reduce the level of retransmissions of new data packets.

Fragments of a partition are not intermixed in the same network packet, but are transmitted sequentially in separate data packets between producer and consumer processes. Consuming processes merely read from their ports, receiving subsequent pages of a partition that may be from any one of the producing processes. The number of pipelines associated with a partitioned data stream between two consecutive operators in a query tree is equal to the product of the number of processes representing the producing and consuming operators. Significantly, however, the cost of constructing these pipelines is determined only by the sum of the number of processes executing the two operators because a single split table controls the entire matrix of pipelines. As the number of partitions in the split table is equal to the number of processes consuming tuples, the cost of constructing the split table increases linearly with the number of consuming processes. The additional cost of distributing the split table increases linearly with the number of processes representing the producing operator.

Since the hash-partitioned algorithms do not require ordered input streams, the pages of a fragmented partition stream may be read in arbitrary order. This characteristic of the hash-partitioned algorithms simplifies the design and enhances the performance of Gamma. That is, the data stream requirements of Gamma can be served by a very simple communications protocol. In particular, Gamma does not require a transport level communications protocol [TANE81] that guarantees in-order delivery of messages arriving from multiple senders. By comparison, a multiprocessor, sort-merge join operation would require a more complex communications protocol or additional synchronization primitives to guarantee the in-order delivery of fragmented streams of tuples.

An additional advantage also is possible as the result of the interaction of the hash-partitioned algorithms with the communications system. A simplified sliding-bit window protocol can be used to increase the throughput of the communications system without increasing the memory requirements of the communications protocol. Normally, a sliding-bit window protocol must be prepared to buffer a number of incoming messages equal to the number of bits in the communications window. However, these buffers are only required as a means of guaranteeing the in-order delivery of messages to the receiving process. Since Gamma does not require such ordering, the additional buffers are unnecessary. Therefore, the performance advantages of a sliding-bit window protocol can be achieved with little additional cost.

6.3.4. Mapping Partitions to Operators

Partitions are important objects in the Gamma environment. They represent a critical, indirect mapping between tuples and operator processes. While it would potentially be possible to compute the address of a destination process directly from the hashed value of a split attribute, Gamma instead maps hashed tuple values to partitions. Partitions are then mapped to operator processes. The indirection provided by this distinction, with partitions being first-class objects and not merely abstractions, allows the mapping between tuples and operator processes to be changed in response to a variety of runtime conditions, e.g. partition overflow, modified intra-query parallelism, etc.. Without this flexibility, the system's ability to control the execution of a query would be much more limited.

All partitioned streams created with the aid of split tables are transferred across the network to waiting processes[24]. However, not all of the destination processes represent relational operators. Sometimes it is necessary to spool a partition of tuples into a temporary relation that can be materialized at a later time.

In order to satisfy these needs, Gamma uses three types of split table entries that are named according to the kind of partitions that they create. There are **pipelined** partitions whose tuples are directly routed to a subsequent relational operator at the next higher level of a query tree. These partitions are so named because tuples are moved in a pipelined fashion between query processes at different levels of a query tree. There are **spooled** partitions that are sent to spooling processes that store tuples in temporary files. Finally, there are **overflow** partitions which contain two smaller sub-partitions, one of which is **pipelined** and the other of which is **spooled.** These **overflow** partitions selectively append tuples to both pipelines and temporary relations.

It is possible for a combination of these partition types to occur within a single split table. Usually, the query optimizer creates split tables containing predominantly **pipelined** partitions and occasionally **spooled** partitions. At runtime, an intra-query scheduler may change a **pipelined** partition to a **spooled** partition if insufficient memory exists to support the processing of the partition. Prior to activating the probing phase of a join operation, the scheduler may also change a **pipelined** partition to a **overflow** partition if a partition overflow occurs. In all cases, however, split tables are not modified once a query process begin execution.

Pipelined Tuple Partitions

These types of partitions are used whenever possible. When tuples are appended to this type of partition, they are effectively routed across the network to a consuming operator process. Since these tuples are not staged to disk, they provide a very efficient means of transferring data between adjacent levels of a query tree.

Spooled Tuple Partitions

If the query optimizer detects that insufficient memory exists in the system to build hash-tables for all partitions of a building relation, then **spooled** partitions are used. Also, at runtime, an intra-query scheduler may

[24]Actually, if the destination of a message is located on the same processor as the sender, then the message is short-circuited locally and is not transmitted across the network.

change a partition from type **pipelined** to type **spooled** if there is insufficient memory. **Spooled** partitions are implemented by spooling processes that can be remotely activated on processors with disks. When a spooling process is used in conjunction with a hash-join operation, a matched pair of partitions from the building and probing relations will be directed to a single spooler. Later, when the tuples of a spooled partition are needed for processing, the spooling process can be directed to materialize the previously stored tuples.

Overflow Tuple Partitions

When a partition overflow occurs, a **pipelined** partition may be changed to type **overflow.** An **overflow partition** effectively separates tuples into two distinct sub-partitions. One of these sub-partitions is a **pipelined** partition stream and the other is a **spooled** partition stream. That is, a tuple that is assigned to an **overflow** partition may be routed to either a subsequent operator process or a spooling process.

The separation of tuples into sub-partitions can be effected by using different bits of a tuple's hashed value than were used by the split function that originally placed the tuple into the partition. For example, if the split function used a modulus function then the sub-partitioning function could use division to separate the tuples into two classes. Alternately, a simple boundary value could be used to separate sub-partitions based on their hashed values. The sub-partition boundaries (and hence the size of the two sub-partitions) are determined during the resolution of partition overflow.

6.4. Relational Operators

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[25] and other complex operations [DEWI85]. 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.

6.4.1. Process Structure

All relational operators in Gamma are executed by one or more query processes that reside on the various processors in the system. The processors that support such query execution processes are referred to as query

[25]An index join method will be added to Gamma in the future.

processors to distinguish them from the other processors in the system, i.e. the hosts. At system startup, each query processor is loaded with a program containing the query execution code. The main query process opens up a globally visible port, the NEWTASK-PORT, and then spawns a predetermined number of query processes.

These query processes provide the ability to activate self-contained relational operators on any processor in the system. The query processes that provide these services constitute an anonymous pool of servers on each processor. These servers are accessible via the global NEWTASK-PORT from which the servers accept requests. To activate a relational operator on a specific processor, a packet describing the operation (including parameters) is sent to the NEWTASK-PORT on the desired processor. Such packets requesting the local execution of a relational operation are removed and serviced by one of the local query processes. When all the query processes are busy, the NEWTASK-PORT acts as a ready queue. Each of the query processes also opens an individual, private port that is used for all communication local to a given query.

Since relational operations may be executed by any one of the query processes on a given processor, the query processes identify themselves to the originators of the query packets that represent work requests. When a query process removes a query packet from the NEWTASK-PORT queue, it determines the network address of the source of the packet by referring to the header of the packet. The query process immediately replies and identifies itself to the originator of the work request. Query processes are identified by the network address of the private, local port which they own. Given such an identification, a query process can be controlled and/or provided with data for the operation which it will perform. These network address identifiers are critical components in the construction of inter-operator pipelines.

6.4.2. 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.

Earlier database machine research has demonstrated that parallelism cannot be used as a complete substitute for indices [BORA83]. Therefore, Gamma uses existing[26] indices for selection operations whenever possible. The Gamma version of WiSS has also optimized the application of restriction predicates through the use of threaded code [BELL73, DEWA75, STON83].

Gamma selection processes must interact with two devices, a disk and the interconnection network. In order to minimize the unproductive time spent waiting on service from these devices, a limited form of read-ahead is used to overlap the processing of one page with the I/O to get the "next" page of a relation from disk. The read-ahead technique employed also overlaps the periodic network write operations that result when a selection operator flushes a buffered page of a partition across the network.

6.4.3. Parallel Hybrid Hash-Join

As in the centralized algorithm, there are two distinct joining phases to the parallel, **Hybrid** HP-join algorithm[27]. In 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 HP-join operator sends a message to the scheduler indicating that the building phase has been completed. Once the scheduler determines that all HP-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.

Partition Overflow Resolution Using Repartitioning

When a partition overflow condition occurs, a join process continues to accept tuples from the affected partition of the building relation. By continuing to accept all tuples from the partition, the static nature of split tables is preserved. Tuples in the building partition are then separated by the join process into two discrete sub-partitions (see Section 2.2). Tuples in the first sub-partition, the **pipelined** sub-partition, are retained in or added to

[26]The cost of dynamically creating an index would not be justified for a selection query.

[27]As in the centralized Hybrid algorithm, the distributed version overlaps the joining and partitioning phases of processing to the greatest extent possible while completing the partitioning of each source relation in a single scan.

the hash-table. All the other tuples belonging to the second sub-partition, the **spooled** sub-partition, are sent to a spooler process that the join process activates on a potentially remote processor with a disk.

Sub-partitions can resolve an occurrence of partition overflow because tuples of the **spooled** sub-partition are removed from the in-memory hash table and the space they occupied is reclaimed. With some care, and due to a nonpreemptive process scheduling algorithm, a separate, newly spawned process may filter tuples of the **spooled** sub-partition out of an overflowing hash table while the original process continues to add new tuples from the **pipelined** sub-partition to the table. Using multiple processes this way is less likely to disturb the flow of tuples arriving on the incoming pipeline than a method which cleans an entire hash table before processing any additional tuples.

During the building phase each join process constructs a coarse histogram of the hashed split values of the tuples that are used to build the hash table. The histogram is consulted when repartitioning is applied in order to chose an appropriate boundary between the **pipelined** and a **spooled** sub-partitions. The boundary between the **pipelined** and **spooled** sub-partitions can be varied dynamically by an individual join process, as only the local join process will be affected. This is an important capability as it allows the join process to iteratively increase the number of tuples that are assigned to a **spooled** sub-partition until an overflow condition is resolved. That is, the boundary between the sub-partitions of an overflowing partition may be changed incrementally until a sufficient number of tuples have been assigned to the **spooled** sub-partition. In fact, by changing the boundaries between the sub-partitions, a specific join process can handle even repeated instances of partition overflow.

If partition overflow has occurred, the scheduler is notified of the boundary value between the sub-partitions once a join process completes the building phase. The scheduler ensures that the probing relation is partitioned identically to the building relation (including sub-partitioning). In this way, all partitions of the probing relation will be subdivided in exactly the same manner as the building relation. Tuples belonging to the **pipelined** sub-partition of the probing relation are sent directly to the joining process and are used to materialize the result relation. Tuples from the **spooled** sub-partition of the probing relation are sent directly to a spooling process. Separating a partition of the probing relation into these two sub-partitions provides a basis for routing all probing tuples directly to the appropriate destination. This separation prevents the tuples destined for the spooler from having to be staged via the join process, saving a network transfer for each page that is appended to the spooled file. Since the probing relation will be the larger of the source relations, these savings can potentially be significant.

Later, after all join processes have finished processing the pipelined partitions of the probing relation, the scheduler reactivates all join processes using the previously spooled, overflow sub-partitions as input. The overflow sub-partitions are resplit using a new split function to ensure a uniform distribution of tuples among the participating join processes. This algorithm can continue to recursively handle additional overflows if they occur.

The described HP-join overflow handling technique can also be applied to the other hash-partitioned relational operators. Such extensions would actually involve a simplification of the described algorithm as the other operators (aggregate functions, duplicate elimination, etc.) only reference a single relation.

Partition Overflow Resolution Using Hashed-Loops

Resolving partition overflow via repartitioning will fail if more tuples have the same attribute value than can be accommodated in the memory available to a join process. If and when a partition overflow occurs, the histogram of hashed values serves the function of providing the information necessary to determine whether a particular hash table overflow can be resolved using repartioning. If the overflow occurred due to an unexpectedly large source relation, the hashed values in the histogram would be expected to be relatively well distributed. If this is not the case, then repartitioning may not be able to solve the partition overflow condition. In this case, the local join operator switches to a hashed-loops algorithm (see Section 4.3).

In the hashed-loops algorithm, the original partition is not separated into sub-partitions. Instead, all newly arriving tuples are sent to the spooler. These spooled tuples represent the remainder of the outer partition that will be iteratively staged into memory by the hashed loops algorithm.

Since the overflowing partition is not sub-partitioned, all tuples from the probing relation will be streamed into the join process during the probing phase of the join operation. These tuples represent the inner relation for the hashed-loops algorithm. All of these inner relation tuples are spooled to disk by the join operator after they have been compared against the tuples of the outer partition that are currently contained in the hash table. The join operator then iteratively stages portions of the outer relation into memory and builds a hash table, each time scanning the entire inner partition.

6.4.4. Updates

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.

The update operations use standard concurrency control and recovery techniques. Concurrency control is implemented with page-level locking and a centralized mechanism for detecting deadlocks. Recovery is being implemented with logging using the techniques described in [AGRA85].

6.4.5. Duplicate Elimination

Random, hash-partitioning is directly applicable to the process of eliminating duplicate tuples. In this case, tuples are partitioned by hashed values that are based on the composite attributes of an entire tuple. Having separated the source relations into discrete partitions, memory resident hash tables are used to filter duplicates from a stream of tuples.

6.4.6. Aggregate Functions

Hash-partitioning also provides the clustering necessary for executing aggregate function operations [JOHN82]. In this algorithm, tuples are partitioned based upon the hashed value of the composite of the operator's grouping attributes.

6.5. Memory Management

The allocation of memory in Gamma is controlled by a global memory manager. Query processes must receive permission from this manager before allocating memory from the large pool of shared memory that exists on each processor. The centralized memory manager maintains a table of the memory available on each processor and grants requests and accepts releases of allocation privileges by adjusting the entries in this table.

The memory manager does not, however, directly interact with the allocation routines of the individual processors. It is assumed that all query processes allocate and free memory according to the directions of the memory manager. Once a query process receives permission to allocate a given amount of memory, the process must physically acquire the appropriate amount of memory using local memory allocation utilities. Likewise, processes are responsible for returning memory via a local memory deallocation utility before notifying the memory management server that memory allocation privileges are being relinquished.

Prior to activating an operator process, an intra-query scheduler acquires an initial memory allocation for the process in accordance with the requirements of the assigned operation. Simple operator processes (selections) are granted sufficient memory to cover the requirements for the buffers associated with the output split tables. Each complex operator process is granted memory for output split table buffers and an initial hash table of moderate size. If an operator process exhausts it's initial memory allocation while building a hash table, the operator process requests additional memory privileges from the memory manager. The operator process is, however, prepared to handle the situation wherein additional memory is not available, i.e. partition overflow.

When a process completes execution, all acquired memory is locally deallocated and a message is sent to the memory manager releasing the related memory allocation privileges.

6.6. Intra-Query Scheduling Policies

Queries originate from query managers on host machines and are sent for execution to schedulers which reside on query processors. These schedulers traverse individual query trees, distributing, activating and controlling query operator processes on multiple processors. At the present time, the scheduler processes 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.

There are three ways in which the schedulers control the actions of individual queries. The schedulers:

- Map operator processes to selected processors.
- Determine the activation order for a query tree.
- Synchronize the interactions of operator processes.

6.6.1. Operator Process Placement

In Gamma, parallelism is applied to the execution of individual relational operators. Operators are represented by multiple processes that are distributed among the distinct processors. The task of mapping operator processes to processors is performed in part by the optimizer and in part by the scheduler assigned to control the execution of the query.

Operator Placements: Query Tree Exterior Nodes

The leaf nodes of a compiled query tree represent selections on permanent relations. For these nodes, the query optimizer specifies which processors should execute the selection operation. The optimizer determines this by comparing the selection predicates with the horizontal partitioning predicates stored in the schema for a relation. If a query's restriction predicates range over a horizontally partitioned attribute, then the optimizer may be able to limit the sites on which the selection operator must be activated. This activation strategy can improve query throughput by reducing the load that is placed on the system.

For example, assume that a relation has been horizontally partitioned by range across two disks on an attribute named 'q'. Furthermore, assume that the value 100 has been defined as the boundary between the two range partitions. In this case, the schema information associated with the relation could be used to determine the placement of a selection operation containing the condition:

q >= 50 and q < 80

Such a selection operator would be activated only on the processor associated with the first disk.

For selections on non-horizontally partitioned attributes, the optimizer prescribes that selection operators be activated on all processors that have attached disks. (Permanent relations in Gamma are always partitioned across all disks, maximizing the potential parallelism of a non-HPA selection operation.)

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. 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. In addition to enhancing the performance of subsequent selection operations, utilizing all disks for the storage of permanent relations allows tuples to be written in parallel to all the disks in the system. 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[28] processor.

[28] 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.

Operator Placements: Query Tree Interior Nodes

For the nodes in the interior of a query tree, the optimizer indicates a general strategy that should be followed for placing operator processes, but the scheduler makes the final placement decisions. Two static strategies are used for determining the number of partitions that are processed by operators in the interior of a query tree. In the first strategy, these operators are activated on **every** processor in the system. In the second strategy, the operators represented by nodes in the interior of a query tree are activated only on processors **without** disks.

The second strategy of utilizing only processors **without** disks for operators in the interior of a query tree has the disadvantage of reducing the level of parallelism that is applied to those operators. However, this disadvantage is counter-balanced by a number of attractive advantages. First, processors without disks are a less expensive resource. Also, processors with disks are potentially a bottleneck in a database system due to the processing requirements of reading, restricting and writing permanent relations. Utilizing processors without disks for the complex relational operators found in the interior of query trees means that these operators will not be competing for CPU and local communication resources (DMA access, network controller buffers, etc.) with those operator processes that must run on processors with disks attached. Thus, offloading complex operations to processors without disks enhances the performance of **selection** and **store** operations.

Placing complex operators on processors without disks does not result in significantly greater communications costs, as the partitioned data streams associated with multiprocessor hash-partitioned algorithms already require that tuples be transmitted to remote sites across the network. Whether the destination processor has a disk or not is of little consequence. With comparison to a policy that places complex operators on processors with disks, additional network traffic is only generated in the case where the source and destination of a partitioned data stream would otherwise have been located on the same processor.[29] These streams would account for 1/N of the total data stream volume of an N processor system. However, Gamma can transfer sequential streams of tuples between processes on two different processors at almost the same rate as that achieved between two processes on the same processor. Therefore, the response times accompanying the placement of complex operators on processors without disks are almost identical to those times that result from a policy of solely using processors with disks. In the process of transferring the additional data across the network (1/N streams), additional system resources are

[29]Recall, that the data messages sent between processes on the same processor are short-circuited.

consumed.

Although the optimizer prescribes the number of processes that should be associated with an operator in the interior of a query tree, a scheduler determines the final binding of an operator process to a processor. That is, the scheduler can attempt to place operators on the less heavily loaded processors in the system. If necessary, the scheduler can assign more than one operator process to a lightly loaded processor, if all other candidate processors are heavily loaded. Currently, the scheduler gives primary consideration to the memory utilization of a processor when activating operators in the interior of a query tree.

Operator Placements: Exceptional Conditions

The optimizer prescribes that a specific number of partitions be created as input to each of the operators in a compiled query tree. There are conditions that can prevent all of these partitions from being immediately processed by a complex relational operator. The following discussion first identifies these conditions and then discusses how these partitions of tuples are eventually processed. There are two circumstances that can prevent partitions from being pipelined to a process executing a specific complex operator:

- The optimizer may detect that the size of the input relation chosen as the building relation will exceed the total memory available to the operator on all processors.
- The scheduler may find that the system is too heavily loaded for a specific operator process to effectively process a specific partition of tuples.

In both cases, selected partitions are spooled to temporary files and are rematerialized at some later time, upon the request of the scheduler. In the case of a join operation, these spooled partitions always come in pairs. That is, when a decision is made to spool a partition of a building relation, the corresponding partition of the probing relation is also spooled. It should be noted, however, that the spooling of partitions in a join operation is always dependent on the building relation. That is, the characteristics or size of partitions of the probing relation are never a factor in the decision to spool partitions during a join operation.

Very Large Building Relations

In the first case requiring **spooled** partitions, the optimizer detects that the building relation will exceed the memory resources available to a set of operator processes. Under these circumstances, the optimizer marks a sufficient number of the input partitions as being **spooled** partitions. The required number of partitions is determined by dividing the expected size of the building relation by the amount of memory necessary for a single, moderately sized hash table.

Heavily Loaded Systems

The second case requiring the use of **spooled** partitions results from an unanticipated lack of system resources due to the demands of other concurrently active query processes. If a scheduler finds that a candidate processor has insufficient memory to support the initial memory allocation required by an operator, then the scheduler may assign the operator process to another, less heavily utilized processor. If another processor cannot be located, the scheduler has the option of assigning multiple partitions to a single processor if sufficient memory is available.

This process is very similar to the **bucket tuning** technique used by the Grace algorithm. Assigning a combination of partitions does not require a reorganization of a split table. Instead, multiple split table entries are simply mapped onto a single output buffer. This mapping of multiple split table entries to a single output buffer effectively makes the partition tuning transparent to the operator processes which produce the partitions of tuples.

If the current load on the system prevents a scheduler from placing an operator process (or reassigning a partition), then that specific operator process is not activated, but instead the affected partition that the operator process would have processed is changed from a **pipelined** partition to a **spooled** partition. While the availability of memory resources is the basis of the current algorithm for restricting the placement of complex operators, other factors could also be considered such as the utilization of a processor's CPU or network interface device.

6.6.2. Query Tree Activation Sequence

Gamma's algorithm for traversing a query tree and activating operators provides a basis for significant intra-query concurrency. The chosen activation sequence also permits data to be pipelined between operators at adjacent levels of a compiled query tree.

When activating nodes of a query tree, conservation of system resources argues for a strict bottom-up activation of query trees. That is, since data availability will flow bottom-up in a query tree, activating operators prematurely at higher levels in a query tree will unnecessarily consume resources that are allocated to the active, but waiting processes. An example of such a resource is the stack space of an active process.

Providing pipelined data streams, however, require some form of top-down activation sequence. For a pipeline to exist, an operator represented by a parent node in a query tree must be prepared to read from a pipe when the operators represented by the children of that node begin producing data. A top-down activation sequence satisfies this requirement.

In Gamma, queries are activated by intra-query schedulers via modified in-order traversals of left-deep query trees. These traversals conserve system resources by activating a query tree bottom-up, but pipelines are enabled by ensuring that immediate ancestor nodes are activated "as needed". These contradictory goals are made possible by separating the activation of a query operator into three stages. In the first stage, a query process is started and permitted to accept input. In the second phase, the process is enabled to produce output tuples. In the last stage, the process is reactivated to process any remaining **spooled** partitions. For complex, hash-partitioned operators, the first and second stages of activation may be separated in time by the activation of other operators in a query tree. For selection operations, the first and second stages of activation are combined and executed simultaneously. In all cases, before an operator is enabled to produce output tuples, the parent of the operation is guaranteed to be already active and ready to accept input tuples.

Operator Activation: Stage One, Startup

In the first stage of activation, a scheduler initiates a process to represent the current operator on each of the processors that have been selected to support the operation. After acquiring initial memory allocation privileges for each process, the scheduler sends a message describing the operation to each processor where the operator will be active. Such packets contain a complete description of a single operation. Given such a packet, a query process can independently execute the specified operation.

Operator Activation: Stage Two, Output Enabling

In order to construct pipelines between operators, the receiving operator processes must be activated prior to the time when split tables are initialized and provided to the processes that will be feeding tuples to a specific pipeline. However, in order to conserve resources, the activation of query operators are delayed until the last minute. In this manner, an operator is activated just prior to the time when tuples from it's source relations are materialized.

However, delaying operator activations in this manner results in operators being activated before their split tables have been completely initiated. For instance, a join operator in the interior of a query tree can be activated so that a hash table can be built from the tuples of the building relation. Since the join operation will not produce any result tuples during this phase of the operation, it is possible to activate the join operation without providing an initialized split table. When the join operation is synchronized between the building and probing phases, the scheduler provides the initialized split table. In this manner, a scheduler can delay activating the operator that will receive the result tuples from a join operation until the time at which the join operation is prepared to enter the probing phase.

Operator Activation: Stage Three, Reactivation

As earlier described, **spooled** partitions can be produced during the processing of a complex relation operator. When a query process finishes execution, it sends a completion message to the scheduler. If any **spooled** partitions remain, the scheduler reactivates the operator process and assigns it one of the remaining **spooled** partitions (or pair of partitions, in the case of a join). Otherwise, the scheduler replies to the query process with a message indicating that no further work exists for the current operator.

6.6.3. Control of Operator Processes

In the course of activating operator processes and assigning them to processors, the scheduler must ensure that the operator processes execute in a coordinated manner. This synchronization basically takes two forms. First, the scheduler initializes the routing information contained in the provided split tables. The second form of synchronization ensures that all the operator processes for a join operation transit from the building to the probing phases at the same time.

Runtime Initialization of Split Tables

The intra-query schedulers are responsible for initializing the routing information that is associated with each entry in the split tables of a query tree. When an operator is activated, the scheduler ensures that the split table provided contains all the necessary routing information. This information will already have been collected by the scheduler during the activation and/or synchronization of the operator's parent node. That is, after activating the processes that will be reading from a pipeline, the scheduler places the destination addresses of these consumers into the partition entries of the split tables that are provided to the tuple producers. Operator processes can then send the tuple production for any given partition directly to the appropriate destination. Furthermore, the query processes can treat the split tables as static data structures that will not change as a query operation proceeds.

Synchronization of Join Phase Transitions

Join operator processes are synchronized after the building phase has completed and before the probing phase is initiated. Following this synchronization of the join processes, the scheduler activates the operator which will be producing the partitions of the probing relation for the join.

As a part of this synchronization, each join process sends a message to the scheduler when the building phase completes. In addition to signaling completion of the building phase, these messages also contain two additional kinds of information. Each message indicates whether or not a particular join operator process encountered partition overflow. Also, the message includes the bit-vector filter constructed during the building of a hash table.

Cost of Scheduler Synchronization

An important characteristic of this synchronization algorithm is the simplicity of the interactions between the scheduler and operator processes. The net cost of activating and controlling an operation in Gamma is four messages per operator process. (The building and probing phases of a HP-join operation are considered separate operations for purposes of control. The net cost of controlling a join operation is, therefore, eight messages per site.) All other data transfers can proceed without further control intervention by the scheduler.

6.7. Inter-Query Scheduling Policies

Queries originate from host processes called query managers in the form of compiled query trees (see section 6.2). Multiple intra-query schedulers can simultaneously be coordinating the execution of the separate queries received from different query managers. As unrestricted inter-query concurrency could lead to unproductive competition for system resources and reduced levels of query throughput, Gamma provides a dispatcher process that controls a queue of incoming queries.

In the case of a multiple site query, the query manager (QM) [30] establishes a connection to an idle scheduler

[30] The QM provides an ad-hoc query interface for users.

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, I/O and memory utilization at each processor. When a query completes, 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.

The dispatcher currently services the queue of incoming queries in FIFO order. However, different service policies could be implemented by the dispatcher. Estimated resource requirements or query priorities could form the basis for such policies.

Certain queries are not directed to either the dispatcher process or to a scheduler. When the query optimizer recognizes that a query will be executed at a single site with the results being returned to the host, the query manager can safely send the compiled query tree directly to the appropriate query processor for execution, bypassing the dispatcher and scheduler. Such queries will only contain a single node and will not require the traversal of a query tree. Range queries and exact match queries on a horizontally partitioned attribute are the only candidates for single site execution.[31]

6.8. Examples of Gamma Execution

6.8.1. 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. The IPC socket associated with the CM process is at an established Internet address. Thus, Gamma can be run from any machine connected to the ARPA Internet. 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.

[31]Bypassing the dispatcher and scheduler enhances the throughput and response time for such short transactions. Queries that update a horizontal partitioning attribute, however, cannot be handled in this way because the update may potentially cause a tuple to migrate to another disk.

6.8.2. 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[32] 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 [HEYT85a, HEYT85b]. A lock manager within the CM ensures catalog consistency.

6.8.3. Query Execution Examples

The following examples of query execution assume the existence of a supplier-and-parts database with three relations:

Suppliers (supplier#, supplier-name, status, city)
Parts (part#, part-name, color)
Supplier-Parts (supplier#, part#, quantity)

The system configuration used contains two processors with disks and one processor without a disk. The Parts and Supplier-Parts relations have been randomly distributed across both disks in the system. The Suppliers relation has been range partitioned according to the following criteria:

[32]Attribute catalog file, relation catalog file, etc..

Distribution Condition	Processor #
supplier# ≤ 100	1
supplier# > 100	2

Clustered B-tree indices are assumed to exist for the Suppliers and Parts relations. The relations are respectively indexed on the supplier# and part# attributes.

Single Site Retrieval

The following query references an attribute of the Suppliers relation that has been horizontally partitioned. Although this query represents a range query that can be executed at a single site in the system, exact match queries can be handled in identical fashion.

```
range of S is Suppliers
retrieve (S.all)
where S.supplier# >= 50 and S.supplier# <= 75
```

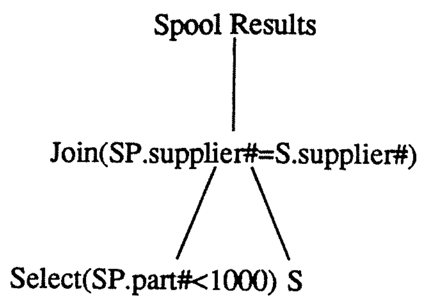
Since the restriction references a HPA, the optimizer compares the predicate against the partitioning predicates. For this range query, the optimizer can determine that the results for the query can be obtained from a single processor. Therefore, the query can be sent directly to a single query processor bypassing the query dispatcher and scheduler.

Query with a Single Join Clause

As an example of the control and synchronization involved with a multiprocessor HP-join operator, consider the following query which retrieves the names of all suppliers who supply certain parts.

```
range of S is Suppliers
range of SP is Supplier-Parts
retrieve (S.name)
where S.supplier# = SP.supplier#
and SP.part# < 1000
```

The query tree corresponding to this query is depicted in Figure 6.9.

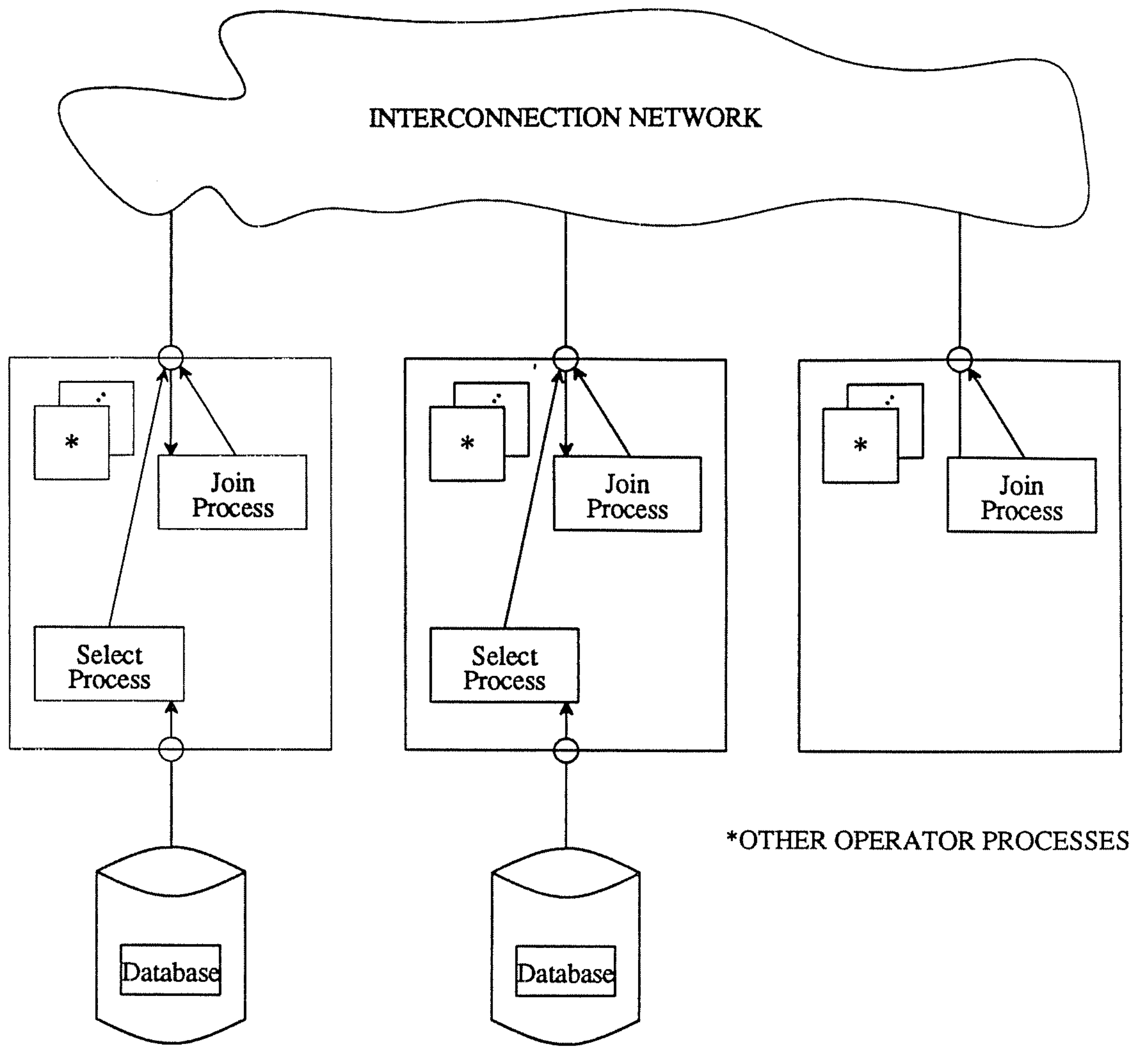


A Left-Deep Gamma Query Tree
Figure 6.9

Note that the optimizer has placed the selection operation on the Supplier-Parts relation in the leftmost leaf of the compiled query tree. This placement reflects the optimizer's decision that the relation resulting from the selection will be smaller than that produced by a scan of the Suppliers relation. The validity of this conclusion is important because the relation produced by the leftmost child of a join operation in Gamma will be used to build in-memory hash tables. The execution of this query proceeds as follows:

(1) The scheduler initiates a join operator on all three processors in the system. Upon activation, each join operator initializes a single, local bit vector to the value zero and allocates an empty hash table. Then each individual join operator reads from its single input stream and add tuples to the hash table that it is constructing. For each tuple, a bit representing a join attribute value is set in the local bit vector.

(2) Having initiated the join operator, the scheduler activates the selection that produces the building relation. The split table for this selection operator is initialized to produce three output streams partitioned according to the hashed values of the supplier# join attribute. The various processes that will be active at this point are displayed in Figure 6.10. The figure depicts an association of operator processes with processors as would exist during the scan of the Supplier-Parts relation. This figure depicts a strategy wherein join processes are activated on every node in the system in order to take advantage of the maximum parallelism possible for the join operation. (Section 6.9 compares the actual performance of this placement policy with an alternative that places join operators only on those processors without disks.)



Active Processes During a Join Query

Figure 6.10

(3) The scheduler waits for completion notices from all the selection operators. The join operators detect the fact that the selection on the Supplier-Parts relation has finished by counting "end of stream" markers.

(4) When a join operator collects all the "end of stream" markers from the Supplier-Parts' pipelines, it sends a message to the scheduler indicating that the building phase of it's join operation has been completed. As a part of this message, the join operator returns the bit vector filter that it constructed during the building phase. This message also includes information concerning the disposition of any hash-table overflow conditions.

(5) Once the scheduler receives notices from all join operators that the building phase has completed, a single spooling operator process is started that controls the return of the result relation to the host. Then, a message is sent from the scheduler to all three join operators that initiates the probing phase of the hash-join algorithm. This message includes the split table that controls the output stream production of the join operation.

(6) After activating the probing phase, the scheduler activates the selection on the Suppliers relation. As a part of the activation process, a packet containing the collected set of bit vector filters is sent to each selection process by the scheduler. One separate bit vector is included for each of the three partitioned streams that will be produced.

(7) As the join operators receive tuples from the selection of Suppliers, the hash table is probed for matching tuples from the Supplier-Parts relation. Matching tuples are joined and output to the spooling process which forwards them to a Query Manager on the host.

(8) The scheduler is notified as the selection, join and spooling operator processes complete.

6.9. Performance Evaluation

There are many facets of the Gamma design that could be measured and analyzed by a performance evaluation. Does the architecture provide adequate levels of aggregate I/O bandwidth? Can processors without disks be effectively used to support join operations? Pipelining tuples between distributed processes can be used to reduce the I/O costs and response time of complex queries. Can these pipelines be established and controlled in the Gamma environment at a reasonable cost in time and resources? Can the conflicting demands of multiple competing queries be satisfied while maintaining high query throughput?

A complete performance analysis of Gamma would have to address these and many other issues. Our current analysis has much more modest goals. The following tests are first of all, single-user benchmarks. Single-user benchmarks are an effective means of identifying and evaluating the performance impact of the separate components of a database system. As such, these benchmarks are a useful tool for evaluating the feasibility of the hardware and software architecture of Gamma. In particular, these single-user tests can identify bottlenecks in the system that might be obscured by the added concurrency of a multi-user environment. Also, even with the constraints of a single user environment, the various forms of intra-query concurrency in Gamma make even the analysis of a single query a complex task.

In the future, a more exhaustive and extensive performance analysis of Gamma will be performed. The impact of multi-user queries, concurrency control and recovery, a more extensive set of operators, and more complex queries will all be examined at that time.

In the following section, the results of our current performance evaluation of Gamma are presented. All of the 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 point at which the query was entered by the user and the point at which it completed execution.

By design, partition overflow did not occur during any of the following performance tests. That is, the compiled query plans generated by the query optimizer specified that a sufficient number of partitions be used such that partition overflow did not occur during join operations. For these initial tests, the occurrence of partition overflow would only have contributed an additional, independent variable to the tests that would have compromised the value of the tests as a means of identifying the specific determinants of the performance of single user queries in the Gamma environment.

For this reason also, join attributes with uniform distributions of values were also used. This constraint guaranteed that the building relations for the following join queries could effectively be partitioned for processing by Gamma. An effective hash function should be able to generate uniform distributions of hashed values even for source relations containing non-uniform distributions of distinct join attribute values. As previously mentioned, the constraint that relations contain a significant degree of unique join attribute values is only important for the building relation, as partition overflow is independent of the characteristics of the probing relation. While the hash function currently employed by Gamma appears to effectively generate uniform distributions of value, we will evaluate the capability and generality of the hash function to randomize various non-uniform distributions of attribute values in future investigations.

In Section 6.9.5, we investigate the ability of Gamma to handle partition overflows. These tests will demonstrate that Gamma can minimize the performance impact of partition overflow even in those cases when the building relation of a join operation is significantly larger than predicted by the query optimizer.

6.9.1. Test Database Design

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 apiece. Each tuple contains thirteen, four byte integer attributes followed by three, 52 byte character string attributes. As discussed in [BITT83], these relations enable one to generate a wide range of retrieval and update queries while precisely controlling the selectivity and the number of result tuples.

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 of use to query i+1 in a buffer pool. The response times for the join queries are the average of a set of two similar join queries that accessed different relations. The variance of response times for each of the tested join queries was less than 0.4 seconds^2. The response times for the selection queries were averaged from a set of four queries that accessed different relations. The variance of response times for each of the tested selection queries was less than 0.02 seconds^2.

6.9.2. 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

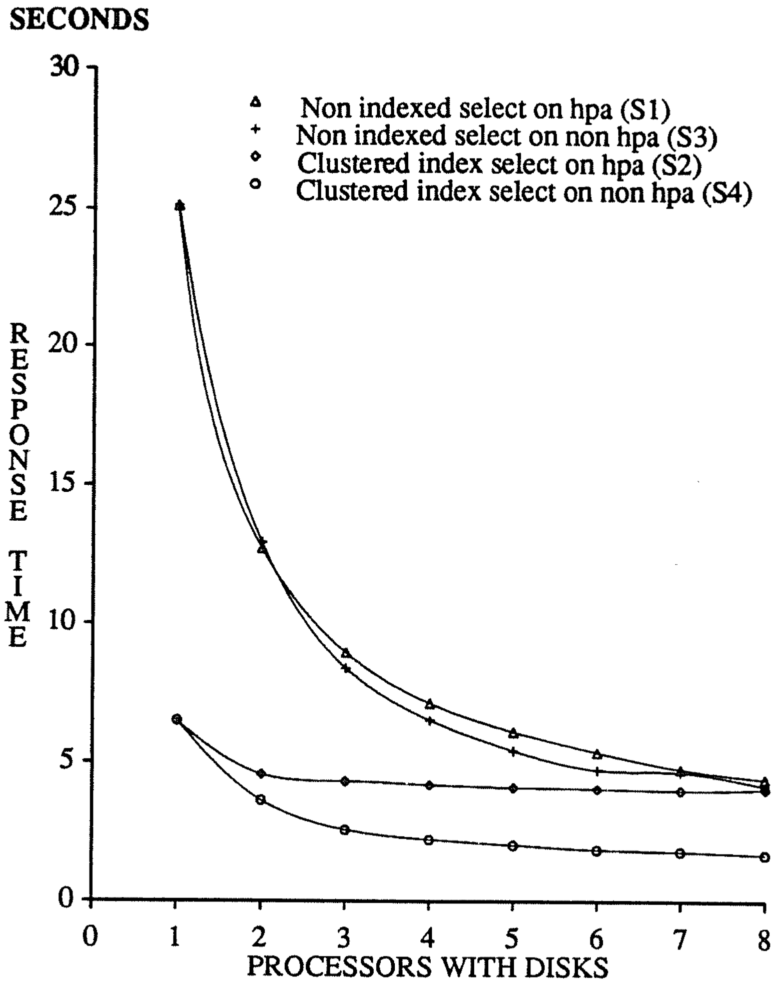
We restricted classes S1 and S2 further by designing the test queries such that the operation is always executed on a single site. This was accomplished by having the horizontal partitioning ranges cover the qualifications of the selection queries. Since queries in both classes S3 and S4 reference a 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 queries tested were of the form:

```
S1,S2:
retrieve into temp (tenKtup.all)
where tenKtup.Unique1 > m and tenKtup.Unique1 < n

S3,S4:
retrieve into temp (tenKtup.all)
where tenKtup.Unique2 > m and tenKtup.Unique2 < n

(m and n are constants between 0 through 9999 and n-m = 1001.)
```

The results from these selection tests are displayed in Figure 6.11.



10,000 Tuple Selection Query Response Times
Figure 6.11

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 6.11 contains a number of interesting results. First, as 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.[33] 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 6.11. While both have approximately the same response time, S1 would have a significantly 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 anomalous shape of the S3 curve for systems with 7 or 8 disks can be attributed to the fact that the seventh and eighth disks that were added to the system have only 82%[34] the performance of each of the other six disks. With evenly partitioned source relations, these slower disks increased the time required for scanning the source relation.

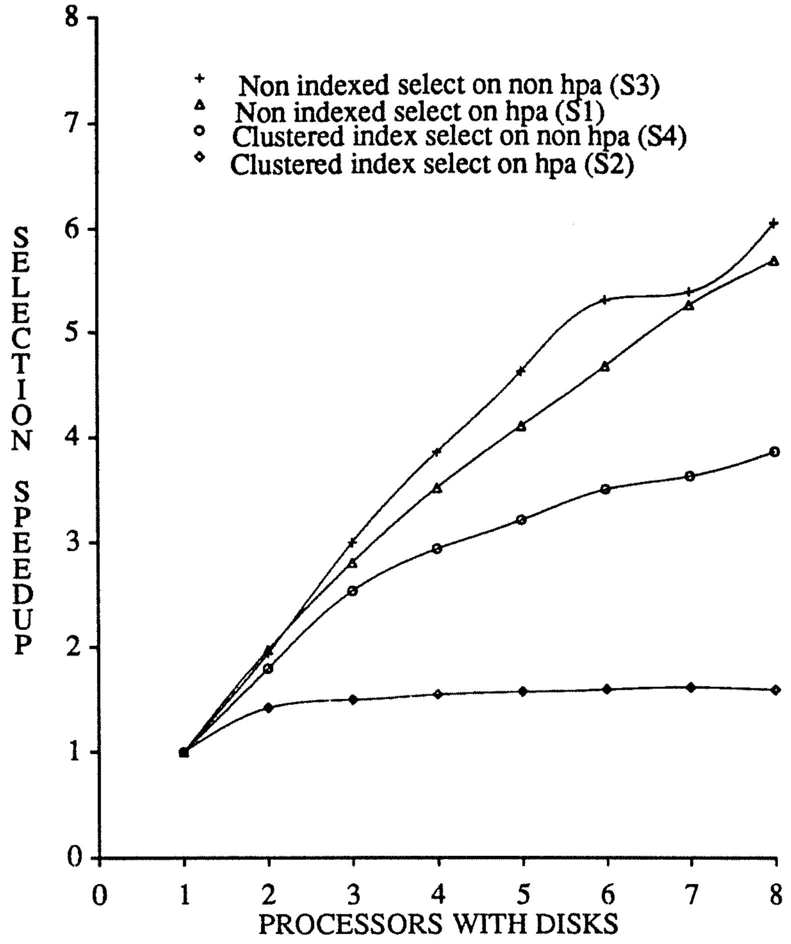
[33]The source relations accessed by this query have been uniformly partitioned across all disks.

[34] 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.

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.

In Figure 6.12, the performance speedups obtained for the selection query are illustrated.



Selection Query Performance Speedup
Figure 6.12

With the exception of the speedups for the non-homogeneous seventh and eighth disks, the speedup factors for the non-indexed selections S1 and S3 are fairly close to linear. As previously noted, selection S1 performs better due to the fact that more than one processor is incurring the cost of distributing the result relation. As described earlier, the lack of speedup for S2 is to be expected, for this query always executes on a single processor and accesses a constant amount of data independent of the total number of processors in the system. However, the decline in improvement in the response time for selection query S4 as additional processors are employed is troublesome. 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.[35] 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 the 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.

[35] A round-robin strategy is used to insure that the result relation is distributed uniformly, no matter how many tuples each site produces. In this particular test, we could have achieved the same result without any network traffic by storing the result tuples on the sites at which they are produced.

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. A side benefit of this new strategy will be to reduce the number of partially filled pages that are sent at the end of the query (see [DEWI79b] for another look at the same problem). A drawback is that the distribution of result tuples may not be as uniform às before.

A solution to the problem of partitioning result relations without triggering network congestion is explored in Chapter 7. The performance of a modified Gamma system that includes a solution for this problem is presented in Chapter 8. We choose to leave this initial, rather negative result in this chapter, however, for a couple of reasons. First, it illustrates how critical communications issues can be. One of the main objectives in constructing the Gamma prototype was 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, the initial Gamma prototype 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 (with a 10 MHz CPU, 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.

6.9.3. Ad-Hoc 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:

```
retrieve into temp (tenKtupA.all, tenKtupB.all)
where (tenKtupA.Unique2A = tenKtupB.Unique2B)
and (tenKtupB.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 is initiated. This phase constructs a hash table using tuples resulting from the selection of the tenKtupA relation. One hash table is built 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.

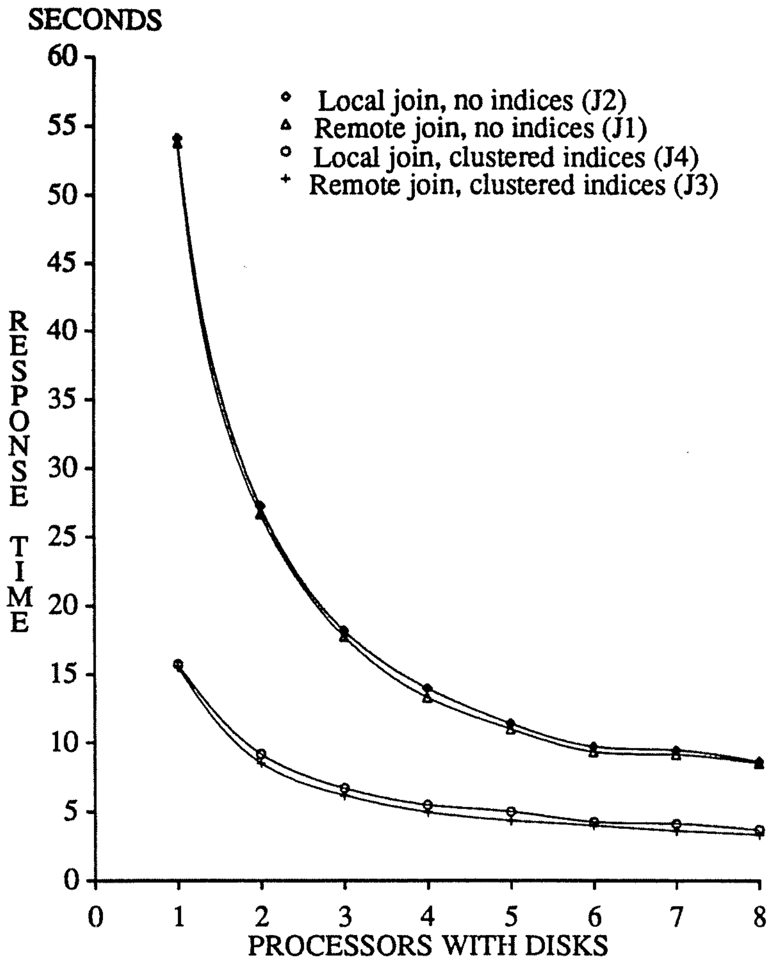
Since Unique2 is not the HPA for either source relation, all sites participate in the execution of the selection operations.[36] The relations resulting from the selection and join operations each contain 1,000 tuples.

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 6.13.

[36] Unique2 was purposely chosen to maximize the rate at which tuples were produced in order to insure that the selection was not a bottleneck.



Join Query Response Times
Figure 6.13

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. The response times of join queries J3 and J4 are much faster due to the fact that the selection operations used clustered indices.

Figure 6.13 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 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.

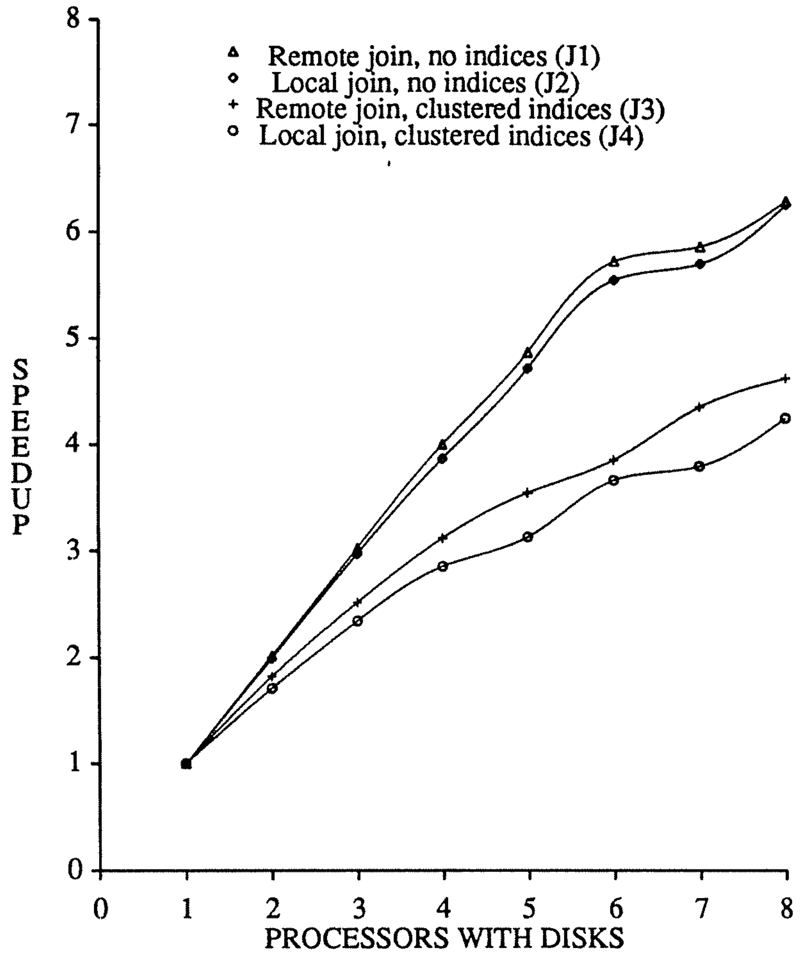
Since twice as many processors are used by the remote join design, one might wonder why the response times for remote joins were not half those of the corresponding local joins. Since the building and probing phases of the join operator are not overlapped, the response time of the join is bounded by 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 6.13, the IDM500 took 84.3 seconds for J2 joins and 14.3 seconds for joins of type J4.

6.9.4. Speedup of Join Elapsed Times

In Figure 6.14, response-time speedup curves are presented for the join tests described in the previous section.



Join Query Performance Speedup
Figure 6.14

These results confirm our hopes that multiprocessor, partitioned hash-join algorithms can effectively provide a basis for a highly parallel database machine.

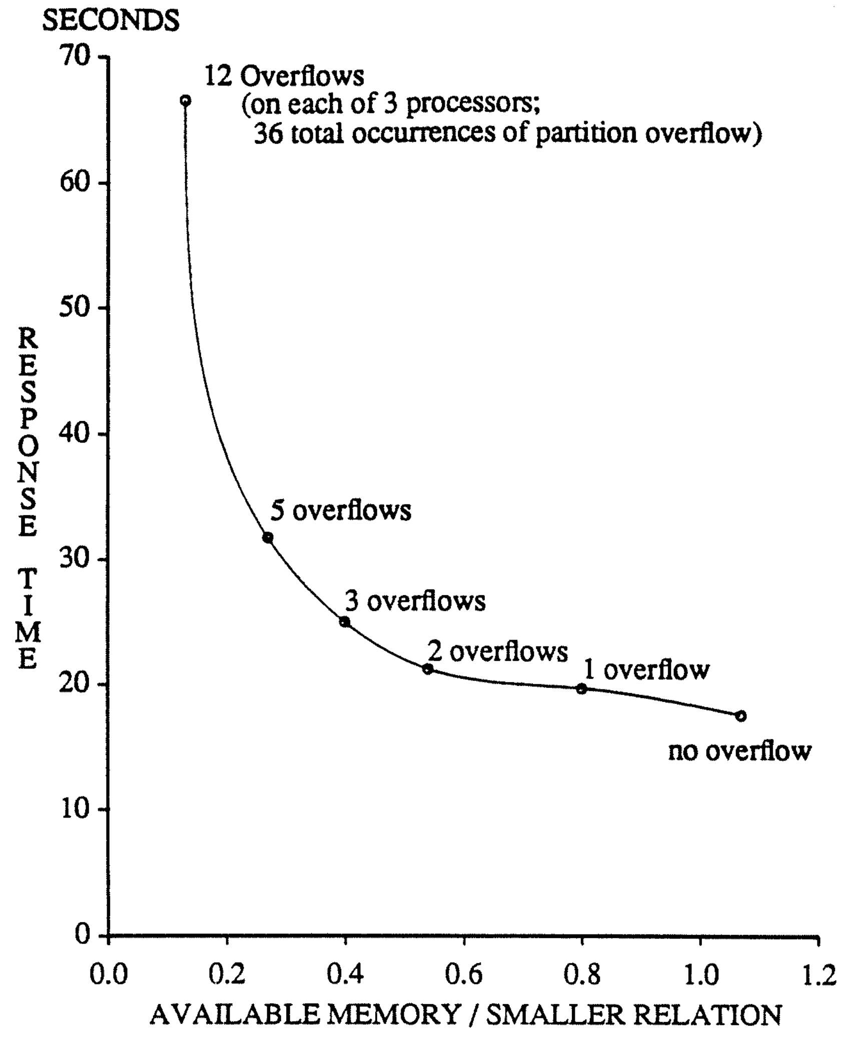
The anomalous shape of the speedup curves for systems with 7 or 8 disks reflects the increased time that is required for scanning each of the evenly partitioned source relations using the two slower, non-homogeneous disks. All of this additional time is directly reflected in increased response times 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 send/receive eight 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.

6.9.5. Partition Overflow Resolution

In Figure 6.15, the impact of encountering multiple partition overflows during the course of the previously tested 10,000 tuple join query is demonstrated for a Gamma system containing 3 processors with disks, 3 processors without disks and a scheduling processor.



Join Query With Multiple Partition Overflows
Figure 6.15

For this test, the join operations were always executed remotely on the processors without disks. Therefore, tuples belonging to overflow sub-partitions were spooled to remote disks during the resolution of the partition overflow. In this test, the total amount of memory available on the join processors (processors without disks) was adjusted to induce varying repetitions of partition overflow. In all cases, this available memory was equally divided among the three join processors. As a result, all join processes encountered identical partition overflow conditions.

The shape of the graph in Figure 6.12 illustrates the fact that Gamma is currently using a distributed version of the Simple hash-partitioned join algorithm (see Section 4.3) to resolve occurrences of partition overflow during join queries. Whenever the size of an overflow sub-partition of a building relation exceeds the amount of memory available in the system, the Simple hash-join algorithm is guaranteed to trigger an additional partition overflow. These subsequent overflows occur when the overflow sub-partition is subsequently used as input for the next pass of the join operation, i.e. when the join operator processes are reactivated for purposes of joining the overflow sub-partitions of the building and probing relation. In the current test, as the available memory was decreased, the partition overflow mechanism was increasingly invoked in a repeated fashion due to the nature of the Simple hash-join algorithm. That is, after the initial overflow occurred, the partition overflow resolution mechanism did not recognize the fact that the overflow sub-partition of the building relation exceeded the memory available to the current join operator processes.

The performance of the partition overflow resolution mechanism will be modified in the future to use a distributed Hybrid hash-join algorithm. For those cases where repartitioning is applicable, a Hybrid hash-partitioned join algorithm can greatly decrease the response time of resolving multiple overflows by repartitioning the overflow sub-partitions into a sufficient number of separate, smaller sub-partitions prior to attempting to join any of the overflowing tuples. This repartitioning is possible because the minimum amount of memory available to the join operator processes is known by the end of the building phase of the initial join operation. Therefore, a basis for repartitioning the spooled overflow sub-partitions can be decided prior to the reception of tuples from the probing relation. In this manner, only the spooled, overflow sub-partition of the building relation has to be repartitioned.

CHAPTER 7

A SIMULATION MODEL OF THE GAMMA DATABASE MACHINE

7.1. Simulation Rationale

In this chapter, we present the results of using a simulation model to explore the performance of Gamma for larger multiprocessor systems with varying hardware capabilities. Measurements taken from the Gamma prototype and applied to the model provide a basis for an accurate simulation of the performance of Gamma.

A larger spectrum of issues that affect the feasibility of the Gamma design for large multiprocessors deserve attention. The interactions of multi-user queries, concurrency control and recovery, and alternative scheduling algorithms are all appropriate topics for investigation with the simulation model. In a similar fashion, alternative hardware configurations and architectures require study. In particular, it will be important to investigate the impact of varying the number of disks associated with each processor in Gamma. The factors that determine the optimum ratio of processors with and without disks should also be examined.

In this chapter, however, we will initially focus our investigation on those software components that were identified as significant (or potential) performance bottlenecks by the single user Gamma benchmark tests. That is, round-robin partitioning, the effective bandwidth of operators that access base relations, and network congestion will be examined. Next, the impact of altering the capabilities of the existing hardware components of Gamma is illustrated. Finally, we examine how well the Gamma design extends to larger configurations containing as many as a hundred disks and two hundred processors.

7.2. Simulation Tools

The simulation models have been built using the facilities of the the Starpak simulation package [GERB81]. The Starpak simulation package was originally written for use with the Starmod programming language [COOK79] and has been modified for use in conjunction with UW-Modula [FINK83]. The simulation package was influenced by the designs of the earlier simulation packages of [KNUT64] and [ARMS68].

The simulation package provides tools for maintaining a sequenced set of future events. The package also provides queuing variables called **stores** that enable the modeling of contention for the shared, finite resources of a system. These features in combination with the process control structures of Modula provide an environment that enables the simulation to closely model a distributed system.

The process scheduling primitive of Starpak provides that a process can be delayed for a specified period of time. The granularity of time is under a user's control. All processes scheduled for a given time execute before any processes scheduled for later times. The simulation clock is advanced to the time of the next scheduled process when all current processes have been suspended or terminated. All processes scheduler to run at a particular time appear to run in parallel.

The **store** variables contain a discrete amount of a resource that is to be shared among competing processes. A process may control all or part of a **store** by request. A requesting process must specify the number of units of the **store** that it wishes to control. If the request cannot be satisfied, i.e. a process has requested more units than are currently available in a particular **store**, then that process will be queued until the request can be met. These queues of waiting processes are sorted by priority and arrival time. Higher priority processes are given preference when **store** units become available. Within the same priority group, a first-in, first-out scheduling policy is enforced. The priorities that control a process' interaction with a **store** are attributes of the simulation package and not extensions of the Modula language. Processes can control their own priorities dynamically as a simulation progresses. A process can release control of all or part of the **store** units it controls. Multiple processes may concurrently hold units within the same **store**.

Occupancy and utilization statistics are maintained for the **store** variables during a simulation. Additionally, the simulation package provides objects called **table** variables that collect information concerning the distribution of values that are assumed by specified user variables during the course of a simulation. The mean and standard deviation for such variables is available at the end of a simulation. A histogram displaying the distribution of values assumed by a given variable can also be displayed.

7.3. Multiprocessor Model

The simulation model is composed of a number of distinct modules that represent the various hardware and software components of Gamma. These modules provide a functional interface to high level abstractions of the components of a database machine, and they encapsulate the implementation and policies that control the various components. The modules are, in turn, implemented using the facilities of the simulation package. The procedural interface provided by the modules enables the implementation of a module to be changed transparently with respect to the other modules in the system. For example, the one bit stop-and-wait protocol module could be replaced by a sliding bit window protocol module that exported the same procedural interface.

The hardware components that are represented in the model are intended to be examples of current, commercially available components such as those used by the Gamma database machine. Unless otherwise stated, all modeled systems have 2*N +1 processors where N is the number of disks in the system. As disks are assumed to be associated on a one to one basis with processors, each system will have N processors with disks and N processors without disks. The single additional processor in excess of 2*N processors is used to support the query scheduler processes.

The processors in the current simulation are loosely coupled via a token ring network. Processes transfer packets between main memory and the network interface of a processor using the FIFO services of a DMA controller. A limited number of read buffers are assumed on each network interface. All of the critical determinants of network access and performance are parameters to the model and can be altered for each separate simulation run.

A one bit stop-and-wait communications protocol provides for the reliable delivery of messages between pairs of processors. As in Gamma, ports are used as queuing and addressing primitives on each processor. The CPU costs, message acknowledgment strategies, and retransmission policies are modeled after those provided by the NOSE operating system[37] and the CRYSTAL nugget [DEWI84b] with the exception that the simulation model does not piggyback acknowledgements. Messages and acknowledgements are allowed to fail and retransmissions and backoff algorithms are used to guarantee the eventual delivery of a reliable message. Messages may fail when a receiver's network interface is not able to accept a message. This may occur when all the buffers on the receiver's network interface are full or when the network device has not yet been reset following a previous network access.

[37]The transmission of a reliable 2 kilobyte message across the network to a remote destination requires 12.4 milliseconds. A ''short-circuited'' message of the same size can be sent to another process on the same processor in 4.4 milliseconds.

An operating system similar to the NOSE kernel used by Gamma is assumed. The processes on each processor are scheduled using a nonpreemptive scheduling policy. Only interrupt handlers are able to preempt control of a processor. The performance of the processors in the system is a parameter to the model and is used to scale the amount of time that a process will hold control of a specific processor.

Write operations to a disk are always blocking. Read operations may selectively block or alternately a single page read-ahead scan can be opened on a file. Latency times representing sequential or random disk I/O are attributed depending on the access pattern for a disk. Interleaved requests from different file scan instances are attributed with random latency times. Non-interleaved disk accesses on the same file scan are attributed with latency times representing sequential disk I/O. These employed disk times[38] represent the measured performance of Gamma using Fujitsu 8 inch disk drives.

All queries in the simulation study are assumed to reference relations of the Wisconsin Database (see section 6.9.1). Tuples in all source relations are 208 bytes long. Source relations with either ten thousand, one hundred thousand, or a million tuples are used by the simulated queries.

In the following analysis, the resource usages of various hardware components are presented in the context of the executions of individual queries. These resource usages represent the absolute amount of time that a resource was busy during the execution of a query. By comparing this time to the response time of a query, the relative utilization of that resource during the query can be calculated. CPU usage includes all time spent executing user or system routines, including interrupt handlers. Average and aggregate CPU usages are presented separately for processors with and without disks. Aggregate CPU usage is calculated by summing the resource usages of all similar CPUs, i.e. those CPUs with or without disks. Disks are considered busy from the time an I/O operation is initiated until the operation completes.[39] Both average and aggregate disk usages are presented. Aggregate disk usage is calculated by summing the disk usages of all disks in a system. The token ring network is considered busy between the time when a token is acquired until a message has been transmitted and the token has been released.

[38]The modeled, combined seek and latency times for sequential and random disk accesses are respectively, 12 and 28 milliseconds. An additional 2 milliseconds of CPU time is required to control the I/O operation. The disk provides a transfer rate of 1.8 Mbytes/second.

[39]There is some overlap between CPU and disk usages because the CPU time required to control an I/O operation is included in both usage categories.

7.4. Analysis of Selection Query Performance

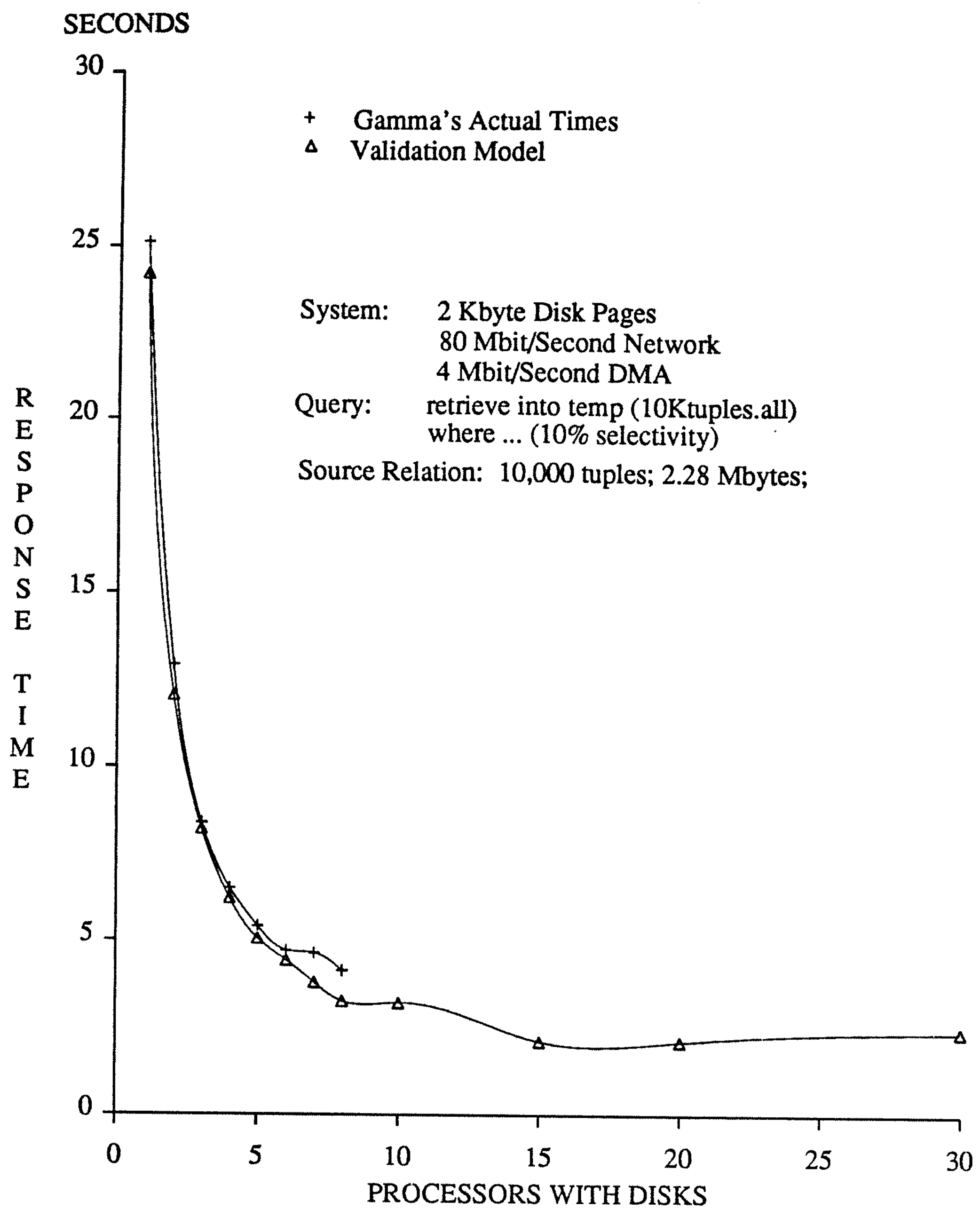
Selection queries that produce a result relation are the simplest query processing task that utilize most of the software and hardware components of Gamma. An analysis of the simulated performance of selection queries can thus provide a basis for identifying important determinants of the performance the query processing system. As indices are an effective way of avoiding some of the potential bottlenecks in a database system, indices would only make the simulated queries run faster and would not significantly help in the identification of performance bottlenecks. Thus, only selection queries on a non-indexed relation will be analyzed. Also, all selection operations will reference non-HPA attributes. (This ensures that each query accesses data on all disk drives.)

In the following section, we first compare the performance of the modeled system with measurements taken from Gamma. Then, the simulation model is used to explore alternative query processing and scheduling algorithms that provide enhanced performance. Finally, the impact of varying the number and capability of the available hardware resources is investigated.

7.4.1. Validation of the Simulation Model

An initial task facing any simulation study is demonstrating the accuracy of the model. In the current simulation study, we are modeling discrete events, i.e. the response time of individual queries. For this reason, the accuracy of the model will be demonstrated by comparing the model with measured response times of identical queries run on Gamma.

In Figure 7.1, the measured performance of Gamma for a selection on a 10,000 tuple relation is compared with the performance predicted by the Gamma simulation model.



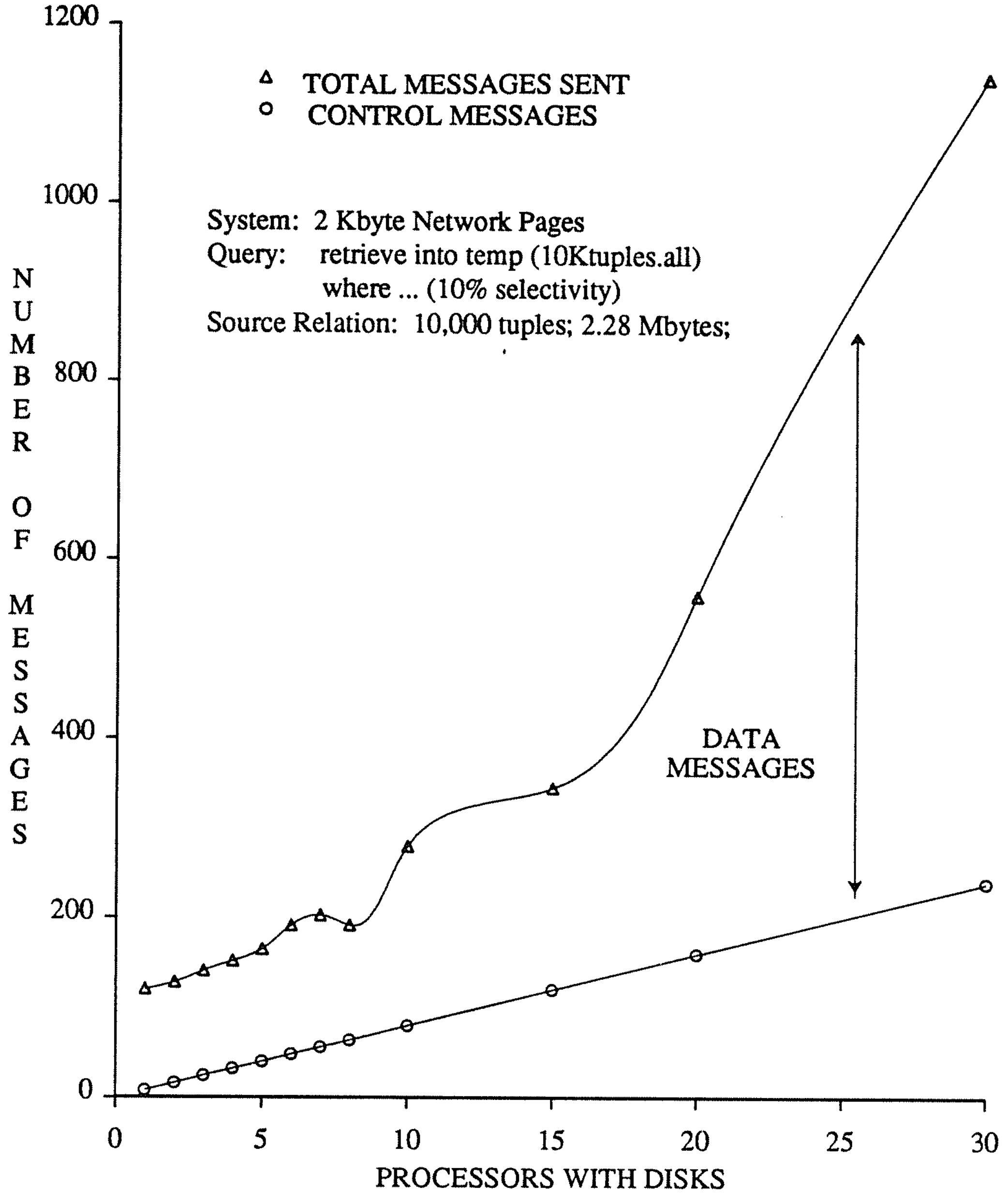
10,000 Tuple Selection Query Response Times
Figure 7.1

This model, the **validation** model, has been designed to reflect as closely as possible the policies and algorithms of the current implementation of Gamma. For systems with up to six disks, the performance of the validation model is within 93% agreement with the measured performance of Gamma. The anomalous performance of Gamma for the systems with seven or eight processors with disks is due to the use of slower, non-homogeneous disks (see section 6.9.2). As the validation model assumes identical disks, the model does not reflect the anomalous behavior of Gamma for these cases.

7.4.2. Identifying Design Flaws in Gamma

In Figure 7.1, the performance of the validation model is also presented for larger configurations of processors and disks than were available to Gamma. The periodic fluctuations in the performance of the validation model for these larger systems results from the policy that Gamma employs when distributing tuples to a permanent result relation. Gamma distributes tuples of all result relations among output buffers on a tuple-by-tuple round-robin basis. This can lead to a degradation in performance when, at the end of a query, all the output buffers are equally, but minimally filled. For example, in the case of the 10,000 tuple selection query for a system with ten disks, 100 tuples are produced by each participating processor. Each processor will use 10 output buffers, as the result relation is to be distributed among all of the disks in the system. Nine result tuples (208 bytes each) will fit in each of the ten buffers (2048 bytes each). After all the buffers have been filled and flushed once, they will be filled with the remaining 10 tuples to a capacity of one tuple per buffer. When the selection finishes, each of these almost empty buffers will have to be written across the network, resulting in a minimum, total transfer of 200 data pages for the entire query.[40] Figure 7.2 illustrates the periodic fluctuations in the number of data pages that are sent during the 10,000 tuple selection query for systems of varying sizes.

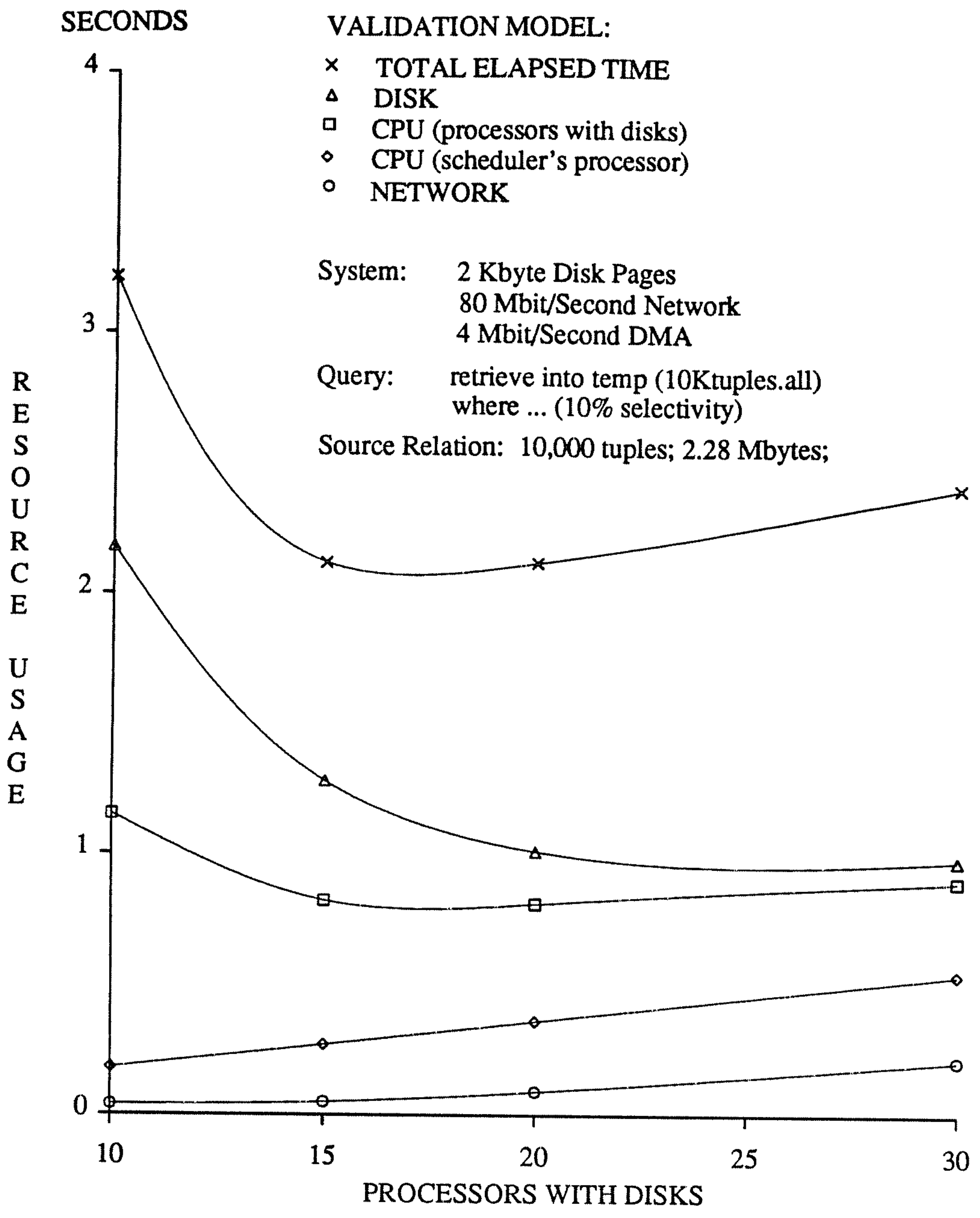
[40] Actually, 1/10 of these data pages are transferred between **select** and **store** operators on the same processor. This results from the fact that tuples from one of the partitions being produced by each **select** operators are sent to a **store** operator located on the same processor. These local messages are not sent across the network, but do consume local resources as the messages are copied between local memory buffers.



10,000 Tuple Selection Query: Total Messages Sent
Figure 7.2

Also, as illustrated in Figure 7.2, the tuple-by-tuple round-robin partitioning strategy used by Gamma has another, more serious drawback. As the size of the system is increased, Gamma transfers many more data pages across the network than necessary. As previously mentioned, this results from the fact that, at the end of a query, most of the selection output buffers will only be partially full. Nevertheless, these buffers must be flushed across the network. As the size of the configuration grows, the number of partially filled pages that are flushed at the end of a selection operation using tuple-by-tuple round-robin partitioning grows as the product of the number of producing and consuming processes associated with the output data stream of the selection. In this example, the number of flushed pages grows as the square of the number of processors with disks as a **store** operator process on each processor with a disk receives a fragment of the result relation. Furthermore, the receivers of these data streams (the **store** operators) expect to receive an end-of-file message from each selection operator, so a message is required even if an output buffer is completely empty.

Figure 7.3 illustrates the impact that flushing these increasing numbers of partially filled buffers has upon the resource consumption of the various components of the system.



10,000 Tuple Selection Query: Resource Consumption
Figure 7.3

As more processors and disks are added, the workload of each component ideally should decrease. That is, in principle, the fixed processing requirements associated with a particular query should be shared among all the components of the distributed system resulting in a smaller workload for each component. However, due to the processing requirements associated with the flushing of buffers at the end of this query, the CPU consumption of each processor with a disk levels off and actually begins to increase for systems with 15 or more disks.

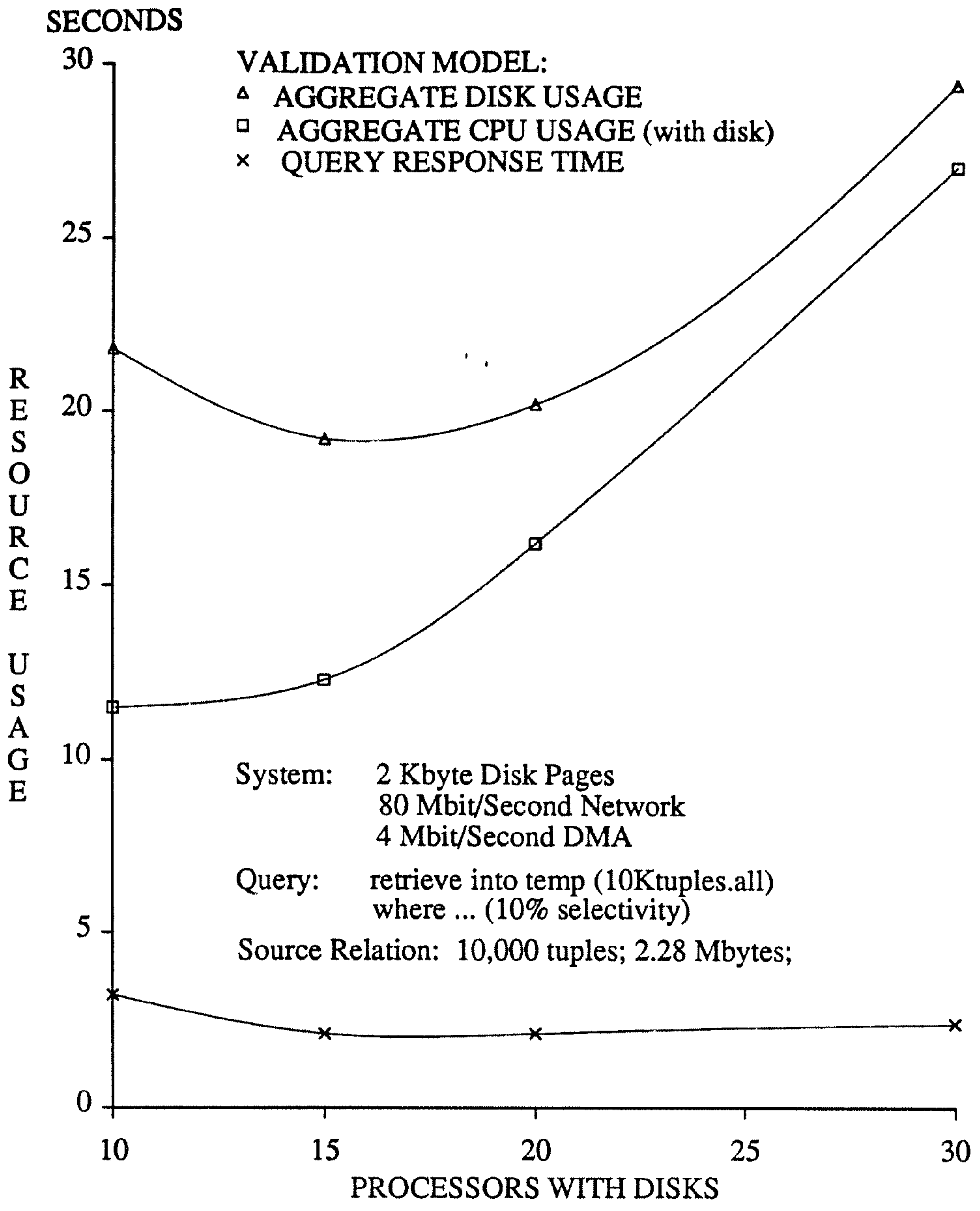
In Figure 7.3, individual disk usage also does not decline proportionally for larger systems as additional disks are added to the system. This results from the fact that the flaw in the round-robin strategy used by Gamma is compounded by the way pages of result relations are written to disk. Currently Gamma (and the validation model) read pages of the result relation off the network directly into disk buffers. It was hoped that this strategy would avoid the extra overhead of copying tuples from network buffers to disk buffers prior to storing them on disk. However, this strategy fails when a large proportion of the pages received are only partially full. Since the tuple-by-tuple round-robin partitioning strategy creates just such a situation, the validation model stores an increasingly larger number of partially full pages of the result relation on disk as the size of the system is increased. For larger systems, this increased disk activity more than counterbalances the benefit of adding additional disks. In fact, for the validation model, when more than 25 disks are used, the addition of disk/processor pairs results in a net increase in the consumption of disk resources on every disk in the system.

Another serious complication occurs in the validation model due to the policy of reading pages directly from the network into disk buffers. This policy requires that the employed data page format satisfy the constraints of both the message passing system and the secondary storage system. In Gamma, this resulted in the transmission of fixed sized data pages across the network. That is, in order to have the control information associated with a data page correctly aligned in a disk buffer, complete, maximum size data pages were always transmitted across the network. In Figure 7.3, the increase in the consumption of network bandwidth reflects the fact that full sized data pages are flushed at the end of a query irregardless of the amount of data that is contained in the buffers.

The increase in the CPU usage on the processor that supports the scheduler processes reflects the fact that schedulers in Gamma (and the validation model) communicate directly with each operator processor in the system. The query scheduler sends and receives a reliable message for each operator on each processor during both the activation and deactivation of the operator. Thus, a total of four control messages are exchanged by the scheduler and each processor with a disk in the process of scheduling each of the **selection** and **store** operations. Figure 7.2

illustrates the number of control messages that are exchanged in the course of the example selection query. Figure 7.3 illustrates the CPU usage that is associated with the processing of these messages by the scheduler process for this query.

Figure 7.4 illustrates the total disk and CPU resources that are consumed by the example selection query as the size of the Gamma system is increased.



10,000 Tuple Selection Query: Aggregate Resource Consumption
Figure 7.4

The significance of the previously described design flaws is evident in Figure 7.4 as the total consumption of disk and CPU resources rapidly increases for larger systems. However, as Figure 7.4 illustrates, Gamma and the validation model are able to effectively use parallelism to offset the negative performance impact of the increased resource consumption. That is, the net response time of the selection query merely levels off and does not increase in proportion to the resource consumption of the query. These increased levels of resource consumption would, however, certainly diminish the multi-user performance of the system.

7.4.3. Corrections to the Gamma Design

As the previous discussion indicated, a number of serious problems exist in the manner in which Gamma (and the validation model) manage data streams. The following discussion investigates solutions to these problems. The software design changes accompanying these solutions have been incorporated into a new model of Gamma that will be referred to as the experimental model.

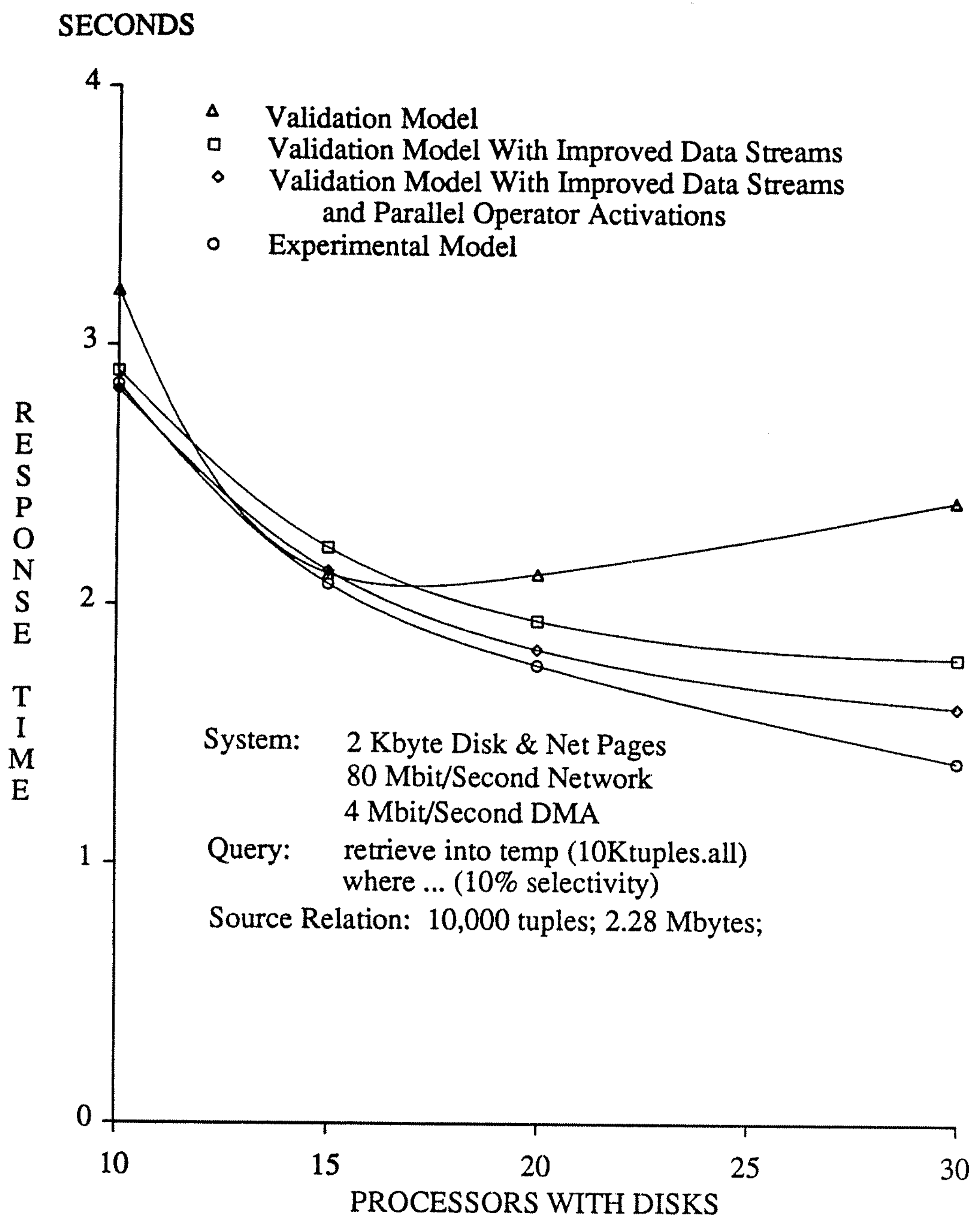
The experimental model partitions result relations using a page-by-page round-robin strategy. That is, subsequent tuples of the result relation are assigned to the same output buffer until the buffer fills and is flushed across the network. At that point, the next buffer (associated with a different destination **store** operator) is assigned the subsequent tuples from the fragment of the result relation that is being produced by the local selection operator process. When an operation completes, the last remaining, partially full page containing tuples from the local fragment of the result relation is flushed across the network. For the selection query previously examined in Figure 7.1, this results in the transfer of a number data pages across the network that is linear with the number of selection operator processes employed.

In the original Gamma design, an end-of-stream packet was transmitted across every pipeline at the end of a query, irregardless if any data remained in the output buffer for the pipeline. In the experimental model, all destinations (**store** operators) do not expect to receive end-of-file messages from every selection operator. Instead, the scheduler assumes the responsibility of notifying the **store** operators when they have received all of their input data.

As previously described, in the original Gamma design, pages of a result relation were read directly from the network into disk buffers in order to avoid the costs of copying tuples between network and disk buffers. The experimental model, by contrast, reads these pages into a network input buffer. Tuples are then copied from the network buffer into a disk buffer. In this manner, the disk pages containing a result relation can be completely filled. Except for the last page of a local fragment of a relation, only completely full pages of tuples are written to disk.

Actually, with a page-by-page round-robin strategy, it would even be feasible to retain the policy of writing pages of the result relation directly from the network to the disk. However, this policy would prevent Gamma from changing the size of a disk page to be different from that of a network page. As will be demonstrated shortly, this would be a severe and costly restriction.

Figure 7.5 illustrates the performance impact of improving the management of data streams with respect to the selection query previously used in Figure 7.1.



Performance Impact of Selection Query Control Improvements
Figure 7.5

The line labeled as incorporating improved data streams represents the performance impact of using page-by-page round-robin partitioning and collecting full disk buffers prior to writing the pages of a result relation to disk. While these improved data streams correct the periodic behavior of the validation model (caused by the relative fullness of the final pages produced by the selection operation), extra CPU overhead is incurred when copying tuples of the result relation from network buffers to disk buffers. This additional cost allows the validation model to perform slightly better than the improved model in systems with 15 disks.

As previously described, Gamma exchanges four control messages with each of the query processes that are assigned to a particular operation. Currently, half of these messages are exchanged in a synchronous manner. Gamma (in particular, the scheduler assigned to a query) sequentially sends and receives a reliable message to each targeted processor when an operator is activated. That is, after sending an activation message, Gamma waits for a reply from the activated operator process before progressing to the activation of an operator process on the next processor. In contrast, during the deactivation of an operation, Gamma sends deactivation messages to all operator processes before waiting for the replies from any of the operator processes. Clearly, the asynchronous nature of the deactivation sequence is a more advantageous choice. In retrospect, the synchronous activation sequence of Gamma was a mistake and is not a requirement of the scheduling algorithm.

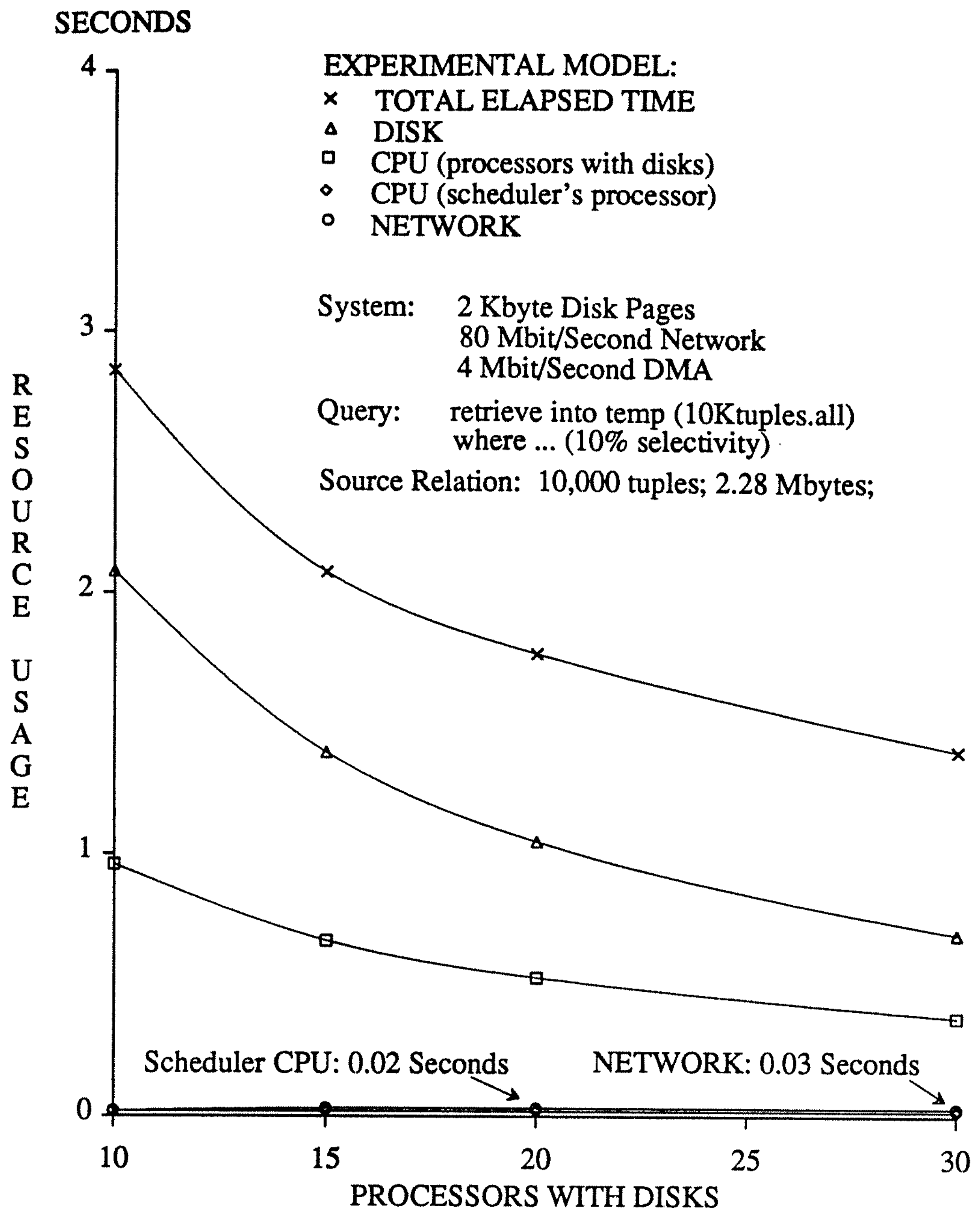
Changing the scheduling algorithm to allow activation messages to be sent to all nodes before accepting any reply messages produces performance improvement, as it shortens the time period necessary to start up query operations. This performance improvement is reflected in the third curve of Figure 7.5 labeled as using parallel operator activations.

Even with a more asynchronous reception of control messages, the centralized scheduler remains a potential bottleneck in the Gamma design. This results from the fact that the current design requires the scheduler to interact individually with each of the processes that are assigned to an operation. The experimental model contains a modified scheduling algorithm that applies increased levels of parallelism to the control of individual queries. Instead of requiring that the scheduler interact directly with each destination processor supporting a particular query operation, the experimental model passes the responsibility for distributing control messages to the query operators themselves. A binary tree structure is logically imposed upon the query operator processes for purposes of transmitting control information.[41] The scheduler sends a control message only to the operator process at the root of the control tree. When this operator process receives the control message, it sends the message on to the operator processes that are it's immediate descendents in the control tree. These descendent processes continue the parallel propagation of the message through the control distribution tree. The processes at the leaves of the control tree terminate the propagation of the control message and initiate the reply messages. These replies are returned up through the control tree, preserving the control information that is specific to each operator process. The routing information for the control distribution tree is determined by the scheduler and is included in the initial control packet that is sent to each query operator process.

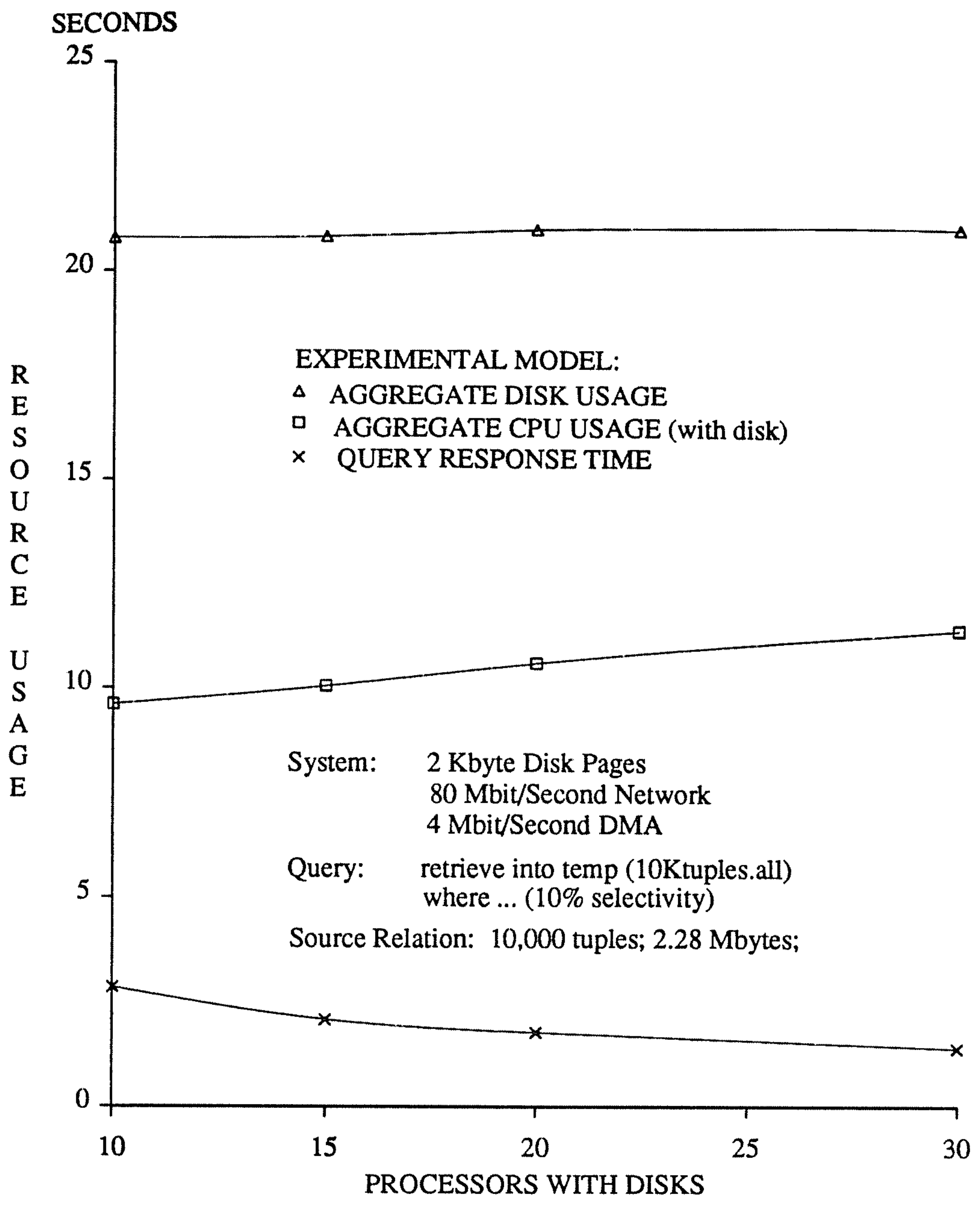
The experimental model includes all of the previously described data stream improvements plus the improved scheduling algorithm that uses binary, control distribution trees. In Figure 7.5, the performance of the experimental model for the example selection query is presented. The performance improvement produced by the experimental model increases with the size of the configuration because of the increased levels of parallelism that are applied to the distribution of control messages.

Figures 7.6 and 7.7 illustrate the resource usage that accompanies the experimental model for the 10,000 tuple selection query.

[41]This binary control tree is completely independent of the compiled query tree that represents the operations that comprise a query. The control tree merely imposes an order among the various operator processes that have been assigned to execute a single operator. That is, the control tree contains no semantic information and is only used to determine the routing of control messages.



10,000 Tuple Selection Query: Resource Consumption
Experimental Model
Figure 7.6



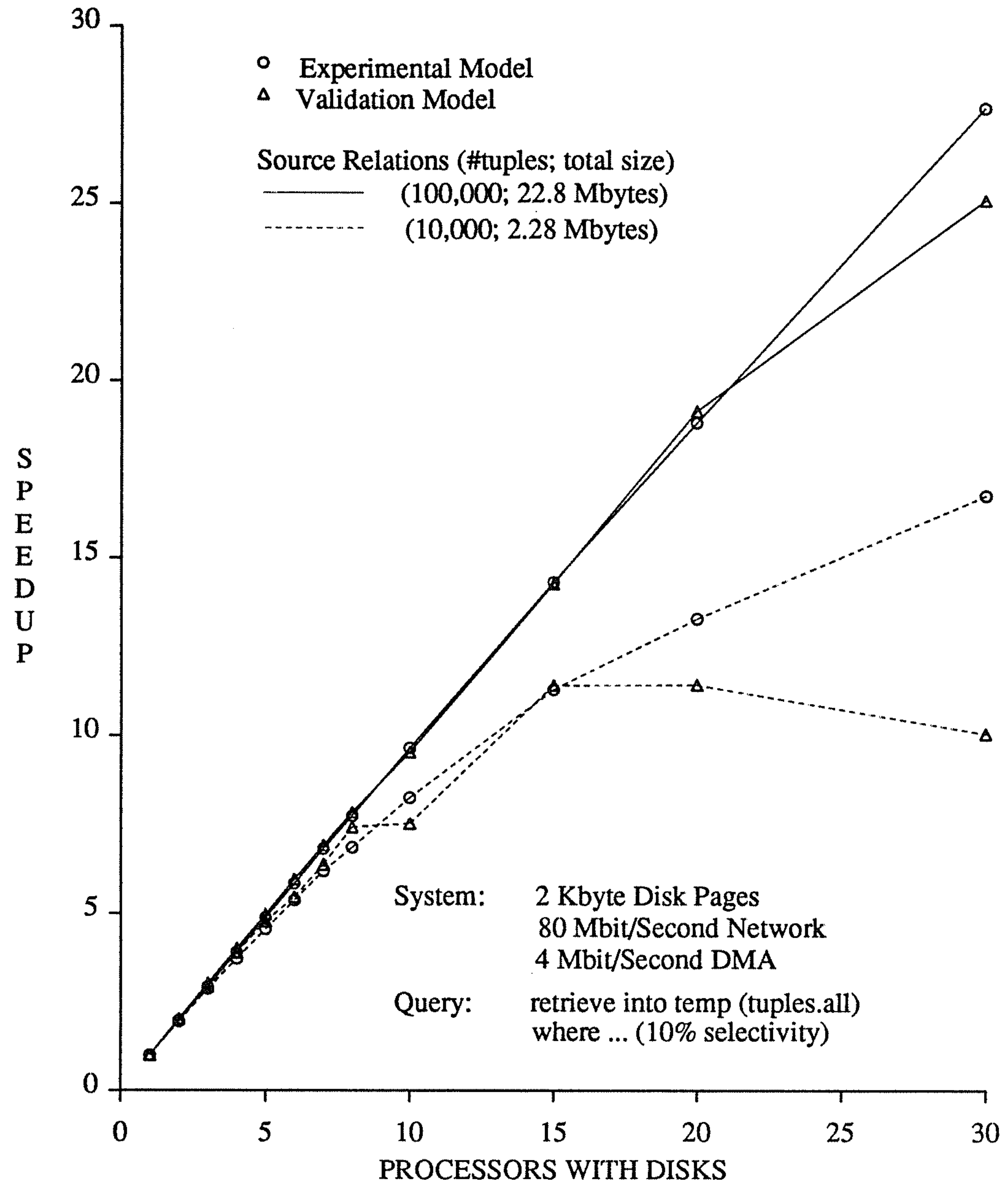
10,000 Tuple Selection Query: Aggregate Resource Consumption
Experimental Model
Figure 7.7

The total consumption of disk resources increases only slightly in Figure 7.7 due the fragmentation of the source and result relations across larger numbers of disks. That is, the last page of each fragment of a relation is only partially full and the number of these partially full pages increases linearly with the number of disks. While the disk remains the most heavily used resource in the experimental model, the workload associated with the use of the disks is effectively distributed as additional disks are added to the system.

The increase in the aggregate level of CPU resources by the query operators reflects the fact that additional control messages are required as the size of the system is increased. In Figure 7.6, the success of removing the scheduler as a centralized bottleneck in the system is indicated by the fact that the scheduler consumes only a small fraction of CPU resources irregardless of the size of the system. The scheduling algorithm evenly distributes to the query operator processes the costs of managing the exchange of control messages. Significantly, the CPU consumption and message passing latency accompanying the exchange of these control messages are incurred in parallel among the various operator processes.

Figure 7.6 demonstrates that the use of page-by-page round-robin streams and a fixed cost scheduler results in a consumption of network bandwidth that is highly independent of the size of the system. That is, the number of data and control messages that are exchanged by the experimental model increases at a low, linear rate as the size of the system is increased.

Finally, Figure 7.7 demonstrates that the experimental model can effectively use parallelism to reduce the response time that is associated with the example selection query. The aggregate disk and CPU consumption far exceed the net response time for the selection query. Figure 7.8 also reflects this fact by illustrating the performance speedup that accompanies the validation and experimental models for both the previously examined 10,000 tuple selection query and for a selection query that accesses 100,000 tuples.



Selection Query Speedup
Figure 7.8

Both selection queries apply predicates with a 10% selectivity and are identical except for the sizes of the source relations.

Figure 7.8 demonstrates that increased levels of performance can be expected for selection queries that access sufficiently large amounts of data with respect to the number of resources that are used. One of the reasons for this result is that increasing the size of the system increases the number of control messages that are required for the management of a query. For the 10,000 tuple selection query, the performance speedup of both the validation and experimental models diminish as the size of the system configuration is increased. This decline results from the fact that the 10,000 tuple selection query produces only a small result relation (1,000 tuples; 228 Kbytes). For a configuration with 30 disks, each processor produces only 33 or 34 tuples of the result relation. Thus, a minimum of 120 data messages[42] must be exchanged in the course of the query for that configuration of the system. In contrast, as previously described, the query scheduler sends and receives a reliable message to each operator on each processor during both the activation and deactivation of the operator. Thus, a total of four control messages are exchanged by the scheduler and each processor with a disk in the process of scheduling each of the **selection** and **store** operations. For the 10,000 tuple selection query, therefore, twice as many control messages[43] (240) are sent as data messages (120). The experimental model is able to partially reduce the impact of this imbalance of control messages to data message via it's scheduling algorithm that increases the parallelism that is applied to the flow of control messages.

In order to avoid an imbalance of control messages with respect to data messages, it may be advisable to store small result relations across a subset of the disks in large configurations. In this manner, subsequent selections or scans on the relation could be performed in a more efficient, albeit slower fashion.

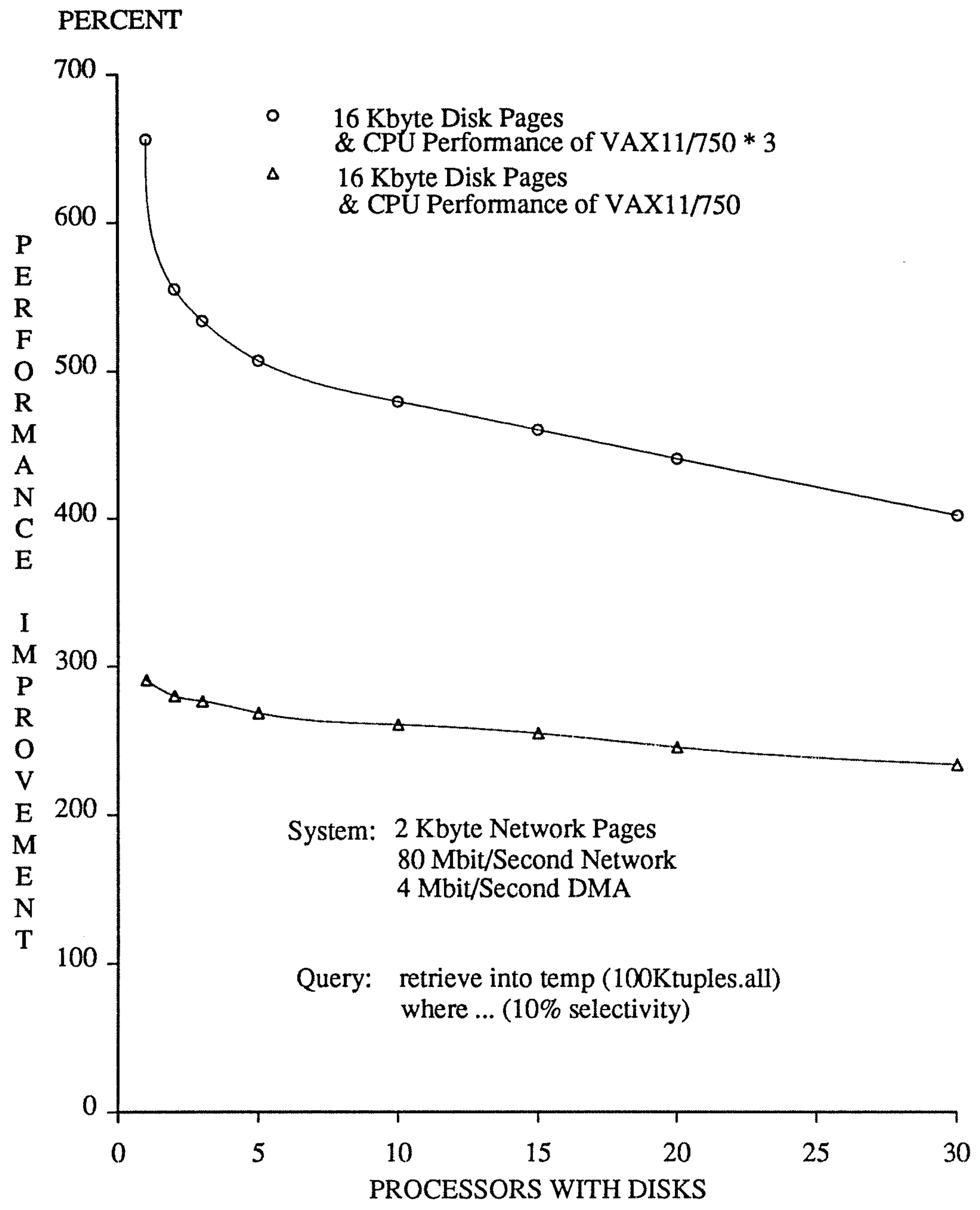
7.4.4. Enhancements to the Gamma Design

For the previously tested selection queries, the disks were the most heavily utilized resource in the system for Gamma. This bottleneck can be reduced by increasing the size of the blocks that are accessed by a single disk operation. Currently, Gamma uses 2 Kbyte disk pages. Figure 7.9 indicates that for selection queries, very

[42]Each processor uses 4 data pages to contain the 33 or 34 tuples that are produced locally.

[43]For each of the 30 processors, a total of eight control messages are required. Four control messages are exchanged when scheduling each of the **select** and **store** operators.

significant performance improvements can be achieved by changing the disk block size from 2 Kbytes to 16 Kbytes.

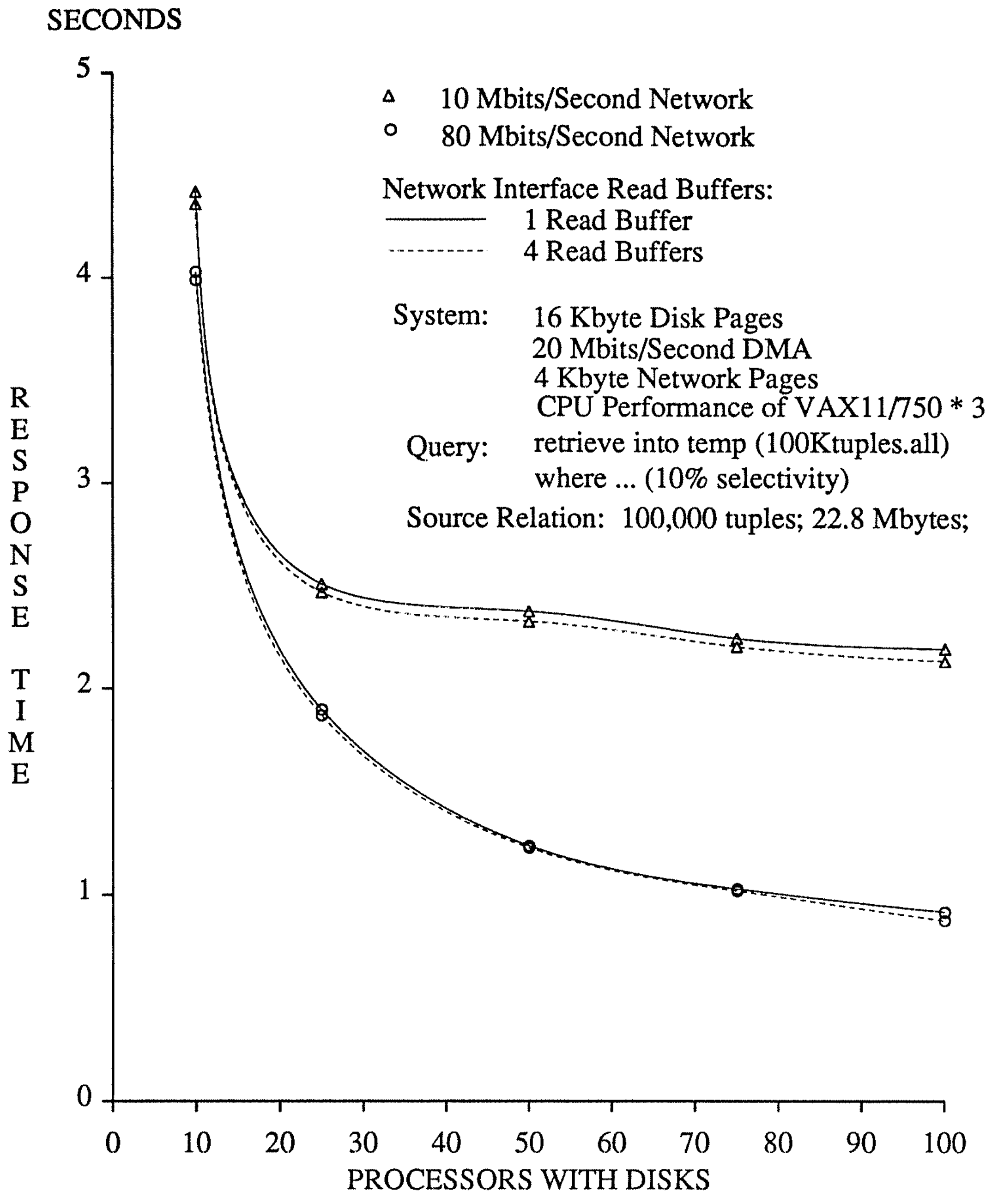


Percent Improvement in Performance of 100,000 Tuple Selection Query
Figure 7.9

The figure presents the percentage improvement in performance[44]. Once the disk block sizes are increased, the performance of the processors themselves becomes a critical factor. The figure illustrates the further performance improvement that results from a 3-fold increase in the CPU performance for systems using 16 Kbyte disk blocks. The decline in the percentage of performance improvement for the systems with faster CPUs results from the use of short circuited messages. When the destination of a message resides on the same processor as a sending process, Gamma (and the validation model) do not send the message across the network, but instead short circuit the message within the confines of the local processor. The CPU overhead of making a copy of such messages and requeuing the messages to the destination port are more significant in smaller systems since a larger proportion of the total messages for a query will be short circuited. For the round-robin partitioned data streams of the selection query, the proportion of data messages that are short circuited is inversely proportional to the number of processors with disks in the system. For example, the 10,000 tuple selection query results in the short circuiting of 100% of the total number of data messages for systems with a single disk. In contrast, for systems with four disks, 25% of the data messages for this query will be short circuited. Thus, smaller systems will tend to be more CPU bound than larger systems and the performance of these smaller systems will thus be enhanced to a greater degree by an increase in CPU performance.

Figure 7.10 illustrates the ability of a single selection operation to saturate a token ring of moderate bandwidth.

[44]Defined as the inverse of response time.

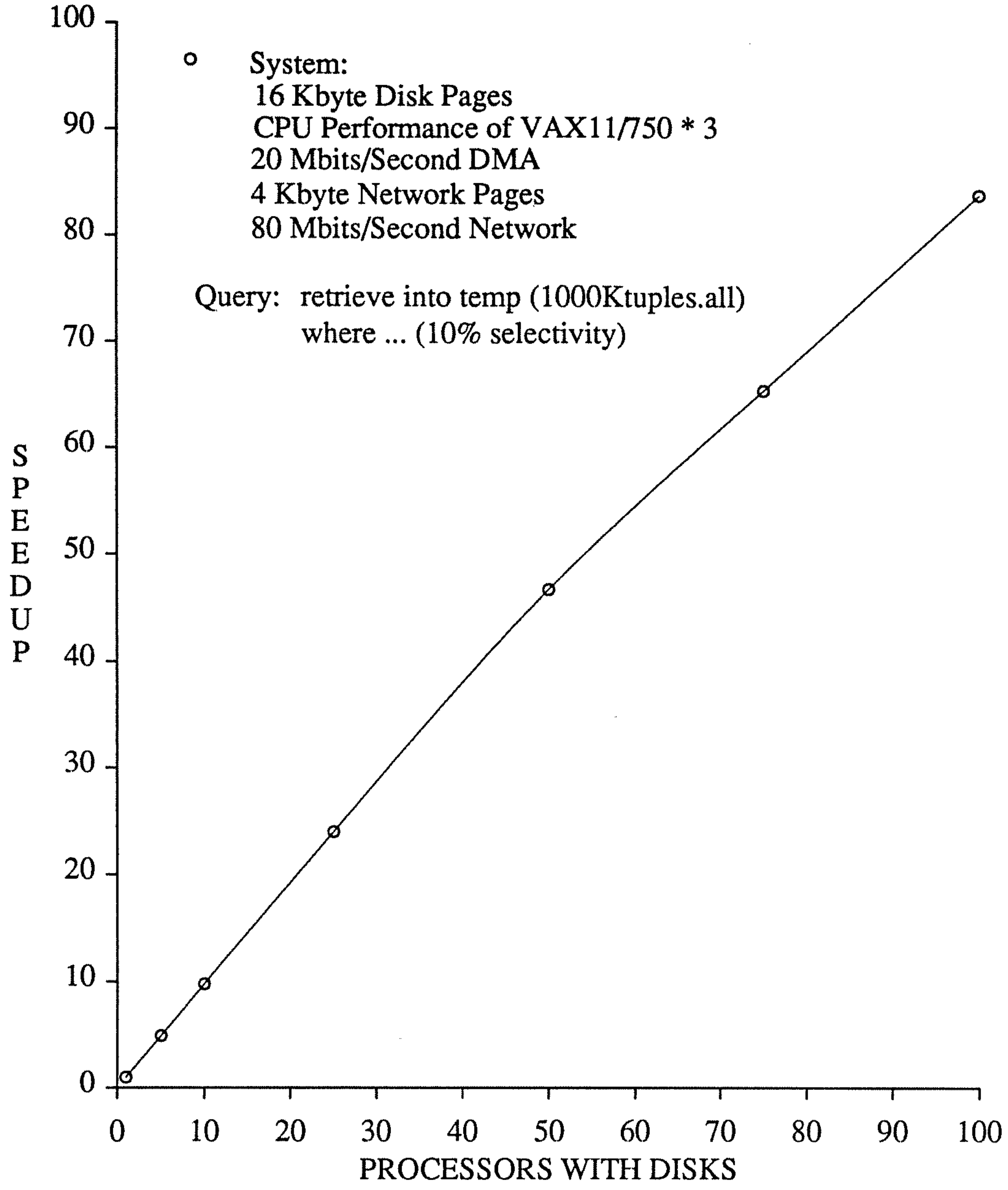


100,000 Tuple Selection Query with Varied Network Capabilities
Figure 7.10

In this experiment, other potential bottlenecks in the system were removed by using large disk blocks, a fast CPU, a fast DMA controller (40 megabits/second instead of 10) and larger (4 kilobyte), maximum size[45], network packets. The number of read buffers on the processors' network interfaces had little effect in this experiment. Once the token ring became saturated, the existence of 4 read buffers only marginally improved the performance of the selection query in comparison to a system with network interfaces that only contained a single read buffer. If the data flow between producers and consumers of tuples streams is evenly distributed, it appears that the network bandwidth has more of a potential to become a bottleneck than does contention for read buffers at the destination sites. This observation is reinforced by the fact that all the selection query tests encountered very low levels of message retransmissions.

Figure 7.11 provides an indication of the extent to which a Gamma system with fast CPUs using large disk blocks can be effectively expanded.

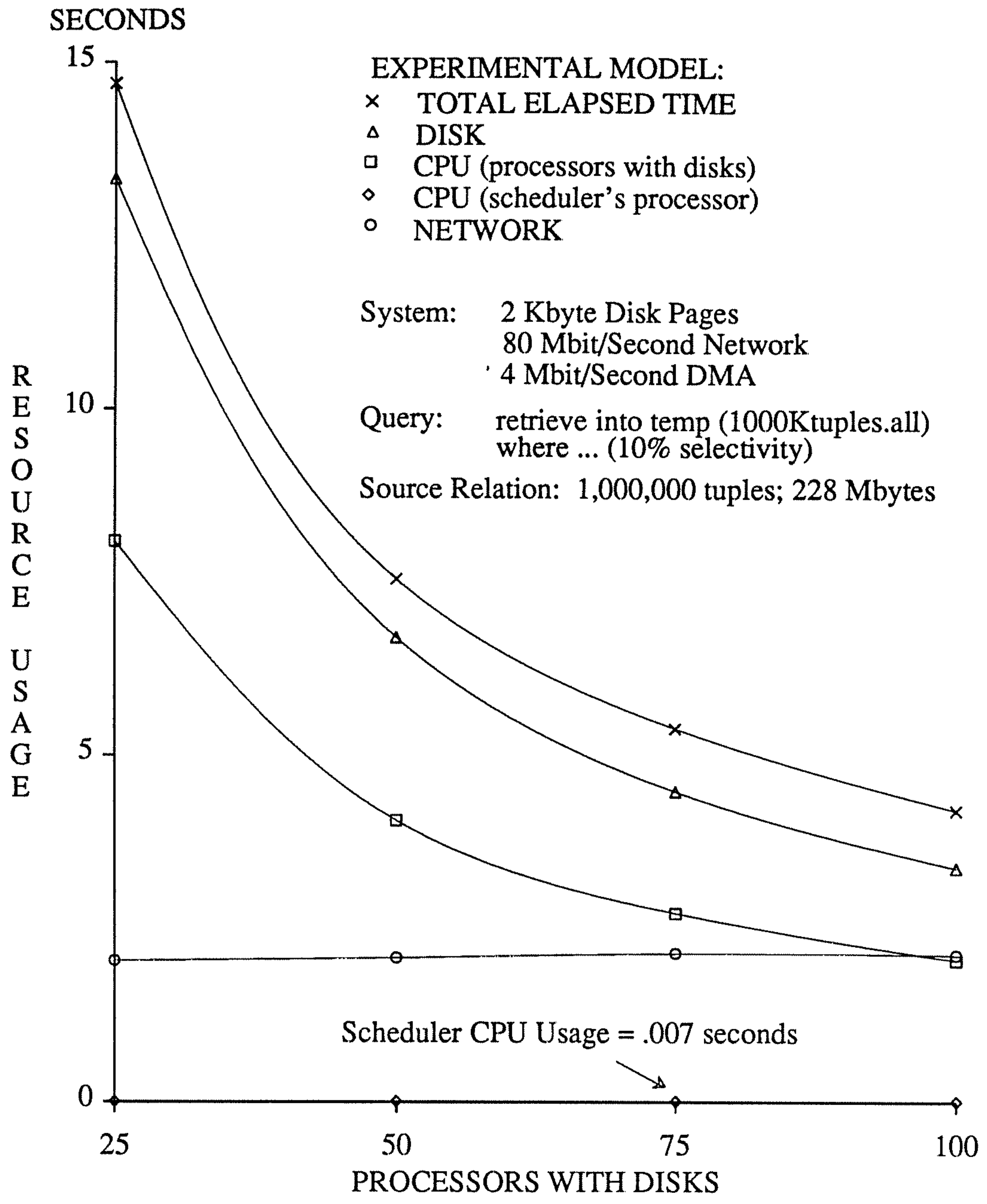
[45]The maximum size of packets is an attribute of the modeled network interface.



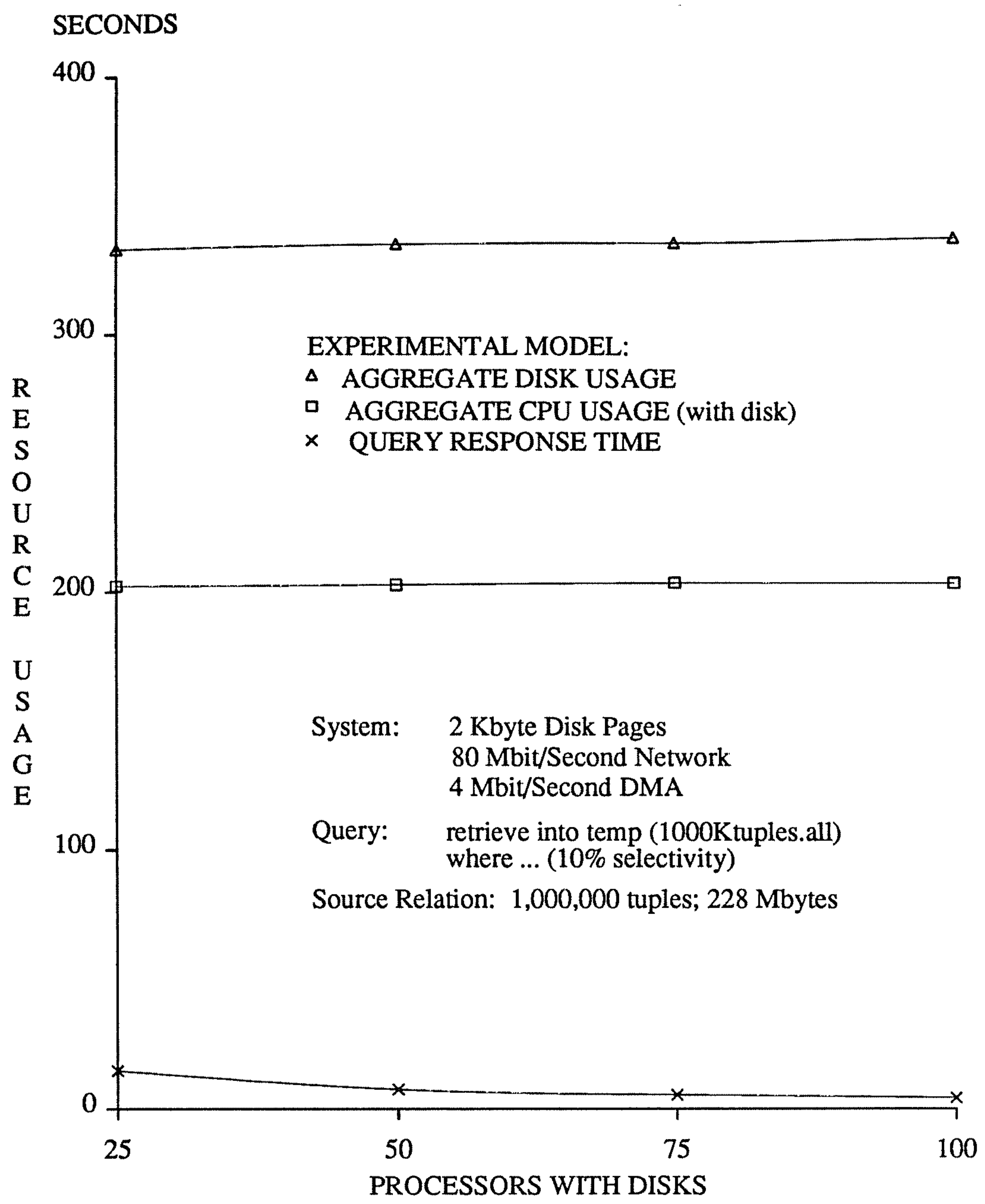
1,000,000 Tuple Selection Query Speedup
Figure 7.11

The indicated selection query accesses a source relation containing 1,000,000 tuples. The figure indicates that significant speedups can be expected for large configurations. However, performance does start to decline in the systems containing 50 or more disks.

As indicated in the accompanying resource usage figures of Figure 7.12 and Figure 7.13, the workload associated with the 1,000,000 tuple selection query can be effectively distributed among systems containing up to 100 processors with disks.



1,000,000 Tuple Selection Query Resource Usage
Figure 7.12



1,000,000 Tuple Selection Query: Aggregate Resource Usage
Figure 7.13

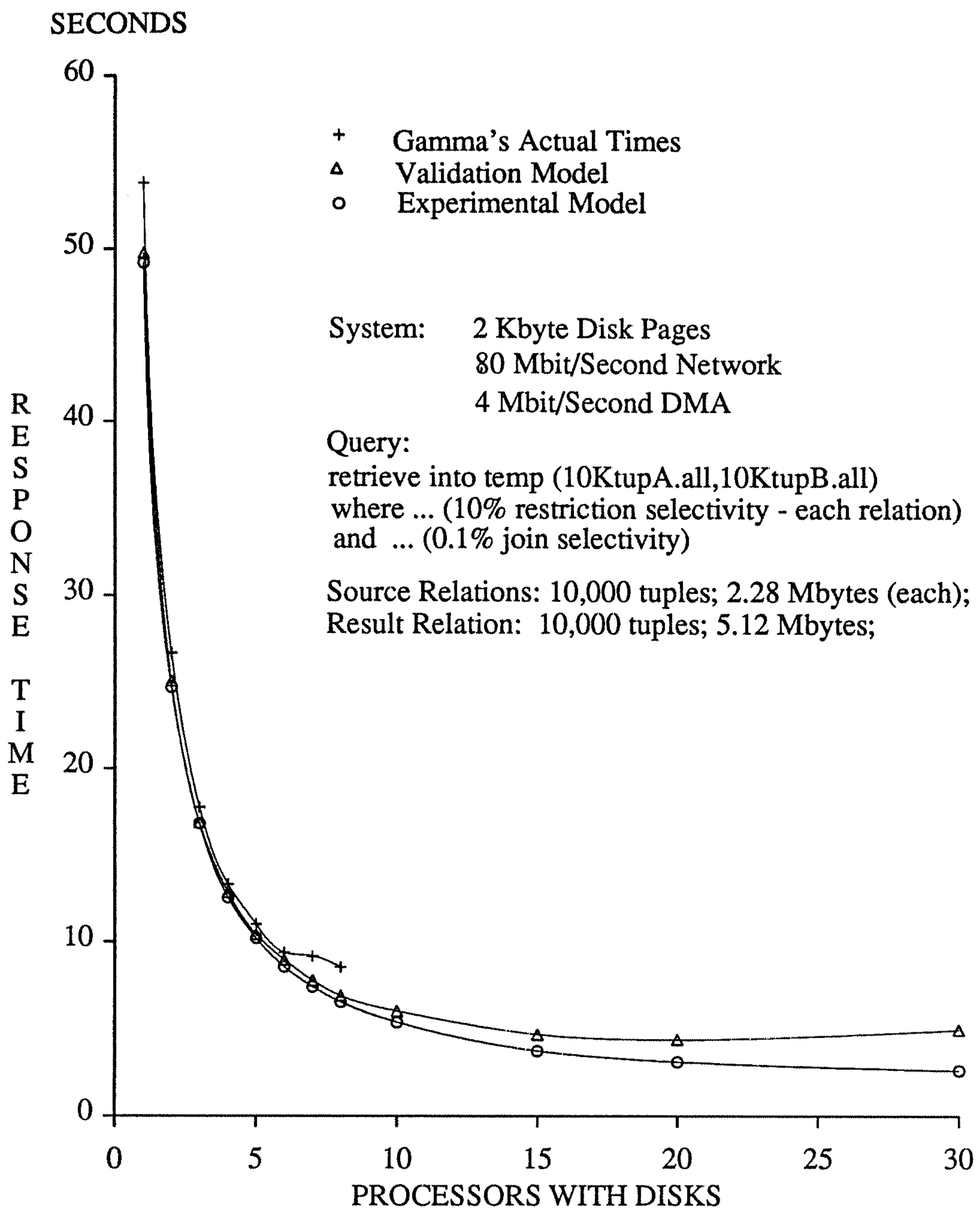
The cost of scheduling and controlling this query remains at a low level. That is, the synchronization and control of a large Gamma system does not become a bottleneck for the tested query.

In Figure 7.12, the ratio of the time that the network is busy with respect to the total elapsed time for the query indicates that the network is becoming more highly utilized. For systems with 100 processors/disks, the effective network utilization exceeds 50%. The effect of this increased utilization is that operator processes are more frequently blocked during a network access waiting for a token. However, for all systems tested, very few retransmissions occurred. That is, acknowledgements were delivered before senders timed out and retransmitted messages. Also, contention for the network buffers of the receivers did not develop. The availability of a non-blocking network write facility might be expected to reduce the negative impact of increased network utilizations on the performance of queries in large systems by reducing the latency incurred in waiting for tokens. A non-blocking network write facility might also enhance query throughput in a multi-user environment as a query would not be blocked by the write operations of other concurrent query processes. Gamma will soon be converted to a new version of the NOSE operating system which provides just such a facility.

7.5. Analysis of Ad-Hoc Join Query Performance

One of the promising features of Gamma is the ability to execute join operations efficiently on processors that are remote from the sites where the source relations are produced. The following section examines the impact of applying the design enhancements of the experimental model to the execution of a join query. The simulation model is then used to explore the level to which the parallelism of a large multiprocessor Gamma system can be effectively applied to a join query that accesses large amounts of data.

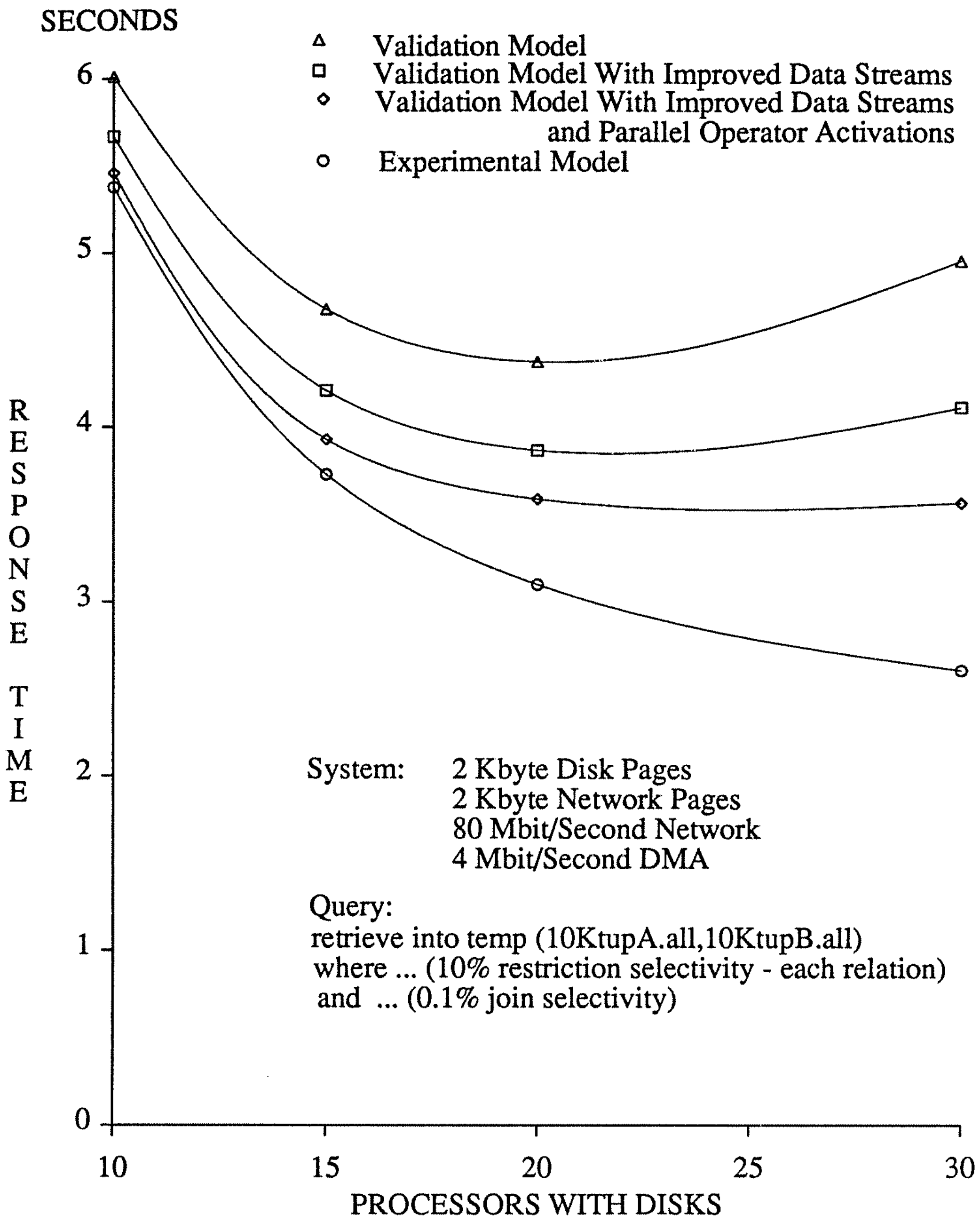
Figure 7.14 illustrates the ability of the validation model to accurately simulate the measured performance of a join query.



10,000 Tuple Join Query Response Times
Figure 7.14

Each of the source relations of the join operation are produced by a 10% selection on a 10,000 tuple base relation. For both source relations, each of the 1,000 tuples qualified by the selection participate once in the result relation of the join operation which thus has a join selectivity of 0.001. The performance of the validation model ranges between 90% and 96% agreement with the measured performance of Gamma for systems with up to twelve processors (half of which have disks). As in the case of the simulated selection query, the validation model does not attempt to model the performance anomaly caused by Gamma's non-homogeneous seventh and eighth disks. The periodicity evident in the modeled selection query of Figure 7.1 is less obvious in the join query, as the hash-partitioned data streams produced by the selection operators flush pages across the network at more irregular intervals than the tuple-by-tuple round-robin partitioning strategy. However, at the end of the selection operation on the building and probing relations, the hash-partitioned strategy still flushes an increasing large number of partially filled pages as the size of the system is increased. However, in contrast to the selection query, these partially filled pages do not result in any additional disk I/O, as they are consumed by join operator processes and not store operator processes.

Figure 7.15 reflects the relative performance impact of applying the improvements to the data stream and scheduling algorithms that were analyzed for the selection queries in the previous section.

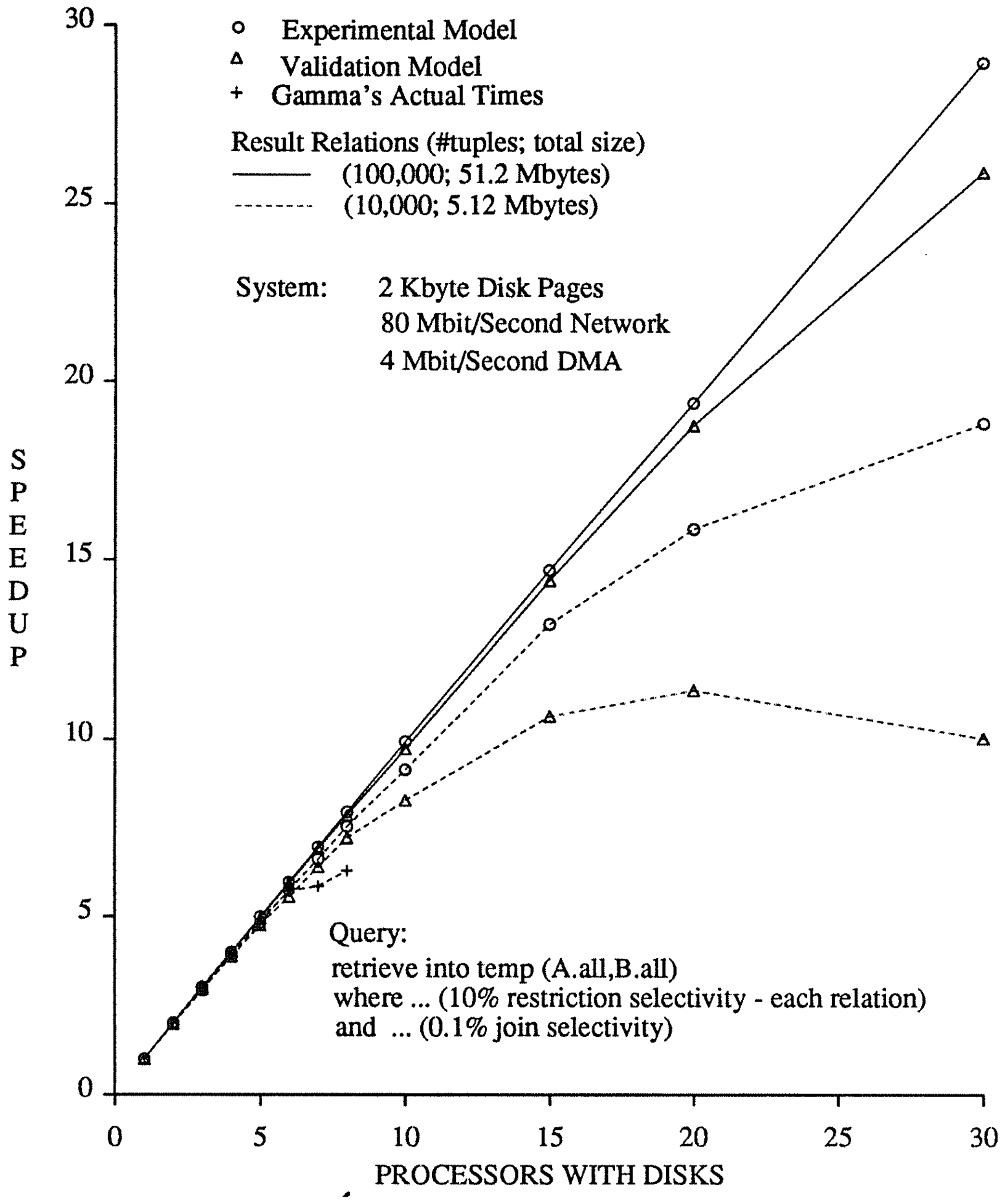


Performance Impact of Query Control Improvements
Figure 7.15

The join operator processes apply round-robin partitioning to the generated result relation. The curve representing the system with improved data streams illustrates that the performance of the join is significantly improved by using a page-by-page round-robin strategy and collecting full pages of tuples before writing pages to the result relation. The beneficial impact of using a parallel activation strategy is also significant. However, most noticeable is the improvement provided by the distributed scheduling algorithm[46] of the experimental model. The increased number of operators in the join query magnifies the relative performance benefits of the control strategy of the experimental model for large systems. Five separate operators have to be activated in the course of the query (2 selections, the build and probe phases of the join, and the store operation) whereas only two operators were activated for the selection query. Therefore as the size of a system is increased, the distributed scheduling algorithm of the experimental model become increasingly important

The speedup factors for the 10,000 tuple join queries are shown in Figure 7.16.

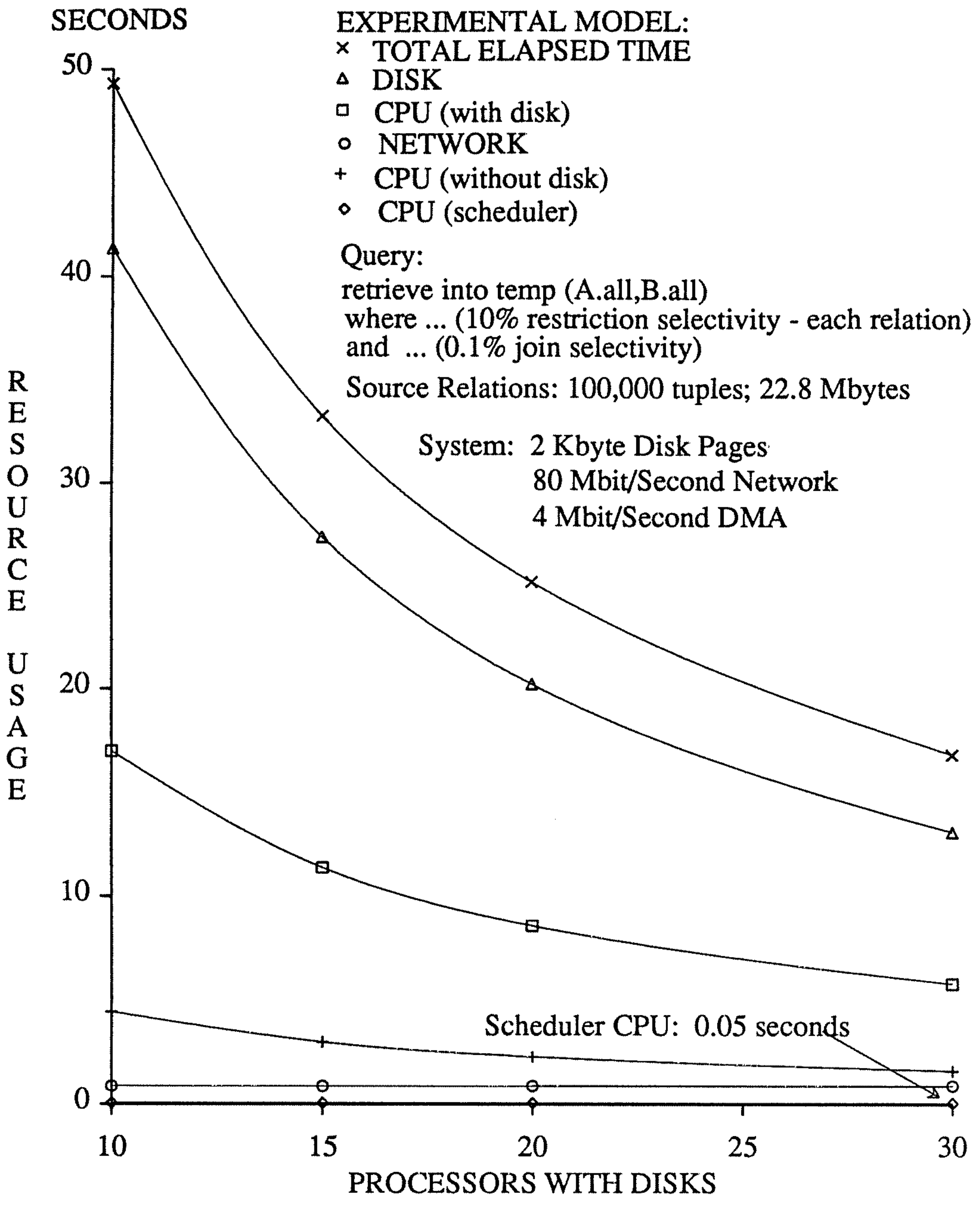
[46]Parallel distribution of control messages through a binary control tree.



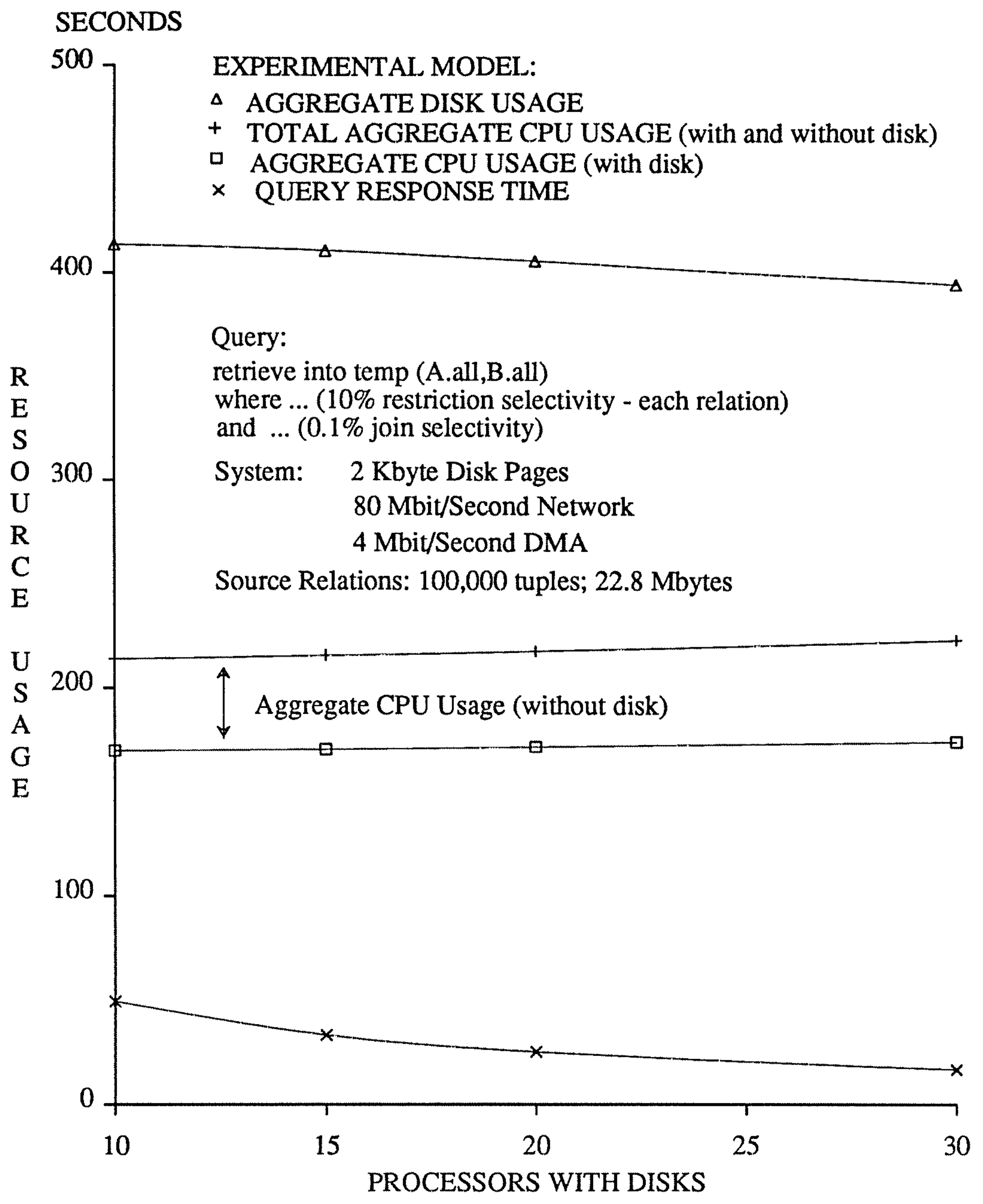
Join Query Speedup
Figure 7.16

For small configurations, Gamma and both of the simulation models produce linear speedups. However, as was the case for the 10,000 tuple selection, the problems caused by the design flaws of the validation model are compounded in larger configurations because the ratio of control to data messages becomes significant. The performance of the validation and experimental models is also presented for a join of two 10%-selected 100,000 tuple relations. The high data volumes of this query allow the experimental model to extend it's 100% linear speedup across the entire spectrum of the presented systems. The high data volumes of this join query also mask most of the negative performance aspects of the validation model.

Figures 7.17 and 7.18 present the resource usage of the 100,000 tuple join query that provide the basis for the 100% linear speedup in the performance of the query.



100,000 Tuple Join Query: Resource Consumption
Figure 7.17

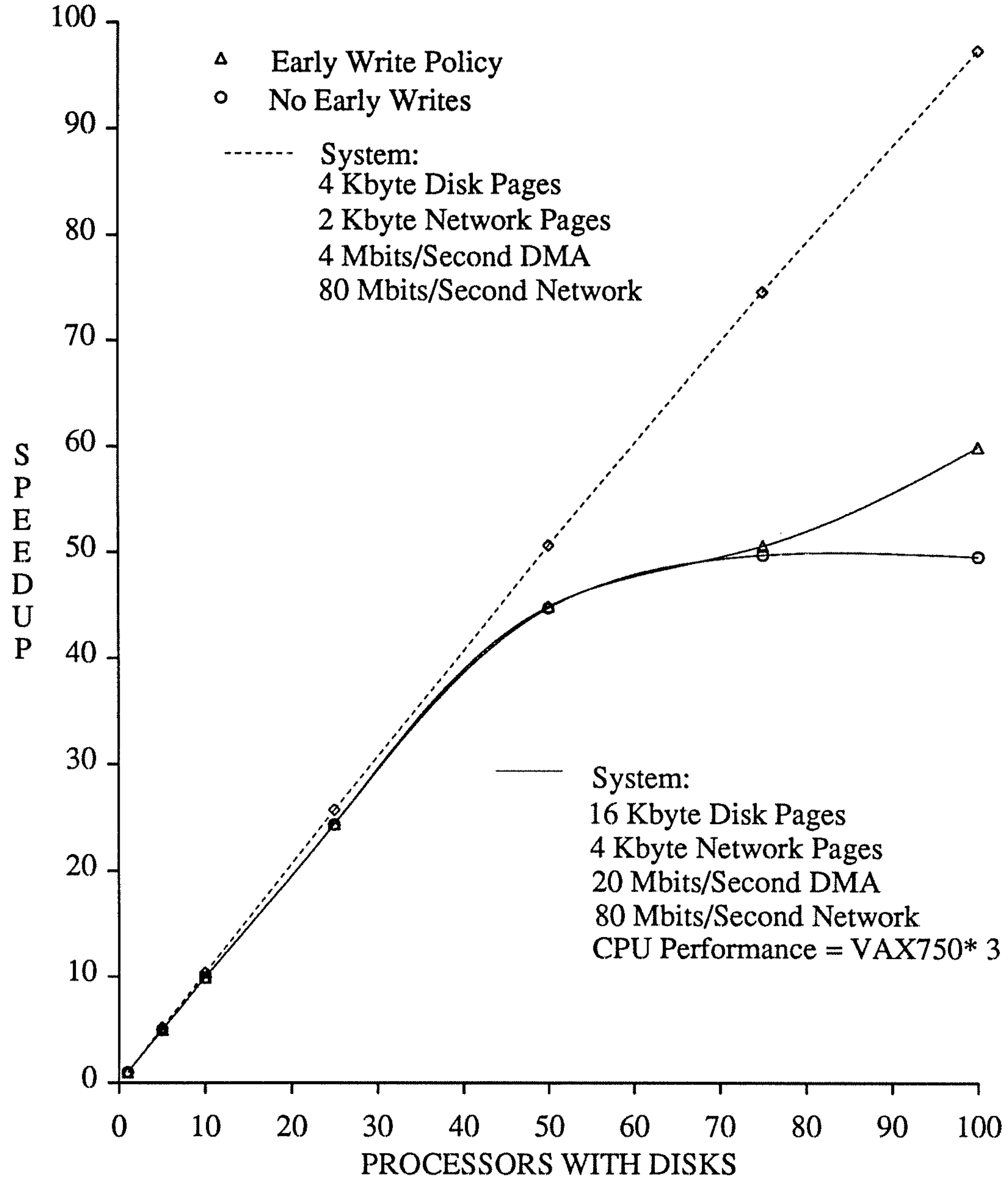


100,000 Tuple Join Query: Aggregate Resource Consumption
Figure 7.18

As the size of the system is increased, the disk and CPU workload are evenly shared. That is, as resources are added to the system, Figure 7.17 shows a proportional decrease in the resource consumption of the individual disks and processors. Similarly, Figure 7.18 does not show any increase in the total disk or CPU workload as the size of the system is increased. Also, as was the case for the selection queries, the CPU resources consumed by the scheduler are minimal.

The slight decrease in the aggregate disk usage in Figure 7.18 is due to the fact that in larger systems a higher proportion of pages of the result relation reside in the buffer pools of the processors with disks at the end of a query. Before the **store** operation completes, these remaining pages of the result relation are flushed to disk. However, at the end of a query the flushing of these pages does not interfere with the disk accesses of the selection operation. That is, for this query, as the size of the system is increased, there is less movement of the disk arm as the relation scans of the **selection** and **store** operations conflict less frequently. It should be noted, however, that this effect is an artifact of the single user benchmark query and would not be expected to persist in a multi-user environment.

Figure 7.19 presents the performance speedups that accompany a join of two 10%-selected 1,000,000 tuple relations.

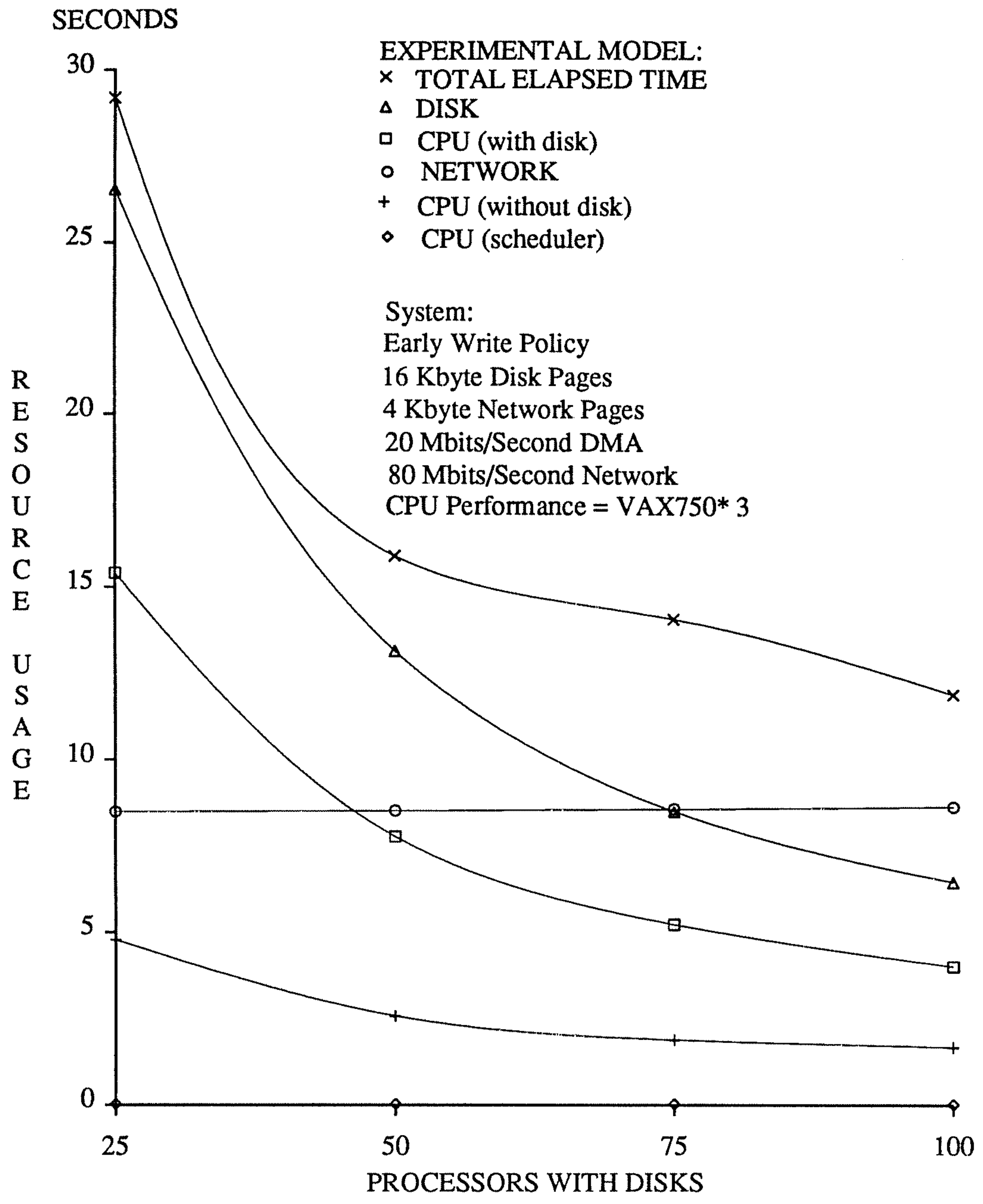


1,000,000 Tuple Join Query Performance Speedup
Figure 7.19

For systems with larger disk blocks, etc., and up to 50 disks (and 101 processors), the speedup factors for this join query are linear, with almost a 100% speedup for each additional disk[47] However, beyond this point the speedups for the query fall off dramatically. As Figure 7.20 illustrates, there are still significant levels of available network bandwidth.[48]

[47]With each additional disk, two processors are also added to the system. Only one of these processors is associated with the additional disk. That is, the size of the system is increased by adding a processor with a disk and another processor without a disk.

[48]The ratio of network usage to the response time of the query for a system with 50 disks indicates that the network is 54% utilized.



1,000,000 Tuple Join Query: Resource Usage
Figure 7.20

In Figure 7.20, the relatively uniform consumption of network bandwidth reflects the fact that the decreased response times for larger systems are not caused by significant numbers of retransmissions. Instead, the problem appears to be related to the highly periodic pattern[49] with which the join query loads the network. Convoys of processes end up waiting for tokens for significant amounts of time. For the 100 disk system, the average time spent waiting for a token when writing a message was 1.27 milliseconds. In contrast, systems with less than 50 disks seldom waited for a token at all. For a fully saturated network, such waits would be expected. However, the network is not completely saturated by this configuration.

In an attempt to break up the periodic nature of network accesses generated by this join query, a policy of writing some of the output buffers before they completely were filled was implemented. In this **early write** policy, each operator process writes out one half of its output buffers initially at the time when they become half full. As Figure 7.19 illustrates, this policy is mildly successful.

Figure 7.19 seems to indicate that a more extensive policy of early writing or randomizing the production of hash-partitioned data pages is required.[50] The current policy of flushing partially full output buffers (early write policy) is only applied the first time the buffers are filled. However, this policy is only effective in those cases when the output buffers are only filled and flushed a small number of times.[51] In smaller configurations (40 to 80 disks), the output buffers are filled and flushed a greater number of times and the current, limited early write policy is ineffective in preventing the eventual formation of convoys. That is, for the tested selection query using 16 kilobyte disk pages, source tuples for the join operation can be produced quickly enough to negate the initial randomness introduced by the writing of partially full output buffers. In the future, a more aggressive policy of randomly writing out partially full output buffers will be tested. Potentially, the performance benefits of introducing additional randomness into the pattern of network access for large systems using large disk pages will outweigh the additional CPU resources that will be consumed transmitting a larger number of partially full pages.

In Figure 7.19, the performance speedup for a system using smaller, 4 kilobyte disk pages and more modest hardware is also presented. The speedup for this system demonstrates that linear speedups for join operations can

[49]This is a artifact of the single-user operation.

[50]However, this problem may not exist in a multi-user environment due to the random effects of competing for resources with other concurrently executing queries.

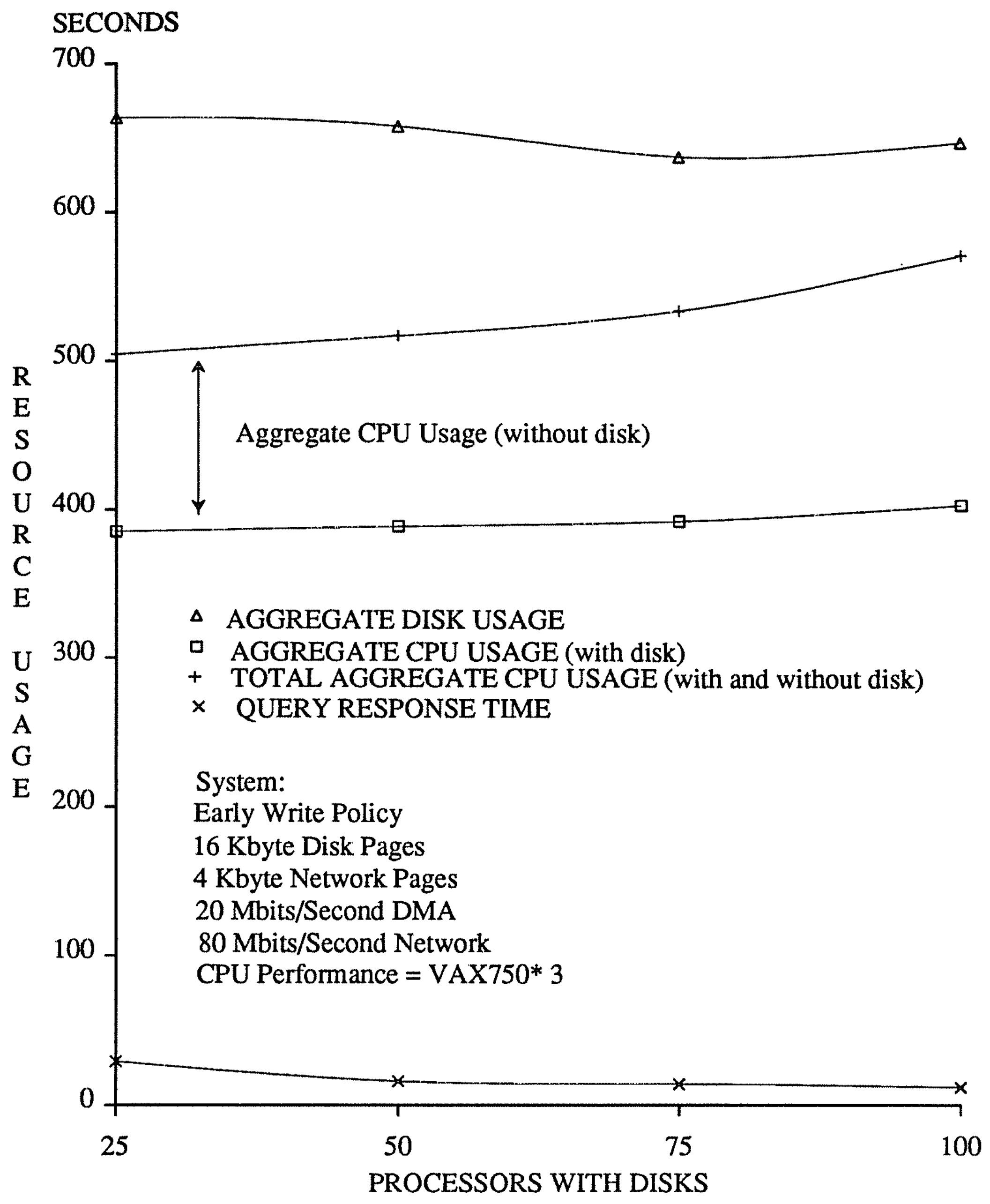
[51]For the system with 100 disks the output buffers of each processor with a disk are flushed at most twice.

indeed be maintained for very large systems. That is, the scheduling algorithm does not become a bottleneck when join operations are executed on configurations with large numbers of processors.

Despite the decline in speedup, both systems using 16 kilobyte disk pages still provided better absolute performance than the system using 4 kilobyte disk pages. For configurations containing one hundred disks, the system with 16 kilobyte disk pages and early writes of partition buffers performed the 1,000,000 tuple join query in 11.9 seconds, whereas the system with 4 kilobyte disk pages executed the query in 28.1 seconds.

While the system using 4 kilobyte disk pages also generated a highly periodic load on the network, the effect was less severe than for the system using 16 kilobyte disk pages. The reason for this difference results from the fact that systems with larger disk pages fill and flush partition buffers at a faster rate. That is, a larger disk page will yield more qualified tuples, each of which may potentially fill a partition buffer and trigger a network access. Furthermore, since systems with larger disk pages have lower response times, they generate higher levels of overall network utilization. With higher network utilizations, the convoys of processes waiting for tokens are formed more frequently and persist for longer periods of time.

The aggregate resource consumption for the 1,000,000 tuple join query that used early writes of partition buffers is presented in 7.21.



1,000,000 Tuple Join Query: Aggregate Resource Usage
Figure 7.21

CHAPTER 8

A MODIFIED GAMMA IMPLEMENTATION

The simulation model identified a number of software modifications capable of increasing the performance of Gamma. Specifically, the following changes were found to provide significant improvements in the response time of individual queries:

- Page-by-page round-robin partitioning.
- Larger disk pages.
- Collecting and writing full pages of tuples to disk.
- Parallel scheduling algorithms.

The first three of these modifications have been incorporated into the design of the Gamma prototype. In the following discussion, we investigate the impact that these changes have upon the measured performance of Gamma. The impact of the modified scheduling algorithm will be investigated in a future revision of the Gamma design.

8.1. Improved Data Streams

Gamma has been modified to partition result relations on a page-by-page round-robin basis. In the revised implementation, a single output buffer is used when a result relation is partitioned. When the buffer fills, it is written to the destination process associated with the current partition. The partitions receiving pages of tuples are alternated on a round-robin basis. Also, each of the processes producing fragments of a result relation use different initial partitions. While this method of round-robin partitioning does not provide as uniform a distribution of tuples as the earlier tuple-by-tuple method, it consumes fewer system resources and reduces network congestion. With page-by-page round-robin partitioning, only a single page has to be flushed across the network when a operator process that is producing a permanent result relation completes.

A second modification was also made to the way data streams are handled by Gamma. Previously, the **store** operators read pages of a result relation directly from the network into a page of the WiSS buffer pool. Later, these buffer pages were flushed to disk. This method of reading pages of a result relation directly from the network into disk buffers saved the CPU resources that would otherwise be spent copying tuples from network buffers to disk buffers. The simulation study indicated, however, that this strategy resulted in large numbers of partially full pages being appended to a file.[52]

To solve this problem, the **store** operator was modified to collect a full page of tuples before appending a page to a result relation on disk. While this change requires copying tuples between network and disk buffers, it results in at most one partially full page being appended to a file. In addition to the savings in time associated with fewer disk I/O operations, both for storage and later retrieval of the data, this change also makes it possible to separate the size and the format of the network and disk pages from one another. The impact of using distinct network and disk page formats is investigated in the following section.

The effect of the network congestion resulting from tuple-by-tuple round-robin partitioning was most evident in the earlier Gamma performance benchmarks when a clustered index was used for the following selection query[53].

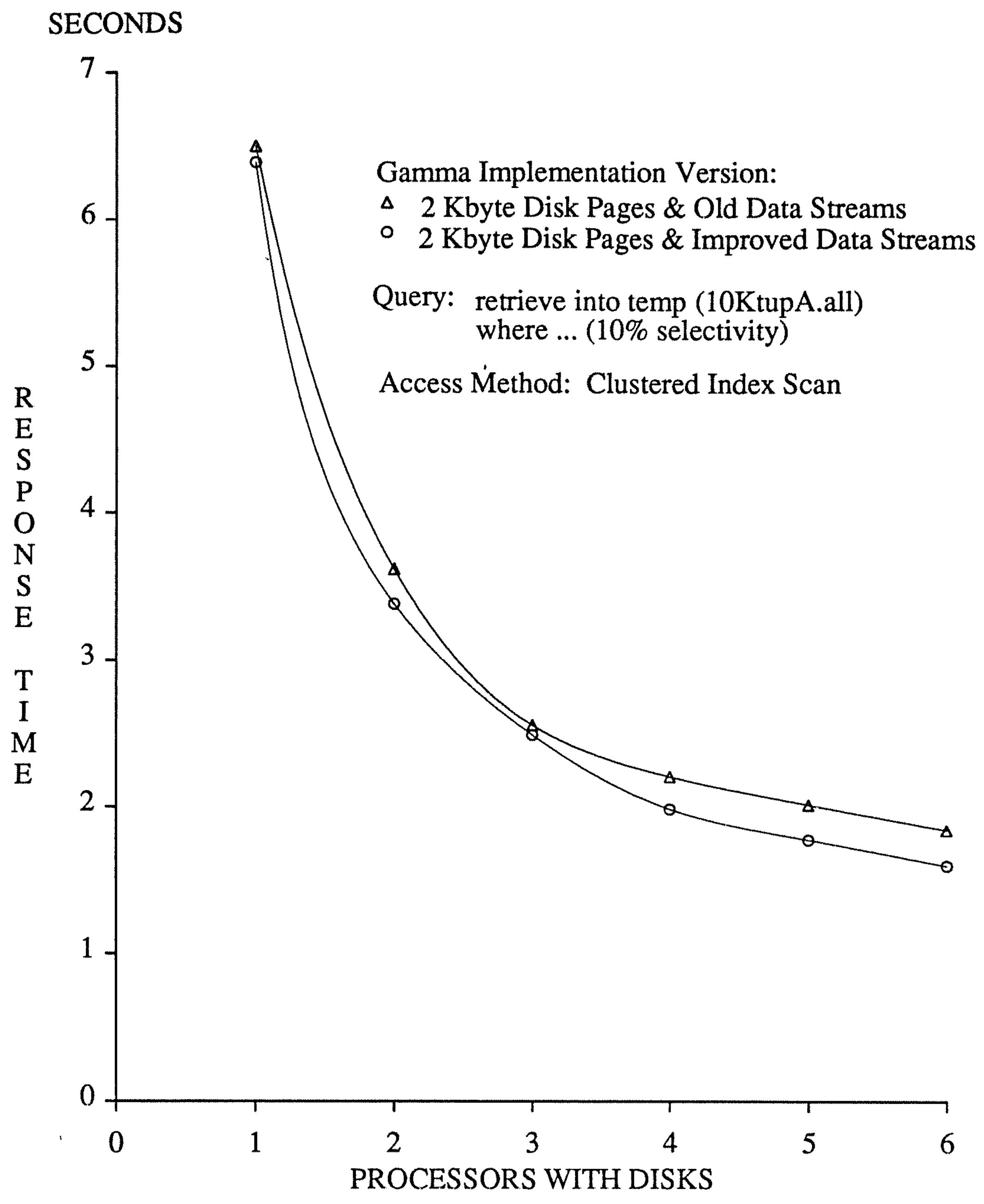
```
retrieve into temp (tenKtup.all)
where tenKtup.Unique1 > m and tenKtup.Unique1 < n
(m and n are constants between 0 through 9999 and n-m = 1001.)
```

Figure 8.1

For this selection query, Figure 8.2 illustrates the measured performance of Gamma when a clustered index is used.

[52]Recall that, potentially, one partially full page would be received for each of the input streams to a **store** operator.

[53]See Section 6.9



Gamma Selection Query Performance Benchmark
Figure 8.2

For a configuration with six query processors, the response time of the selection query was decreased by 14%. As suggested by the simulation model, this performance improvement can be expected to increase in importance as larger numbers of processors are employed. This results from the fact that tuple-by-tuple round-robin partitioning flushes an increasingly large number of partially full pages while page-by-page round-robin partitioning flushes a number of pages that increases linearly with the number of employed partitions.[54]

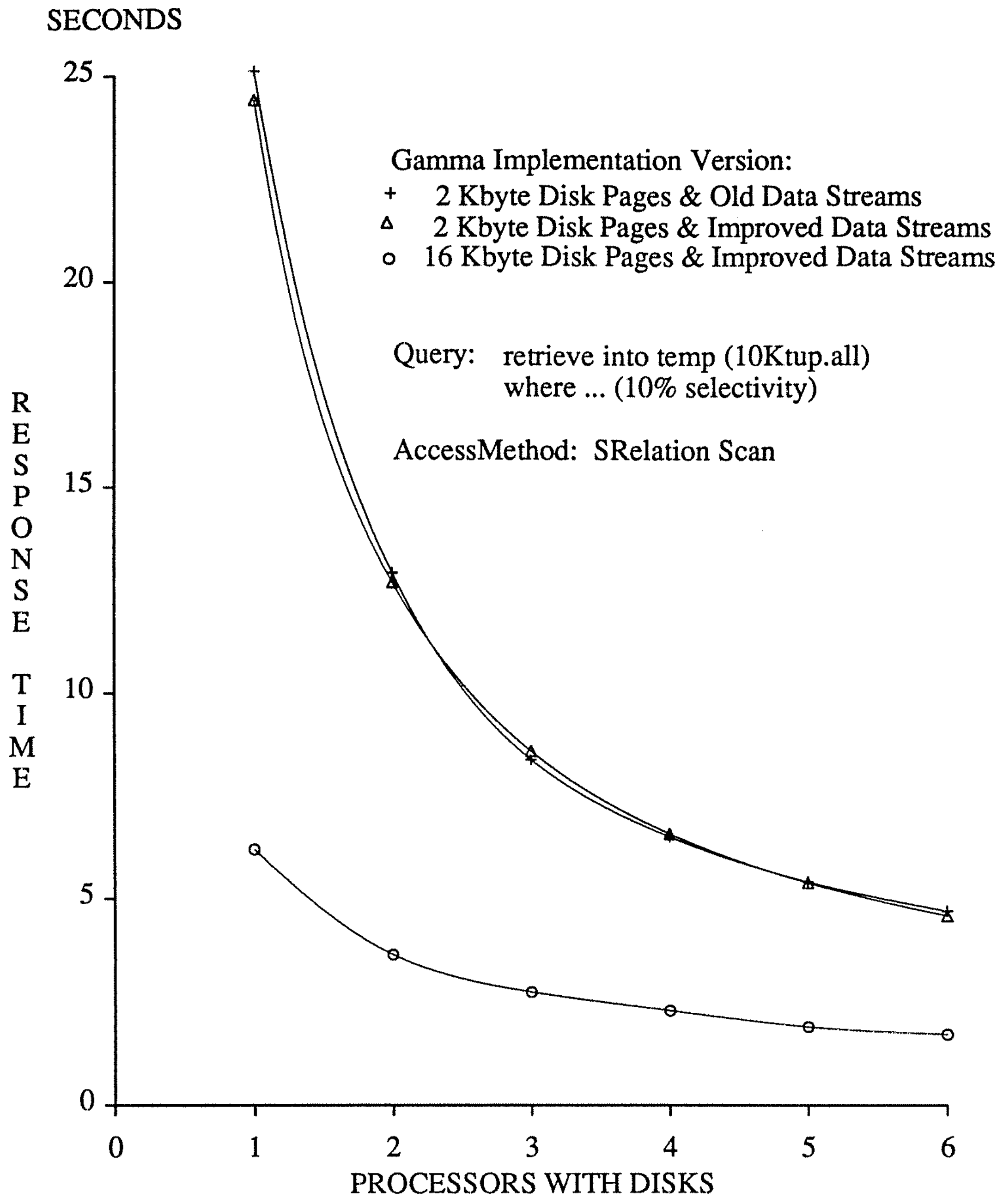
8.2. Larger Disk Pages

The simulation study also indicated that larger disk blocks could significantly decrease the response time of queries in Gamma. The time for Gamma to read and write pages of relations to and from disk was a large component of the execution time for many queries. For the benchmark queries, Gamma was found to be I/O bound. In the following section, we discuss one solution that reduces this bottleneck.

The initial Gamma implementation used 2 kilobyte disk pages. The modifications to data streams described in the previous section removed the requirement that disk pages be identical in size and format to the network pages. Thus, while Gamma continues to use the maximum 2 kilobyte network pages, disk pages have been increased to 16 kilobytes, the size of a track on a Fujitsu eight inch disk.

The impact of varying the size of the disk pages is most noticeable for queries that access a large amount of data. Therefore, in Figure 8.3,

[54]The number of partitions for a result relation is equal to the number of processors with disks.



Gamma Selection Query Benchmark
Figure 8.3

the effect of increasing the size of disk pages is illustrated for the selection query of Figure 8.1 for the case where indices do not exist. The performance of three implementation versions of Gamma are illustrated. All three versions used a relation scan to locate qualifying tuples. The first Gamma system used tuple-by-tuple round-robin partitioning. The second system included the data stream improvements described in the previous section along with 2 kilobyte disk pages. The similarity of performance of these two systems indicates that the data stream improvements have relatively less impact on the response time of queries that access larger amounts of data for systems of modest size.

The third system used the improved data streams and 16 kilobyte disk pages. The performance[55] of this version of Gamma improved approximately 300% with respect to the similar version that used 2 kilobyte pages. These results indicate that Gamma is indeed I/O bound for this query when 2 kilobyte disk pages are used. From a performance viewpoint, the increased disk page sizes are a simple yet effective enhancement to the Gamma design.[56]

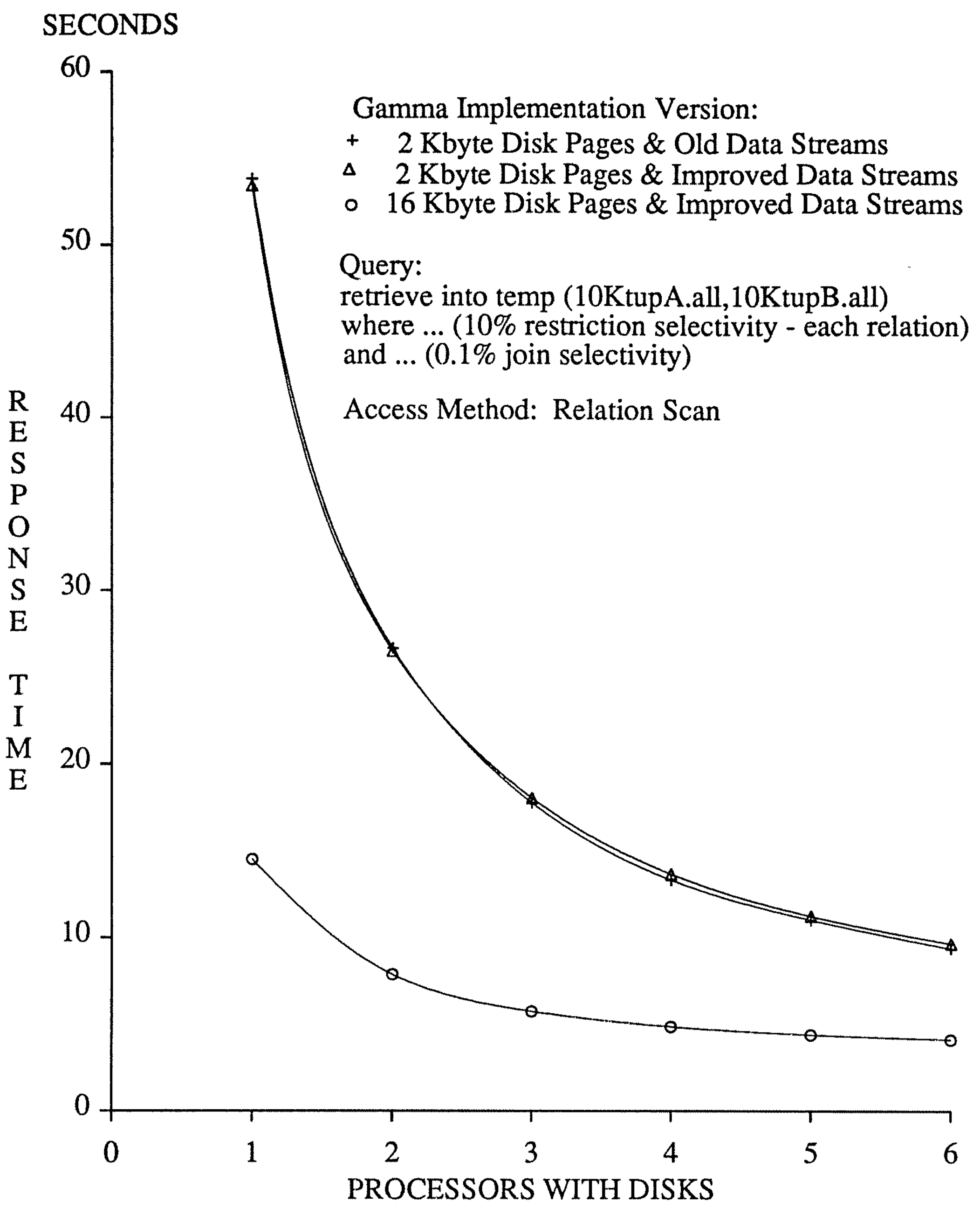
In Figure 8.4, the performance impact of larger disk pages is depicted for the following join query:

```
retrieve into temp (tenKtupA.all, tenKtupB.all)
where (tenKtupA.Unique2A = tenKtupB.Unique2B)
and (tenKtupB.Unique2B < 1000)
```

The same three versions of Gamma that were used for the previous selection tests were used for this join query benchmark, i.e. 2 kilobyte systems with and without data stream improvements and a 16 kilobyte system with the data stream improvements described in the previous section.

[55]Performance, in this case, was defined to be the inverse of the measured response time for the query.

[56]In fact, for this query, a Gamma system with 16 kilobyte data pages and no indices performed as well as a 2 kilobyte system with a clustered index on the selection attribute.



Gamma Join Query Performance Benchmark
Figure 8.4

The use of larger disk pages again provided very substantial improvements in performance.

CHAPTER 9

SUMMARY

9.1. Conclusions

In this dissertation, we have demonstrated that parallelism really can be made to work in a database machine context. The design and implementation of Gamma, a new relational database machine has been presented. Gamma uses hash-partitioned query processing algorithms for complex relational operations. These algorithms have been shown to provide an effective basis for the development of dataflow query processing techniques that pipeline data between operators at different levels of a query tree.

The highly independent, distributed processes that are used by Gamma can be controlled with an extremely low-cost scheduling algorithm that provides high levels of both intra-query and inter-query parallelism. A highly parallel scheduling algorithm has been presented that can effectively distribute queries across at least two hundred processors while providing nearly 100% linear speedups for complex queries containing selection and join operations. Furthermore, the results obtained for a single processor configuration of Gamma were demonstrated to be very competitive with those of a commercially available database machine.

The parallelism provided by Gamma has been demonstrated to complement the use of indices for selection queries.

A database machine architecture comprised of conventional, commercially available components has been constructed that provides high I/O bandwidths and can be expanded incrementally. This architecture illustrates that a high performance database machine can be constructed without the assistance of special purpose hardware components.

A simulation model of a distributed database machine has been developed that accurately reflects the measured performance of an actual, implemented database machine, Gamma. This simulation model has proven to be an invaluable tool during both the initial and retrospective design phases of Gamma. The model has also proven a valuable tool for exploring the impact of employing hardware resources of varying number and capability.

Finally, a useful testbed for database research has been developed. Gamma will be utilized for research into other aspects of relational database systems that can benefit from a highly parallel, high performance environment.

9.2. Future Research Directions

Using the Gamma prototype as a research vehicle we intend to explore a number of issues. Some of these issues include the use of adjustable join parallelism as a technique for load balancing and low priority queries, the effectiveness of alternative techniques for implementing bit vector filtering, index balancing algorithms and the effect of duplicating the root node at multiple sites, evaluation of alternative techniques for handling bucket overflows, and different strategies for processing complex queries. As mentioned earlier, an index join method should also be added to Gamma. Additionally, Gamma appears to a be a promising testbed for the development of distributed algorithms that compute the transitive closure of a relation [SCHN86].

A more extensive performance evaluation is needed that includes multi-user queries and considers the impact of concurrency control and recovery on the throughput of the system. The performance of other complex operators[57] also needs to be measured in Gamma.

[57]Aggregate functions, duplicate elimination, union, etc.

CHAPTER 10

REFERENCES

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

[ARMS68] Armstrong, J., Ulfers, H., Miller, D., Page, H., SOLPASS, A Simulation Oriented Language Programming and Simulation System, Technical Report, 1968.

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

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

[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.

[BELL73] Bell, J.R., "Threaded Code," Communications of the ACM, Vol. 16, No. 6, (June 1973), pp. 370-372.

[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.

[BLAS77] Blasgen, M. W., Eswaran, K. P., Storage and Access in Relational Data Bases, IBM Systems Journal, No. 4, 1977.

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

[BORA82] Boral, H. and D. J. DeWitt, "Applying Data-Flow Techniques to Database Machines," Computer, Vol. 15, No. 8, August, 1982.

[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.

[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.

[COOK80] Cook, R. P., "*MOD - A Language for Distributed Computing", IEEE Transactions on Software Engineering, 6, 6, November, 1980.

[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.

[DEWA75] Dewar, R.B.K., "Indirect Threaded Code," Communications of the ACM, Vol. 18, No. 6, (June 1975), pp. 330-331.

[DEWI79a] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June, 1979.

[DEWI79b] DeWitt, D. J., "Query Execution in DIRECT," Proceedings of the 1979 SIGMOD International Conference on Management of Data, May 1979, Boston, Mass.

[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.

[DEWI86] DeWitt, D., and R. Gerber, "GAMMA, A High Performance Dataflow Database Machine", Proceedings of the 1986 VLDB Conference, Tokyo, Japan, August, 1986.

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

[EPST78] Epstein, R., Stonebraker, M., Wong, E., "Distributed Query Processing in a Relational Database System", ACM-SIGMOD Conference Proceedings, Austin, Texas, May, 1978.

[FINK83] Finkel, R., Cook, R., DeWitt, D. J., Hall, N., Wisconsin Modula: Part III of the First Report on the Crystal Project, University of Wisconsin -- Madison, Technical Report 501, April, 1983.

[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.

[FUJI82] Fujitsu, Limited, M2351A/AF Mini-Disk Drive CE Manual, 1882.

[GARC84] Garcia-Molina, H. and Kenneth Salem, Disk Striping, to appear Proceedings of the 1985 SIGMOD Conference, also Dept. of Electrical Engineering and Computer Science Technical Report #332, December, 1982.

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

[GERB81] Gerber, R., Cook, R., and P. Clancy, The Starpak Modula Simulation Package, University of Wisconsin -- Madison, 1981.

[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.

[HAGM86] Hagmann, R. B., "An Observation on Database Buffering Performance Metrics", Proceedings of the 1986 VLDB Conference, Tokyo, Japan, August, 1986.

[HE83] He, X. et. al. The Implementation of a Multibackend Database Systems (MDBS), in Advanced Database

Machine Architecture, edited by David Hsiao, Prentice-Hall, 1983.

[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 Manager," Gamma internal design documentation, December, 1985.

[HEYT85b] Heytens, M., "The Gamma Catalog Manager," 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.

[JOHN82] Johnson, R. R. and W. C. Thompson, "A Database Machine Architecture for Performing Aggregations," Tech. Report UCRL - 87419, June 1982.

[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.

[KAMI85] Kamiya, S., et. al., "A Hardware Pipeline Algorithm for Relational Database Operations and Its Implementation Using Dedicated Hardware," Proceedings of the 1985 SIGARCH Conference, Boston, MA, June, 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, Tcchnical Report, 1983.

[KITS83c] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, Relational Algebra Machine Grace, RIMS Symposia on Software Science and Engineering, 1982, Lecture Notes in Computer Science, Springer Verlag, 1983.

[KNUT64] Knuth, D. E., and J. L. McNeley, A Formal Definition of SOL, IEEE Transactions on Electronic Computers, August, 1964.

[KNUT64] Knuth, D. E., and J. L. McNeley, SOL -- A Symbolic Language for General Purpose Systems Simulations, IEEE Transactions on Electronic Computers, August, 1964.

[LEHM86] Lehman, T., "Design and Performance Evaluation of a Main Memory Database System", Technical Report #656, University of Wisconsin, August, 1986.

[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.

[LIN76] Lin, S. C., Smith, D., and J Smith, The Design of a Rotating Associative Memory for Relational Database Applications, TODS Volume 1, No. 1, March, 1976.

[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.

[LOHM85] Lohman, G., et. al., "Query Processing in R*", in **Query Processing in Database Systems**, edited by Kim, W., Reiner, D., and D. Batory, Springer-Verlag, 1985.

[LU85] Lu, H. and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network" Proceedings of the 11th VLDB Conference, Stockholm, Sweden, August, 1985.

[MURA83] Murakami, K., et. al., "A Relational Data Base Machine: First Step to Knowledge Base Machine," Proceedings of the 10th Symposium on Computer Architecture, Stockholm, Sweden, June 1983.

[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.

[OZKA77] Ozkarahan, E. A., Schuster, S. A., and K. C. Sevcik, Performance Evaluation of a Relational Associative Processor, TODS Volume 2, No. 2, June, 1977.

[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.

[SACC86] Sacco, G., "Fragmentation: A Technique for Efficient Query Processing", ACM Transactions on Database Systems, Volume 11 Number 2, June, 1986.

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

[SCHN86] Schneider, D. and M. Skarpelos, "Design and Implementation of a Distributed Transitive Closure Algorithm", Working paper, University of Wisconsin, May, 1986.

[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.

[SHAW80] Shaw, D. E., A Relational Database Machine Architecture, Proceedings of the 1980 Workshop on Computer Architecture for Non-Numeric Processing, March, 1980.

[SHAW85] Shaw, D. E., "Relational Query Processing on the NON-VON Supercomputer", in **Query Processing in Database Systems**, edited by Kim, W., Reiner, D., and D. Batory, Springer-Verlag, 1985.

[SLOT70] Slotnick, D. L., Logic per Track Devices, TODS Volume 1, No. 3, September, 1976.

[STON71] Stone, H. S., Parallel Processing with the Perfect Shuffle, IEEE Transactions on Computers, February, 1971.

[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, Michael, "MUFFIN: A Distributed Data Base Machine", Proceedings of the 1st International Conference on Distributed Computing, Hunstville, AL, October, 1979.

[STON83] Stonebraker, M., et. al., "Performance Enhancements to a Relational Database System," ACM Transactions on Data Systems, Vol. 8, No. 2, (June 1983), pp. 167-185.

[SU75] Su, S. Y. W., Lipovski, G. J., CASSM: A Cellular System for Very Large Data Bases, Proceedings of the International Conference on Very Large Data Bases, September, 1975.

[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.