

Copyright © 1985, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

ARCHITECTURAL ALTERNATIVES FOR  
DATABASE MACHINES

by

Marguerite Clare Murphy

Memorandum No. UCB/ERL 85/87

5 November 1985

COVER PAGE

ARCHITECTURAL ALTERNATIVES FOR  
DATABASE MACHINE

by

Marguerite Clare Murphy

Memorandum No. UCB/ERL 85/87

5 November 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering  
University of California, Berkeley  
94720

**ARCHITECTURAL ALTERNATIVES FOR  
DATABASE MACHINES**

**Copyright © 1985**

**Marguerite Clare Murphy**

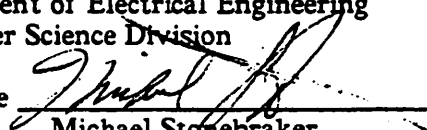
ARCHITECTURAL ALTERNATIVES FOR  
DATABASE MACHINES

Marguerite Clare Murphy

Ph.D.

Department of Electrical Engineering  
Computer Science Division

Signature



Michael Stonebraker  
Chairman of Committee

Sponsored by:

Defense Advanced Research Projects Agency (DoD)  
ARPA Order No. 4871  
Monitored by Naval Electronic Systems Command  
under contract No. N00039-84-C-0089

ABSTRACT

The goal of this thesis is to determine which architectural features are viable alternatives for relational database machines. This requires consideration of both the algorithms used during query processing and of the underlying hardware used to support those algorithms.

Two general approaches have been proposed for augmenting database management system performance: specialized hardware (database machines) and fast access data structures coupled with query optimization. The database machine approach attempts to increase the performance of key functions by implementing them in either firmware

or hardware. The query optimization approach attempts to minimize the computing resources required to execute a query by storing relations in specialized data structures and considering various alternative processing strategies. Although these two approaches are clearly related, database machines typically restrict processing techniques to those directly supported by the specialized hardware and optimization strategies largely ignore any unique characteristics of the underlying hardware.

The purpose of this research is twofold: to study and describe relational database queries in sufficient detail to allow the functional resource requirements to be accurately ascertained; and to evaluate various uniprocessor and multiprocessor architectures in an implementation independent manner.

The principal technique used in this investigation is to extend the cost functions used by query optimization routines to provide a more precise description of the functional resource requirements of database workloads.

The results of this thesis include a new high performance algorithm for distributed relational join processing (the Bloom-join algorithm), a new definition for selectivity which is consistent and which permits unbiased estimation of query resource requirements, and detailed cost functions for both the uniprocessor and multiprocessor join algorithms. Some specific results which follow immediately from these equations are: reducing the amount of data during query processing does not universally increase performance, adding additional processors to a multiprocessor does not necessarily increase performance, and the response oriented speedup for at least one class of multiprocessor algorithms and queries is closer to  $\log_2 N$  than  $N$ .

## TABLE OF CONTENTS

Chapter 1. INTRODUCTION .....	1
1.1. Database Machine Architectures .....	1
1.2. Motivations .....	1
1.3. Relational Operations .....	2
1.4. Survey of Database Machines .....	4
1.5. Overview of Thesis .....	17
Chapter 2. UNIPROCESSOR JOIN MODEL .....	19
2.1. Join Optimization .....	19
2.1.1. Strategies and Cost Functions .....	20
2.1.2. Data Buffering .....	20
2.1.3. Fast Access Structures .....	22
2.1.4. Storage Modification Algorithms .....	24
2.1.5. Join Processing Algorithms .....	26
2.2. Model Description .....	27
2.2.1. Parameters and Performance Metrics .....	27
2.2.2. Basic Operations and Cost Functions .....	29
2.3. Experiments .....	34
2.3.1. Workload Parameters and Output Statistics .....	34
2.3.2. Algorithm Choice .....	35
2.3.3. Fast Sort .....	37
2.3.4. Fast Search .....	38

2.3.5. Linear Relation Sort and Merge Join .....	39
2.4. Summary .....	41
Chapter 3. MULTIPROCESSOR JOIN MODEL .....	43
3.1. Multiprocessor Join Processing .....	43
3.1.1. Performance Considerations .....	45
3.1.2. Distributed Join Algorithms .....	47
3.1.2.1. D-Join Algorithm .....	47
3.1.2.2. Semi-Join Algorithm .....	49
3.1.2.3. Bloom-Join Algorithm .....	53
3.1.2.4. Fragment & Replicate Algorithm .....	55
3.1.2.5. Fragment & Rotate Algorithm .....	57
3.1.2.6. Distributed Hash Join Algorithm .....	59
3.2. Multiprocessor Join Model Description .....	61
3.2.1. Multiprocessor Performance Metrics .....	62
3.2.2. Selectivity Definition and Estimation .....	63
3.2.3. Cost Functions .....	69
3.3. Experimental Results .....	70
3.3.1. Total Time Minimization .....	70
3.3.1.1. Estimating Transmission Costs .....	71
3.3.1.2. Discussion of Transmission Costs .....	74
3.3.1.3. Estimating Total Costs .....	75
3.3.1.4. Simulation Results .....	79
3.3.1.4.1. Network Speed .....	81
3.3.1.4.2. Selectivity .....	83



3.3.1.4.3. Number of Duplicates .....	85
3.3.1.4.4. Algorithm Choice .....	87
3.3.2. Response Time Minimization .....	87
3.3.2.1. Estimating Response Time Costs .....	88
3.3.2.2. Discussion of Response Time Costs .....	90
3.3.2.3. Simulation Results .....	93
3.3.2.3.1. Total Time & Response Time .....	93
3.3.2.3.2. Associative Search Hardware .....	96
3.3.2.3.3. Broadcast and Ring Network Facilities .....	98
3.3.3. Multiprocessor vs. Uniprocessor Optimization .....	98
3.4. Summary .....	101
Chapter 4. CONCLUSIONS AND FUTURE RESEARCH .....	103
4.1. Conclusions .....	103
4.2. Future Research .....	104
Appendix A. Distributed Cost Functions .....	106
Appendix B. Total Time Calculations .....	112
Appendix C. Response Time Calculations .....	124

## ACKNOWLEDGEMENTS

I would like to thank and acknowledge some of the many people here at Berkeley who have contributed to my education both as an undergraduate in Electrical Engineering and as a graduate student in Computer Sciences:

Professor Michael Stonebraker, my advisor, for his guidance, support, encouragement and prodding over the past six years.

My dissertation committee members, Professors Alan Smith and Ronald Wolff, for reading through earlier drafts of this dissertation and providing invaluable criticism and insights.

Professors Eugene Wong, Larry Rowe and Domenico Ferrari for their advice and support, particularly during the final stages of writing this dissertation.

And all of the members of the Ingres research project, past and present, who have contributed to making a congenial environment for studying and exploring new ideas in database management.

Finally, I would like to acknowledge the financial support that I have received during my graduate career: from the Gee Foundation during 1980-81, from a University Opportunity Fellowship during 1980-81 and 1981-82 and from the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, Monitored by Naval Electronic Systems Command under contract No. N00039-84-C-0089 from 1983 through 1985.

The more abstract the situation in which knowledge occurs, the more clear and distinct this knowledge can be. Mathematical knowledge— at least in the first place— is so clear and distinct because the degree of abstraction is correspondingly high. It is possible, however, to limit and delimit even the most fully living object of knowledge, to fit it in, so to speak— as any spatial object can be fitted in— to geometrical figures. In this way, a mathematical-geometrical clarity and distinctness is attained: but, as a result of such geometrizing, the living object loses its concreteness and vitality. Any abstraction and reduction for the sake of clarity and distinctness is at the expense of concrete substantiality and fullness. Kant in his day drew attention to the fact that concrete knowledge is richer than mathematical. Clarity and unclarity cannot be so unequivocally distinguished in real life. There is a continual transition from the obscurity to the clarity of the image, with infinitely many degrees and stages.

Hans Kung, Does God Exist?

## CHAPTER 1

### INTRODUCTION

#### 1.1. Database Machine Architectures

This thesis describes a variety of algorithms for processing relational joins and ways in which the underlying machine architecture may be modified to better support this processing. Although the result of a relational join operation is well defined, there exist a multitude of processing strategies that will produce the correct result and which have execution costs that vary greatly depending on the characteristics of the relations being joined. There is no consensus on the "best" join processing technique, even when the alternatives are restricted to software solutions, and the problem becomes more complex with the potential of enhancing various operations through hardware assists.

#### 1.2. Motivations

Database machines (computers designed specifically for database applications) have been popular in the literature for many years. It is only within the last few years, however, that they have gone beyond paper designs and prototype implementations to become commercial products (for example, IDM from Britton Lee [Epst80a], iDBP from Intel [Inte82a], and the DBC/1012 from Teradata [84a] ).

One of the major benefits claimed for database machines is enhanced performance of database operations. Presumably some machines will attain this goal to a greater degree than others, depending on the individual design decisions made by the developers. While the existing literature provides a wealth of possible design alternatives, it does not provide criteria to guide ones' choice among them. The goal of this research is

to determine which architectural features are viable alternatives for database machines.

### 1.3. Relational Operations

In the relational model, all data appears to the user as two-dimensional tables called *relations*. Each relation contains an arbitrary number of records or *tuples* which have identically named fields or *attributes*. While every tuple in a relation has the same attributes, the values of those attributes are such that no two tuples are identical in all their attribute values (i.e. relations contain no duplicate tuples). Figures one and two contain examples of two relations, PILOTS and PLANES, with attributes "Name, License, Duty" and "Number, Type, Status" respectively. Each relation contains three tuples.

Of the various operators used for expressing relational queries (i.e. queries to data stored as relations), three appear much more frequently than the others [Ullm82a]. In this section we define these three operations, select, project and natural join, and discuss the amount of processing required for each.

The *select* operation creates a relation which is a subset of another relation. Each tuple in the result has attribute values which satisfy a qualification clause, where the

Name	License	Duty
Abe	727	on
Bob	727	off
Dee	707	on

Figure 1. PILOTS relation.

---

Number	Type	Status
101	727	ready
102	707	hold
103	707	ready

Figure 2. PLANES relation.

qualification clause may be any boolean expression involving at most a single attribute in every term. For example, a select operation against the PILOTS relation with qualification clause "LICENSE = 707" would create a relation whose tuples are a subset of the PILOTS relation such that every tuple in the result has a LICENSE attribute equal to 707. The result relation is shown in Figure 3. The maximum amount of processing required to calculate the select result is a single scan of the original relation (e.g. the PILOTS relation).

The *project* operation creates a relation each of whose tuples contains a subset of the number of *attributes* appearing in the original relation. Duplicate tuples are removed from the result so the number of tuples may decrease as well as the number of attributes per tuple. For example, if the LICENSE attribute is projected from the PILOTS relation, both the NAME and DUTY attributes are removed from each tuple, and the resulting duplicates removed from the result. Figure four contains the projection result. The processing required for this operation is at most a single scan to remove the attributes plus the overhead to remove duplicate tuples from the result, typically by sorting the result tuples.

The relational join operation provides a means of combining data from two relations. Conceptually, the join operation selects those tuples from the cross product of the two relations which have "qualifying" join attributes. In the case of an *equi-join*

Name	LICENSE	DUTY
Dee	707	on

Figure 3. SELECT (LICENSE = 707) FROM PILOTS

License
707
727

Figure 4. PROJECT LICENSE FROM PILOTS

qualifying tuples have equal join attributes. This is the only kind of join that will be considered in this thesis: when the term "join" appears, equi-join will be implied. Figure five contains the result of a join between PILOTS and PLANES on the License and Type attributes. Note that the result of a join operation is itself a relation. A number of algorithms exist for calculating the natural join and the overhead varies greatly depending on the storage structure of the two relations. In the best circumstances, one relation must be scanned and for every tuple encountered one page of the other relation read <sup>1</sup>. In the worst circumstances, the cross product of the two relations must be formed and scanned to determine the result. Since the natural join overhead tends to dominate the overhead required by the other two operations, it is the only operation considered in this thesis. Several of the distributed join processing techniques evaluate joins by breaking them into a sequence of simpler relational operations, including projects and joins between various intermediate results. For these algorithms, we explicitly include the project processing costs in addition to the join costs.

#### 1.4. Survey of Database Machines

Database machines attempt to increase overall performance by implementing key functions in either firmware or hardware. There is little agreement on which functions are the most important and many of the proposals seem to be motivated by properties of the hardware technology rather than a realistic appraisal of database management

Name	License	Duty	Number	Type	Status
Abe	727	on	101	727	ready
Bob	727	off	101	727	ready
Dee	707	on	102	707	hold
Dee	707	on	103	707	ready

Figure 5. JOIN PILOTS ON LICENSE WITH PLANES ON TYPE

---

<sup>1</sup> We assume that the number of logical page accesses is equal to the number of physical page accesses, as explained below.

system resource requirements. This section gives a brief taxonomy of representative machines of various kinds. Note that the machines do not fit neatly into exclusive categories and alternate taxonomies exist which are equally justifiable. See, for example [Hawt79a, Good80a, Bray79a]. Our categories are hierarchically arranged. At the highest level the machines are divided into uniprocessor and multiprocessor database machines. These broad categories are further broken down into subcategories. Each subcategory is defined below and several representative machines are described for each.

#### **1.4.1. Uniprocessor Database Machines**

Uniprocessor database machines attempt to increase performance by increasing the execution speed of particular functions without using multiprocessors. This may involve firmware implementation of selected functions, hardware assists for cpu operations or data filters which operate on data as it streams from the disk to the cpu.

##### **1.4.1.1. Custom Microcode**

Perhaps the most straightforward way to design a database machine is to begin with a traditional von Neumann architecture and add custom microcode to enhance the performance of database applications. This approach has been successfully used by the IDMS and ADABAS database systems running on IBM 370 hardware [Ston83a].

A recent Japanese proposal [Seki83a] concluded that microcoded assists would result in a factor of three to five decrease in the execution time of "basic database operations" (e.g. index search and scan operations, bit-map operations, sorting operations, address translation and tuple fetch). In addition, 1000 microsteps of microcoded database management routines were found to take about 40% of the execution time of their software counterparts, resulting in a factor of two improvement in total execution time. Another recent proposal [Ston83a] evaluated the potential for performance



improvement through custom microcode for the Ingres database management system running on a Digital Vax 11/780 computer. The conclusion was that the estimated 3-5% improvement in total execution time did not justify the expense of microcoding an estimated 1000 lines of microcode.

There are several reasons for these widely varying conclusions. First, the Vax 11/780 has an extensive collection of high level instructions, including variants of all the "basic database operations" described above except tuple fetch. It was deemed unlikely that a novice programmer could write microcode that would outperform that of the machine's developers. In addition, as explained in [Patt85a] the falling cost of high speed memory has decreased the difference in access time between the fast, expensive memory used for microstore and the slower, less expensive memory used for main memory. For modern implementations, the factor of three difference assumed in [Seki83a] is not realistic. Finally, most recent high performance implementations of von Neumann architectures utilize pipelined instruction fetch and hence do not incur instruction fetch overhead, except for failed branch instructions. The Vax 11/780 has an instruction prefetch unit. A non-pipelined architecture was assumed in [Seki83a] and avoiding instruction fetch is one of the primary sources of performance improvement in that study.

A general conclusion from these studies is that older, simpler implementations of architectures are more amenable to microcoded assists [Ston83a]. However [Patt85a] claims that, with current technology, *no* machine with complex microcode is likely to outperform a simple processor.

#### 1.4.1.2. Sort/Search Hardware

The next most straightforward method to implement a database machine is to augment the microarchitecture of a von Neuman machine with supplementary (nonprogrammable) hardware to enhance the performance of database operations.

[Dohi82a] propose an architecture with a tree structured sorting network that operates on compressed data to produce ordered relations. Algorithms for executing the relational set operations and data manipulations on sorted relations are presented. In [Tana84a] a similar architecture is proposed which uses VLSI to implement an interval search engine and a two-way-merge sorter. The hardware is bit-sliced to allow for variable wordlengths.

#### 1.4.1.3. Filters

The previous two sections have described techniques for increasing the speed at which data is processed. This section describes a collection of techniques to increase performance by reducing the amount of data that is read from disk into the cpu for processing. Database machines which use these techniques are called filters. This subsection concludes with descriptions of three database machine filters: CAFS, IDM and TUNABLE FILTERS.

#### CAFS

The CAFS machine [Ward84a, Bray79a] is a very early example of a data filter. CAFS was designed to accept data in parallel from a channel multiplexor connected to several disks, perform a fast keyword search to filter out superfluous data, and output the result to the host computer. [Babb79a] describes ways in which the machine may be enhanced with a hashed bit array for rapid execution of joins and removal of duplicate data after a project operation.

#### IDM

One shortcoming of the CAFS machine is the requirement that the entire relation be scanned during processing. The IDM produced by Britton Lee [Epst80a] attempts to overcome this difficulty by using low level access methods (e.g. index structures) to reduce the amount of data that must be examined by the database machine itself. A

multi-threaded environment permits several queries to execute concurrently and specialized hardware allows data to be processed at the rate it is read from disk. In addition, a disk cache is used to further reduce disk traffic.

## TUNABLE FILTERS

Tunable filters are a recent proposal [Kies84a] which use dynamic filters (i.e. filters whose search criteria may be modified during execution) to allow more complex search criteria for relations and sets of tuple id's. The filters are tunable: they will always retrieve a superset of the qualifying tuples and may be made as precise as desired at runtime by increasing the amount of time spent calculating the filter's parameters. Tunable filters allow a variety of join processing algorithms. [Kies84b] presents query optimization techniques using dynamic filters.

### 1.4.2. Multiprocessor Database Machines

The basic principle behind multiprocessor database machines is the contention that a collection of many small slow processors can provide the same processing power as one large, fast processor at a greatly reduced cost. The arithmetic of adding together the effective execution rate of the two options and comparing the purchase cost is straightforward, the implementation of algorithms to *realize* the potential execution rate of a multiprocessor database machine is not. Many proposed machines are limited by "Amdahl's argument": the number of nodes that can usefully be put to work concurrently on a given problem is limited by the reciprocal of the fraction of the computation that must be done sequentially [Seit85a]. As an extreme example, if an algorithm requires that individual tuples be processed in a fixed order, the addition of more processors will not increase performance since each processor will wait until one of its tuples is ready to be processed, and only one tuple at a time will ever be ready. The common counterargument that in a multiprogramming environment with a well

balanced load of independent problems, the *overall* throughput of a multiprocessor is higher does not apply directly since most of the proposed database machines do not support multi-threaded execution. The performance gain for these machines results from decreased response time of individual queries, allowing a greater number of queries to be processed sequentially in a given period of time and hence allowing greater throughput.

#### 1.4.2.1. Parallel Multiprocessor Database Machines

Parallel multiprocessor machines attempt to increase performance by splitting a problem into smaller, identical problems that can be solved in parallel on identical processors. The earliest parallel database machines utilized a processor per disk track to search for tuples as they were being read from secondary storage to memory [Bray79a]. More recent proposals attempt to incorporate more complex functions, such as sort and join, and support for concurrent, secure, crash resilient execution [Good80a].

##### 1.4.2.1.1. Associative Search Processors

Associative memory allows access of any data element in one access without prior knowledge of its location. This is accomplished by associating a processor with every word of data stored in the machine and performing key comparisons in parallel. STARAN is an example of an associative search processor.

#### STARAN

The earliest commercial associative processor was the STARAN machine [Rudo82a]. Although originally designed to process image data, STARAN was modified to execute database management functions by rewriting the software [Bray79a]. The primary difficulty was the need for high speed transfer of data into the processor from disks via the host and output of results from the processor back again to the host. In [Berr79a] various interfaces and buffering techniques are evaluated for bit-slice

associative processors of the STARAN type. It is claimed that these techniques will overcome both the problems of staging data into the processor rapidly enough and of building a processor large enough to handle realistic problems.

#### 1.4.2.1.2. Logic per Track Devices

Logic per track devices are based on the same principle as filters: increase performance by reducing the amount of data read from the disks into memory for further processing. Conventional rotating memory devices have a collection of storage areas (tracks) and one or more read/write heads which transfer data to and from the device. Associating a processor with every track allows data to be processed as it is read from the device, permits multiple heads to be more effectively utilized and allows data to be addressed by content instead of by address [Su79a].

A major difficulty with this approach is providing disk error detection and recovery. Common techniques based on cyclic redundancy codes group the data stored on the disk into blocks. In order to read a single record, the entire block must be fetched from the disk and processed to verify that it is error free. Typically a *disk controller* is located between the processor memory bus and the disk hardware. Logic per track devices assume a much simpler interface and it is not clear how they map into this more realistic model of disk activity.

### CASSM

The CASSM database machine [Su79a] is a logic per track device which utilizes one processor per fixed-head disk track to provide efficient parallel access to an hierarchically organized database. Multi-threaded execution of queries is not supported.

#### 1.4.2.1.3. Join Processors

The machines described above were designed to *search* through data rapidly. In relational database management systems the join operation tends to be a greater source of overhead, as explained above. This section describes database machines designed to optimize join performance. The generic multiprocessor architecture contains a collection of disks (possibly augmented with filters), a memory buffer area and a collection of query processors. Various interconnections between processors, disks and memory have been proposed. In addition, there are many techniques for allocating processors to queries and executing joins using multiprocessors. Six examples of join processors follow.

#### RAP

The RAP [Schu78a] machine organizes processors into cells containing a processor and a private memory. A single bus links the host processor and the cells and data is staged from secondary storage into the memory for processing. Joins are implemented using a "cross-mark" operation: one relation is scanned and all qualifying tuples are marked. Then, for every tuple marked in the first relation, all tuples in the second relation with a matching join field are marked [Bray79a]. In order to permit parallel operation, the relations are partitioned among the cells. Each cell executes a "cross-mark" operation on its partition of the data, then the data is redistributed among cells and the process repeated. If  $N$  processors participate in the join,  $N$  processing steps are required. Note that multi-threaded execution is not supported by RAP.

#### JOH

The Join Operations Hardware (JOH) proposal [Meno83a] connects processors with private memories in a ring network. The connections to secondary storage are not described, although the discussion of expected performance in the papers would indi-

cate that there is a direct connection. Joins are processed by first distributing the source relation evenly among the processors. Each processor stores its allocation of tuples in a hash structure and places the join attributes in a private associative memory. Tuples from the target relation are routed among all the processors in broadcast sequence (i.e. around the ring). When a target tuple arrives at a processor, the associative memory is checked and if a match is found the tuple is retrieved from the hash structure to form a result tuple. The target tuple is then sent to the next processor in the ring.

The proposal contains a detailed analysis of the sizes of memories needed to store intermediate results and queue tuples propagating around the network without overflow. The architecture is compared with other machines on the basis of the join repertoire supported (e.g. natural, implicit, inequality and m-way joins), the time complexity of the join and whether processing may overlap data staging. JOH compares very favorably.

## **DIRECT and SABRE**

In both the DIRECT [DeWi78a, DeWi82a] and SABRE [Vald82a] machines the processors are connected to the memory units via a cross-point switch. This allows data to be staged into the machine once, then accessed by any processor without relocation. [Bora81a] investigated the performance of various processor allocation strategies for Direct/Sabre and concluded that MIMD or data-flow techniques<sup>2</sup> lead to maximum performance. [Bitt83a] evaluated various join processing algorithms (excluding those which utilize indices) and concluded that the merge join algorithm is superior to the nested loops algorithm for relations of approximately equal size and the converse is

---

<sup>2</sup> The DIRECT architecture is sufficiently general to support either parallel or pipelined execution. The data-flow strategy proposed is a generalization of a pipelined strategy. The MIMD strategies allow multi-threaded parallel execution of queries.

true when one relation is much larger than the other.

Two multiprocessor join algorithms based on semi-join techniques for a DIRECT-like architecture augmented with data filters on the disks have been proposed by [Vald82a]. The bit-array semi-join uses a technique similar to that described in [Babb79a]. Each bit of the array is associated with a single hash value. The array is set by sorting one relation <sup>3</sup>, hashing the tuples and setting the corresponding bit for each tuple. The join is computed by hashing each tuple of the second relation, checking if the appropriate bit is set in the bit-array and, if the bit is set, checking if the first relation contains any qualifying tuples. A method for performing these computations in parallel on a variable number of processors is described.

The second semi-join algorithm, selection semi-join, creates a projection of the join attribute from the first relation, then uses this projection to drive the data filters to select out those tuples of the second relation which qualify. Performance comparisons indicate that for a machine containing eight processors and eight data filters, the nested loops semi-join algorithms will outperform nested loops algorithms in all cases except when the join selectivity approaches that of the cartesian product of the two relations.

## GRACE

The GRACE database machine [Kits83a] performs joins by hashing relations into buckets (memory buffers), then sorting each bucket and performing merge joins between corresponding pairs of buckets. A hardware sorter is associated with each bucket and a hash bit-array (similar to that proposed in [Babb79a]) is associated with each disk module. Data is staged into the buffers in parallel. Since sorting cannot be

---

<sup>3</sup> Sorting is used to detect duplicate join attribute values, although it is not clear that removing these values is more efficient than "resetting" the bit vector when the duplicates are encountered.



initiated before the last data element has arrived, a two phase execution strategy is required. It is demonstrated that the computational complexity of the join operation is reduced from  $O(n^2)$  to  $O(n)$ , however the coefficients and low order terms are not given so it is not clear what the expected performance will be for non-asymptotic problems.

### DBC/1012

The DBC/1012 is a new product recently announced by Teradata [84a]. A collection of processors are associated with disk storage units and connected by a tree structured network. Data is distributed among processors by using a randomizing hash function which partitions the data evenly. The network has broadcast capability in addition to the direct tree connection. This permits very efficient sorting (using a Tournament Sort algorithm) and allows commit acknowledgements to be merged and processed in a straightforward manner by using numeric codes for commit and abort such that one is strictly less than the other. Joins are processed using the merge join algorithm. Performance studies [Nech84a] indicate that this architecture will provide a lower cost machine with the same response time as other architectures with different interconnections, memory technologies and expected number of concurrent users.

### MBDS

The MBDS database machine [Demu85a, Demu84a] associates processors with private disks over an ethernet-like broadcast network. One processor acts as the controller and all other processors execute identical software in parallel. The database is evenly distributed across the processors using a so called cluster-based data placement algorithm. Benchmark studies performed using a simulation built on a VAX-11/780 (VMS-OS) and two PDP-11/44s validate the claims that if the database size remains constant, the response time is inversely proportional to the number of backends; and if

the number of backends grows as the database size increases, the response time will remain invariant. The benchmark does not contain join queries and their implementation is not described, however the architecture is similar to a join processor as described above and could execute many of the algorithms presented in this section.

#### 1.4.2.1.4. Distributed Architectures

Distributed architectures resemble conventional distributed systems. A collection of processors with private memory and secondary storage are connected by a communications medium. The processors are capable both of executing independently and of co-operating with each other on a single query. Many "database machines" of this type could also properly be called "distributed database management systems."

#### MUFFIN

The Muffin database machine [Ston78a, Ston79a] is a distributed database machine fundamentally oriented towards multiprocessing of database commands. The basic design goals are high transaction rates through specialization of function, resiliency to failure and support for a wide range of database sizes and transaction complexity. The hardware is conventional, however the operating system is minimal and only provides those functions required for database management. Distributed join algorithms (e.g. the fragment-and-replicate algorithm [Epst78a] ) are proposed for this architecture.

#### JASMIN

The JASMIN database machine [Fish84a, Lai84a] is a functionally distributed database machine designed to support large databases and high transaction rates. It is implemented on conventional hardware using a multiprocessing operating system which permits implementation of replicated software modules. Performance may be tuned by redistributing modules among processors. Concurrency control, crash recovery and version consistency are directly addressed by the proposal. Join

processing is not described.

#### 1.4.2.2. Pipelined Multiprocessor Database Machines

Pipelined multiprocessor machines attempt to increase performance by splitting the problem into a sequence of smaller problems and pipelining them through a series of specialized processors. Peak performance is determined by the longest segment of the pipeline, however in many applications dependencies between different segments of a computation reduce the average performance to below that of the peak rate.

##### DBC

The DBC database machine [Bane78a] has two pipelined processing loops, the structure loop and the data loop, and seven major processing elements. The data base command and control processor provides an interface between the two loops and a host computer. The data loop contains a mass memory unit and a security filter processor. The structure loop contains an index translation unit, a keyword transformation unit, a structure memory (used for performing set intersections) and a structure memory information processor. A performance analysis comparing DBC with a relational system supported on a conventional computer [Bane78b] concluded that while the DBC requires a factor of one-to-two more storage space, query execution times are likely to be at least an order of magnitude better. Benchmark studies on prototype hardware have not yet been published.

##### iDBP

The iDBP Database Machine is an Intel product and is one of the few commercial database machines currently on the market [Inte82a]. The iDPB functions between a host computer and up to four disk spindles [84a]. Two 8086 processors are divided functionally between four subsystems: operating system and database management system software; memory subsystem; communication protocol processing; and mass

storage subsystem. Execution is multi-threaded and multiple iDBPs may be used in complex systems, e.g. to manage so called "disk farms."

This concludes our discussion of database machines.

## 1.5. Overview of Thesis

This thesis will examine a variety of uniprocessor and multiprocessor join processing techniques and evaluate the performance improvements resulting from various architectural enhancements. In the following two chapters we shall present a series of models representing uniprocessor join processing and multiprocessor join processing. For each model, a collection of experiments are run to evaluate both the techniques and the potential for performance improvement through architectural enhancement. The thesis concludes with a summary of the major conclusions and outlines areas for future research.

Basic techniques for processing a single join in a uniprocessor environment are presented in chapter two. These techniques include a collection of fast access structures, several join processing algorithms and a method of query optimization. Query optimization is the process of generating alternate execution plans for a given query and selecting the best (i.e. minimum cost) one for execution. The method used in this dissertation generates an exhaustive collection of plans and provides a detailed description of both the total estimated cost and the number of elementary operations performed. The join processing model accepts as input a collection of parameters describing the query workload characteristics and the relative costs of the elementary operations and produces as output the average execution cost per query and the number of times each operation was executed. The chapter concludes by describing a collection of experiments evaluating various architectural enhancements. Some specific questions addressed are: Which combinations of join algorithms generate the best (i.e. minimum cost) execution plans during query optimization? What are the effects of enhancing the

performance of various basic operations (e.g. page sort and page search)? How would the performance of a machine built around a linear time sorting device and the merge join algorithm compare with a database system built around a collection of storage structures and join processing algorithms?

In chapter three, the join processing techniques are extended to a multiprocessor environment. Multiprocessors are abstracted by a collection of execution nodes with private storage connected by a communication medium. A wide variety of algorithms based on semi-joins have been proposed for this environment [Yu84a], however they are all based on the assumption that the network transmission costs dominate and neither cpu nor disk access costs need be considered in selecting an execution plan. This assumption is not valid for tightly coupled multiprocessors nor loosely coupled multiprocessors linked by a local network. We examine six multiprocessor join algorithms in detail, including semi-join algorithms, and consider all processing costs: processing, input/output and data transmission. One of these algorithms, the bloom-join, is a new algorithm which tends to require fewer data transmissions than an equivalent semi-join and less total processing than any of the other algorithms. The join model presented in this chapter is a direct extension of the uniprocessor join model described in chapter two. We assume that the two relations and the join result are located on distinct nodes. Some specific questions addressed by the experiments are: Do algorithms which minimize data transmissions ever win if total processing costs are taken into account? Do distributed join strategies which minimize data transmissions perform well compared with those which minimize response time? How significant are accurate selectivity estimates for effective query optimization? What are the effects of assuming that join attributes contain no duplicate values?

The final chapter, chapter four, contains the conclusions and suggestions for future research.

## CHAPTER 2

### UNIPROCESSOR JOIN MODEL

This chapter introduces uniprocessor join optimization and presents a performance evaluation of a variety of database machine architectures. The evaluation is based on an extension of the cost functions used in query optimization which more precisely estimates the resource requirements as the cost of executing the basic operations varies. The chapter begins with a definition of the join optimization problem. Various fast access structures and algorithms used for query processing are then described and form the basis for the performance model. This model is described in detail and the results of a series of experiments are presented.

#### 2.1. Join Optimization

In the relational model, all data appears to the user as relations, as defined in the previous chapter. Relations provide a simple conceptual view of the data for the user. When relations are implemented, the data is broken into fixed size blocks or *pages* (which may or may not correspond to the data as seen by the user) and stored on secondary storage devices, referred to generically as disks. One of the primary functions of a database management system is to automate the mapping between the logical relations and the physical data files. The user queries the database by logically describing the data he wishes to have retrieved using a high level query language. The system translates these high level queries into a series of operations on the underlying data files. Typically a complex query is broken into a sequence of simple relational operations (e.g. select, join), which are then translated into the actual file accesses.

### 2.1.1. Strategies and Cost Functions

For any given relational query there are a number of different sequences of operations, or *strategies*, that will yield the correct result. The optimization problem is to generate a set of strategies and select the one with the minimum cost. For the simplest queries, an exhaustive collection of strategies can be generated (within the range of options supported by the database management system). As the number of operations in the query increases, however, the number of strategies increases rapidly and the problem becomes one of generating the smallest subset that will contain at least one "good" strategy. In this chapter we will discuss optimization of a single join operation and optimization will be performed by generating an exhaustive collection of strategies. We describe these strategies in detail below. Briefly, each strategy consists of a possible modification to the storage structure of each relation followed by calculation of the join result using one of the available algorithms.

The cost associated with a strategy is usually a rough measure of the computing resources required for the sequence of operations. A typical metric is the number of disk accesses [Ceri84a]. Although [Kooi80a] describes a regression analysis of benchmark traces indicating that the number of disk accesses is a good predictor for the total processing required by a query, this metric is based on an implicit assumption that the amount of data referenced is sufficiently large that the fixed overhead associated with executing the query is negligible (i.e. that the query is "data intensive" [Hawt79b, Hawt79a]). The cost model described below includes a detailed accounting of both CPU and I/O processing.

### 2.1.2. Data Buffering

Data buffering strategies allow multiple pages to reside in main memory during query processing. If a query requests access to a page that is in the buffer pool (i.e. in memory), that page is not read in from disk. If the page is not in the buffer pool, it

must be read in from secondary storage, possibly displacing another page already in the buffer pool. The number of page accesses requested by the query (the number of *logical* page accesses) and the actual number of accesses to secondary storage (the number of *physical* page accesses) must be carefully distinguished. Query optimization cost formulas are typically in terms of the number of *logical* page accesses and do not consider buffering explicitly.

Regardless of the buffering strategy, every data page accessed by the query must be read from secondary storage at least once. If the page is processed in multiple steps (e.g. if tuples are processed sequentially with additional processing interleaved) the page may be overwritten before all of the tuples have been processed. When this happens, an additional transfer from secondary storage will be required, tending to make the number of physical page accesses *greater* than the number of logical page accesses. If, on the other hand, the buffer pool is large and pages are logically accessed multiple times, there is a possibility that a page will already be in the buffer pool when it is logically accessed. This will tend to make the number of physical page accesses *less* than the number of logical page accesses.

This thesis assumes that the buffer pool is small to moderate sized and that the effects of premature overwriting of data pages are balanced by random re-references to pages already in the pool. In this case, the number of logical accesses is equal to the number of physical accesses. Many techniques exist for decreasing the relative number of physical page accesses. For example, increasing the size of the buffer pool [Sacc81a], using a "predictive" page replacement algorithm to reduce the number of times a page is reread [Smit76a], and modifying the query processing algorithms to take advantage of buffering by creating small temporaries which can fit into the buffer pool in their entirety [Kris84a]. Although these techniques are important in their own right, they are not considered further here.



### 2.1.3. Fast Access Structures

Relations consist of some number of tuples which are stored on data pages. These tuples and pages may be organized into a storage structure for more efficient access either by the placement of tuples on data pages or by auxiliary indexing structures. Five structures are considered here: heap, ordered heap, hash, key sequential index and secondary hash index. Each structure is assumed to be built on the join attribute. The first four are primary structures, that is they contain the entire tuple. The fifth is a secondary structure which contains join attribute values and pointers to the tuples, which are themselves stored in some primary structure on an another (non-join) attribute.

#### HEAP

A heap <sup>1</sup> is an unordered structure with tuples placed randomly on data pages. A relation stored in a primary structure on an attribute other than the join attribute will have a heap structure with respect to the join attribute. In order to retrieve all tuples with a given value in the join attribute, the entire relation must be searched.

#### ORDERED HEAP

An ordered heap structure stores the tuples on data pages in join attribute value order. Both the tuples on pages and the pages themselves are ordered. All tuples with a given value for the join attribute may be retrieved by a binary search of the relation: the middle page is examined to determine whether the tuples are in the upper or lower half of the relation, then the middle page of the appropriate half relation is examined and so on until the required page has been located. This page is in turn searched using a binary search to locate the required tuples.

---

<sup>1</sup> Relational database terminology uses the term "heap" to refer to an unstructured file which is unrelated to the "heap data structure" defined in [Knut73a]

## HASH

A hash structure stores tuples by applying a hash function to the attribute value to obtain a "bucket" address where the tuple is stored. Tuples within a bucket are unordered. A bucket may be as small as a tuple address within a page or as large as several pages. Collisions occur when multiple attribute values hash to the same bucket and overflows occur when the number of collisions to some bucket is greater than the bucket size. The simplest technique to handle overflow is to chain an additional hash bucket to the first and treat both as one large (unordered) bucket. Subsequent overflows are handled analogously, with the size of the resulting bucket increasing at each step. If no overflows are present, then a bucket size of one tuple address requires the minimum access time. However, if overflows are present, the chains tend to become unduly long and the access time increases as larger buckets are searched for matches. A bucket size of one page and no overflows are assumed. In order to retrieve all tuples with a given attribute value, one page is read and searched (the page address is calculated from the attribute value). This overhead is independent of the relation cardinality.

## KEY SEQUENTIAL INDEX

A key sequential index is a primary storage structure which stores the tuples in an ordered heap and builds an auxiliary index structure for fast access on join attribute value. Both ISAM (Indexed Sequential Access Method) [IBMa] and B-tree indices [Baye70a, Come79a, Held78a] are key sequential. The index is a tree structure, where each node contains attribute and address pairs. The attribute value is the maximum (or minimum) value stored on the page pointed to by the associated address. The leaf nodes of the tree point to the data pages. In order to retrieve all tuples with a given join attribute value, the index is traversed by reading the root, performing a binary search to locate the address of the next page and continuing down the tree until the

data page is located and searched.

## SECONDARY INDEX

Secondary indices are hashed. A structure analogous to a primary hash structure is built containing join attribute values and tuple addresses. Since the relation is stored in another primary structure both the page address and location within the page are known. There is one index entry for every tuple of the relation. Tuples with a given attribute value are located by searching the index page to determine the tuple address then, for each address, reading the data page and directly accessing the tuple.

### 2.1.4. Storage Modification Algorithms

In this dissertation, we assume that query processing proceeds in two stages: structure modification followed by join processing. Three basic operations are required to modify any storage structure to any other: sort a relation, hash a relation, and build a key sequential index. For example, to create a key sequential index structure from a hash structure, the relation must be sorted and the key sequential index built. The remaining modifications are straightforward. This section discusses various implementations of these functions. We consider two sorting algorithms, one software and one hardware. Since fast hashing may be critical to the performance of several of the join algorithms, we include a simple hash algorithm (slow hash) and a more complex, but somewhat faster algorithm (fast hash).

## SORT

Binary merge sort [Knut73a] is used to sort relations. The individual data pages are first sorted, then merged pairwise in increasing length runs until the entire relation is sorted.

## LINEAR SORT

Sorting with one processor requires  $O(n \log n)$  page operations, where  $n$  is the number of pages in the relation. Many proposals for parallel sorting devices have appeared in the literature. The fastest of these devices accept  $p$  unordered numbers and produce a sorted output after  $O(\log^2 p)$  comparator delays using  $O(p \log^2 p)$  comparators [Batc68a]. The amount of data that may be sorted is limited by the amount of hardware in the device. These devices are modeled by a linear sort algorithm which requires time proportional to the number of pages the relation is stored on. The linear sort algorithm reads the data pages, sorts them, then writes the result back to disk.

## SLOW HASH

The slow hash algorithm reads the relation then iterates through the tuples on each data page writing them one by one to the proper hash bucket. Secondary index creation uses the same algorithm.

## FAST HASH

The fast hash algorithm attempts to minimize the number of random disk accesses by using sequential access whenever possible. The algorithm has three phases. During the first phase, the data is read, the bucket address calculated for each tuple join attribute and the augmented relation written back to disk. During the second phase the relation is sorted by bucket address and during the third phase the relation is written to the hash buckets (in hash bucket order). Secondary indices are created by hashing using either algorithm in an analogous manner.

## BUILD KEY SEQUENTIAL INDEX

A key sequential index is built from the bottom up by reading each data page, in join attribute order, to find the minimum (maximum) attribute value stored on the page and adding the attribute value and page address to the index page under

construction. When all of the data pages have been read, the process is repeated for the first level of index pages and so on until a single index page, the index root, is created.

### **2.1.5. Join Processing Algorithms**

The join algorithm is similar to other computationally intensive tasks in that the choice of a good algorithm is critical to high performance. The problem is further complicated by the wide range of parameter values that may occur in practice, making the choice of any single algorithm which is near optimal over most of the expected situations a difficult if not impossible task. Three of the most promising algorithms are considered here.

#### **TUPLE SUBSTITUTE JOIN**

The tuple substitute join algorithm distinguishes the two relations as the inner and outer relations. For each tuple of the outer relation, all tuples with matching join attribute value are retrieved from the inner relation. The cost to retrieve these tuples will depend on the storage structure of the inner relation, as described above. As matches are encountered, they are written to the result relation.

#### **MERGE JOIN**

The merge join algorithm first sorts the two relations, if they are not already ordered, then performs the join by iterating over both relations in parallel looking for matching join attribute values. Matches are written as they are encountered.

#### **HASH JOIN**

The hash join algorithm first hashes the two relations, if they are not already hashed. Next, individual hash buckets are sorted. Since buckets are a single page, the sort merge phase is not required. Finally, the tuples in corresponding buckets are com-

pared and matches are written as they are encountered.

## 2.2. Model Description

This section presents the query workload and cost estimation models. Individual queries are described by a collection of parameters. A set of queries (workload) is described by a set of parameter values and relative frequencies. Performance is estimated for a workload by calculating the average minimum cost per query. Query costs are calculated by estimating the cost of each execution strategy to determine which is minimum. The costs themselves are calculated by estimating the number of basic operations required, based on knowledge of the behavior of the different storage structures and algorithms.

### 2.2.1. Parameters and Performance Metrics

Eight parameters, listed in Table 1, are used to describe queries. The selectivity is defined as the fraction of the smaller relation which appears in the join result. The selectivity is assumed known. The selectivity parameter describes the join operation. The remaining parameters describe the two relations. Indices are storage structures used for fast access by join key value, as described above. The index key fanout is the number of page addresses appearing in one index page. Since the only relevant index for join processing is on the join key itself, the index key fanout must be identical for

Symbol	Parameter
p	Selectivity
K	Index Key Fanout
C1	Block Cardinality of Relation 1 (pages)
C2	Block Cardinality of Relation 2 (pages)
T1	Tuples per Page in Relation 1
T2	Tuples per Page in Relation 2
S1	Storage Structure of Relation 1
S2	Storage Structure of Relation 2

Table 1. Query Parameters

the two relations and it appears as a single parameter. The block cardinality is the number of data pages in the relation. The number of tuples per page is assumed constant for all pages in a given relation and the storage structure is one of heap, ordered heap, hash, key sequential index or secondary index, as described above.

A single join query is described by a set of values for the eight parameters described above. A *workload* is a set of queries described by a collection of parameter values. Associated with each parameter value is a *relative frequency* which describes the number of times that the parameter value is repeated in the workload relative to all other values of that parameter. The set of queries in the workload is generated by selecting the eight parameter values according to their relative frequencies (the parameters are assumed to be independent of each other). For example, if the values and relative frequencies of a workload are:

Parameter	Value	Relative Frequency
Selectivity	.05	1
Index Key Fanout	20	1
Block Cardinality	10	1
	100	2
Tuples per Page	1	1
Storage Structure	heap	1

there will be nine queries in the workload with parameter values:

p	K	C1	T1	S1	C2	T2	S2
.05	20	10	1	heap	10	1	heap
.05	20	10	1	heap	100	1	heap
.05	20	10	1	heap	100	1	heap
.05	20	100	1	heap	10	1	heap
.05	20	100	1	heap	10	1	heap
.05	20	100	1	heap	100	1	heap
.05	20	100	1	heap	100	1	heap
.05	20	100	1	heap	100	1	heap
.05	20	100	1	heap	100	1	heap

### 2.2.2. Basic Operations and Cost Functions

Six basic operations, listed in Table 2, are used to estimate data access costs. The *read* operation transfers one page of data from secondary storage to memory for further processing and the *write* operation transfers one page from memory back to secondary storage. The read and write operations are assumed to take the same amount of time and any effects due to data buffer caching or sequentiality are ignored. The *search* operation examines all of the tuples on a page to determine if any have a matching attribute value. The *sort* operation orders the tuples on a page by ascending (or descending) attribute value. Since the number of tuples on a page is typically small, the page sort operation is assumed to operate in a fixed time independent of the number of tuples on a page. The last two operations include a number of less frequently occurring operations and are distinguished by the expected amount of overhead. The *look* operation examines a small number of tuples on a page and requires a small amount of overhead. Examination of index pages, binary search of data pages and direct address access into data pages are operations classified as looks. The *scan* operation examines most of the tuples on a page and requires a large amount of overhead. Page operations classified as scans include merge of sorted data pages and iterative examination of every tuple on a page during query processing. If searching and sorting were not of particular interest, they would be considered scan operations since

Operation	Definition
Read	Read a page of data from disk
Write	Write a page of data to disk
Search	Linearly search a page for tuples with a given attribute value
Sort	Sort the tuples on one page
Look	Process a few tuples on a page
Scan	Process most of the tuples on a page

Table 2. Basic Page Operations



they examine all tuples on the page.

Given an execution strategy, the total cost to execute a query is estimated by tabulating the number of page operations and summing their associated costs. The cost functions for the various storage structures and algorithms are described below. If several alternate strategies can be used to process a given query, the cost for each is estimated and the minimum cost strategy selected. The performance of a workload is estimated by determining the minimum cost execution strategy for each query in the workload. In addition to the estimated minimum cost, the number of page operations required and the algorithm used are tabulated. Various performance metrics can be calculated directly from these statistics (e.g. the average cost per query or the average number of disk operations per query).

## STORAGE STRUCTURE COST FUNCTIONS

This section presents the cost functions to retrieve all tuples with a given join attribute value from a relation stored in each of the five storage structures.

A relation stored in a *heap* structure must be searched in its entirety to retrieve all tuples with a given join attribute value. If the number of tuples were known a priori the search could be terminated after the last tuple was found, however this information is typically not available. If the relation has block cardinality  $C$ , the search requires  $C$  page reads and  $C$  page searches.

Binary search is used to retrieve the tuples from an *ordered heap* as described above. If the relation has block cardinality  $C$ , then  $\log_2 C$  pages are read<sup>2</sup>. During the relation binary search, the first and possibly the last tuple on each page read are examined. When the final data page has been located, a binary search is required to locate any qualifying tuples. Since both of these operations examine a small number of

---

<sup>2</sup>  $\log_k$  denotes log to the base  $k$

tuples on each page, a total of  $\log_2 C$  page look operations are required.

All tuples may be retrieved from a *hash* structure by reading and searching one page. This is independent of the relation cardinality.

*Key sequential index* structures require a root-leaf traversal to locate the data pages containing the qualifying tuples. If there are  $C$  pages in the relation and the index page fanout is  $K$ , this search will require  $(\log_k C) + 1$  page reads. A binary search will be performed on each index page and the one data page will be searched for a total of  $(\log_k C)$  page looks and one page search.

Tuples are located using a *secondary index* by searching the index hash bucket to determine the data page addresses then, for each address, reading the data page and directly accessing the tuple. The overhead per tuple is two pages read, one page searched and one page directly accessed (a look operation).

## STORAGE MODIFICATION COST FUNCTIONS

This section presents the costs to execute the storage modification algorithms described above.

The *software sort* algorithm described above has two phases: first the individual pages are sorted, then the sorted pages are merged pairwise. The first phase requires that all  $C$  data pages of the relation be read, sorted and written. The merging phase requires  $\log_2 C$  pairwise merges of the entire relation, for a total of  $C \log_2 C$  page reads, writes and scans.

The *linear sort* algorithm assumes a device which can sort a relation in time proportional to its cardinality. The device is modeled by another basic page operation, *rsort*, although in this case the intuitive meaning is not as clear since pages are not operated on in isolation as they are in the other page operations. The overhead to sort a relation with  $C$  data pages is  $C$  page reads, writes and *rsorts*.

The *slow hash* algorithm writes tuples to buckets one by one. If there are  $T$  tuples per data page and  $C$  pages in the relation, slow hashing requires  $C$  page reads,  $C$  page scans and  $T * C$  page writes. Secondary index creation requires the same number of reads, writes and scans.

The *fast hash* algorithm has three phases as described above. There are  $T$  tuples per data page,  $C$  pages in the relation and  $K$  value-address pairs per index page. We assume that twice as many addresses as key plus address pairs can be placed on a page. Under this assumption, the hash bucket addresses added to each tuple during the first phase will require  $T * C / (2 * K)$  additional pages and the relation size will increase by this amount. Define  $C'$  as the number of pages in the augmented relation,  $C' = C + T * C / (2 * K)$ , and  $pmerge(C')$  as the number of page accesses to pairwise merge the  $C'$  pages, where  $pmerge(C') = C' * \log_2 C'$ . The overhead for phase one is then  $C$  reads,  $C$  scans and  $C$  writes; the overhead for phase two is  $C' + pmerge(C')$  reads,  $C' + pmerge(C')$  writes and  $C'$  sorts; and the overhead for phase three,  $C'$  reads and  $C'$  writes. This yields a total overhead of:  $C + 2 * C' + pmerge(C')$  reads,  $3 * C' + pmerge(C')$  writes,  $C + pmerge(C')$  page scans and  $C'$  page sorts. The equations can be modified in a straightforward way if a linear time sorting device is being modeled.

If a secondary index is being built using the fast hash algorithm, the equations will be identical except for the term  $C'$ , which will be replaced by a term  $I$  equal to the block cardinality of the projection consisting of the hash keys (attribute values) and addresses. This projection contains the same number of tuples as the original relation,  $C * T$ , and there are  $K$  tuples per page by definition so  $I$ , the block cardinality, equals  $T * C / K$ .

*Key sequential index creation* begins with the sorted data pages (which are the leaves of the final structure) and reads these pages to build the first level of index pages, which are then themselves read to build the next level of index. The process

continues until the root is built. This requires

$$\sum_1^{\log_K C} K^i$$

page reads and looks and

$$\sum_0^{\log_K C - 1} K^i$$

writes.

### JOIN ALGORITHM COST FUNCTIONS

This section presents the costs to execute the three join algorithms. In addition to the join processing cost, there is a small cost associated with writing the result relation which is the same for all algorithms and equal to the product of the selectivity and the cardinality of the smaller relation page writes.

The *tuple substitute join* algorithm designates one relation as the inner relation and the other as the outer. Assume the outer relation has block cardinality  $C1$  and tuples per page  $T1$ . During join processing, every tuple in the outer relation is examined once at a cost of  $C1$  reads and scans. The cost to access the inner relation depends on the storage structure. For all primary structures, the cost is equal to the product of the number of tuples in the outer relation,  $C1 * T1$ , and the cost to retrieve all tuples with a given attribute value from the inner relation. Note that this cost is independent of whether or not any qualifying tuples exist since the lack of a qualifying tuple will not become known until the data pages have been examined.

In the case of a secondary index, the data pages are only accessed if in fact they contain tuples with the given join key value. For each tuple in the outer relation, the secondary hash index is read and searched and for each tuple in the result, the corresponding data page is read and directly accessed. The number of data pages accessed depends on both the selectivity of the join and the number of duplicate

attribute values. The model assumes a worse case of no duplicate values, hence the number of data pages accessed is equal to the product of the selectivity and the cardinality of the smaller relation.

The *merge join algorithm* first sorts the two relations, if they are not already ordered, then performs the join by iterating over the two relations in parallel looking for matching join attribute values. If the two relations have block cardinalities  $C_1$  and  $C_2$ , the merge requires  $C_1 + C_2$  page reads and scans.

The *hash join algorithm* first hashes each relation, if it is not already hashed, then sorts the hash buckets and compares the tuples in corresponding buckets. The sort and comparison phase requires  $C_1 + C_2$  page sorts,  $2 * (C_1 + C_2)$  page reads,  $C_1 + C_2$  page writes and  $C_1 + C_2$  page scans.

### 2.3. Experiments

Four experiments were performed to evaluate various architectural alternatives for database applications. The first experiment varied the algorithms used for query processing. The second and third experiments varied the architecture by modifying the cost of the page sort and page search operation respectively. This modification could reflect either a faster software implementation or a hardware implemented "assist." The fourth experiment evaluated the performance of a database machine architecture designed around a linear time sorting device and the merge join algorithm.

#### 2.3.1. Workload Parameters and Output Statistics

The workload parameters for a "standard" query workload and an "p-standard" workload are given in Table 3. The parameters for the standard workload were chosen as typical of applications using small relations. All parameter values are assumed equally likely to occur. In many applications, however, the queries to be run against the data base are known a priori. This permits the physical storage structures to be

chosen so that a fast access path will always exist for a query. The p-standard workload models this situation by excluding the heap data structure from all queries in the workload.

In order to assign values to the basic page operations, the cost of one disk read or write was assumed equal to one. The costs of the other basic operations were assigned relative to this cost. Table 4 lists the page operation costs used in the experiments. The page scan cost was assumed equal to the disk read/write cost and the page look cost to one-tenth the disk read/write cost. Page search was assumed to equal the disk cost and page sort to equal three times the disk cost. All numeric costs given below will be in terms of a disk read/write cost equal to one.

### 2.3.2. Algorithm Choice

This section describes the results of a series of experiments to evaluate different software architectures. A typical architecture includes the fast hash algorithm and

Parameter	Values	Relative Frequency
Selectivity	5.25	1.1
Keys per Index Page	20,100	1.1
Block Cardinality	10,1000	1.1
Tuples per Page	1.20	1.1
Storage Structure		
Standard	heap, oheap, hash, key seq, sindex	1.1.1.1.1
P-standard	heap, oheap, hash, key seq, sindex	0.1.1.1.1

Table 3. Standard and P-standard Workloads

Page Operation	Cost
Read	1
Write	1
Search	1
Sort	3
Scan	1
Look	.1

Table 4. Typical Cost Values

both the tuple substitute and merge join algorithms. The average query cost for a standard workload is 5,559<sup>3</sup> and for an p-standard workload 3,577. This cost is the average minimum cost over all queries in the workload. Since all possible execution plans are considered for each query and the minimum cost one chosen, these average costs are the best possible.

The first alternative considered is to use the simpler hash algorithm. This modification results in an increase of the average cost for a standard workload to 5,625 and for the p-standard workload to 3,583, as listed in Table 5. This indicates that the hash function (used to create both hash primary structures and hash secondary indexes) is not critical and the time spent to implement a complex algorithm would not be well spent.

The remaining alternatives considered various combinations of join algorithms. The results are listed in Table 6. An architecture which only included the merge join algorithm would clearly not perform well compared with any of the other alternatives, although the addition of merge join to an architecture which already included tuple substitute join would provide a significant cost reduction. This is a result of the

Alternative	Standard	P-standard
Fast Hash	5,559	3,577
Simple Hash	5,625	3,583

Table 5. Simple Hash vs. Complex Hash

Alternative	Standard	P-standard
Tuple Substitute Only	7,248	5,401
Merge Join Only	24,170	20,653
Merge and Tuple Substitute Joins	5,559	3,577
Tuple Substitute, Merge Join and Hash Join	5,494	3,474

Table 6. Join Alternatives

<sup>3</sup> The units of all costs are disk access equivalents. The absolute time is equal to the product of the cost and the disk access time.

merge join requiring that both of the relations be sorted. Since the workload is evenly distributed among heap, ordered heap, hashed, key sequential and secondary index structures, the merge join algorithm will be effectively operating on heaps three fifths of the time, whereas the tuple substitute algorithm can take advantage of unordered storage structures. This effect is enhanced as the relation cardinality is increased, since sorting is an  $O(n \log n)$  function of the block cardinality, although the cost of performing a merge join on two large *ordered* relations tends to be less than that of tuple substitute join.

Addition of the hash join algorithm would appear to have little effect on overall performance, despite the existence of queries in the workload for which it is the optimal choice. This is an indication that there are few queries in the workload for which the hash join algorithm is optimal and furthermore for those for which it is, the difference between the cost of the hash join strategy and the next best strategy is small.

To summarize, the particular hashing algorithm implemented is probably irrelevant. Of the various combinations of join algorithm, a system consisting of tuple substitute join and merge join would probably give the best performance for the least implementation effort.

### 2.3.3. Fast Sort

Page Sort Cost	Standard	P-standard
0	5.426	3.477
5	5.683	3.628
10	5.934	3.711
15	6.146	3.791
20	6.356	3.871

Table 7. Average Query Cost as a Function of Sort Cost



There have been many proposals in the literature for fast sorters designed to operate on a (small) fixed number of elements. This experiment evaluates such devices for inclusion in a database management system by running a model containing tuple substitute join and merge join over a range of values for the page sort cost. The resulting average query cost for the standard and p-standard workloads are given in Table 7. A page sort cost of zero corresponds to the limiting value of the performance metric as sort cost is decreased. As the page sort cost increases from zero to 20, only a 11-17% increase in average query execution cost is observed. This indicates that the page sort function is not critical for the system and workload under consideration: that is, if the cost is low it is not heavily used and if the cost is high alternate strategies which do not require sorting will be used. This can be more clearly seen from the additional statistics for the standard workload listed in Table 8. The number of sorts (#Sorts) is the number of page sort operations required by the workload. The number of merge joins (#Mjoin) is the number of optimal strategies in the workload which use the merge join algorithm. The final three columns list the number of storage transformations: building a secondary hash index (#Build Sindx) and building an ordered heap (#Build Oheap). At each incremental increase of sort cost, the number of page sorts decreases. This decrease results from using the tuple substitute algorithm instead of the merge join algorithm or from avoiding building either ordered heaps or secondary index structures. To conclude, a fast page sort function would probably not lead to a significant improvement in performance. This result was verified under various additional workloads.

#### 2.3.4. Fast Search

Another popular function for hardware implementation is the page search function. For example, the "database accelerator" in the Britton Lee IDM-500 is built around a fast search engine [Epst80a]. This function was evaluated in an analogous

Sort Cost	#Sorts	#Mjoin	#Build Sindex	#Build Oheap
0	106,884	250	198	20
5	90,404	206	198	20
10	68,044	206	182	--
15	67,484	206	166	--
20	67,008	192	164	--

Table 8. Other Statistics as a Function of Sort Cost

Page Search Cost	Standard	P-standard
0	4,416	2,821
.5	5,026	3,207
1	5,559	3,577
1.5	5,909	3,805
2	6,188	4,001

Table 9. Average Query Cost as a Function of Search Cost

manner, with results summarized in Table 9. In this case, as the search cost is increased from zero to two, a 40-42% increase in average query cost is observed. Similar results were obtained for a variety of different workloads. This indicates that the page search operation is a critical function which would be a good candidate for faster implementation. Additional statistics are listed in Table 10, where the number of searches (#Searches) is the number of page search operations required by the workload, the number of tuple substitutions (#Tsubs) is the number of times the tuple substitution algorithm was selected and the number of merge joins (#Mjoins) the number of times the merge join algorithm was selected. These additional statistics also indicate that the page search operation is critical: if the search cost is low, a large number of searches will be performed; if the search cost is high, alternate strategies which do not require searching will be used, however the absolute number of page searches in the optimal strategies is still sufficiently great that their cost is significant.

### 2.3.5. Linear Relation Sort and Merge Join

Several recent database machine proposals describe architectures built around parallel hardware sorting devices [Dohi82a, Acce85a]. This experiment estimates the

Search Cost	#Searches	#Tsubs	#Mjoins
0	1,976,960	1410	190
.5	1,933,160	1394	206
1	1,205,880	1358	242
1.5	1,087,800	1348	252
2	835,800	1336	264

Table 10. Additional statistics for Standard Workload.

performance of a similar architecture which uses a sorting device as an assist to a more traditional database management system utilizing the merge join algorithm exclusively. The standard workload described above contains a large number of small relations. In order to test this architecture under the most favorable conditions, an additional "large relation" workload was run. The parameters for this workload are listed in Table 11. Since merge join is the assumed processing strategy, all unordered relations must be sorted, including ones that have hashed and secondary index structures. Hence none of these structures accelerate processing and as a result we modified the standard workload to include only heaps and ordered heaps in equal quantities.

Table 12 contains the experimental results. The numbers in parenthesis on the first line of the table are the baseline costs for this workload on a traditional architecture, as described above. The remainder of the table indicates workload cost for various values of Rsort. When Rsort is equal to one, the cpu cost to sort the relation is equal to the disk access cost to read it from disk and simulates a very high speed sorter. Other rows in Table 11 estimate the performance of slower sorters as well as the limiting case of zero sort cost.

Parameter	Values	Relative Frequency
Selectivity	5.25	1.1
Keys per Index Page	50,100	1.1
Block Cardinality	500,1000	1.1
Tuples per Page	10,15	1.1
Storage Structures	heap,oheap	1.1

Table 11. "Large Relation" Workload Parameters

	Standard	Large
Baseline Cost	5,559	19,990
Rsort per Data Page		
20	14,178	21,093
15	11,653	17,343
10	9,128	13,593
5	6,603	13,843
3	5,593	11,343
1	4,583	9843
0	4,078	6093

Table 12. Average cost as a function of rsort cost.

These results indicate that an architecture composed of linear time relation sorting and the merge join algorithm will only outperform a traditional architecture if Rsort is less than three for the standard benchmark or less than twenty for the large benchmark. This is a consequence of using both the tuple substitute and the merge join algorithm in the traditional architecture rather than the merge join algorithm only. There are many cases when tuple substitution outperforms merge join regardless of the cost to sort a relation. For example, if the tuple substitute algorithm is being used with a small outer relation and a large inner relation, it will tend to examine a small fraction of the inner relation and be less costly than merge join processing. To conclude, in order to provide a net speed up in average execution cost on this architecture, the linear time sorting element must either be capable of handling very large relations (i.e. roughly 1000 or more disk pages in length) without performance degradation, or of operating at disk access speed (i.e. rsort roughly equals one).

## 2.4. Summary

This chapter has described uniprocessor join optimization and presented a model of query workload resource requirements. This model allows design architectures to be compared in an implementation independent manner. It provides a vehicle for testing new or alternative algorithms for inclusion in a database management system and for evaluating the effect of various performance enhancements. By varying the

workload descriptions, the queries can be made as specific to a given application as desired. This allows both well defined and poorly defined applications to be described in the same manner and used for performance evaluation.

This chapter also presented the results of several experiments evaluating various architectures. Conclusions are that the choice of a good query processing algorithm is of more importance than the use of "assists" to speed up execution. Given that efficient processing algorithms have been chosen, the page search operation is shown to be a good candidate for enhancement but the page sort operation is not. It is unlikely that an architecture built solely around hardware assisted sorting of relations in linear time and the merge join algorithm would outperform a more traditional database management system which has multiple software supported processing tactics. These results are, of course, somewhat workload dependent. Similar results to those presented in the text were obtained for other workloads.

## CHAPTER 3

### MULTIPROCESSOR JOIN MODEL

A variety of join algorithms for multiprocessors are presented and analyzed in this chapter. These algorithms allow for parallelism both between and within queries (i.e. both inter- and intra-query parallelism). The analysis is based on an extension of the cost functions and model presented in the previous chapter. Some specific questions addressed are: Over which sets of queries do each of the algorithms offer optimal performance? Are algorithms based on minimizing data transmissions ever optimal in a local network environment where data transmissions are inexpensive compared with disk accesses? How effective is increasing the number of processors in increasing performance?

#### 3.1. Multiprocessor Join Processing

A multiprocessor consists of a collection of processors, associated secondary storage and a communication medium. All processors are equivalent in processing capability. The communication medium is referred to generically as a network, although it could be implemented in many ways (e.g. by an ethernet network, shared disks, shared memory, etc.). We assume that the communication medium allows one page of data to be sent between any pair of processors at a fixed cost that is independent of the particular processors chosen. Initially, we assume that no more than one page of data may be transmitted simultaneously and that no more than two processors may communicate simultaneously (i.e. there is no broadcast facility). Secondary storage devices are assumed to be associated with individual processors and memory is not shared. This implies that all data to be accessed by more than one processor must be duplicated or explicitly transmitted over the communication medium.

Within this framework a number of different kinds of specialized hardware are possible. Among the alternatives considered are high speed implementations of basic operations and variations of the structure of the communication medium (e.g. adding a ring or broadcast transmission capability). In a processing environment with large relations and buffered paging of data from disk, shared memory is of limited utility beyond providing a fast communication pathway between processors. Random access to data in shared memory requires synchronized execution and coordinated paging of data from secondary storage and is not considered further here.

As in the case of uniprocessor query optimization, efficient utilization of storage structures is critical to high performance. Data on individual sites is stored in one of the five structures described in the previous chapter: heap, ordered heap, hash, key sequential index or secondary index. When data is transmitted over the network it may lose its structure, depending on how files are implemented, addressed and transmitted. If a structure is built using *physical* page addresses and pages within a file are *physically contiguous*, transmission which retains contiguity will allow ordered relations to be transmitted intact. All other structures which rely on the physical addresses will effectively become heaps, however the addresses may be patched in a straightforward manner. If physical contiguity is not maintained, all structures will effectively become heaps. If *logical* page addresses are used, all of the structures can be transmitted intact. The model presented in this thesis assumes that files have sequential logical addresses, as in the UNIX operating system [Ritca], and can be transmitted intact along with any auxiliary indices.

Several of the multiprocessor algorithms calculate a single join using a sequence of simpler operations, including projections and joins over intermediate results. In general, the result of any join operation has a heap structure (i.e. is unstructured), however in certain special cases it may retain some structure. Both the result of a

merge join and the result of a tuple substitute join with an ordered outer relation are ordered.

When joins are executed on a uniprocessor, the individual steps in a strategy are executed sequentially. Multiprocessors allow several modes of parallel execution. Intra-query parallelism (parallelism within a single query) results from *pipelining* the individual steps in a strategy through a sequence of processors, executing the steps in *parallel* on a set of processors, or *overlapping* data transmission with on-site processing. In all cases, the nodes need only support *single-threaded* execution, that is queries are executed serially in their entirety and only one query is being processed at any one time. This is the mode of execution supported by many of the database machines which implement specific algorithms in hardware. *Multi-threaded* execution allows multiple queries to be in progress at the same time. The simplest way to achieve inter-query parallelism is to partition the nodes among distinct queries. The nodes need not be multi-threaded. More complex modes of inter-query parallelism require that the nodes support multi-threaded execution with interleaved requests from distinct queries.

### 3.1.1. Performance Considerations

One of the motivations behind multiprocessor architectures is the claim that a collection of slow, inexpensive processors can utilize parallelism to attain the same performance as a single fast, expensive processor at a lower overall price. Multiprocessor architectures can be roughly broken into three categories [Patt85b]: throughput oriented, availability oriented and response oriented. Throughput oriented multiprocessors are general purpose systems which attempt to increase performance by running multiple independent jobs in parallel on a balanced system. Performance is increased by decreasing the amount of resources required by each job. Availability oriented multiprocessors provide fail-safe or fail-proof operation and attempt to maximize the



number of independent tasks done in parallel. The goal is to provide reliable execution rather than high performance and we do not consider this category further in this thesis. Response oriented multiprocessors tend to be specific to a particular application and attempt to maximize the number of cooperating processes done in parallel. Performance is increased by decreasing the response time of individual queries by splitting them into subqueries which are executed in parallel on multiple processors.

Performance is typically measured in terms of either throughput or response time. The throughput is the number of queries per unit time that are executed by the system. In an open system, this is the same as the rate at which queries leave the system. In a closed system, this is the rate at which queries "start over" and begin another cycle of processing. The response time is the amount of time an individual query spends in an open system or the time for one complete cycle in a closed system. The response time has two components: the time spent receiving service and the time spent waiting at servers for other queries to complete their service. [Wolf82a, Heym82a]

Our model minimizes either total time (by considering all processing costs) or response time (by considering the amount of processing on the critical path from query initiation to query termination). All estimates are for an unloaded system and we do not consider device contention as a processing cost. A typical model will calculate the costs for several thousand distinct queries. We considered several queueing models for the multiprocessor organizations, however the problem of assigning branching probabilities on the basis of the workload characteristics led to models that were sufficiently complex that analytic solution was not feasible. We also concluded that a simulation model with more than a thousand distinct query classes would be unlikely to converge in a reasonable amount of time. The metrics we use are averages over all queries in a workload, as described below.

### 3.1.2. Distributed Join Algorithms

This section describes six distributed join algorithms. The first three algorithms are throughput oriented and attempt to minimize the total amount of resources required. The distributed join algorithm uses the most obvious technique of transmitting the two relations to a single site and performing a uniprocessor join. The next two algorithms, semi-join and bloom-join, attempt to decrease the number of data transmissions generated during query processing by transmitting encoded information about the join attribute rather than the entire relation. The semi-join algorithm has been extensively explored in the distributed database management literature (see [Ceri84a] for a partial bibliography) and the bloom-join algorithm is our extension to a popular database machine algorithm [Vald82a]. The remaining three algorithms (fragment & replicate, fragment & rotate and distributed hash join) appear in the database machine literature. They are response oriented and attempt to minimize the response time by dividing the work as evenly as possible among the processors.

We explain each algorithm in detail and describe the actual sequence of steps that are required to execute the join example of the previous chapter. All six algorithms are designed to operate on distributed data. We assume that each algorithm begins with an identical problem: both relations on distinct sites in their entirety and the join result required on a third site.

#### 3.1.2.1. D-Join Algorithm

The d-join algorithm proceeds in the most obvious way by either transmitting one relation to the site of the other to perform the join or transmitting both relations to the result site and performing the join there. In more concrete terms:

- (1) Select one or both relations for transmission.

- (2) If one relation is selected, transmit it to the site of the other, otherwise transmit both relations to the final site.
- (3) Perform the join.
- (4) If the join is not on the final site, transmit it there.

Referring to the example presented in section 2.1, assume that the PILOTS relation is stored on site A, the PLANES relation on site B and that the result of JOIN PILOTS ON LICENSE WITH PLANES ON TYPE is required on site C. The algorithm could proceed by the following steps:

- (1) Select the PILOTS relation on site A for transmission.
- (2) Transmit PILOTS to site B.
- (3) JOIN PILOTS ON LICENSE WITH PLANES ON TYPE on site B.
- (4) Transmit the join result to site C.

The total processing includes the time to transmit one or both relations, the uniprocessor join processing time and possibly the time to transmit the join results. For simplicity, we assume that data which is transmitted cannot be accessed until the entire transmission is complete and that join processing must be complete before the result may be accessed. Under these assumptions, there is no parallelism possible and the minimum response time<sup>1</sup> is equal to the total processing time.

Since storage structures are preserved during transmission, the query optimization problem is almost identical to that of the previous chapter: the number of strategies will increase by a factor of three when transmissions are taken into account and the cost to execute every query will increase, however the join processing itself will not be modified. Optimization will select an identical uniprocessor join processing strategy

---

<sup>1</sup>The minimum response time is the single-threaded response time, i.e. the response time when there is no contention for resources with other concurrently running queries.

for each of the three transmission options, then select the transmission option with the minimum cost, i.e. the one that requires transmission of the minimum number of pages.

### 3.1.2.2. Semi-join Algorithm

The semi-join algorithm was first proposed for the SDD-1 distributed database management system [Good79a, Bern81a]. The base hardware for this system is a collection of processors linked by the ARPA network [Hear82a]. The ARPA network is a "long-haul network" designed to connect geographically remote machines. The protocol is involved and imposes a significant overhead to transmit data from one site to another. In this environment, assuming that the only significant source of overhead is data transmission may be reasonable, although it has recently been questioned [Seli80a]. Many authors however, e.g. [Vald82a], propose using this algorithm in a local network environment where messages are inexpensive compared with cpu processing and I/O costs. Later in this chapter we will examine the performance of semi-joins in this environment more carefully.

Semi-join reduction uses a semi-join to remove tuples from a relation which do not appear in the join result. If two relations are being joined, either one or both relations may be semi-join reduced. If one relation is being reduced, the algorithm proceeds by the following steps:

- (1) Select one of the two relations for *semi-join reduction*
- (2) Project the join attribute from the *other* relation and transmit this projection to the site of the relation to be semi-join reduced.
- (3) Join the projection with the relation to be reduced (the *semi-join* ) and transmit the result to the final site. Note that the semi-join result is guaranteed to contain only tuples which appear at least once in the result to

the original join query.

- (4) Transmit the (entire) relation that was projected to the final site.
- (5) Join the semi-join reduced relation with the relation that was projected to calculate the final join result.

In terms of our example above and assuming again that the PILOTS relation is on site A, the PLANES relation B and the result is required on site C, the join will be solved by reduced the PLANES relation in the following sequence of steps:

- (1) Select the PLANES relation on site B for reduction.
- (2) Project the License attribute from PILOTS, call the projection LICENSE and transmit LICENSE to site B.
- (3) JOIN LICENSE WITH PLANES ON TYPE, call the result SJR-PLANES, and transmit it to site C.
- (4) Transmit PILOTS to site C.
- (5) JOIN SJR-PLANES ON TYPE WITH PILOTS ON LICENSE.

If one relation is semi-join reduced, the total processing includes calculation of one projection, two uniprocessor joins, and transmissions of a projection, a semi-join reduced relation, and an un-reduced relation. Semi-join reduction of both relations (i.e., both of the two relations are reduced by a semi-join before transmission) is straightforward. The final join is between the two semi-join reduced relations, which are each guaranteed to contain only tuples which appear at least once in the result. Note that the final join is still required to obtain the result to the original query. If both relations are reduced, the total processing includes calculation of two projections, three joins and transmission of two projections and two semi-join reduced relations.

Although the algorithm is basically sequential, there are a number of ways parallelism can be utilized to decrease the response time. For simplicity, we assume that

on-site processing (all processing except transmissions, e.g. projection processing) must be completed before transmission can begin and, likewise, transmission must be completed before on-site processing can begin. Under these assumptions, the only way to achieve parallel execution when one relation is being reduced is to transmit the relation that is not semi-join reduced during the the time that it is being projected or during the semi-join processing itself, whichever is longer. If both relations are being reduced, the projection processing on each site can be done in parallel and the semi-joins can overlap with transmission of one projection and one semi-join result.

Traditional semi-join optimization [Good79a, Bern81a] is based on minimizing the estimated number of bytes transmitted. On site processing (e.g. projection or join calculation) is assumed to have negligible cost. Reductions in data transmission are relative to that required to transmit both relations in their entirety to the result site for join processing.

The query optimization technique we assume is exhaustive and considers all processing costs. It first chooses which relation to semi-join reduce, then produces a detailed sequence of operations to perform the projection, transmissions, semi-join and final join. The subproblems of optimizing the semi-join and final join are identical to the optimization problem presented in the previous chapter.

In order to estimate the processing costs, the cardinality of the projection and semi-join results must be estimated. The projections eliminate all attributes except the join attribute. In the limiting case of a relation which has a single attribute, the join attribute, projection will have no effect. If there are multiple attributes then the join attribute may contain duplicate values. When the non-join attributes are eliminated by the projection, all duplicate join attribute values are also removed. This will decrease the number of tuples in the projection and hence decrease its cardinality. In addition, when attributes are eliminated, tuples become smaller and the number of

tuples per page will increase. This will also tend to decrease the cardinality of the projection. The cost functions described below quantify both of these tendencies.

Join selectivity is used to estimate the cardinalities of the overall join and any semi-joins which may be used during processing. The join selectivity definition given in the previous chapter implicitly assumed that there were no duplicate attribute values. Under this assumption, the *maximum* number of tuples in the result is equal to the number of tuples in the smaller of the two relations being joined. The only way the number of tuples in the result can be any larger than this is to have duplicate values in one or both of the relations. The limiting case is when only a single join attribute value appears in both relations: the number of tuples in the result will then equal the *product* of the numbers of tuples in each relation. One of the major goals motivating semi-join techniques is avoiding transmission of join results whenever possible because of the potential for large increases in size. Clearly a model which does not allow for this expansion is inappropriate. In this chapter we extend the join selectivity definition to allow each value appearing in the join attribute to be duplicated a constant number of times.

In addition to having associated cardinalities, the intermediate result relations may be stored in fast access structures. When an attribute is projected from a relation, duplicate values are removed. If the relation is initially ordered, the duplicate values may be removed as the projection is being generated. If the relation is not ordered, the subrelation of attribute values must be sorted to locate and remove duplicates. In either case, the projection itself is ordered. When the semi-join is performed, the result will be ordered if: the merge join algorithm is used; the projection is the outer relation of a tuple substitute join; or the projection is the inner relation of a tuple substitute join and the outer relation is ordered. In all other cases (i.e. when the projection is the inner relation of a tuple substitute join and the outer relation is stored in a

heap, hash or secondary index structure), the result will be a heap.

### 3.1.2.3. Bloom-Join Algorithm

The bloom-join algorithm is a new algorithm which is similar to the semi-join algorithm except in the way join attribute data is encoded for transmission. Rather than transmitting a projection of attribute values, a *bloom filter* is created and transmitted. When a relation is hashed, the hash function provides a map between attribute values and hash bucket numbers. A hash vector associates one bit with every hash bucket number. If at least one tuple is present in the bucket, the bit is set to one. The size of a hash vector tends to be smaller than that of an attribute projection, however less information is transmitted since many attribute values may hash to the same bucket. *Bloom-filters* [Knut73a] provide a means of encoding more information into the same bit-vector. The technique assumes  $k$  independent hash functions. For each attribute value,  $k$  bits are set in the vector corresponding to the results of the  $k$  hash functions. If there are  $N$  records in the file (each of which has a unique key) and  $M$  hash buckets, the probability that the filter will incorrectly indicate that a value is present will be approximately  $\left(1 - e^{-\frac{kN}{M}}\right)^k$ . If  $k = 1$ , the bloom filter is a standard hash vector. Although all of the references cited below assume  $k = 1$ , we will refer to this algorithm as the "bloom-join algorithm" to prevent confusion with the distributed hash join algorithm described below.

The notion of a bloom-join appeared early in the database machine literature [Schu78a, Bray79a, Babb79a], however it has only recently been proposed and evaluated as a distributed processing technique [Vald82a]. In both environments, bloom filters will encode the most information when the hash functions partition attribute values across as many buckets as possible.



As with semi-joins, one or both relations may be reduced. If one relation is being reduced, the algorithm proceeds by the following steps:

- (1) Select one of the two relations for *bloom-join reduction*
- (2) Create a *bloom filter* for the join attribute of the *other* relation. The bloom filter contains one bit for every hash bucket as described above. Each tuple is hashed on its join attribute value and the bit(s) corresponding to its bucket address is set to one. If the number of hash functions is  $k$ ,  $k$  bits will be set for each join attribute value.
- (3) Transmit the bloom filter to the site of the relation to be reduced and calculate the reduction by hashing each tuple and checking if the corresponding bit(s) in the bloom filter are set. Note that the reduction may contain tuples that are not part of the final join result.
- (4) Transmit the bloom-join reduced relation to the final site.
- (5) Transmit the relation that was not bloom-join reduced to the final site and join with the reduced relation to produce the final result.

Referring back to our example, assume again that the PILOTS relation is stored on site A, the PLANES relation on site B and the result is required on site C.

- (1) Select the PLANES relation on site B for bloom-join reduction.
- (2) Create a bloom filter on the License attribute of the PILOTS relation and call the vector H-LICENSE.
- (3) Transmit H-LICENSE to site B and calculate the bloom-join reduction of PLANES and call it HJR-PLANES. Note that HJR-PLANES will be a subset of PLANES, however it may contain tuples that are not part of the final join result.

- (4) Transmit HJR-PLANES to site C.
- (5) Transmit PILOTS to site C and JOIN PILOTS ON LICENSE WITH HJR-PLANES ON TYPE.

The total processing for a single reduction includes: calculation of the bloom filter, the bloom-join reduction, and the final join; and transmission of the bloom filter, the bloom-join reduced relation and the entire relation that is not reduced. If both relations are reduced, the total processing will include calculation of both bloom filters, both bloom-join reductions and the final join; and transmission of the two bloom filters and the two bloom-join reduced relations. Under the assumptions made above for the semi-join, the only way to achieve parallel execution for a single bloom-join is to transmit the relation that is not reduced either during calculation of the bloom filter or during calculation of the reduction, whichever is longer. If both relations are reduced, parallelism analogous to that described above for semi-join reduction can be achieved.

The only step which allows more than one processing strategy is the last step where the final join is calculated. This optimization problem is identical to the join optimization problem described in the previous chapter, however the possible storage structures are constrained by the earlier processing steps. The relation which is not reduced will have whatever structure it initially had. The relation which is reduced will be ordered if the original relation was ordered, otherwise it will have a heap structure. The result relation will depend on the final join strategy.

#### 3.1.2.4. Fragment & Replicate Algorithm

The fragment and replicate algorithm was proposed for the Distributed Ingres Database Management System [Epst78a] and Muffin database machine [Ston79a]. The algorithm minimizes both response time and communications traffic costs and solutions can be obtained for both site-to-site and broadcast network models. It was designed to

process data which may be partitioned among sites either randomly or by using *distribution criteria* to establish a unique site for each data element based on the values of one or more attributes. Clearly there are an unlimited number of ways that data could be fragmented among sites. We assume that initially the data is not fragmented, the two relations are stored on distinct sites and the result is required on a third site, as in the examples above.

The algorithm proceeds by the following steps:

- (1) Select one of the two relations for fragmentation among  $N$  sites.
- (2) Perform the fragmentation by dividing the relation into  $N$  equal pieces and transmitting each piece to a distinct site.
- (3) Replicate the other relation on all sites by transmitting it to each site<sup>2</sup>.
- (4) Perform a join on each site in parallel then transmit all results to the final site.

In terms of the PILOTS/PLANES example:

- (1) Select the PLANES relation on site B for fragmentation among  $N$  sites.
- (2) Divide PLANES into  $N$  equal pieces and transmit each piece to a distinct site, call the pieces PLANES- $I$ ,  $I=1..N$ .
- (3) Transmit a copy of the entire PILOTS relation to each of the  $N$  sites.
- (4) JOIN PILOTS ON LICENSE WITH PLANES- $I$  ON TYPE,  $I=1..N$  and transmit the join results to site C.

---

<sup>2</sup> If the data is suitably partitioned by value, only part of the data needs to be replicated at each site. In a broadcast environment, the replicated relation is only transmitted once during replication. If each site requires a distinct subset of the "replicated" relation, multiple transmissions are required. In a non-broadcast environment, replication requires one transmission for every site. In this case, replicating a portion of the data on each site will reduce the amount of data transmitted, however the partitioning costs cannot be neglected. The distributed hash join algorithm, described below, presents one way of partitioning the data by value and for that algorithm the partitioning costs dominate.

The total processing includes fragmenting the PLANES relation, replicating the PILOTS relation, performing  $N$  joins and transmitting the results to the final site. Since the joins are performed in parallel, the minimum response time is equal to the sum of the time to process a single join and the transmission times of the fragments, the replicated relations, and the results. Clearly the transmission time for replication will tend to dominate unless a broadcast facility is available.

Query optimization must first decide which relation to fragment and then whether storage structures should be created for either the fragmented or replicated relations. The structures may be created in parallel after transmission<sup>3</sup>. Other than the relative ordering of transmissions and storage structure creation, the join optimization problem at each site is identical to the uniprocessor problem and all sites will have the same optimal strategy. The result structure will likewise depend on the particular strategy chosen.

### 3.1.2.5. Fragment & Rotate Algorithm

The fragment and rotate algorithm was proposed recently in conjunction with a hardware realization that is particularly well suited to LSI and VLSI implementation [Meno83a]. A model based on queueing analysis is used to determine the design constraints between processing speed and memory size. If the cost is fixed, the analysis yields optimal chip designs, where the design decisions include the number of processors to place on a chip and the sizes of the associated memories. The analysis presented here maps the algorithm onto the general multiprocessor model described above for purposes of comparison.

---

<sup>3</sup> A further optimization would be to create the structure for the replicated relation prior to transmission. This will not affect the response time, but will reduce the total time. We leave this as a future extension.

The algorithm proceeds by the following steps:

- (1) Select one of the two relations for fragmentation among  $N$  sites.
- (2) Perform the fragmentation by dividing the relation into equal pieces and transmitting each piece to a distinct site.
- (3) Pipe individual pages of tuples <sup>4</sup> from the other relation to each site in turn (i.e. *rotate* around to each site), performing a retrieval on the join attribute values at every site visited.
- (4) When all pages of tuples have visited all sites, transmit the result tuples to the final site.

Continuing with our PILOTS/PLANES example:

- (1) Select the PLANES relation on site B for fragmentation.
- (2) Divide PLANES into  $N$  equal pieces and transmit each piece to a distinct site. call the pieces PLANES- $I$ ,  $I=1,N$ .
- (3) Pipe pages of tuples from the PILOTS relation to sites 1- $N$ ; at each site  $I$ , retrieve all tuples with matching join attribute values from PLANES- $I$ .
- (4) Transmit the results at sites 1 to  $I$  to site C.

The total processing includes one retrieval against every fragment for each tuple that is rotated from site to site and the transmissions of fragments, rotating pages and results. The rotating pages effectively form a pipeline: if there are  $N$  processors, the first  $N$  pages prime the pipe by sequentially activating each processor in sequence; execution continues on all processors in parallel until the first processor finishes execution of the last page; the pipe then empties over the next  $N$  execution periods as each of the

---

<sup>4</sup> The original algorithm piped individual *tuples* from site to site on a dedicated local network. The overhead for this mode of transmission in our more general framework would be excessive, hence we have modified the algorithm slightly to provide better performance.

processors becomes inactive. If there are  $C$  pages in the relation that is being rotated and each page requires processing at each of the  $N$  processors, there will be a total of  $C * N$  processing steps. While the pipe is full,  $N$  steps will be processed in parallel. During the  $N$  periods over which the pipe is filling,  $N^2 / 2$  pages will be processed and likewise for the  $N$  periods over which the pipe is emptying. The overall execution time will thus be equal to  $(C * N - N^2)/N + 2 * N = C + N$ . The minimum response time is the sum of the time to transmit the fragments, the time to process  $C + N$  pages by executing a retrieve for every tuple on each page, and the time to transmit the results.

The only decision remaining for query optimization is to decide which relation to fragment. Since each tuple will generate a retrieve request to each fragment, fast access to fragments is critical. Since the hash structure provides the fastest access, it is the only structure considered. The JOH proposal [Meno83a] hashes the fragments and uses an associative memory to verify that at least one matching tuple exists before accessing the data itself. The result tuples are unstructured.

### 3.1.2.6. Distributed Hash Join Algorithm

The distributed hash join is another example of a technique to partition the processing as evenly as possible among a collection of processors. The Grace database machine [Kits83a] uses this algorithm exclusively for join processing. The two relations are first hashed on the join attribute using identical hash functions into two sets of hash buckets. Since hashing partitions by value, all tuples with matching join attribute values will be in corresponding buckets. The buckets are then distributed among processors so that corresponding buckets are located on the same processor. Individual hash buckets are then sorted and corresponding buckets merged to produce the query result. This step of the processing is done in parallel. Specialized hardware is used to speed bucket sorting. Hash vectors are associated with each relation and are used to restrict the number of buckets processed by eliminating pairs of corresponding buckets

with at least one empty member before distributing them to processors.

The DBC/1012 also stores data in a distributed hash structure (i.e. the data is hashed and the buckets partitioned among the sites) [84a, Nech84a], however these papers indicate that joins are processed by coalescing the two relations using a distributed sort followed by the (uniprocessor) merge-join algorithm.

The distributed hash join algorithm proceeds by the following steps:

- (1) Perform a distributed hash of each relation by hashing each relation then partitioning the buckets among the sites and transmitting each bucket to its assigned site.
- (2) Sort individual buckets in parallel on all sites.
- (3) Merge individual buckets in parallel on all sites.
- (4) Transmit the results on each site to the final site.

In terms of the PILOTS/PLANES example

- (1) Hash PILOTS and PLANES, partition the buckets evenly among the N sites and transmit each bucket to its proper destination.
- (2) Sort the PILOTS and PLANES hash buckets in parallel.
- (3) Merge the sorted PILOTS and PLANES hash buckets in parallel.
- (4) Transmit the results on sites 1 to N to site C.

The total processing includes hashing both relations, sorting and merging corresponding buckets and transmitting the hash buckets and results. Under the assumptions above, the results of the first hash may be transmitted while the second hash is being processed. Once the data has been completely distributed, individual buckets may be sorted and merged in parallel. The minimum response time is thus equal to the sum of: the time to hash the first relation; the maximum of the times to hash the second relation and transmit the first relation; the time to transmit the second

relation; the time for one site to sort and merge its allocation of buckets; and the time to transmit the results to the final site.

Query optimization requires at least two strategies from which to select an optimal plan. This algorithm fixes the processing steps into a single strategy and hence no optimization can be performed. The only relevant storage structure for the two relations is hash and the result relation will be a heap.

### 3.2. Multiprocessor Join Model Description

In this section we describe our model of multiprocessor joins. This model is an extension of the uniprocessor join model presented in Chapter 2 and includes that model as a special case. It incorporates all six algorithms described in the previous section, although any subset may be used in a particular experiment. The most significant difference between multiprocessor and uniprocessor joins is the addition of multiple processing sites and the possibility of parallel execution. In this environment, the total execution time (across all processors) may differ greatly from the overall duration of the query, or response time. We begin this section with a discussion of distributed performance metrics. These metrics are used as optimization criteria (e.g. the minimum response time strategy or the minimum total time strategy can be selected by the optimization strategy) and as metrics describing the performance of the workload (both the average response time per query and the average total time per query are tabulated, regardless of the optimization criteria). The section continues with a detailed examination of join selectivity in the presence of duplicate attribute values. We derive a more precise definition which allows more accurate estimates of distributed query resource requirements. An additional basic operation is added to model data transmissions and the eight query parameters defined in the previous chapter are augmented by three additional parameters representing the number of processors and the number of unique values appearing in each join attribute. The section concludes



with cost functions for the distributed join algorithms.

### 3.2.1. Multiprocessor Performance Metrics

The minimum response time of a query in a uniprocessor environment is equal to the total processing time. In a multiprocessor environment, intra-query parallelism may be used to decrease the minimum response time by overlapping data transmission with on-site processing or by executing subqueries simultaneously on multiple processors.

The potential for overlap is limited by *dependencies* between various steps in the processing, that is, by requirements that processing on an individual step of a strategy be completed before processing can begin on the next sequential step. For example, an intermediate result cannot be transmitted before the subquery which generates it has begun. Although in some special cases transmission of the result can begin before the subquery has completed, for simplicity we assume that this is never done. Under this assumption, the distributed join algorithm cannot support any intra-query parallelism and the semi-join and bloom-join algorithms allow for limited overlap between on-site processing and data transmission.

The remaining three algorithms, fragment & replicate, fragment & rotate and hash join, allow for parallel execution of identical subqueries. In the first two algorithms, the degree of parallelism is limited by the cardinality of the relation being fragmented. In the third algorithm, the degree of parallelism is limited by the minimum number of values in either of the two relations (i.e. by the maximum number of hash buckets). The query parameters defined in the previous chapter are augmented by an additional parameter, the number of processors in the multiprocessor,  $N$ , to allow evaluation of the response time of these three algorithms. Table 1 contains the complete list of distributed parameters ( $V1$  and  $V2$  are used for selectivity estimation, as described in the following section).

Symbol	Parameter
p	Selectivity
K	Index Key Fanout
C1	Block Cardinality of Relation 1 (pages)
C2	Block Cardinality of Relation 2 (pages)
T1	Tuples per page in Relation 1
T2	Tuples per page in Relation 2
V1	Number of Unique Join Attribute Values in Relation 1
V2	Number of Unique Join Attribute Values in Relation 2
S1	Storage Structure of Relation 1
S2	Storage Structure of Relation 2
N	Number of Processors

Table 1. Distributed Query Parameters

If the system is multi-threaded, the response time of individual queries will tend to increase with increased throughput as contention for resources among different queries increases, however the *minimum* response time will remain unchanged. The minimum response time is estimated by determining the number of basic operations on the critical path from query initiation to termination. In the absence of bottlenecks within the multiprocessor, the overall *throughput* will depend on the total resource requirements of individual queries. If the query resource requirements increase, the throughput will decrease. In building our model, we have included cost functions both for total time and minimum response time. Optimization can either minimize the total time, roughly corresponding to maximum *multi-threaded* throughput, or minimize response time, corresponding to maximum *single-threaded* throughput.

### 3.2.2. Selectivity Definition and Estimation

The selectivity of a relational operation is the fraction of data remaining after the operation is performed. Although the selectivities of select and project operations are well defined, see for example [Seli79a], a plethora of definitions exist for join and semi-join selectivity: the fraction of the product of the cardinalities of the two relations [Jark84a], the fraction of the domain of the join attribute [Yu84a], the fraction of

the sum of the cardinalities of the two relations [Pram85a], the fraction of the larger relation [Seli80a], the fraction of either relation [Aper83a] and so on. In this paper we present an in depth analysis of the join operation and derive a new definition of join selectivity which allows accurate estimates of intermediate results generated during join processing in both a uniprocessor and a distributed environment. The definition is consistent: the cardinality of Relation A join Relation B is the same as the cardinality of Relation B join Relation A. The effects of duplicate join attribute values are explicitly considered and semi-join selectivity is treated as a special case of join selectivity.

The join selectivity is not a critical parameter for estimating the resource requirements of a uniprocessor join: except for a small difference in the number of secondary index accesses and the cost to write the result relation to disk, the amount of processing is independent of the cardinality of the result relation. Estimating the resource requirements of a distributed join, on the other hand, requires accurate selectivity estimates not only for the join itself, but for the intermediate results generated during query processing (e.g. semi-join result cardinalities). In this section we extend the selectivity definition used in Chapter 2 to more accurately estimate the cardinalities of various intermediate results by explicitly considering the number of distinct values appearing in the join attributes of each relation. Using our definition of selectivity, the estimated result cardinality does not depend on the order in which the operations are performed. Some models, e.g. [Yu84a], have the property that the estimated result cardinality depends on the order in which the operations are performed. Clearly, for any specific query and any correct sequence of operations to calculate the result, the result cardinality *cannot* vary. In addition, using our definition of selectivity (presented below) one or both relations may realize a net decrease in cardinality after reduction. Simpler models, e.g. [Sacc82a], based on more restrictive assumptions have the mathematical property that only the larger of the two relations may be reduced by a semi-join.

The definition of join selectivity used in Chapter 2 was based on an implicit assumption that there were no duplicate values in either of the two join attributes<sup>5</sup>. If this is the case, the maximum number of tuples in the result is equal to the number of tuples in the smaller relation (i.e. the number of values in the smaller relation) and defining the selectivity as the fraction of tuples from the smaller relation appearing in the result leads to consistent estimates for all of the quantities needed to estimate uniprocessor join resource requirements.

In the more general case, each attribute value may be duplicated some number of times. If a duplicated attribute appears in the join result, each of the corresponding tuples appears once with *every* matching tuple from the other relation. For example, if there are 4 tuples with identical attribute values from one relation and 6 tuples with the same (matching) attribute value from the other relation, there will be 24 tuples in the result with that attribute value. In order to model this more typical scenario, we add two new query parameters to represent the number of distinct join attribute values appearing in each relation,  $V_1$  and  $V_2$ , and clearly distinguish the number of *tuples* in a relation from the number of *values* in an attribute. In general,  $V_1$  and  $V_2$  will be chosen from the same domain of values. For example, the TYPE attribute of the PLANES relation and the LICENSE attribute of the PILOTS relation have values which come from the same domain, in this case, integers corresponding to airplane types. For the join to be semantically meaningful, the join attributes must come from the same domain. Our *definition* of selectivity does not make this assumption, although the statistical model used for *estimating* the selectivity will require it.

In order to estimate the number of tuples in the join result, we assume that all values in a relation are duplicated a constant and equal number of times. This implies

---

<sup>5</sup> Note that unless the number of duplicates is very large, the amount of resources required during uniprocessor join processing is independent of the number of times each attribute value is duplicated.

that the number of times each value is duplicated in relation 1,  $D1$ , is equal to:

$$\frac{C1 * T1}{V1}$$

where  $C1.T1$  and  $V1$  are the cardinality, number of tuples per page and number of unique values, respectively, and  $C1*T1$  is the number of tuples in the relation. The number of duplicates in relation 2 is defined in an analogous manner. Since the result tuples are composed of one tuple from each relation, the result can contain no more distinct values than are present in either of the two relations initially. This is true independent of the number of duplicate values. In terms of our query parameters, the maximum number of matching *values* is equal to the minimum of  $V1$  and  $V2$ , call it  $VMIN$ , and the selectivity,  $p$ , is defined as the fraction of the minimum number of *values* appearing in the result <sup>6</sup>. The number of *values* in the result is thus equal to  $p * VMIN$ . Since each value is duplicated  $D1$  times in relation 1 and  $D2$  times in relation 2, the number of *tuples* in the result is equal to:

$$p * VMIN * D1 * D2$$

Since each tuple in the result is composed of one tuple from each relation, the number of tuples per page will decrease to<sup>7</sup>:

$$\frac{T1 * T2}{T1 + T2}$$

and the number of *pages* in the result,  $R$ , will be equal to:

---

<sup>6</sup>Note that no assumption is made about the domains of  $V1$  and  $V2$ . If  $V1$  and  $V2$  are selected from disjoint domains,  $p$  will be zero for all queries. If  $V1$  and  $V2$  are selected from the same domain, for every particular selection of values,  $p$  will have a value between 0 and 1 inclusive.

<sup>7</sup>Each tuple from relation 1 is  $1/T1$  pages long, each tuple from relation 2,  $1/T2$  pages long and the result tuples are  $1/T1 + 1/T2$  pages long, hence there are  $1/(1/T1 + 1/T2)$  tuples per page in the result.

$$p * VMIN * \left\lfloor \frac{C1 * T1}{V1} \right\rfloor * \left\lfloor \frac{C2 * T2}{V2} \right\rfloor * \left\lfloor \frac{T1 + T2}{T1 * T2} \right\rfloor$$

Which can be simplified to

$$p * VMIN * \left\lfloor \frac{T1 + T2}{V1 * V2} \right\rfloor * C1 * C2$$

Semi-join reduction removes all tuples from a relation that do not appear in the final result at least once. The number of tuples in a semi-join reduced relation is thus equal to the number of tuples from that relation that appear in the final result. Since the number of values in the result is equal to  $p * VMIN$ , the number of values from relation 1, say, that appear in the result is also equal to  $p * VMIN$ ; the number of tuples from relation 1 is equal to  $p * VMIN * D1$ ; and the number of pages,  $R1$ , is equal to

$$\frac{p * VMIN * \left\lfloor \frac{C1 * T1}{V1} \right\rfloor}{T1} = \frac{p * VMIN * C1}{V1}$$

An analogous equation can be derived for  $R2$ , the number of pages from relation 2 that appear in the result.

During semi-join processing, the join attribute is projected out from one of the two relations being joined, say relation 1. Since projections are relations they contain no duplicates and the number of tuples in the result is equal to the number of values in the result:  $V1$ . Each tuple in the result will contain a single join attribute value. Assuming twice as many attribute values as attribute value-address pairs can be placed on a page, the number of tuples per page will increase to  $2 * K$  and the number of pages in the projection will be equal to

$$\frac{V1}{2 * K}$$

The above discussion assumed that the selectivity parameter was known. This is rarely the case in practice and the selectivity itself must be estimated. Let the domain of the join attribute contain  $M$  values. The relations contain  $V1$  and  $V2$  values randomly selected from  $M$  without replacement and the join result contains those values appearing in both relations. The number of values in the join result has expected value

$$E(\text{No.values in result}) = \frac{V1 * V2}{M}$$

In the discussion above we noted that the number of values in the result is equal to  $p * VMIN$ , hence

$$E(p) = \frac{E(\text{No.values in result})}{VMIN} = \frac{V1 * V2}{M * VMIN}$$

This statistical model agrees with that presented in [Yu84a] although their definition of the term "selectivity" is somewhat different.

Note also that since this formula calculates the *expected value* the result has a variance associated with it. Whenever the selectivity is replaced by its expected value in a calculation, the results will also be expected values and will themselves have associated variances. The effect of using expected values becomes apparent when costs are estimated for a specific query using the expected cost formulas and compared with the actual (measured) costs: unless the variance is very small it is unlikely that the two results will agree. If a large number of queries are compared in a like manner, however, the *average* measured cost should be very close to the expected cost. In the limit, as the number of queries becomes infinite, the two should agree exactly [Hoe171a].

Query optimization compares the costs of various execution plans. If expected costs are used, as they are in many optimization strategies, e.g. [Ceri84a, Yu84a], the

expected costs are ranked and the minimum one chosen. Unfortunately, the fact that one expected value is less than another expected value does not necessarily imply that the actual values for a specific query are likewise ranked. Query optimization techniques based on expected cost functions will *tend* to choose good execution strategies in the long run, however they may choose incorrectly for many specific queries. In order to avoid this problem, many optimization strategies assume a *worst case* selectivity that is equal to one [Seli80a, Bern81a]. Our cost functions include the selectivity as an explicit parameter and calculate the exact cost (within the modeling assumptions) rather than the expected cost. This allows us to set the selectivity equal to either its expected value or its worst case value, or any other value that may be appropriate to a particular query.

### 3.2.3. Cost Functions

Appendix A contains a complete list of the cost functions in terms of the parameters defined above. An additional basic operation, page transmission cost, is added to the six basic operations defined in Chapter 2. Table 2 contains the complete list of basic operations.

Operation	Definition
Read	Read a page of data from disk
Write	Write a page of data to disk
Search	Linearly search a page for tuples with a given attribute value
Sort	Sort the tuples on one page
Look	Process a few tuples on a page
Scan	Process most of the tuples on a page
Xmit	Transmit a page of data

Table 2. Basic Page Operations



### 3.3. Experimental Results

In this section we compare the performance of the six multiprocessor algorithms both analytically and using a simulation model. The simulation model is an extension of the model presented in the previous chapter and includes that model as a special case. The first three algorithms were designed to minimize the total time by minimizing transmission costs. We analytically compare the expected number of page transmissions and discuss the total processing required using both example queries and simulation experiments. The last three algorithms were designed to minimize response time and rely on special hardware to provide high performance of key functions. We demonstrate that the expected amount of data transmitted is almost identical, however the amount of processing varies greatly for the three algorithms. We use the simulation model to evaluate the effectiveness of the special hardware and to estimate the performance as a function of the number of processors. Although the two sets of algorithms were designed under very different assumptions, our model allows us to compare them on the basis of both total execution time and response time. We compare the performance of each algorithm executing in isolation on a variable number of processors and include statistics for the best uniprocessor solution for comparison.

#### 3.3.1. Total Time Minimization

In this subsection we compare the three algorithms that attempt to minimize the total execution time by minimizing the amount of data transmitted: distributed join, semi-join and bloom-join. We begin with an analytic comparison of the transmission costs for the three algorithms, continue with a discussion of total processing costs using an extended example and conclude with the results of a series of simulation experiments.

### 3.3.1.1. Estimating Transmission Costs

Estimates for the amount of data transmitted based on the query parameters defined above are straightforward. In this section we discuss the derivations for each of the algorithms in turn.

#### DISTRIBUTED JOIN ALGORITHM

The distributed join algorithm first determines which uniprocessor join strategy is optimal by assuming both relations are located on a single site and performing (uniprocessor) optimization. It then estimates the amount of data transmitted for each of the three transmission strategies described above and selects the one with minimum cost. Table 3 lists these costs in terms of the parameters defined above.  $R$  is the cardinality of the result relation:

$$R = p * VMIN * \left( \frac{T1 + T2}{V1 * V2} \right) * C1 * C2$$

Strategy	#Page Xmit
Transmit relation 1	$C1 + R$
Transmit relation 2	$C2 + R$
Transmit both	$C1 + C2$

Table 3. Distributed join transmissions.

#### SEMI-JOIN ALGORITHM

The semi-join algorithm *reduces* one or both relations, as described above. A reduction is *effective* if it decreases the amount of data transmitted relative to that required to transmit the entire (un-reduced) relation. This will be the case if the cardinality of the semi-join result plus the cardinality of the projection is less than the cardinality of the (un-reduced) relation. In terms of the query parameters defined above:

$$\frac{p * VMIN * C1}{V1} + \frac{V2}{2 * K} < C1$$

if semi-join reduction of relation 1 actually decreases the amount of data transmitted. The greater the difference between the quantity on the left of the angle bracket and the quantity on the right, the more the semi-join decreases network transmissions and the more effective it becomes. Clearly, as the selectivity decreases, the semi-join will become more effective.

Since the number of times each value is duplicated, D, is equal to  $C * T / V$ , this inequality can be rewritten as:

$$\frac{p * VMIN * D1}{T1} + \frac{C2 * T2}{D2 * 2 * K} < C1$$

Since the value of VMIN depends on that of V1 and V2 (and hence also on that of D1 and D2), the relationship between the effectiveness of semi-joins and the number of duplicates is not straightforward except for two special cases. First, if V1 is less than V2 and all parameters except the number of duplicates in relation 2 are fixed, then *increasing* the number of duplicates in relation 2 will increase the effectiveness of the semi-join. Intuitively this is reasonable since if the cardinality and number of tuples in a relation are fixed, increasing the number of duplicates will decrease the number of values and hence decrease the cardinality of the projection. Second, if V1 is greater than V2 and all parameters are fixed except the number of duplicates in relation 1, *decreasing* the number of duplicates in relation 1 will increase the effectiveness of the semi-join. Again, this is intuitively reasonable since, if  $V2 < V1$ , VMIN is independent of the number of duplicates in relation 1 and the number of values in the semi-join result will be fixed by the values of the other parameters. Since the cardinality of the semi-join result is equal to the product of the number of values and the number of duplicates, decreasing the number of duplicates will decrease the cardinality.

When semi-join reduction is used as a distributed query processing tactic, three possible strategies must be considered: reduce relation 1, reduce relation 2 and reduce both relations. If neither relation is reduced, the strategy is properly a distributed join strategy. Table 4 lists the amount of data transmitted for each of these three alternatives.  $R_1$  is the cardinality of the semi-join reduction of relation 1,

$$R_1 = \frac{p * VMIN * C_1}{V_1}$$

and  $R_2$  the cardinality of the semi-join reduction of relation 2:

$$R_2 = \frac{p * VMIN * C_2}{V_2}$$

$P_1$  is the cardinality of the projection of the join attribute from relation 1,  $P_1 = V_1/2*K$ , and similarly for  $P_2$ .

Strategy	#page Xmit
Reduce Relation 1	$P_2 + R_1 + C_2$
Reduce Relation 2	$P_1 + R_2 + C_1$
Reduce both	$P_2 + R_1 + P_1 + R_2$

Table 4. Semi-join transmission costs

## BLOOM-FILTER JOIN ALGORITHM

The bloom-join algorithm is almost identical to the semi-join algorithm except in the way the join attribute information is encoded for transmission. Semi-join reduction is guaranteed to eliminate *all* tuples which do not appear in the result. If a single hash function is used to generate a bloom filter (i.e.  $k = 1$ ), it is likely that the bloom-join reduced relation will contain tuples which do not appear in the result relation and will be larger than the corresponding semi-join reduced relation. However, as described in section 1.1.2.3, if the number of hash functions,  $k$ , is chosen sufficiently large, the number of superfluous tuples will approach zero and bloom-join reduction will have

the same effect as semi-join reduction. Table 5 lists the estimated transmission costs. BF1 and BF2 are the number of pages in the bloom filters for relations 1 and 2 respectively.

Strategy	# page xmit
Reduce Relation 1	$BF2 + R1 + C2$
Reduce Relation 2	$BF1 + R2 + C1$
Reduce both	$BF1 + BF2 + R1 + R2$

Table 5. Bloom-filter join transmission costs

### 3.3.1.2. Discussion of Transmission Costs

From Tables 3 and 4, it is clear that semi-join reduction may or may not decrease data transmissions. For example, if every tuple in both relations appears in the result, semi-join reduction will not decrease the cardinality of either relation and the *total* transmission time (including transmission of the projection) will be greater than that of the distributed join strategy which transmits both relations to the final site for joining. In most cases, however, the semi-join will successfully decrease the cardinality of one or both relations and the data transmission will be less than that of any of the distributed join algorithms. Semi-join reduction and bloom-join reduction have the same effect on relations, as described above. On examining Tables 4 and 5, it is clear that any difference in transmission costs is a result of differences in the cardinalities of the projections and the bloom filters themselves. Since the projection contains the actual join attribute value (either as a numeric quantity or as a byte string) it will tend to be larger than the corresponding bloom filter, which contains a minimum of one bit per value. This indicates that bloom-joins will tend to generate less data transmissions than semi-joins.

### 3.3.1.3. Estimating Total Costs

Table 6 lists the total processing required for each of the algorithms, excluding transmission costs. We call this processing "on-site." The distributed join algorithm requires the least number of on-site joins, although it tends to generate the maximum number of data transmissions as explained in the previous section. Since we assume that storage structures are preserved during transmission, on-site join processing may take advantage of any such structures. Calculating the join using a single semi-join (i.e. only one of the two relations is reduced before transmission) requires two joins and a single projection and calculating the join using two semi-joins (e.g. both relations are reduced before transmission) requires three joins and two projections. In addition, these joins involve intermediate results that have either a heap or an ordered heap structure. The bloom-join algorithm will require a single on-site join, calculation of one or two bloom-filters and either one or two "hash-joins", as described above. The on-site join will again involve intermediate results with either a heap or an ordered heap structure.

Algorithm	On-site Processing
Distributed join	1 Join
1 Semi-join	1 Projection, 2 Joins
2 Semi-joins	2 Projections, 3 Joins
1 Bloom-join	Calc 1 BF, 1 Hash-join, 1 Join
2 Bloom-joins	Calc 2 BF, 2 Hash-joins, 1 Join

Table 6. On-site Processing Required

In order to introduce our simulation model and gain a clearer intuition about the relative amount of processing required for these three algorithms we present a numeric example at this point. Table 7 contains the values of the query parameters. For the three algorithms of this section, the number of processors is irrelevant so long as it is at least three (one for each relation and one for the result) and we omit it. The amount of processing depends intimately on the storage structure of the two relations.

as will become clear after examining the numeric calculations below. We examine three of the 25 possible combinations of storage structures for the two relations: heap joined to heap, ordered heap joined to ordered heap and hash joined to hash. Rather than considering all possible (uniprocessor) join algorithms, we only consider the optimal algorithm for each of the three storage structures: create a secondary index and use tuple substitution for the heap structures, use the merge join algorithm for the ordered heaps and use tuple substitution with the hash structures<sup>8</sup>. For purposes of comparison we include the "nested loops" algorithm. This algorithm produces strategies which use the tuple substitute algorithm exclusively and do not perform any storage modifications. Each of the optimal cost calculations was verified using the simulation model. The result cardinality,  $R$ , and the cardinalities of the reduced relations,  $R_1$  and  $R_2$  are defined above. The values given in the table are calculated from the other parameters.

---

<sup>8</sup> Note that the optimal join algorithm depends on the relative values of the parameters and, although these algorithms are optimal for this particular query, the association of a single join algorithm with each storage structure is not valid in general.

Parameter	Symbol	Value
Block Cardinality of Relation 1	C1	1000
Number of Unique Join Attribute Values in Relation 1	V1	5000
Tuples per Page in Relation 1	T1	10
Block Cardinality of Relation 2	C2	10,000
Number of Unique Join Attribute Values in Relation 2	V2	50,000
Tuples per Page in Relation 2	T2	25
Index Key Fanout	K	100
Selectivity	p	0.25
Storage Structure of Relation 1	S1	heap, ordered heap, hash
Storage Structure of Relation 2	S2	heap, ordered heap, hash
Result Block Cardinality	R	1750
Block Cardinality of Relation 1 After Reduction	R1	250
Block Cardinality of Relation 2 After Reduction	R2	250

Table 7. Example query parameters

The detailed calculations for the total cost to process a distributed join using each of the algorithms on each of the three storage structures are given in Appendix B. The leftmost column describes the processing step being performed and the next two columns the number of basic operations required. The final two columns contain the numeric cost using the parameter values as named and reversed, corresponding to an exchange of relation 1 for relation 2 in the calculations. The totals are arithmetic sums, corresponding to an assumption that all basic operations cost the same amount, i.e. have the same duration. The three totals correspond to the three strategies listed in Tables 3, 4 and 5 above and are repeated in Table 8.



Heap	Nested Loops			
	D-join	2.00e8	2.00e8	2.00e8
	S-join	1.05e8	1.05e8	2.01e8
	B-join	5.03e6	1.25e8	1.30e6
Heap	Create Sindx			
	D-join	178,000	169,000	177,250
	S-join	179,000	423,125	445,275
	B-join	55,334	174,766	73,250
Oheap	Merge Join			
	D-join	26,500	35,500	34,750
	S-join	27,850	55,750	48,850
	B-join	27,766	54,766	47,782
Hash	Tsubs			
	D-join	26,500	44,500	34,750
	S-join	29,400	183,525	206,925
	B-join	38,266	39,766	73,282

Table 8. Summary of Total Processing Costs for Example Query

The processing times for the various strategies differ by several orders of magnitude, even when the same algorithm is being used on relations stored in the same storage structures. This clearly illustrates the importance of effective query optimization. If the two relations are stored as heaps and the nested loops algorithm is being used, the bloom-join algorithm performs best, with an estimated cost two orders of magnitude better than the best semi-join and distributed join strategies. There are several reasons for this variation. First, the amount of processing required for the nested loops algorithm is proportional to the *product* of the number of tuples in one relation and the number of pages in the other relation. Any strategy which decreases the cardinalities of either or both relations will realize a large decrease in total processing, hence both semi-joins and bloom-joins perform better than distributed joins. Second, although semi-joins decrease the amount of data successfully, reduction still requires a join between the projection and the relation being reduced. Bloom-filter joins eliminate this join and allow the reduction to be calculated with a single scan over the relation at a cost much less than the equivalent nested loops join. Note that data transmissions make up an insignificant part of the total processing required (i.e. less than .01%) and in this case decreased transmission activity per se is not the reason

for higher performance of the algorithms which use reduction.

If a secondary index is created, the amount of processing decreases greatly for all three algorithms. Although the bloom join is still the optimal choice, the distributed join now is less costly than the best semi-join strategy. For this particular query, the reduction in transmissions obtained with a semi-join strategy does not compensate for the increase in on-site processing.

If two ordered heaps are being joined, the variation between the three algorithms is again much less. In this case the distributed join strategy is the optimal choice. The merge join algorithm requires much less on-site processing and, for this query, the reduction in network traffic does not compensate for the increase in on-site processing for either the semi-join or the bloom-join algorithm, although the difference is small.

If the two relations are stored in hash structures, the best overall strategy is again the distributed join. In this case the best semi-join strategy is somewhat better than the best bloom-join strategy. Intuitively this is reasonable since the bloom-join algorithm always performs reduction by scanning the entire relation while the semi-join reduction will only examine one hash bucket per join key value.

#### **3.3.1.4. Simulation Results**

In this section we present the results of a series of simulation experiments. The first experiment varied the cost to transmit a page of data, the second varied the join selectivity and the third the number of duplicate values in each relation. The three algorithms were simulated individually and as a group using query optimization to calculate the minimum cost strategy.

In order to run the simulation model, the standard and p-standard workloads described in section 2.3.1 were modified slightly to reflect the additional parameters used to model distributed queries. For an individual relation, say R1, the number of

duplicates,  $D_1$ , is equal to:

$$D_1 = \frac{C_1 * T_1}{V_1}$$

(See section 3.2.2.) The workload contains a wide range of relation cardinalities. If the number of values were held constant over all relation cardinalities, for some queries the number of duplicates would be sufficiently large that the assumptions made in deriving the uniprocessor cost functions would be violated. For example, if the number of values is fixed at 100 and the number of tuples per page at 20, the number of duplicates in a relation of block cardinality 10 would be 2 ( $20*10/100$ ) and number of duplicates in a relation of block cardinality 1000 would be 200 ( $20*1000/100$ ). The number of duplicates in the latter query clearly violates the assumption that there are a small number of duplicates per attribute value. In order to define a workload with independent parameters, the two parameters representing the number of values,  $V_1$  and  $V_2$ , are replaced by two parameters representing the number of *duplicates* per value,  $D_1$  and  $D_2$ . For the standard workload, all queries have the same number of duplicates, which requires that the number of values vary as the block cardinality and number of tuples per page. Table 9 contains the complete distributed workload parameters used in this chapter. The page operation costs are the same as those given in Table 3 of Chapter 2.

Parameter	Symbol	Values	Relative Frequency
Selectivity	P	10,25	1.1
Keys per Index Page	K	10,100	1.1
Block Cardinality	C1,C2	10,1000	1.1
Tuples per Page	10,20	T1,T2	1.1
Duplicates per Value	D1,D2	3	1
Storage Structure	S1,S2		
	Standard	heap,oheap,hash key seq. indx	1.1,1.1,1
	P-standard	heap,oheap,hash. key seq. indx	0.1,1.1,1

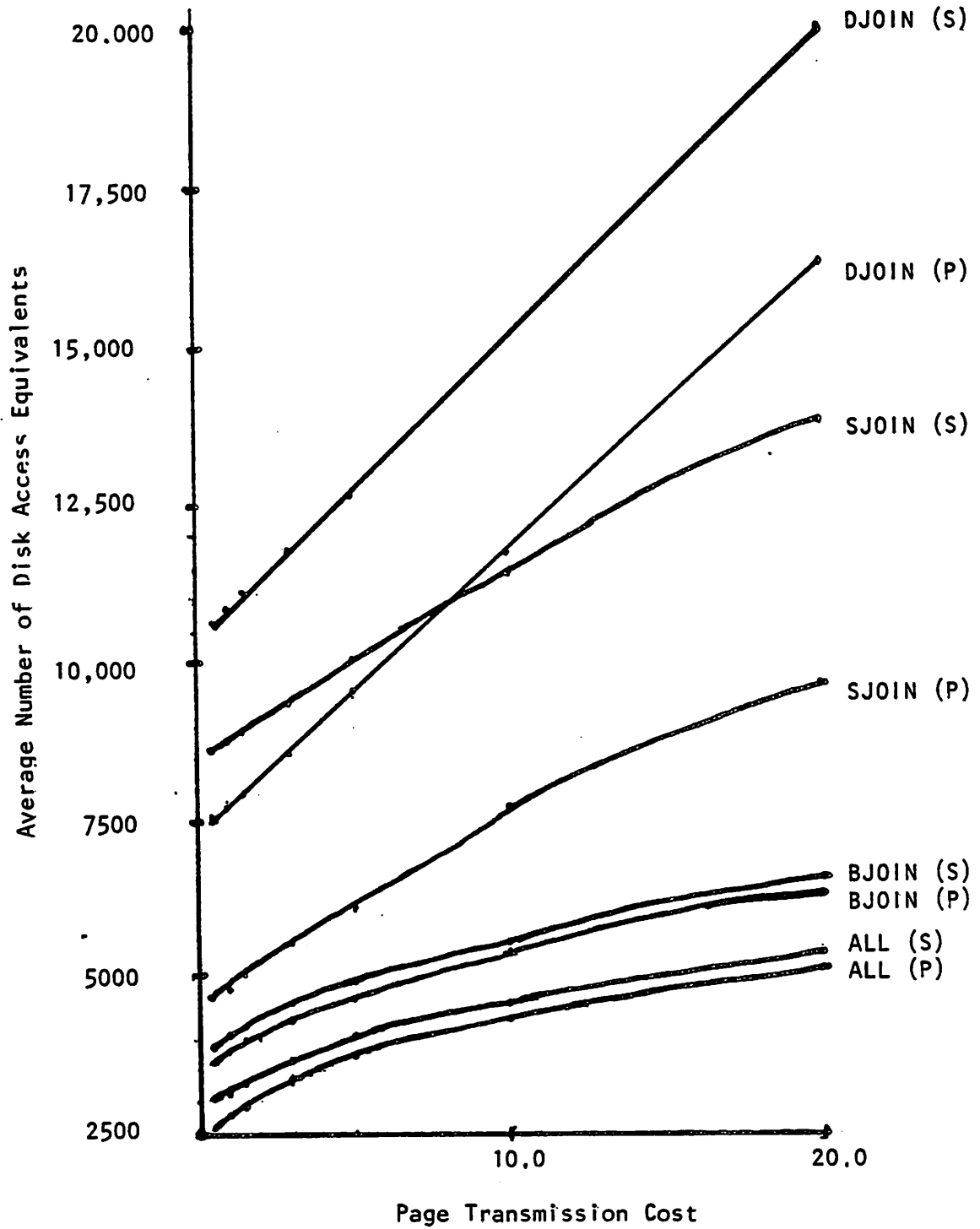
Table 9. Distributed Standard and P-standard Workloads

### 3.3.1.4.1. Network Speed

Figure 1 contains a graph of the total execution cost as a function of the network speed for each of the three algorithms simulated individually. In all cases the execution cost increases as the network speed decreases, with the effect more pronounced for the P-standard workload. The P-standard workload is identical to the standard workload except for the absence of heap structures. Overall, the amount of on-site processing for queries which do not involve heap structures tends to be less than that for queries which do involve heaps. The amount of data transmitted is the same in both cases and tends to be more significant for those queries with less on-site processing, i.e. for queries in the P-standard workload.

At all network speeds the bloom-join algorithm requires fewer resources than the semi-join algorithm, which itself requires fewer resources than the distributed join algorithm. While we demonstrated above that specific queries exist for which each of the three algorithms is optimal, if the costs for all queries in the workload are averaged, the three algorithms have a definite rank. A system built using the bloom-join exclusively will probably have higher performance than one built using semi-join or distributed join exclusively.

Figure 1. Total Cost vs. Network Speed

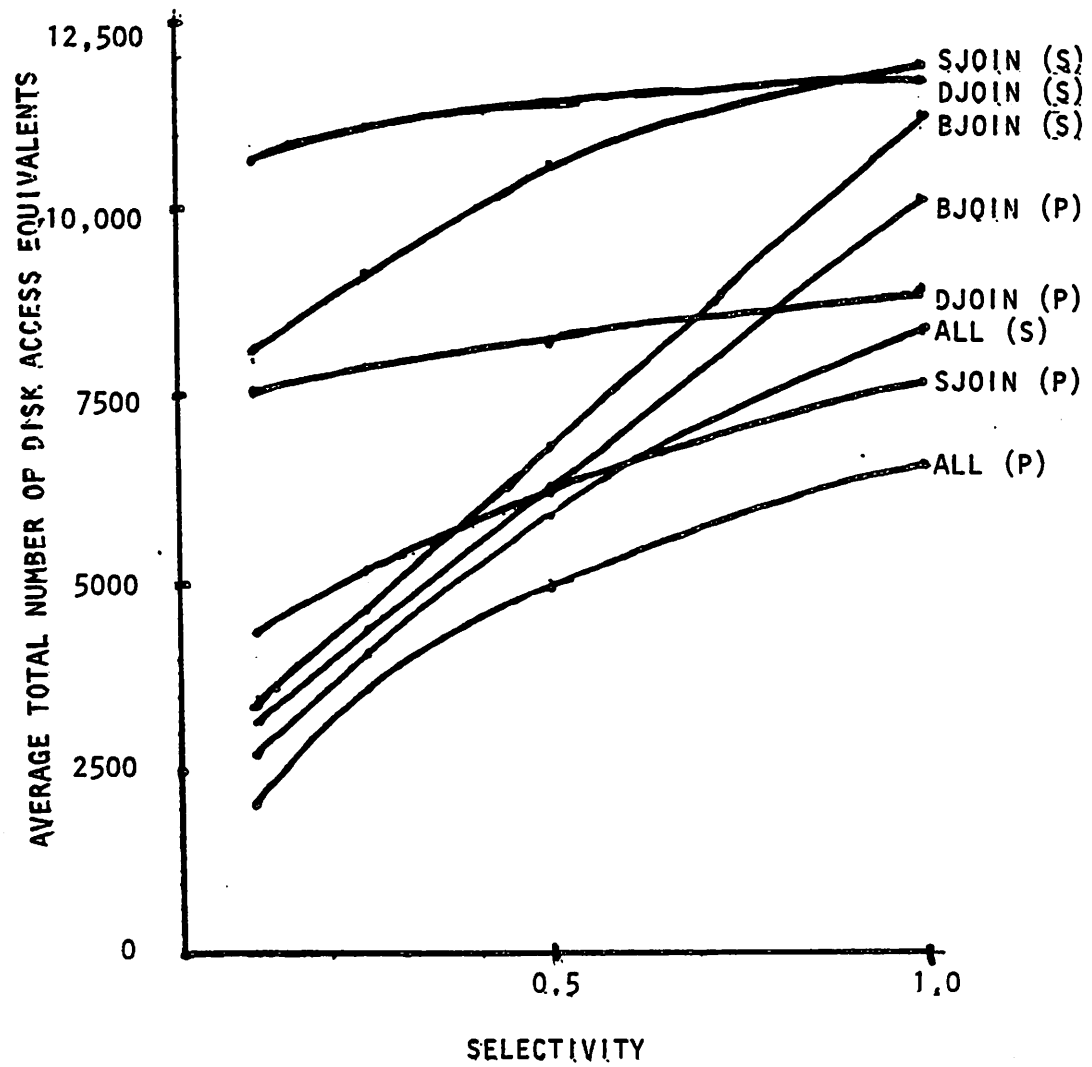


#### 3.3.1.4.2. Selectivity

Our model includes the selectivity as a distinct parameter rather than making any assumptions about its expected value and the resulting estimates of query execution cost are exact, within the assumptions made in formulating the equations. In order to evaluate the effect of selectivity on the performance of the three algorithms, we ran the simulation with the selectivity fixed at a single value for all queries in the workload. The page transmission cost was set equal to one disk read, corresponding to a local network environment. The results are graphed in Figure 2.

In all cases increasing the selectivity increased the average cost per query, with the effect on bloom-joins being by far the most significant. In fact, if the selectivity is assumed to be 1 in all cases, the semi-join strategy will tend to have estimated costs less than those of the bloom-join strategy. If the actual selectivity is near .1 for all queries, however, the semi-joins will require more than twice as much processing as the equivalent bloom-joins.

Figure 2. Total Cost vs. Selectivity



### 3.3.1.4.3. Number of Duplicates

In order to evaluate the effect of duplicates on the performance of the three algorithms, we fixed the number of duplicates at a single value for both relations and varied the remaining parameters over their workload values. The page transmission cost was again set equal to one disk access. The average total cost as a function of the number of duplicates is graphed in Figure 3.

The net effect of increasing the number of duplicates is to *decrease* the average cost for semi-join processing by 25-41% and *increase* the average cost for bloom-joins and distributed joins by 8-9% and 5-6% respectively. This indicates that the effect of reducing the size of the projections generated during semi-join reduction is more significant than that of increasing the join result cardinalities. In addition, it suggests that the common assumption that there are no duplicate values in the join attribute severely overestimates the amount of processing required for semi-join strategies.

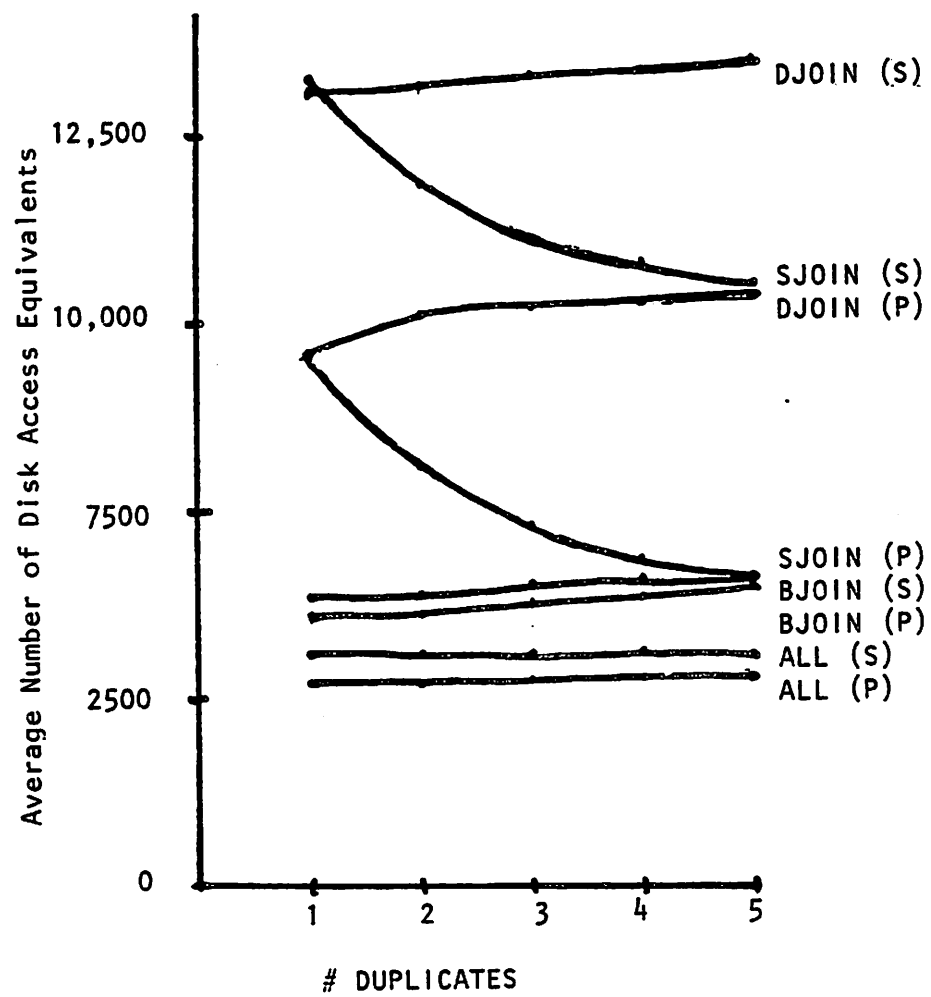
If both the selectivity and the number of times each value is duplicated are assumed to be one, the average costs for the three algorithms are:

Distributed join, Standard Workload	11,166
Semi-join, Standard Workload	15,296
Bloom-join, Standard Workload	10,332
Distributed join, P-standard Workload	8019
Semi-join, P-standard Workload	11,274
Bloom-join, P-standard Workload	9113

Under these assumptions, the distributed join algorithm appears to perform much better than it does when the actual selectivity is small or when each value is duplicated a small number of times.



Figure 3. Total Cost vs. Number of Duplicates



#### 3.3.1.4.4. Algorithm Choice

The final experiment of this section evaluated various multiprocessor software architectures. Each of the three algorithms was evaluated individually and all three algorithms were evaluated together by expanding the strategy space to include every possible strategy using each of the three algorithms and selecting the overall minimum during query optimization. We varied the network speed, number of duplicates and selectivity as above. In all cases the average optimal cost is less than the cost for any one algorithm alone, although the difference between the cost for bloom-join and that for the optimal strategy is not as great as that between bloom-join and distributed join. A system built around all three algorithms will probably perform somewhat better than one built around any one algorithm alone.

Variations in the value of the selectivity are much more significant for the optimal average query cost. This is reasonable: since the average total cost is less, small changes in the intermediate result cardinality become more significant. Note again that assuming selectivity equal to 1 in all cases will overestimate the total cost by nearly a factor of three for queries with (actual) selectivities near 0.1.

The net effect of variations in the number of duplicates is much less when optimization is used. Again, this is reasonable: since the cost of semi-join strategies tends to decrease as the number of duplicates increases and the cost of bloom-joins and distributed joins to increase, combining the three strategies should tend to damp out any effects. This is observed in the simulation results.

#### 3.3.2. Response Time Minimization

This section examines the three algorithms which attempt to increase performance by executing a portion of their total processing in parallel on  $N$  processors. Each processor executes an identical collection of operations. The minimum response time is estimated by estimating the number of basic operations on the critical path from query

initiation to query termination. The section begins with an explanation of the response time cost estimates, continues with a discussion of response time using the numeric example introduced above and concludes with the results of a series of simulation experiments.

### 3.3.2.1. Estimating Response Time Costs

The minimum response time of a query is the time between query initiation and query termination. In this section we present equations for the response time as a function of  $N$ , the number of processors. All of the equations are derived from the complete cost functions listed in Appendix A.

### FRAGMENT & REPLICATE JOIN ALGORITHM

The fragment and replicate algorithm fragments one relation and replicates the other relation on all of the sites containing at least one fragment. It performs best in an environment which supports simultaneous *broadcast* to all sites over the network. If a broadcast facility is not available, the data transmission costs increase by nearly a factor of  $N$  and become dominant, as can be seen in Table 10. (The equations in this table and in the following two tables are derived from the cost equations in Appendix A by substituting the uniprocessor join cost equations into the multiprocessor equations.) In order to generate estimates for the numbers of other basic operations, we assumed that both relations were initially heap structured, that the tuple substitute uniprocessor join algorithm would be used on every site and that a secondary index structure would be created for the replicated relation. This is the optimal strategy for these particular parameter values.

Basic Operation	Estimated Number (R.T.)
Read	$C1/N + I + pmerge(I) + C2/N + 2*C2*T2/N$
Write	$3*I + pmerge(I) + R/N$
Scan	$C1/N + pmerge(I) + C2*T2/N$
Search	$C2*T2/N$
Sort	$I$
Xmit	$C1 + N * C2 + R$ (no broadcast)
	$C1 + C2 + R$ (broadcast)
	$I = T1 * C1 / (K * N)$
	$pmerge(I) = I \log_2 I$

Table 10. Fragment &amp; Replicate Response Time

### FRAGMENT & ROTATE JOIN ALGORITHM

The fragment and rotate algorithm fragments one relation and then rotates pages of the other relation from site to site in order to calculate the join result. The algorithm assumes that a ring network is available. If this is not the case, the transmission costs will increase by a factor of  $N$ , as can be seen in Table 11. The processing strategy is fixed by the algorithm and no storage structure other than hash is considered. The JOH proposal [Meno83a] includes special purpose hardware to allow the data to be searched very rapidly using an associative memory to determine whether a data value is present and a hash structure for tuple storage. The effect of this hardware is to reduce both the number of pages read and the number of pages searched.

Basic Operation	Est. Number (R.T.)
Read	$C1/N + Q/N + pmerge(Q/N) + C2*T2 + N$
Write	$3*Q/N + pmerge(Q/N) + R/N + N$
Scan	$C1/N + pmerge(Q/N)$
Search	$C2 * T2 + N$
Sort	$Q/N$
Xmit	$C1 + N * C2 + R$ (no ring network)
	$C1 + C2 + N + R$ (ring network)
	$Q = C + B * C/(2*K)$

Table 11. Fragment &amp; Rotate Response Time

### DISTRIBUTED HASH JOIN ALGORITHM

The distributed hash join algorithm hashes both of the relations, distributes the hash buckets among the sites (i.e. fragments both relations by value), then uses a merge join algorithm to join individual pairs of hash buckets. The algorithm does not assume any particular network structure. The GRACE database machine [Kits83a] implements the page sort operation in hardware, despite the relatively small number of sort operations that are required.

Basic Operation	Est. Number (R.T.)
Read	$C1+Q1+pmerge(Q1)+2C1/N+Q2+pmerge(Q2)+2C2/N$
Write	$3*Q1+pmerge(Q1)+C1/N+3*Q2+pmerge(Q2)+C2/N+R/N$
Scan	$C1+pmerge(Q1)+C1/N+C2+pmerge(Q2)+C2/N$
Sort	$Q1+Q2+C1/N+C2/N$
Xmit	$C1 + C2 + R$

Table 12. Distributed Hash Join Response Time

#### 3.3.2.2. Discussion of Response Time Costs

The transmission times for the three algorithms are nearly identical, assuming any special network facilities are available. The response times for fragment & replicate and fragment & rotate are similar: the fragment & replicate algorithm will read and write more, the fragment & rotate algorithm will search more. Overall, the distributed hash join algorithm will read, write and scan more pages than the other two, however it will use a small number of page sorts rather than any page searches. In practice, the fragment and replicate algorithm may perform much better than is indicated here since it can select from a range of uniprocessor join algorithms and storage structures.

The numeric example presented in the previous section is again used to gain a clearer intuition about the amount of processing required by the three algorithms. We assume that both relations are initially stored as heaps and vary the number of proces-

sors,  $N$ , from 1 to 20. Appendix B contains calculations for the total processing costs, assuming again that all page operations have equal cost. The totals are summarized in Table 13. Calculations are given for both fragmentation strategies (i.e. fragment relation 1 or fragment relation 2). We assume that the fragment & replicate algorithm uses a broadcast network and the fragment & rotate algorithm a ring network. In addition, we investigate the effects of an associative store, as proposed in [Meno83a], by reducing the number of read and search operations to just those required to actually retrieve the result tuples for both the fragment & replicate and fragment & rotate algorithms.

# Processors	1	5	10	20
Frep	1,039,100 179,000	217,900 41,500	115,295 26,375	64,008 19,188
Frot	556,404 583,254	529,610 129,370	515,895 77,715	514,213 53,587
Frot w/Hardware	61,404 575,754	24,610 121,870	20,895 70,215	19,213 46,087
DHjoin	660,150	614,750	609,075	606,237

Table 13. Summary of Response Time Costs for Example Query.

In these examples, the variation in processing time is not as great as in the previous section. It should be noted, however, that the best strategy of the previous section requires the same amount of processing as the best strategy of this section using specialized hardware and 5 processors or using 10 standard processors, suggesting that effective use of storage structures is as important as effective use of specialized hardware and multiple processors. Overall, the fragment and replicate algorithm performs better than the fragment & rotate algorithm if specialized hardware is not being used and the converse is true if specialized hardware is being used. The fragment & rotate algorithm has a clearly distinguished "better" strategy: one of the two fragmentation strategies results in a dramatic decrease in processing as the number of processors is increased and the other is virtually unaffected. The explanation for this

behavior lies in the algorithm used to retrieve qualifying tuples: as the number of processors is increased, the number of fragments increases and the amount of processing also increases since each fragment must be queried once by each tuple that is being rotated.

If specialized hardware is used to decrease the number of data pages searched, the algorithm performs somewhat better than the fragment & replicate algorithm. The same strategy which performed worst without the hardware performs best with the hardware. This is because most of the processing (i.e. 80-90%) is concentrated in reading and searching pages, however most of these searches are eliminated by the specialized hardware resulting in better overall performance. With the hardware support, both of the fragment & replicate strategies can make effective use of multiple processors, as can be seen from the inverse relationship between the query cost and number of processors.

The best distributed hash join algorithm performs worse than either of the other algorithms on any number of processors. By far the bulk of the processing results from hashing and distributing the two relations. The same hashing algorithm was used to estimate costs for all three distributed join algorithms so any improvement in hashing cost is likely to be reflected equally among the algorithms. Similar results for uniprocessor joins in environments with a small number of data buffers are presented in [DeWi84a] If a single processor is used, the sort operation represents less than 2% of the total processing and the fraction decreases as the number of processors increases. If 20 processors are being used, sorting is 0.1% of the total processing. It is unlikely that hardware enhancement of the sort operation will result in any overall performance improvement.

### 3.3.2.3. Simulation Results

The simulation experiments were run on the standard and p-standard workloads described above. Note that the fragment & rotate algorithm does not distinguish between the storage structures so the performance of the standard and p-standard workloads is identical. Initially we assume that any special hardware used by the algorithms is present in the system, i.e. a broadcast network facility for the fragment & replicate algorithm and both a ring network and associative search hardware for the fragment & rotate algorithm<sup>9</sup>. The first experiment compares the algorithms on the basis of total time and response time. The second experiment evaluated the performance without the associative search hardware and the third the performance with neither the associative search hardware nor the special network facilities.

#### 3.3.2.3.1. Total Time & Response Time

Figure 4 displays the total time and response time for each of the algorithms as a function of the number of processors.

Both the fragment & replicate and the fragment & rotate algorithms exhibit a linear increase in total execution time as the number of processors increases. This results primarily from the additional network transmissions which occur in parallel on the specialized networks. The total execution cost of the distributed hash join algorithm is independent of the number of processors. As the number of processors increases, all three algorithms exhibit a decrease in the average response time, however the rate at which the response time decreases becomes less as the number of processors increases, so the difference between 9 and 10 processors is almost negligible. Note that the decrease in response time is clearly not linear.

---

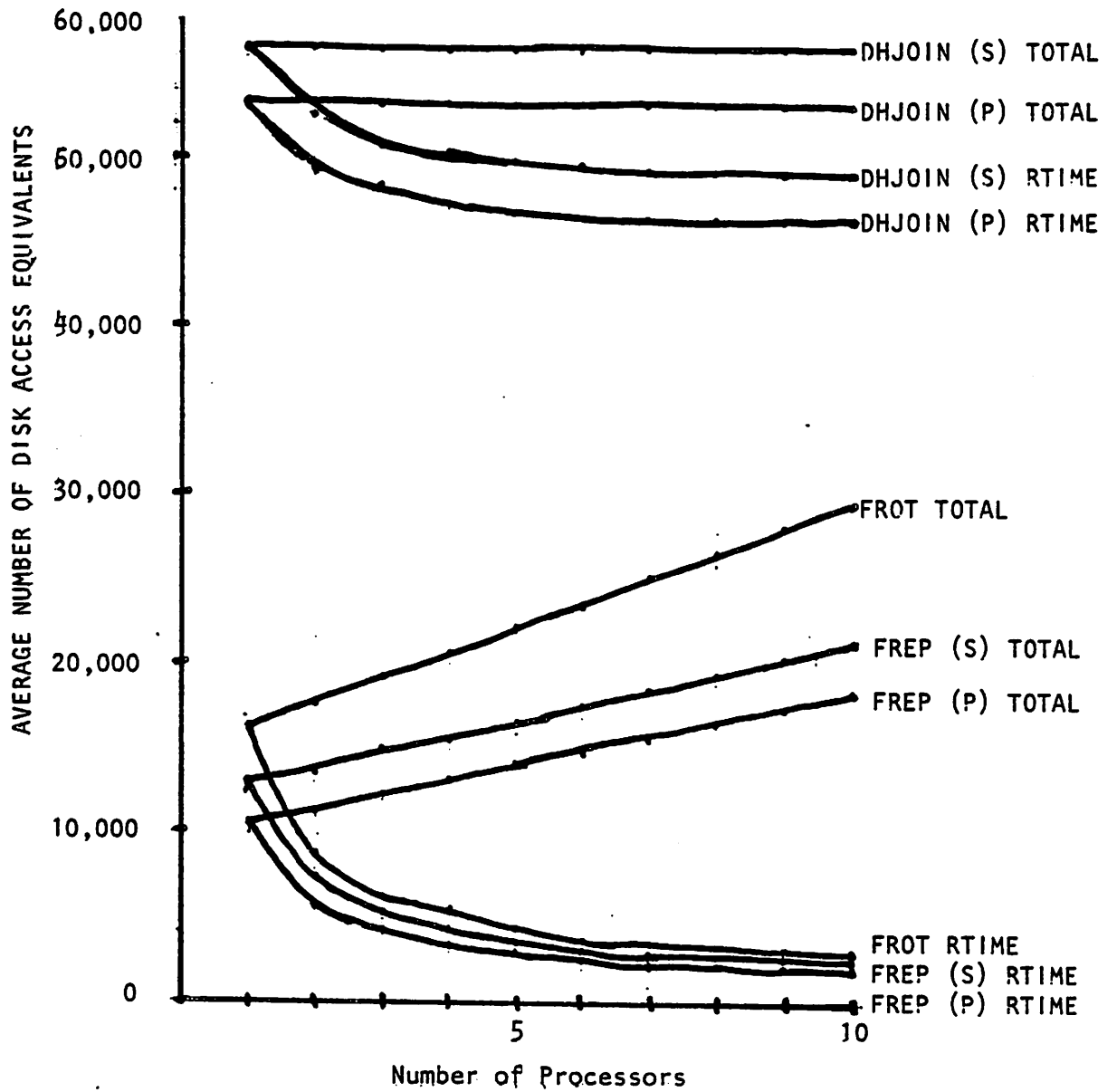
<sup>9</sup> The associative search hardware is not used for the fragment & replicate algorithm since the storage structures of the replicated relations are variable.



Ideally multiprocessors should exhibit a linear decrease in response time as the number of processors increases. Although there is no generally accepted model of multiprocessor performance, many authors have conjectured that a linear decrease is overly optimistic. A recent article [Patt85b] contains a discussion of various conjectures about the performance of multiprocessor systems as the number of processors becomes large. The consensus appears to be that the speedup should be somewhere between  $N$  and  $\log_2 N$ . Our results substantiate these conjectures.

Figure 4. Response Time and Total Time vs. Number of Processors

(Algorithms evaluated individually)

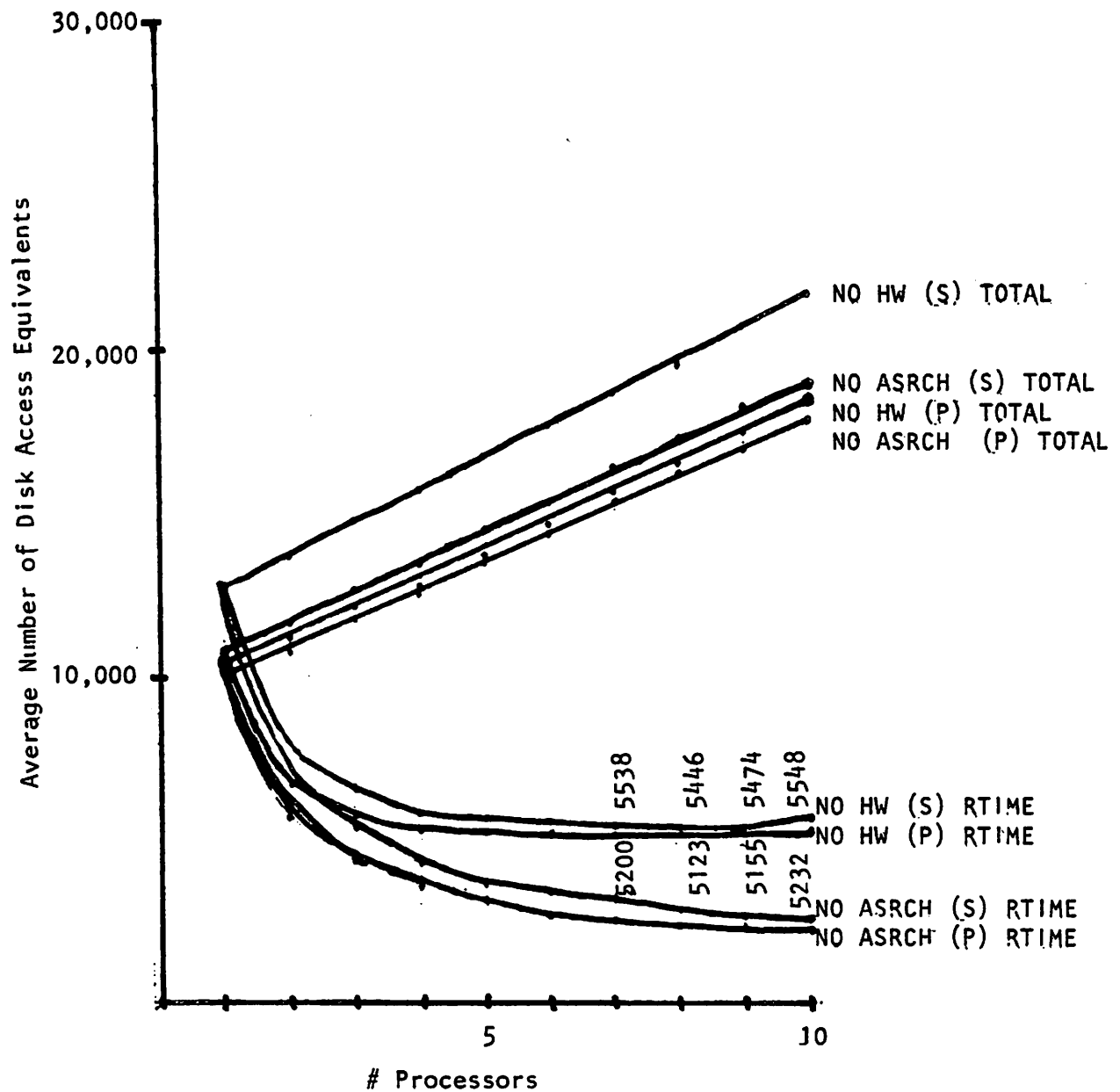


### 3.3.2.3.2. Associative Search Hardware

The associative search hardware proposed by [Meno83a] is used to reduce the number of data pages read and searched by verifying that one or more tuples with the given search key exist before accessing the data pages. If the number of search keys is large and the selectivity is small, the hardware will significantly reduce the number of data pages accessed, as can be seen from the equations in Appendix C. In order to evaluate the *overall* effect of the associative search hardware, we ran the simulation model with all three algorithms, as above, but used cost functions which assume that the fragment & rotate algorithm always accesses the data pages.

As can be seen from comparing the the traces in Figures 5 and 6, including the hardware produces a significant improvement in the optimal response time. This indicates that the addition of associative search hardware to a system which uses optimization over the three algorithms is likely to improve performance. When the trace for the optimal solution without hardware is compared to the traces for the algorithms in isolation, however, the optimal solution is almost equal to that of the fragment & replicate algorithm alone. This indicates that if the associative hardware is *not* included in the system, the additional complexity of using query optimization over the three algorithms is not warranted and a simpler system built using the fragment & replicate algorithm alone will perform as well.

Figure 5. Response Time and Total Time vs. Number of Processors  
 (Algorithms evaluated together, Associative Hardware, Network Facilities)



### 3.3.2.3.3. Broadcast and Ring Network Facilities

The effects of using a network which does not have broadcast or network facilities were evaluated by assuming that transmissions which would occur in parallel if the network facilities were present are performed serially. The results are displayed in Figure 5. The model did not include any special hardware and the average total time is equal to that of the model with all three algorithms evaluated together (with the network facilities) and any differences result from loss of parallelism. Note that when the number of processors is increased from 8 to 9, the increase in total average time to add an additional processor is *greater* than the reduction in response time resulting from more parallelism and the response time *increases*. This occurs because the increase in overhead to add one more processor is fixed and independent of the number of processors, while the decrease in response time that results from adding one more processor is dependent on the number of processors. For this workload, the decrease in response time is very small when the number of processors is increased from 8 to 9 and does not compensate for the larger fixed overhead to add the processor, hence the average response time increases. An analogous result for transaction processors was reported in [Hell85a]. Since the average total cost for the distributed hash join algorithm is independent of the number of processors, its response time cannot increase as processors are added.

### 3.3.3. Multiprocessor vs. Uniprocessor Optimization

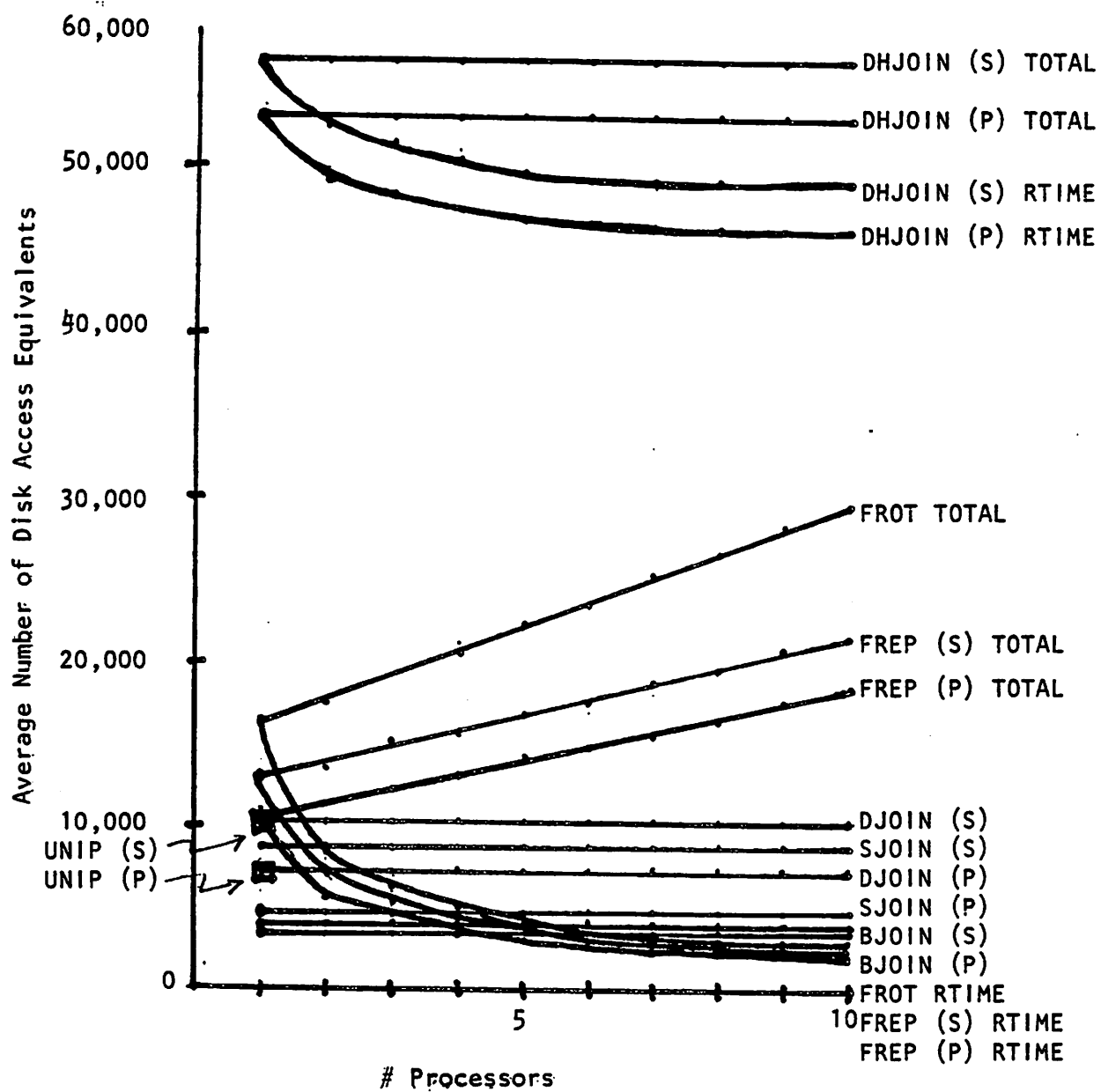
To conclude the series of experiments, we compared the algorithms which minimize total time with those that minimize response time and with the best uniprocessor solution. Although our model will generate response time estimates for the algorithms which minimize total time, many assumptions were made and the results are very close to the total time estimates in all cases and we omit them. Clearly the response time can be no greater than the total time. Although the algorithms which minimize

total time assume three processors, the equations are valid for less than three processors if it is assumed that a message may be sent and received by the same processor. If more than three processors are present, they will be ignored by the algorithms.

As can be seen from the graph in Figure 6, the two algorithms which effectively minimize response time, fragment & replicate and fragment & rotate, require more total resources than those which minimize total time and the difference widens as the number of processors increases. If there are less than 3 processors, the response time is least for the algorithms which minimize *total* time. If there are 4 processors, the fragment & replicate algorithm will have lower response time than the best total time algorithm and if there are 7 processors the fragment & rotate algorithm will likewise have lower response time. If there are more than 7 processors, the response time will be lower for the algorithms which minimize response time, although by this point the addition of more processors is no longer as effective as it was for smaller number of processors and the difference between 9 and 10 processors is negligible.

The overall conclusion is that the bloom-join algorithm in conjunction with effective physical database design (i.e. selecting storage structures for the two relations to insure that most queries will have a fast access path available to them) will provide the highest performance and the lowest implementation overhead using the least amount of hardware, specialized or otherwise.

Figure 6. Uniprocessor vs. Multiprocessor Performance



### 3.4. Summary

In this chapter we have developed a model for multiprocessor join processing and examined six distributed join algorithms in detail. The algorithms are designed to either minimize the total execution time or to minimize the response time. We argue that the former corresponds to maximum multi-threaded throughput and the latter to maximum single-threaded throughput. In formulating our model, we developed a new definition of join selectivity which is based on much less restrictive assumptions than other definitions. Not only does our definition have great intuitive appeal, but it has the mathematical properties that either or both relations may be reduced and the estimated size of the various sub-results does not depend on the order in which they are calculated.

The three algorithms which minimize total time (distributed join, semi-join and bloom-join) were compared analytically and using a simulation model. We analyzed the amount of data transmitted by the three algorithms and concluded that the bloom-join algorithm would probably require less than the semi-join and both would require less than the distributed-join algorithm. If total processing is taken into account, we demonstrated that classes of queries exist for which each of the three algorithms is optimal. We also demonstrated that reduction is not universally beneficial: queries exist for which reducing one or the other relation produces substantial reduction in processing cost, however reducing *both* relations produces a substantial increase in processing cost.

The simulation experiments to evaluate the average total query cost as a function of network speed demonstrated that at all speeds the average cost for bloom-joins is less than that for semi-joins which is less than that for distributed joins. Using query optimization results in performance that is somewhat better than using the bloom-join alone. We demonstrated that, for this workload, assuming that the selectivity is



always equal to one in the query optimization cost functions will more than double the amount of processing performed. Duplicate values in the join attribute tend to decrease the amount of processing required for semi-joins and increase the amount of processing for the other two algorithms, although the effect is negligible if optimization is being used.

The three algorithms which minimize response time (fragment & replicate, fragment & rotate and distributed hash join) were also compared analytically and with a series of simulation experiments. The distributed hash join algorithm had the worst overall performance and its response time decreased by a negligible amount as the number of processors increased, although if the number of processors becomes very large its performance may become comparable to that of the other two algorithms. The fragment & replicate algorithm requires a broadcast network facility for acceptable performance and the fragment & rotate algorithm a ring network facility. If these facilities are not present, the amount of overhead required to include an additional processor may be greater than the savings resulting from more parallelism and the query response time may *increase* rather than decrease as processors are added to the calculation. If associative search hardware is used, a system optimizing over the fragment & replicate and the fragment & rotate algorithm will probably have the best performance. If the hardware is not used, the fragment & replicate algorithm alone will probably have the best performance.

If a multiprocessor database machine is implemented using a single algorithm, the bloom-join in conjunction with effective physical database design should result in the highest performance for the least implementation overhead and require the least amount of hardware, specialized or otherwise.

## CHAPTER 4

### CONCLUSIONS AND FUTURE RESEARCH

#### 4.1. Conclusions

In this thesis we have presented a technique for system performance evaluation of uniprocessor and multiprocessor relational database machines. The analysis includes both specialized hardware and software techniques involving data structures and algorithms. The results are implementation independent. We also present a new definition for join selectivity which is consistent: the estimated cardinality of  $A \text{ join } B$  is equal to that of  $B \text{ join } A$ . In addition, the definition allows for various statistical estimates based on a variety of assumptions which may be appropriate to a given application. Semi-join selectivity is defined as a special case of join selectivity.

In a uniprocessor environment, a system built around the tuple substitution and merge join algorithms using query optimization in conjunction with effective physical database design is likely to yield the highest performance for small to moderate sized queries. If a single algorithm is chosen for implementing a multiprocessor database machine for the same workload, the bloom-join algorithm in conjunction with effective physical database design will probably yield the highest performance for the lowest implementation overhead using the least amount of hardware, specialized or otherwise.

In implementing our simulation model we developed a collection of detailed cost functions which estimate the resources consumed during query execution. Some results that follow immediately from the equations are:

- (1) Reducing the amount of data does not universally increase performance: a large class of queries exist for which reducing the amount of data at intermediate

stages during processing actually increases the total amount of processing required.

- (2) Adding additional processors does not necessarily increase performance: Algorithms which have total processing costs that increase as the number of processors increases (e.g. if copies of data must be sent to every processor participating in the query) may realize a net *decrease* in performance if the savings resulting from using an additional processor do not compensate for the additional total cost.
- (3) Response oriented performance speedup for the algorithms and workload studied in this thesis is closer to  $\log_2 N$  than  $N$ : All three of the algorithms which optimize response time have performance which not linear in the number of processors. For the workload under consideration, no more than seven or eight processors can be effectively utilized.

#### 4.2. Future Research

The results we have presented in this thesis are valid for small to moderate sized applications with low selectivities and a small number of duplicates. Other application areas can be characterized in an analogous manner: large relation; very low selectivity, selectivity equal to one (corresponding to functional and multivalued dependencies); and, number of duplicates equal to 1 (corresponding to queries over relation keys). A parallel set of evaluations for these other application areas would be a good complement to this work. In addition, many assumptions were made in making our estimates concrete. While all of these assumptions are valid for some processing environments, other assumptions which are equally valid exist. For example, we assumed that storage structures would be preserved during data transmission. Many environments, i.e. [Seli80a], do not have this property. Repeating the evaluation with assumptions corresponding to different processing environments would yield results which are

comparable to our results. In both cases, the addition and evaluation of new hardware features, data structures and algorithms is straightforward.

Our analysis and simulation experiments evaluated one join operation. Many of the algorithms were designed to execute a sequence of join operations in an efficient manner. Extending our technique and simulation models to this problem is straightforward. The existing (single join) simulation programs execute quickly on a VAX 11/780 and this addition would not make simulation infeasible. Since the distributed optimization problem is so difficult, many algorithms rely on heuristic techniques to attempt to find a non-optimal execution strategy which is better than a randomly chosen one. A multiple-join simulation model which calculates the true minimum strategy would allow us to evaluate the effectiveness of these heuristic algorithms.

Along another vein, a multiprocessor implementation of a rudimentary distributed database machine which supports the data structures and algorithms described above would allow validation of our model through benchmark studies. This would be a valuable addition to our work.

## APPENDIX A

### Distributed Cost Functions

Distributed join cost functions:

Transmit one relation:

Calculation	Operation	Number
Transmit relation	XMIT	C1
Final join	*	join(R,K,C1,T1,V1,S1,C2,T2,S2,V2)
Transmit result	XMIT	R

Transmit both relations:

Transmit relation	XMIT	C1
Transmit relation	XMIT	C2
Final join	*	join(R,K,C1,T1,V1,S1,C2,T2,S2,V2)

Semi-join cost functions for total time:

Reduce relation 1:

Calculation	Operation	Number
Project	*	$\text{project}(K, C2, T2, S2, V2)$
Transmit projection	XMIT	$V2/(2*K)$
Semi-join	*	$\text{join}(R1, K, C1, T1, S1, V1, V2/2K, 2K, \text{OHEAP}, V2)$
Transmit sjoin result	XMIT	R1
Transmit relation	XMIT	C2
Final join	*	$\text{join}(R, K, R1, T1, p*V_{\text{MIN}}, SS1, C2, T2, S2, V2)$

Reduce both relations:

Project	*	$\text{project}(K, C2, T2, S2, V2)$
Transmit projection	XMIT	$V2/(2K)$
Semi-join	*	$\text{join}(R1, K, C1, T1, S1, V1, V2/2K, 2K, \text{OHEAP}, V2)$
Transmit sjoin result	XMIT	R1
Project	*	$\text{project}(K, C1, T1, S1, V1)$
Transmit projection	XMIT	$V1/2K$
Semi-join	*	$\text{join}(R2, K, V1/2K, 2K, \text{OHEAP}, V1, C2, T2, S2, V2)$
Transmit sjoin result	XMIT	R2
Final join	*	$\text{join}(R, K, R1, T1, SS1, p*V_{\text{MIN}}, R2, T2, SS2, p*V_{\text{MIN}})$

SS1 = Storage structure of relation 1 after semi-join reduction

SS2 = Storage structure of relation 2 after semi-join reduction

Bloom-join cost functions for total time:

Reduce relation 1:

Calculation	Operation	Number
Create vector	READ	C2
	SCAN	C2
	WRITE	BF
Transmit vector	XMIT	BF
Hash-join	READ	C1
	SCAN	C1
	WRITE	R1
Transmit result	XMIT	R1
Transmit relation	XMIT	C2
Final join	*	join(R.K,R1,T1,V1,SS1,C2,T2,S2,V2)

Reduce both relations:

Create vector	READ	C2
	SCAN	C2
	WRITE	BF
Transmit vector	XMIT	BF
Hash-join	READ	C1
	SCAN	C1
	WRITE	R1
Transmit result	XMIT	R1
Create vector	READ	C1
	SCAN	C1
	WRITE	BF
Transmit vector	XMIT	BF
Hash-join	READ	C2
	SCAN	C2
	WRITE	R1
Transmit result	XMIT	R2
Final join	*	join(R.K,R1,T1,p*VMIN,SS1,R2,T2,p*VMIN,SS2)

BF = cardinality of bloom vector

## Fragment &amp; Replicate cost functions for total time and response time

## Total Time:

Calculation	Operation	Number
Fragment relation 1	XMIT	$C_1$
Replicate relation 2		
(no broadcast)	XMIT	$N * C_2$
(broadcast)	XMIT	$C_2$
Join fragments	*	$\text{join}(R/N, K, C_1/N, T_1, V_1/N, \text{HEAP}, C_2, T_2, S_2, V_2) * N$
Transmit results	XMIT	$R$

## Response Time:

Fragment relation 1	XMIT	$C_1$
Replicate relation 2		
(no broadcast)	XMIT	$N * C_2$
(broadcast)	XMIT	$C_2$
Join fragments	*	$\text{join}(R/N, K, C_1/N, T_1, V_1/N, \text{HEAP}, C_2, T_2, S_2, V_2)$
Transmit results	XMIT	$R$



## Fragment &amp; Rotate cost functions for total time and response time

## Total Time:

Calculation	Operation	Number
Fragment relation 1	XMIT	C1
Hash fragments	*	$\text{hash}(C1/N, K, T1)$
Rotate & join (no hardware)	XMIT	$N * C2$
(hardware)	READ	$N * C2 * T2$
	SEARCH	$N * C2 * T2$
	WRITE	R
	READ	$N * B1$
	SEARCH	$N * B1$
	WRITE	R
Transmit results	XMIT	R

## Response Time:

Fragment relation 1	XMIT	C1
Hash fragments	*	$\text{hash}(C1/N, K, T1)$
Rotate & join (no hardware)	XMIT	$C2 + N$
(hardware)	READ	$C2 * T2 + N$
	SEARCH	$C2 * T2 + N$
	WRITE	$R/N + N$
	READ	$B1 + N$
	SEARCH	$B1 + N$
	WRITE	$R/N + N$
Transmit results	XMIT	R

$B1 = \text{number of tuples from relation 1 in result} = R1 * T1$

## Distributed Hash Join cost functions for total time and response time

Total Time:

Calculation	Operation	Number
Hash relation 1	*	$\text{hash}(C1, K, T1)$
Distribute	XMIT	$C1$
Hash relation 2	*	$\text{hash}(C2, K, T2)$
Distribute	XMIT	$C2$
Sort buckets	READ	$C1 + C2$
	SORT	$C1 + C2$
	WRITE	$C1 + C2$
Merge buckets	READ	$C1 + C2$
	SCAN	$C1 + C2$
	WRITE	$R$
Transmit results	XMIT	$R$

Response Time:

Hash relation 1	*	$\text{hash}(C1, K, T1)$
Distribute	XMIT	$C1$
Hash relation 2	*	$\text{hash}(C2, K, T2)$
Distribute	XMIT	$C2$
Sort buckets	READ	$C1/N + C2/N$
	SORT	$C1/N + C2/N$
	WRITE	$C1/N + C2/N$
Merge buckets	READ	$C1/N + C2/N$
	SCAN	$C1/N + C2/N$
	WRITE	$R/N$
Transmit results	XMIT	$R$

## APPENDIX B

### Total Time Calculations

Distributed Join, Heap joined to heap using nested loops.

Step	Basic Operation	Number	As named	Reversed
Transmit		C1	1000	10,000
Final Join	Read	$C1(1+T1*C2)$	1.00e8	1.00e8
	Scan	C1	1000	10,000
	Search	$C1*T1*C2$	1.00e8	1.00e8
	Write	R	1750	1750
Transmit		R	1750	1750
		Total	2.00e8	2.00e8

Transmit		C1+C2	11,000
Final Join	Read	$C1(1+T1*C2)$	1.00e8
	Scan	C1	1000
	Search	$C1*T1*C2$	1.00e8
	Write	R	1750
		Total	2.00e8

Semi-Join. Heap joined to heap using nested loops.

Step	Basic Operation	Number	As named	Reversed
Project	Read	$C1 + pmerge(A1)$	1300	2.38e4
	Write	$A1 + pmerge(A1) + P1$	375	1.75e4
	Scan	$C1 + pmerge(A1)$	1.30e3	2.38e4
	Sort	A1	50	1250
Transmit		P1	25	2500
Semi-join	Read	$C2 + V1 * C2$	5.00e7	5.00e7
	Scan	C2	1.00e4	1.00e3
	Search	$V1 * C2$	5.00e7	5.00e7
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	$R2 + R2 * T1 * C1$	2.50e6	2.50e6
	Scan	R2	250	250
	Search	$R2 * T1 * C1$	2.50e6	2.50e6
	Write	R	1750	1750
Total			1.05e8	1.05e8
Project	Read	$C2 + pmerge(A2)$	2.38e4	
	Write	$A2 + pmerge(A2) + P2$	1.75e4	
	Scan	$C2 + pmerge(A2)$	2.38e4	
	Sort	A2	1250	
Transmit		P2	2500	
Semi-join	Read	$C1 + V2 * C1$	5.00e7	
	Scan	C1	1.00e3	
	Search	$V2 * C1$	5.00e7	
	Write	R1	250	
Transmit		R1	250	
Final Join	Read	$R2 + R2 * T1 * R1$	6.25e5	
	Scan	R2	250	
	Search	$R2 * T1 * R1$	6.25e5	
	Write	R	1750	
Total			2.01e8	

Note:  $A1 = (C1 * T1) / (2 * K)$  = cardinality of "projection" with duplicates  
 $P1 = V1 / (2 * K)$  = cardinality of projection  
 $pmerge(A) = A \log_2 A$ .  $\log_2$  = log to the base 2

Bloom-Join, Heap joined to heap using nested loops.

Step	Basic Operation	Number	As named	Reversed
BF-Calc	Read	C1	1000	10,000
	Scan	C2	1000	10,000
	Write	BF	4	4
Transmit		BF	4	4
Bloom-join	Read	C2+BF	10,004	1004
	Scan	C2+BF	10,004	1004
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	$R2+R2*T1*C1$	2.50e6	6.25e7
	Scan	R2	250	250
	Search	$R2*T1*C1$	2.50e6	6.25e7
	Write	R	1750	1750
Total			5.03e6	1.25e8
BF-Calc	Read	C2	10,000	
	Scan	C2	10,000	
	Write	BF	4	
Transmit		BF	4	
Bloom-join	Read	C1+BF	1004	
	Scan	C1+BF	1004	
	Write	R1	250	
Transmit		R1	250	
Final Join	Read	$R2+R2*T1*R1$	6.25e5	
	Scan	R2	250	
	Search	$R2*T1*R1$	6.25e5	
	Write	R	1750	
Total			1.30e6	

Note: BF= Cardinality of bloom-filter = 4

Distributed Join. Heap joined to heap using Tuple substitution  
and creating a secondary index

Step	Basic Operation	Number	As named	Reversed
Transmit		$C_2$	10,000	1000
Build indx	Read	$C_1 + I + pmerge(I)$	42,500	42,500
	Write	$3*I + pmerge(I)$	37,500	37,500
	Scan	$C + pmerge(I)$	40,000	40,000
	Sort	$I$	2500	2500
Final join	Read	$C_2 + 2*C_2*T_2$	21,000	21,000
	Scan	$C_2$	1000	1000
	Search	$C_2*T_2$	10,000	10,000
	Look	$C_2*T_2$	10,000	10,000
	Write	$R$	1750	1750
Transmit		$R$	1750	1750
Total			178,000	169,000

Transmit		$C_1 + C_2$	11,000
Build indx	Read	$C_1 + I + pmerge(I)$	42,500
	Write	$3*I + pmerge(I)$	37,500
	Scan	$C + pmerge(I)$	40,000
	Sort	$I$	2500
Final join	Read	$C_2 + 2*C_2*T_2$	21,000
	Scan	$C_2$	1000
	Search	$C_2*T_2$	10,000
	Look	$C_2*T_2$	10,000
	Write	$R$	1750
Total			177,000

Note:  $I = T*C/K = 2500$

Semi-join Join, Heap joined to heap using Tuple substitution  
and creating a secondary index

Step	Basic Operation	Number	As named	Reversed
Project	Read	$C1+pmerge(A1)$	1350	25,000
	Write	$A1+pmerge(A1)+P1$	375	15,025
	Scan	$C1+pmerge(A1)$	1300	23,750
	Sort	A1	50	1250
Transmit		P1	25	250
Build Sindx	Read	$C2+I+pmerge(I1)$	42,500	1800
	Write	$3*I1+pmerge(I1)$	37,500	1000
	Scan	$C2+pmerge(I1)$	40,000	1700
	Sort	I1	2500	100
Semi-join	Read	$P1+2*V1$	10,025	100,250
	Scan	P1	25	250
	Search	V1	5000	50,000
	Look	V1	5000	50,000
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Build Sindx	Read	$C1+I2+pmerge(I2)$	1800	42,500
	Write	$3*I2 + pmerge(I2)$	1000	37,500
	Scan	$C1+pmerge(I2)$	1700	40,000
	Sort	I2	100	2500
Final Join	Read	$R2+2*R2*T2$	12,750	12,750
	Scan	R2	250	250
	Search	$R2*T2$	6250	2500
	Look	$R2*T2$	6250	2500
	Write	R	1750	1750
Total			179,000	423,125
Project	Read		25,000	
	Write		15,025	
	Scan		23,750	
	Sort		1250	
Transmit		250		
Build Sindx	Read	$R1+I3+pmerge(I3)$	400	
	Write	$3*I3 + pmerge(I3)$	200	
	Scan	$R1+pmerge(I3)$	375	
	Sort	I	25	
Final Join	Read	$R2+T2*R2*2$	12,750	
	Scan	R2	250	
	Search	$T2*R2$	6250	
	Look	$T2*R2$	6250	
	Write	R	1750	
Total			445,275	

$$I1 = T2*C2/K$$

$$I2 = T1*C1/K = 100$$

$$I3 = T1*R1/K = 25$$

Bloom Join. Heap joined to heap using Tuple substitution  
and creating a secondary index

Step	Basic Operation	Number	As named	Reversed
BF-Calc	Read	C1	1000	10,000
	Scan	C2	1000	10,000
	Write	BF	4	4
Transmit		BF	4	4
Bloom-join	Read	C2+BF	10,004	1004
	Scan	C2+BF	10,004	1004
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Build Sindx	Read	C1+I1+pmerge(I1)	1800	42,500
	Write	C1+I1+pmerge(I1)	1000	37,500
	Scan	C1+pmerge(I1)	1700	40,000
	Sort	I1	100	2500
Final Join	Read	R2+2*R2*T2	12,750	12,750
	Scan	R2	250	250
	Search	R2*T2	6250	2500
	Look	R2*T2	6250	2500
	Write	R	1750	1750
Total			55,334	174,766
BF-Calc	Read	C2	10,000	
	Scan	C2	10,000	
	Write	BF	4	
Transmit		BF	4	
Bloom-join	Read	C1+BF	1004	
	Scan	C1+BF	1004	
	Write	R1	250	
Transmit		BF	4	
Build Sindx	Read	R1+I2+pmerge(I2)	400	
	Write	3*I2+pmerge(I2)	200	
	Scan	R1+pmerge(I2)	375	
	Sort	I	25	
Final Join	Read	R2+T2*R2*2	12,750	
	Scan	R2	250	
	Search	T2*R2	6250	
	Look	T2*R2	6250	
	Write	R	1750	
Total			73,250	

$$I1 = T1 * C1 / K = 100,2500$$

$$I2 = T1 * R1 / K = 25$$



Distributed Join, ordered heap joined to ordered heap using merge join

Step	Basic Operation	Number	As named	Reversed
Transmit		C1	1000	10,000
Final Join	Read	C1+C2	11,000	11,000
	Scan	C1+C2	11,000	11,000
	Write	R	1750	1750
Transmit		R	1750	1750
Total			26,500	35,500

Transmit		C1+C2	11,000
Final Join	Read	C1+C2	11,000
	Scan	C1+C2	11,000
	Write	R	1750
Total			34,750

Semi-Join, Ordered heap joined to ordered heap using merge join

Step	Basic Operation	Number	As named	Reversed
Project	Read	C1	1000	10,000
	Scan	C1	1000	10,000
	Write	P1	25	250
Transmit		P1	25	250
Semi-join	Read	C2 + P1	10,025	1250
	Scan	C2 + P1	10,025	1250
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	R2+C1	1250	10,250
	Scan	R2+C1	1250	10,250
	Write	R	1750	1750
Total			27,850	55,750
Project	Read	C2	10,000	
	Scan	C2	10,000	
	Write	P2	40	
Transmit		P2	250	
Semi-join	Read	C1 + P2	1250	
	Scan	C1 + P2	1250	
	Write	R1	250	
Transmit		R1	250	
Final Join	Read	R1 + R2	500	
	Scan	R1 + R2	500	
	Write	R2	1750	
Total			48,850	

Note:  $P1 = V1 / (2 * K) = \text{cardinality of projection}$

## Bloom-Join, Ordered heap join to ordered heap using merge join

Step	Basic Operation	Number	As named	Reversed
BF-Calc	Read	C1	1000	10,000
	Scan	C1	1000	10,000
	Write	BF	4	4
Transmit		BF	4	4
Bloom-join	Read	C2+BF	10,004	1004
	Scan	C2+BF	10,004	1004
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	R2 + C1	1250	10,250
	Scan	R2+C1	1250	10,250
	Write	R	1750	1750
Total			27,766	54,766
BF-Calc	Read	C2	10,000	
	Scan	C2	10,000	
	Write	BF	4	
Transmit		BF	4	
Bloom-join	Read	C1+BF	1004	
	Scan	C1+BF	1004	
	Write	R1	250	
Transmit		R1	250	
Final Join	Read	R1 + R2	500	
	Scan	R1 + R2	500	
	Write	R	1750	
Total			47,782	

Note: BF= Cardinality of bloom-filter = 4

Distributed Join, hash joined to hash using tuple substitution

Step	Basic Operation	Number	As named	Reversed
Transmit		C1	1000	10,000
Final Join	Read	C1+C1*T1	11,000	11,000
	Scan	C1	1000	10,000
	Search	C1*T1	10,000	10,000
	Write	R	1750	1750
Transmit		R	1750	1750
Total			26,500	44,500

Transmit		C1+C2	11,000
Final Join	Read	C1+C1*T1	11,000
	Scan	C1	1000
	Search	C1*T1	10,000
	Write	R	1750
Total			34,750

## Semi-Join, Hash joined to hash using tuple substitute algorithm

Step	Basic Operation	Number	As named	Reversed
Project	Read	$C1 + pmerge(A1) + A1$	1350	25,000
	Write	$A1 + pmerge(A1) + P1$	375	15,025
	Scan	$C1 + pmerge(A1)$	1300	23,750
	Sort	A1	50	1250
Transmit		P1	25	250
Semi-join	Read	$P1 + V1$	5025	50,250
	Scan	P1	25	250
	Search	V1	5000	50,000
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	$R2 + R2 * T2$	6500	2750
	Scan	R2	250	250
	Search	$R2 * T2$	6250	2500
	Write	R	1750	1750
Total			29,400	183,525
Project	Read	$C2 + pmerge(A2)$	25,000	
	Write	$A2 + pmerge(A2) + P2$	15,025	
	Scan	$C2 + pmerge(A2)$	23,750	
Transmit		P2	250	
Semi-join	Read	$P2 + V2$	50,250	
	Scan	P2	250	
	Search	V2	50,000	
	Write	R1	250	
Transmit		R1	250	
Build Sindx	Read	$R1 + I + pmerge(I)$	400	
	Write	$3 * I + pmerge(I)$	200	
	Scan	$R1 + pmerge(I)$	375	
	Sort	I	25	
Final Join	Read	$R2 + T2 * R2 * 2$	12,750	
	Scan	R2	250	
	Search	$T2 * R2$	6250	
	Look	$T2 * R2$	6250	
	Write	R	1750	
Total			206,925	

Note:  $A1 = (C1 * T1) / (2 * K)$  = cardinality of "projection" .  
 with duplicates  
 $P1 = V1 / (2 * K)$  = cardinality of projection  
 $Sort(A) = A \log_2 A$ .  $\log_2$  = log to the base 2

## Bloom-Join. Hash joined to hash using tuple substitution

Step	Basic Operation	Number	As named	Reversed
BF-Calc	Read	C1	1000	10,000
	Scan	C2	1000	10,000
	Write	BF	4	4
Transmit		BF	4	4
Bloom-join	Read	C2+BF	10,004	1004
	Scan	C2+BF	10,004	1004
	Write	R2	250	250
Transmit		R2	250	250
Transmit		C1	1000	10,000
Final Join	Read	R2+R2*T2	6500	2750
	Scan	R2	250	250
	Search	R2*T2	6250	2500
	Write	R	1750	1750
Total			38,266	39,766
BF-Calc	Read	C2	10,000	
	Scan	C2	10,000	
	Write	BF	4	
Transmit		BF	4	
Bloom-join	Read	C1+BF	1004	
	Scan	C1+BF	1004	
	Write	R1	250	
Transmit		R1	250	
Build Sindx	Read	R1+I+pmerge(I)	400	
	Write	3*I+pmerge(I)	200	
	Scan	R1+pmerge(I)	375	
	Sort	I	25	
Final Join	Read	R2+T2*R2*2	12,750	
	Scan	R2	250	
	Search	T2*R2	6250	
	Look	T2*R2	6250	
	Write	R	1750	
Total			73,282	

Note: BF= Cardinality of bloom-filter = 4

## APPENDIX C

### Response Time Calculations

#### Fragment & Replicate Response Time Calculations

Basic Operation		1	5	10	20
Read	$C1/N + I1 + pmerge(I1)$	1800	320	150	70
	$C2/N + 2 * C2 * T2 / N$	510,000	102,000	51,000	25,500
Write	$3 * I1 + pmerge(I1)$	1000	160	70	30
	$R/N$	1750	350	175	88
Scan	$C/N + pmerge(I1)$	1700	300	140	65
	$C2/N$	10,000	2000	1000	500
Search	$C2 * T2 / N$	250,000	50,000	25,000	12,500
Look	$C2 * T2 / N$	250,000	50,000	25,000	12,500
Sort	$I$	100	20	10	5
Xmit	$C1 + C2 + R$	12,750	12,750	12,750	12,750
Total		1,039,100	217,900	115,295	64,008

Read	42,500	7000	3250	1500
	21,000	42,000	21,000	1050
Write	37,500	6000	2750	1250
	1750	350	175	88
Scan	40,000	6500	3000	1375
	1000	200	100	50
Search	10,000	2000	1000	500
Sort	2500	500	250	125
Xmit	12,750	12,750	12,750	12,750
Total	179,000	41,500	26,375	19,188

$I = \text{cardinality of index} = T * C / K$

## Fragment &amp; Rotate Response Time Calculations

Basic Operation	Number	N			
		1	5	10	20
Read	$C1/N + Q1/N +$ $pmerge(Q/N) + C2 * T2 + N$	263,601	252,095	250,950	250,434
Write	$3 * Q1/N +$ $pmerge(Q/N) + R/N + N$	16,451	2665	1235	575
Scan	$C1/N + pmerge(Q/N)$	12,550	1880	835	362
Search	$C2 * T2 + N$	250,001	250,005	250,010	250,020
Sort	$Q/N$	1050	210	105	52
Xmit	$C1 + C2 + R + N$	12,751	12,755	12,760	12,770
Total		556,404	519,610	515,895	514,213

Read	188,751	41,255	24,510	16,702
Write	193,001	34,105	15,935	7413
Scan	167,500	29,000	13,375	6120
Search	10,001	10,005	10,010	10,020
Sort	11,250	2250	1125	562
Xmit	12,751	12,755	12,760	12,770
Total	583,254	129,370	77,715	53,587

## HW Search

Read	$C1/N + T1/N +$ $pmerge(Q/N) + B1 + N$	16,101	4,595	3,450	2,934
Write		16,451	2,665	1,235	575
Scan		12,550	1,880	835	362
Search	$B1 + N$	2,501	2,505	2,510	2,520
Sort		1,050	210	105	52
Xmit		12,751	12,755	12,760	12,770
Total		61,404	24,610	20,895	19,213

Read	185,001	37,505	20,760	12,952
Write	193,001	34,105	15,935	7,413
Scan	167,500	29,000	13,375	6,120
Search	6,251	6,255	6,260	6,270
Sort	11,250	2,250	1,125	562
Xmit	12,751	12,755	12,760	12,770
Total	575,754	121,870	70,215	46,087

$Q$  = cardinality of augmented relation =  $C + B * C / (2 * K)$



## Distributed Hash Join Response Time Calculations

Basic Operation	Number	N			
		1	5	10	20
Read	$C1+Q1+pmerge(Q1)+2*C1/N$ $+C2+Q2+pmerge(Q2)+2*C2/N$	214,350	196,750	194,550	193,450
Write	$3*Q1+pmerge(Q1)+C1/N$ $+3*Q2+pmerge(Q2)+C2/N$	218,700	208,500	207,225	206,587
Scan	$C1+pmerge(Q1)+C1/N$ $+C2+pmerge(Q2)+C2/N$	191,050	182,250	181,150	180,600
Sort	$Q1+Q2+C1/N+C2/N$	23,300	14,500	13,400	128,50
Xmit	$C1+C2+R$	12,750	12,750	12,750	12,750
Total		660,150	614,750	609,075	606,237

$Q$  = cardinality of augmented relation =  $C + B * C / (2*K)$

$pmerge(Q) = Q \log_2 Q$

## REFERENCES

- [84a] . "The Genesis of a Database Computer: A Conversation with Jack Shemer and Phil Neches of Teradata Corporation," *Computer* 17(11) pp. 42-56 (November 1984).
- [Acce85a] . Accel Technologies, Inc., "Database hardware sorter offers increased speed," *Computer* 18(3) p. 130 (March 1985). (New Product Announcement)
- [Aper83a] Apers, Peter M.G., Alan R. Hevner, and S. Bing Yao, "Optimization Algorithms for Distributed Queries," *IEEE Transactions on Software Engineering* SE-9(1)(January 1983).
- [Babb79a] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware," *ACM Transactions on Database Systems (TODS)* 4(1) pp. 1-29 (March 1979).
- [Bane78a] Banerjee, Jayanta, David K. Hsiao, and Richard I. Baum, "Concepts and Capabilities of a Database Computer," *TODS* 3(4)(December 1978).
- [Bane78b] Banerjee, Jayanta and David K. Hsiao, "Performance Study of a Database Machine in Supporting Relational Databases," *4th Int. Conf. on VLDB*, (1978).
- [Batc68a] Batcher, K.E., "Sorting networks and their applications," *Proc. AFIPS SJCC* 32 pp. 307-317 (April 1968).
- [Baye70a] Bayer, R. and E. McCreight, "Organization and Maintenance of Large Ordered Indices," *Proc. 1970 ACM-SIGFIDEI Workshop on Data Description and Access*, pp. 107-141 (November 1970).
- [Bern81a] Bernstein, Philip A., Nathan Goodman, Eugene Wong, Chistopher L. Reeve, and James B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM TODS* 6(4) pp. 602-625 (December 1981).
- [Berr79a] Berra, P. Bruce and Ellen Oliver, "The Role of Associative Array Processors in Data Base Machine Architecture," *Computer*, (March 1979).
- [Bitt83a] Bitton, Dina, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS* 8(3) pp. 324-353 (September 1983).
- [Bora81a] Boral, Haran and David J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," *ACM Transactions on Database Systems* 6(2) pp. 227-254 (June 1981).
- [Bray79a] Bray, Olin H. and Harvey A. Freeman, *Data Base Computers*, Lexington Books, Lexington, MA (1979).
- [Ceri84a] Ceri, Stefano and Giuseppe Pelagatti, *Distributed Databases Principles and Systems*, McGraw-Hill, New York (1984).
- [Come79a] Comer, Douglas, "The Ubiquitous B-Tree," *Computing Surveys* 11(2)(June 1979).
- [DeWi78a] DeWitt, David J., "Direct- A Multiprocessor Organization for Supporting Relational Data Base Management Systems," *Proceedings Fifth Annual Symposium on Computer Architecture*, (1978).
- [DeWi82a] DeWitt, David J., "Query Execution in Direct," *ACM-SIGMOD*

- Conference on Management of Data*, (1982). (mimeograph copy)
- [DeWi84a] DeWitt, David J., Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David Wood, "Implementation Techniques for Main Memory Database Systems," *ACM-SIGMOD*, (1984).
  - [Demu85a] Demurjian, S. and D.K. Hsiao, "Benchmarking Database Systems in Multiple Backend Configurations," *Database Engineering* 8(1) pp. 29-29 (March 1985).
  - [Demu84a] Demurjian, Steven A., David K. Hsiao, Douglas S. Kerr, Jai Menon, Paula R. Strawser, Robert C. Tekampe, and Robert J. Watson, "Performance Evaluation of a Database System in a Multiple Backend Configurations," Memo # NPS52-84-019, Naval Postgraduate School, Monterey, CA (October 1984).
  - [Dohi82a] Dohi, Yasunori, Akira Suzuki, and Noriyuki Matsui, "Hardware Sorter and its Application to Data Base Machine," *The 9th Annual Symposium on Computer Architecture Conference Proceedings*, (April 1982).
  - [Epst78a] Epstein, Robert, Michael Stonebraker, and Eugene Wong, "Distributed Query Processing in a Relational Database System," *ACM SIGMOD*, (1978).
  - [Epst80a] Epstein, Robert and Paula Hawthorn, "Design Decisions for the Intelligent Database Machine," *NCC*, (May 1980).
  - [Fish84a] Fishman, Daniel H., Ming-Yee Lai, and W. Kevin Wilkinson, "Overview of the Jasmin Database Machine," *SIGMOD Conference Proceedings*, (June 1984).
  - [Good80a] Goodman, James Richard, "An Investigation of Multiprocessor Structures and Algorithms for Data Base Managment," Doctoral Dissertation, Memo #UCB/ERL M80/24, Electronics Research Laboratory, University of California, Berkeley, CA 94720 (June 9, 1980).
  - [Good79a] Goodman, Nathan, Philip A. Bernstein, Eugene Wong, Christopher L. Reeve, and James B. Rothnie, "Query Processing in SDD-1: A System for Distributed Databases," Technical Report CCA-79-06, Computer Corporation of America (October 1, 1979).
  - [Hawt79b] Hawthorn, P. and M. Stonebraker, "The Use of Technological Advances to Enhance Data Management System Performance," *Proc. ACM-SIGMOD 1979 Int. Conf. Management of Data*, pp. 1-12 (May 1979).
  - [Hawt79a] Hawthorn, Paula Birdwell, "Evaluation and Enhancement of the Performance of Relational Database Management Systems," Doctoral Dissertation, Memo #UCB/ERL M79/70, Electronics Research Laboratory, University of California, Berkeley, CA 94720 (November 7, 1979).
  - [Hear82a] Heart, F.E., R.E. Kahn, S.M. Ornstein, W.R. Crowther, and D.C. Walden, "The Interface Message Processor for the ARPA Computer Network," pp. 402-415 in *Computer Structures: Principles and Examples*, ed. Siewiorek Bell and Newell, McGraw-Hill Book Company, New York (1982).
  - [Held78a] Held, Gerald and Michael Stonebraker, "B-trees Re-examined," *CACM* 21(2)(February 1978).
  - [Hell85a] Helland, Pat, "Transaction Monitoring Facility (TMF)," *Database Engineering* 8(2) pp. 11-18 (June 1985).
  - [Heym82a] Heyman, Daniel P. and Matthew J. Sobel, *Stochastic Models in Operations Research, Volume 1: Stochastic Processes and Operating Characteristics*, McGraw-Hill Book Company, New York (1982).
  - [Hoel71a] Hoel, Paul G., Sidney C. Port, and Charles J. Stone, *Introduction to Probability Theory*, Houghton Mifflin Company, Boston (1971). basic probability theory

- [IBMa] IBM., "OS ISAM Logic." CY28-6618, IBM, White Planes, New York ( ).
- [Inte82a] Intel., "iDBP DBMS Reference Manual. Preliminary Release 3," Order Number: 222100, Intel Corporation, Austin, TX 78766 (February 17, 1982).
- [Jark84a] Jarke, Mathias and Jurgen Koch, "Query Optimization in Database Systems," *Computing Surveys* 16(2) pp. 111-152 (June 1984).
- [Kies84a] Kiessling, Werner, "Tuneable Dynamic Filter Algorithms for High Performance Database Systems," *Proc. Int'l Workshop on High Level Architecture*, pp. 6.10-6.20 (May 21-25, 1984).
- [Kies84b] Kiessling, Werner, "Access Path Selection in Databases with Intelligent Disk-Subsystems," Unpublished working draft (1984).
- [Kits83a] Kitsuregawa, M., H. Tanaka, and T. Moto-oka, "Application of Hash to Data Base Machine and its Architecture," *New Generation Computing*, (1) pp. 62-74 (1983).
- [Knut73a] Knuth, Donald, *The Art of Computer Programming: Volume 3/ Sorting and Searching*, Addison-Wesley, Reading, Massachusetts (1973).
- [Kooi80a] Kooi, Robert Philip, "The Optimization of Queries in Relational Databases," Doctoral Dissertation, Dept. of Computer and Information Sciences, Case Western Reserve University (September 1980).
- [Kris84a] Krishnamurthy, Ravi and Stephen P. Morgan, "Query Processing on Personal Computers: A Pragmatic Approach," *Proceedings of the Tenth International Conference on Very Large Data Bases*, (August 1984).
- [Lai84a] Lai, Ming-Yee and W. Kevin Wilkinson, "Distributed Transaction Management in Jasmin," *Proceedings of the Tenth International Conference on Very Large Data Bases*, (August 1984).
- [Meno83a] Menon, M.J. and David K. Hsiao, "Design and Analysis of Join Operations of Database Machines," in *Advanced Database Machine Architecture*, ed. David K. Hsiao, Prentice Hall, Inc., Englewood Cliffs, New Jersey (1983).
- [Nech84a] Neches, Philip M., "Hardware Support for Advanced Data Management Systems," *Computer* 17(11) pp. 29-41 (November 1984).
- [Patt85a] Patterson, David A., "Reduced Instruction Set Computers," *Communications of the ACM* 28(1) pp. 8-21 (January 1985).
- [Patt85b] Patton, Peter C., "Multiprocessors: Architecture and Applications," *Computer* 18(6) pp. 29-40 (June 1985).
- [Pram85a] Pramanik, S. and D. Ittner, "Use of Graph-Theoretic Models for Optimal Relational Database Accesses to Perform Join," *ACM Transactions on Database Systems* 10(1) pp. 57-74 (March 1985).
- [Ritca] Ritchie, D.M. and K. Thompson, "The UNIX Time-sharing System," *The Bell Systems Technial Journal* 57(6) pp. 1905-1929 (July 1974 ). Part 2
- [Rudo82a] Rudolph, Jack A. and Kenneth E. Batchner, "A Productive Implementation of an Associative Array Processor: STARAN," pp. 317-331 in *Computer Structures: Principles and Examples*, ed. Siewiorek Bell and Newell, McGraw-Hill Book Company, New York (1982).
- [Sacc81a] Sacco, Giovanni Maria and Mario Schkolnick, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," Technical Report #RJ3354 (40310), IBM Research Laboratory, San Jose, CA 95193 (12-29-81).
- [Sacc82a] Sacco, Giovanni Maria and S. Bing Yao, "Query Optimization in Distributed

- Data Base Systems," in *Advances in Computers*, Academic Press (1982).
- [Schu78a] Schuster, S.A., H.B. Nguyen, E.A. Ozkarahan, and K.C. Smith, "Rap. 2 - An Associative Processor for Data Bases," *Proceedings Fifth Annual Symposium on Computer Architecture*, (1978).
- [Seit85a] Seitz, Charles L., "The Cosmic Cube," *Communications of the ACM* 28(1) pp. 22-33 (January 1985).
- [Seki83a] Sekino, Akira, Ken Tadeuchi, Takenori Makino, Katsuya Hakozaki, Tsugunori Doi, and Tatsuo Goto, "Design Considerations for an Information Query Computer (IQC)," in *Advanced Database Machine Architecture*, ed. David K. Hsiao, Prentice Hall, Inc., Englewood Cliffs, New Jersey (1983).
- [Seli79a] Selinger, P. Griffiths, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of ACM-SIGMOD*, (1979).
- [Seli80a] Selinger, Patricia Griffiths and Michel Adiba, "Access Path Selection in Distributed Database Management Systems," *Aberdeen Conference on Databases*, (July 1980).
- [Smit76a] Smith, Alan Jay, "Sequentiality and Prefetching in Database Systems," IBM Technical Report (March 19, 1976).
- [Ston79a] Stonebraker, Michael, "Muffin: A Distributed Data Base Machine," *Proc. First Annual Conference on Distributed Computing*, (1979).
- [Ston83a] Stonebraker, Michael, John Woodfill, Jeff Rangstrom, Marguerite Murphy, Marc Meyer, and Eric Allman, "Performance Enhancements to a Relational Database System," *ACM TODS* 8(2) pp. 167-185 (June 1983).
- [Ston78a] Stonebraker, M., "A Distributed Data Base Machine," Memo. # UCB/ERL M78/23, Electronics Research Laboratory, University of California, Berkeley, CA 94720 (23 May 1978).
- [Su79a] Su, Stanley Y.W., "Cellular Logic Devices: Concepts and Applications," *Computer*, (March 1979).
- [Tana84a] Tanaka, Yuzuru, "Bit-Sliced VLSI Algorithms for Search and Sort," *Proceedings of the Tenth International Conference on Very Large Data Bases*, (August 1984).
- [Ullm82a] Ullman, Jeffrey D., *Principles of Database Systems*, Computer Science Press, Rockville MD (1982). Second Edition
- [Vald82a] Valduriez, Patrick, "Semi-Join Algorithms for Multiprocessor Systems," *Proceedings ACM-SIGMOD Conference on Management of Data*, (1982).
- [Ward84a] Ward, A. G., "Name Tracing Using the ICL Content Addressable File-store," *Proceedings of the Tenth International Conference on Very Large Data Bases*, (August 1984).
- [Wolf82a] Wolff, R.W., *Queueing Theory*, Class Notes 1982.
- [Yu84a] Yu, C.T. and C.C. Chang, "Distributed Query Processing," *ACM Computing Surveys* 16(4) pp. 399-433 (December 1984).