ON THE USE OF DATA-FLOW TECHNIQUES IN DATABASE MACHINES

by

HARAN BORAL

Computer Sciences Technical Report #432

May 1981

ON THE USE OF DATA-FLOW TECHNIQUES IN DATABASE MACHINES

by

HARAN BORAL

A thesis submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1981

ABSTRACT

The past decade has seen a number of design efforts in the area of database machines. Research has shown that all of the major designs suffer from some flaw leading to the inefficient execution of one or more operations. In this thesis we show that the lack of systematic study of the algorithms to be used by an architecture before a hardware design is picked is the reason for these flawed designs. We then consider a number of possible algorithms for all the relational algebra operators and introduce a new design based on a group of these. The proposed machine utilizes a local network communication mechanism and employs a data-flow strategy for query processing.

Previous research has shown both advantages and disadvantages of using a data-flow query processing strategy. In particular, it was shown that data movement between the mass storage devices and processors is minimized at the expense of additional control messages. In this design we show how such a strategy can be employed without the large control overhead.

We also consider the problem of associating logic with a disk for the implementation of certain operations "on the fly". Three design approaches are examined and compared.

It is shown how an associative disk can be incorporated into a database machine that supports both on-the-disk-and off-the-disk processing.

ACKNOWLEDGEMENTS

There are numerous people who have helped me along the way and I would like thank them at this time. My advisor, Professor David DeWitt, helped me formulate a number of my ideas; provided more than generous support; was a good friend; arranged for an excellent working environment; etc. Professor Marvin Solomon was willing to provide some of his time to help me prepare for the Foundations qualifier. Professors Bob Cook and Jim Goodman read earlier drafts of this dissertation and made numerous helpful comments. Professor Cook also made a number of important suggestions during my prliminary examination. My colleagues and friends, Kevin Wilkinson and Dina Friedland, were always willing to listen to my (sometimes jumbled) ideas and provide constructive criticisms.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The past decade has seen several design efforts in the area of database machines. These efforts were initiated partly in response to the needs expressed by the user community, and partly due to the availability of cheap and new hardware. Data processing was changing from the traditional off-line, single-user, batch mode to the on-line, concurrent, multi-user mode. Also, the advent of terminal-oriented, time-sharing computer systems, and user-friendly software systems, caused computer resources to become more easily usable. These changes brought about an increase in the user population, and to a larger extent a growth in the size of databases. Databases of billions and even trillions of bytes already exist today and there are projections for further growth.

The concept of database machines originated with the XDMS back-end computer at Bell Laboratories [1]. The motivation for the design of XDMS (and future database machines) lay in the size and complexity of both data and programs to manage it. It was felt that by shifting the database management system (DBMS) from a general-purpose computer (termed the host) to another computer (termed the "back-end"), dedicated to its execution, several gains

would be made. Of particular interest are: a possible improvement of response time and better utilization of the general-purpose computer by users of programs other than the DBMS.

However, the performance enhancement achievable by XDMS-like database machines is limited for three reasons. The first reason is that XDMS was designed for the DBTG Network model [2]. Application programs are written in a high-level procedural language and executed on the host computer. Calls to the DBMS are trapped by the host and forwarded to the back-end for execution. In this type of an environment the communication overhead may well offset any advantages gained by executing a relatively simple operation, such as reading a single record, on the back-end. The second, and more important, limitation is that in such systems the back-end computer would be a conventional von-Neumann machine. Such machines were designed for the execution of numeric operations. In querying and updating a database very little numeric processing is used; rather, specialized instructions for text manipulation are needed. Finally, we note that while a powerful machine can be used as the back-end computer, no parallelism can be employed in the manipulation of the data. Thus, database machine researchers have been concentrating on the design of special-purpose architectures that can support parallel operations on data.

By 1980 a number of designs had emerged that use parallelism for performance enhancement of query execution in relational databases. However, it is only recently that an attempt to compare the performance of the various designs for a benchmark of queries was made by Hawthorn and DeWitt [3]. Since the machines compared were designed for the same purpose: the execution of queries on a database stored in a tabular form[1], one would expect that the study would have concluded that a specific design, or a design of a particular type, is the best. It was thus surprising that the authors found that no one machine executed all the benchmarked queries well.

In a subsequent paper, Hawthorn [4] shows that one could classify databases according to the transaction types that are executed on them. For example, transactions on statistical databases will place radically different requirements on the DBMS (and thus on the database machine) than would bibliographic search transactions on text databases. However, as stated above, this result does not apply to the database machines studied in [3] since all of these machines were designed to execute a small number of

---

1 Although some of the machines studied were designed to handle a variety of data models, the underlying storage structure viewed by the hardware is either the relational model or a tabular structure similar to it.

well-defined high-level operations on a well-defined data structure.

In this thesis we describe a new database machine design. Our purpose in undertaking this task is threefold. First, we feel that a database machine should execute all its operations (in this case relational algebra operators) efficiently. As has been shown in [3], this is not the case with the present architectures. Second, we believe that in order for a database machine design to be viable, it must support a large number of transactions per time unit. Many of the designs that have been described in the literature make use of intra-instruction parallelism to achieve this goal. The number of transactions that can be supported by a machine that uses only intra-instruction parallelism is limited because only a single transaction can be active at a time. We thus feel that to be successful, a database machine must support intra-instruction as well as inter-instruction and inter-program (query) parallelism. That is, the machine must be of the <u>m</u>ultiple <u>i</u>nstruction stream, <u>m</u>ultiple <u>d</u>ata stream (MIMD) variety.

Third, we feel that because the operations as well as the underlying data structures in relational databases are well understood, a top down design method should be used in the design of an architecture. The design process should start with the study of a number of algorithms for <u>all</u> the high-level operations to be executed by the machine.[2] The results of such a study would then be used to specify the types of services (low-level primitives) required of the hardware. Finally, an architecture which implements these low-level operations should be designed using off-the-shelf "non-exotic" technology.

In Chapter 2 we survey the database machine literature. The survey is divided into two major parts. In the first part we classify the existing designs into four distinct classes. Each machine type is characterized in terms of architectural features that distinguish it from the others, and the algorithms it uses to implement the high-level operations. Examples from the literature are used to illustrate the ideas described. The second part discusses two performance evaluation studies of database machines.

In Chapter 3 we describe the method used to arrive at the final design. We begin with a critique of database machine research. We argue that the reason database machine research has been unsuccessful is the lack of method in the area. The goals of this dissertation are described in some detail. Specifically, we advocate an "algorithm directed" approach to the design of database

[2] In the event that not all the operations are considered to be equally important one may wish to study only a subset of them. We believe that in the case of the relational algebra all the operations are important and have therefore included all of them in our study.

machines rather than an "architecture directed" one. We follow this with a description of the algorithms that form the basis for the architecture.

One of the performance evaluation studies discussed in Chapter 2 compares a number of database machine designs of two types and concludes that future machines should include features from both types. In Chapter 4 we present some results based on simulations of architectures of these two classes. In particular, we examine the problem of processor allocation strategy in DIRECT [5], an MIMD database machine. We also study the effect of bus contention on a number of different architectures that process selections directly on a disk.

Chapter 5 contains the description of the architecture which implements the low-level operations required by the algorithms used. The description of the architecture is broken down into a number of parts. We begin with a discussion of the physical components. In the description of each component-type we discuss our choice of a particular implementation and contrast it with a number of other possibilities. We continue by showing how the algorithms specified earlier are actually implemented. Then, the actions of the various components are characterized. We conclude with a discussion of how data integrity is maintained through the implementation of concurrency control and crash recovery mechanisms.

Chapter 6 will include our conclusions and an outline of future research in this area.

CHAPTER 2

DATABASE MACHINES LITERATURE SURVEY

2.1. A Brief History of Database Machine Development

A database machine is a collection of specialized hardware intended to support basic database management functions [6]. This hardware is to be organized as an independent machine which is able to communicate with any number of general-purpose computers (called hosts) and satisfy their requests. Some advantages claimed for a separate machine approach are: economy through specialization for hardware, less complex software systems, data sharing among many (possibly different) machines, isolation of the protection function and its implementation free of operating system constraints [1], increased performance [6], and, simpler storage structures in the absence of indices [7]. Some claimed disadvantages are: unbalancing of resources and response time overhead [1].

Typically, indices in a database occupy between 1 and 10 percent of the database size [8]. By their elimination that much space becomes free. This is quite significant in databases that store 10^{10} bytes. Maintenance of these indices is also eliminated. It is not clear, however, whether one wants to eliminate the use of indices altogether in a database machine. DBC [9] is an example of a database machine that uses indexing to reduce the data space to be searched for each query. We shall consider this issue in a later section.

One can basically ignore the claimed disadvantage of the difficulty of load balancing given the trend of computer hardware costs. Response time overhead has been studied by Hawthorn and Stonebraker [10]. They classify relational queries into three classes according to their execution time components. The classes are: overhead intensive, data intensive, and multi-relation data intensive.

Overhead intensive queries are those queries for which the DBMS spends more time performing overhead functions such as directory look up than it does executing code directly related to the query. Data intensive queries are those queries that require the DBMS to scan large amounts of data and therefore spend little of its time performing overhead functions. Finally, multi-relation data intensive queries are those queries that are data intensive but reference more than a single relation (e.g. a query that includes a join).

Hawthorn and Stonebraker show that for overhead intensive queries a DBMS running on a conventional processor can search an index, or use a hashing function, to find the data as quickly, or faster, than a database machine would

because of the high communication cost between the machines. They also show, though, that for data intensive queries the response time in database machines is significantly better.

Most database machine researchers have concentrated on the relational model [11]. Operations are performed on relations, which are basically non-hierarchical collections of objects. It is thus possible to view a relation as a vector, where the base element is either a tuple, or a fixed-size block of tuples. Processors can be allocated to operate on some number (possibly one) of base elements. Query languages for relational systems are typically non-procedural and thus amenable to execution by a number of processors.

It is not clear whether parallelism can be used in a similar way to enhance the performance of all DBMSs. Other, older, data models, such as the DBTG Network model [2], were designed to optimize the execution of database query programs by requiring the programmer to incorporate access path information into his program. In such programs, access to the database is performed a record at a time using physical links between the stored records. Also, access to the database is through inherently sequential procedural programs which constrain the amount of parallelism attainable.

As indicated earlier the first database machine implemented was XDMS [1]. XDMS was designed to enhance the performance of network type DBMSs. It is a single instruction stream, single data stream (SISD) system and as such can only achieve a minimal performance enhancement over a general-purpose computer handling all database activities. A large number of database machine designs have been offered since. We classify them into four categories. Our classification is similar to that of [12] and is intended only to simplify the description of the multitude of machines found in the literature.

Slotnick [13] pioneered the idea of associating logic with read/write heads of rotating storage devices. Slotnick's argument was that the logic could be employed to search the data on the mass storage unit and thus limit the amount of data to be transferred to the host for further processing. The idea received some attention and has been used as the basis for a number of designs. The feature that is common to all of these designs is that a query is executed on the disk, usually with the assistance of a single controlling processor. For this reason we term such database machines associative disks. In Section 2.2.1 we describe the features of a number of associative disks.

The performance of RAP [14], an associative disk type database machine, is compared with that of an hypothetical uni-processor DBMS in [15]. It is shown that for operations that can be processed in linear time on a uni-processor, RAP outperforms the conventional systems by orders of magnitude. However, for operations that require non-linear time on a uni-processor, RAP performs only marginally better. A number of "off-the-disk" database machines have been offered as alternatives to associative disks as a result of this. In Section 2.2.2 we survey two of these designs.

Our next classification includes all database machines that contain features from both associative disks and off-the-disk machine categories. We term this group Hybrid architectures. Two designs of this type are described in Section 2.2.3. Recently, Epstein and Hawthorn [16] pointed out that database machines that run on a specially designed SISD computer can serve the needs of a large number of medium size user groups better than expensive high-performance machines that use parallelism. We thus describe one such database machine in Section 2.2.4. The following descriptions include both the architectural specifications and algorithms for query processing.

In discussing the query processing algorithms we assume that each machine must support a number of important operations. These are: selection, join, projection, scalar aggregates, aggregate functions, append, delete, and modify. For a definition of these operations, particularly the aggregates, see Appendix A.

2.2. Associative Disks

Following Slotnick's paper, a number of researchers (Parker [17], Minsky [18], and Parhami [19]) focused their attention on the design of logic-per-track systems. None of these is a comprehensive proposal for the implementation of a database machine, but they served as a source of ideas for future efforts.

2.2.1. CASSM

CASSM [20,21] was the first database machine design to employ parallelism. It was designed to support all three major data models. The storage medium used is a fixed head disk with some logic in each head. The logic is considerably more complex than that proposed by Slotnick (for example, it can perform arithmetic operations).

Each data item is stored as an ordered pair:

<attribute name , value>

Data items belonging to the same record are stored in a physically contiguous block preceded by record and relation identifiers. A fixed number of mark bits are associated with each attribute and each record. These are used to identify result data of one operation that is the input to a subsequent operation (this includes I/O and garbage

collection operations). Strings are stored only once in the database, separately from the records in which they are values. In these cases the value field of the ordered pair is a pointer to the string. These pointers are also used for the implementation of databases using the Hierarchical or Network data models.

The processing elements are controlled by a single processor which is responsible for communication with the "outside world" (one or more host computers), distributing instructions to the processors, and collating and processing intermediate results. This includes forming the result relation at the end of a query.

When executing a selection query, a processing element marks tuples belonging to the relation in one disk revolution. A second revolution is used to search the marked records for the desired attribute and check its value; qualifying attributes are marked. A third revolution is used to output the marked attributes. In the event that the marked attribute is a string, the third revolution is used to chase the pointer in the value field of the marked ordered pair. An additional, fourth, revolution is required to output the marked string.

Joins are implemented using a hashing scheme and an auxiliary memory. This scheme was first proposed for use in the CAFS [22] and LEECH [23] database machines. The hash function is applied to the joining attribute of the smaller of the two relations. The result of each application is used as the index to a bit vector in the auxiliary memory (RAM is used). Associated with the set bit are the attribute values that hashed to that index. Next, the hash function is applied to the joining attribute of the tuples in the second relation. The result is checked against the bit vector, and if that bit is set, against the list of values associated with it. If a match occurs the attribute (in the second relation) is marked for output, the corresponding bit in a new vector is marked, and the joining value saved. In the next step the hash function is applied to the first relation, this time checking the bit position indexed by the hash value in the new vector. If the bit is set the values are compared. A match causes the attribute (in the first relation) to be marked for output. In the final step the marked attributes are collected by the controller which forms the result relation by actually materializing the join.

Since the CASSM processors can perform arithmetic functions, aggregate operations can be processed locally. The results from each processor are sent to the controller for a collation of the intermediate results. Aggregate functions are implemented as a sequence of selection subqueries, each designed to handle one partition.

Queries are executed in a <u>s</u>ingle <u>i</u>nstruction stream, <u>m</u>ultiple <u>d</u>ata stream (SIMD) mode, although the output of values can be overlapped with the execution of another instruction.

One of the problems with the CASSM is that the processors are connected via a single bus to the controller and the auxiliary memory used in joins. Contention for the bus among processing elements with output can severely hamper the performance of the machine for all query types, but especially the join [3].

2.2.2. RAP

In RAP [14,24,25], a tabular data structure, similar to relations but allowing duplicates, is used for the storage structure. Tuples are stored bitwise along a track. Only tuples from one relation are allowed on a track, although numerous tracks can be used to store a relation. As in CASSM, a tuple is augmented with a fixed number of mark bits (attributes are not) that serve the same purpose. Processing of a selection operation is similar to CASSM, although it is faster because of the simpler data structure. Also, the processing elements have the capability of scanning for a number of different values in parallel.

Joins are processed as a series of selection subqueries on the larger relation, using the values of the joining attribute in the smaller relation as the selection criteria. Like CASSM, RAP implements only an implicit join. In an implicit join tuples to be joined together are marked as such. In order to materialize the join the marked tuples must be sent to the controller, where the result relation is formed. The processing elements do not have arithmetic capabilities associated with them, necessitating the transfer of tuples to the controller for the execution of aggregates. Aggregate functions are implemented as in CASSM, except that the processing elements must send all of the tuples participating in the operation to the controller for all computations.

Ozkarahan and Sevcik [24] describe a virtual RAP machine. In this organization the database resides on some number of conventional mass storage devices. The RAP system consists of a number of cells, each with a pair of tracks. The controller assumes the additional responsibilities of loading the tracks with data to be examined. Each processor can examine only one track at a time. However, while one track is being examined, the second can be loaded under the supervision of the controller. This organization is further described in [25] with an emphasis on the internal structure of the cells and data organization. It is

expected that the cells will employ one of the new memory technologies, such as <u>c</u>harged <u>c</u>ouple <u>d</u>evices (CCDs), <u>m</u>agnetic <u>b</u>ubble <u>m</u>emories (MBMs), or <u>e</u>lectronic <u>b</u>eam <u>a</u>ddressable <u>m</u>emories (EBAMs), to implement their storage component. The main advantage of these memory technologies (especially MBMs) over disks is that the movement of the stored bit stream can be halted at any time instance. This feature helps to reduce time losses due to channel contention (see Section 4.3). Another advantage is their higher data delivery rates.

2.2.3. RARES

In RARES [26] a different storage format from RAP's is used. Tuples are stored across tracks in byte parallel fashion. That is, byte 0 of a tuple is stored on track 0; byte 1 of the same tuple is stored in the same position on track 1; and so on. The tracks used to store a tuple form a band. Band sizes vary according to the tuple length. It may be necessary to use more than one disk "radius" to store a single tuple. The rationale for using this orthogonal storage layout is that in outputting selected tuples, contention between the processing elements for the bus can be reduced. This is based on the observation that in RAP, a single processing element will tie the bus up for a relatively long period of time (proportional to the tuple length). While the bus is in use, the other processing elements can continue searching for qualifying tuples as long as they have sufficient temporary storage to hold them (RAP'S original design had none). The amount of temporary storage required by each processing element must be sufficient to hold a few "average" size tuples (One of RAP's latest redesigns [25] mentions the figure of 1 Kbyte of temporary storage). In RARES, on the other hand, the amount of temporary storage needed is only a few bytes, since each tuple is distributed across a large number of tracks.

One feature of any design like RARES is that the data must be sent to the controlling processor in a particular sequence in order to allow the controller to construct the tuple correctly. This places a constraint on the design of the hardware (the storage medium) which may make it impossible to construct.

RARES was designed to be part of a database machine and thus there are no specifications of other relational operators.

2.2.4. Purely Associative Arrays

The three systems described so far have one feature in common, they are pseudo-associative devices. Berra and Oliver [27] discuss the use of fully associative arrays in database machines. This approach calls for the use of a

bit slice associative processor with a fast staging buffer. The processor is organized as a number of two dimensional modules whose total storage capacity equals that of a disk track. The buffer memory consists of a number of modules, each of which has a capacity equal to that of the processor's (all of it). Loading the processor is performed by a custom-designed I/O device which is capable of selecting a module from the buffer, and distributing its contents to all the modules in the processor in parallel. While one buffer memory module is emptied into the processor, one or more of the other modules can be loaded from the disk.

One advantage to this approach is that the complex operations that require repeated scanning of data will be executed much more efficiently than they would be on a disk, if the data fits into memory. However, since the amount memory in purely associative systems is limited (due to its very high cost), generally, the data will have to be swapped in and out of the device a large number of times, possibly offsetting this gain. Oliver [28] describes and compares the performance of RELACS, an associative processor designed for relational data management, to RAP. The comparison is based on a performance study of RAP [15] (see Section 2.6.1). It is shown that in the worst case, RELACS performs at about the same level that RAP does.

However, in the best case it is about 3 orders of magnitude faster. In the above, worst and best case refer to the size of the associative memory.

2.2.5. VERSO

Since fixed head disks are now an (almost) obsolete technology, more recent associative disk designs have been using moving head disks. In VERSO [29], a single processor is placed between the disk (which could be a conventional moving head disk or one modified to allow parallel read out) and the memory device into which the desired data is to be delivered. The processor acts as an I/O filter, scanning the tuples and forwarding only those that match the selection criteria. In order to do this, the processor must be able to scan the data as fast as the disk delivers it. This problem is quite complicated because selections can have varying complexity. To achieve this, the processor is organized as a finite state machine which executes very simple microcode instructions. The processor must, however, have the capability of directing the disk controller to stop data delivery in the case that it cannot keep up with its rate.

Bancilhon and Scholl [29] also discuss the possibility of using VERSO to execute other relational operations. It is shown, that if a new normal form is used for organizing relations, the number of joins to be executed will be much

smaller than in the relational model. Thus, one could afford to pay an occasional performance penalty and execute the joins by the filter (the new normal form also simplifies the join algorithm).

2.2.6. SURE

SURE [30] uses a moving head disk modified to enable parallel read out from all of the recording surfaces simultaneously. The output is collected into a single high-speed broadcast channel from which it is read by a number of processors. Each processor is a very high-speed pipelined unit with a simple instruction set geared towards searching. A selection query is broken down into as many simple components as possible, each of which is assigned to one of the processors for execution. The actual number of processors used for the execution of a selection query depends on its complexity.[1]

Although both SURE and VERSO seem to be feasible today, advances in disk technology may change that. Recently, new disks have appeared on the market that provide higher data transfer rates. An example is the IBM 3380 [31] which has a data transfer rate of 3 Mbytes/second. Such disks place unusually high speed

---

[1] As an aside, it should be noted that SURE is an actual example of a multiple instruction stream, single data stream (MISD) architecture.

requirements on I/O filters such as VERSO and SURE which must operate in real time. It may be possible that as new and faster disks are manufactured, processors can be designed to keep up with the data delivery rates. However, an alternative approach is currently under investigation in the Technical University in Brauschweig for the design of their new database machine RDBM [32].

Rather than broadcasting the data directly to the processors it is delivered to a buffer memory (RAM in their current design). Some number of processors are connected via a high-speed bus to the memory, from which blocks of data are read for processing. The number of processors can be increased to offset an improved data transfer rate of a new disk. In this type of an organization the critical component is the bus, which must be able to support a large number of processors, as well as very high data transfer rates.

Like RARES and VERSO, SURE is intended to function only as a search processor, perhaps in the context of a database machine or serving a DBMS on a general-purpose computer.

2.2.7. CAFS

The final machine in this category is CAFS [8,22] which is commercially available from ICL Ltd. The architecture of CAFS is quite similar to that of SURE. A parallel readout disk is used, its output is placed on a high-speed broadcast channel where it is scanned by a number of processing elements, each of which executes a part of the selection operation. However, unlike SURE, CAFS is intended to serve as a database machine.

Joins and projections are executed using bit vectors as in CASSM. However, a major difference between the two machines is that only a single processor (specially designed) is used to do this in CAFS. Using a single processor (which must operate at the data delivery rate) eliminates all the memory conflict and bus contention problems of CASSM.

2.2.8. Associative Disks that Use New Technologies

There are several research efforts taking place which are examining the use of new memory technologies, particularly MBMs, for the intelligent storage of relational databases [33,34,35,36,37,38,39]. However, none of these efforts have yet culminated in the design of a database machine. Most of the research is geared towards the design of chips that would have capabilities similar to those that the RAP and CASSM cells have.

2.3. Off-the-Disk Machines

2.3.1. INFOPLEX

INFOPLEX [40] realizes database management functions by way of a functional hierarchy. Some examples of possible functional levels would be a language interface level and a data access path level. Each function in the hierarchy is implemented using a microprocessor complex. Data is pipelined between levels. Thus, both intra- and inter-instruction parallelism can be attained. An intelligent storage hierarchy employing different technologies of varying speed and cost is used to store the database. The design of the organization is based on locality of reference observations in databases. Specifications of the implementation of both the functional and storage hierarchies are very sketchy and, thus are not presented here.

2.3.2. DIRECT

In this section we describe the architecture and query processing in DIRECT [5,41,42]. The design of DIRECT is accorded considerably more attention than other machines for a number of reasons. First, it was through work on the implementation of DIRECT that we became throughly familiar with issues in database machine design. Second, our work on processor allocation strategies (reported in Section 4.2) was the prime catalyst for this dissertation.

Finally, by discussing the architecture of DIRECT and query processing in it, we shall be able to clearly illustrate the problems which must be addressed in designing an MIMD database machine.[2]

In its original design DIRECT was intended to serve as a back-end database machine to INGRES [43] although it should be able to support any relational DBMS. The INGRES parser converts all queries into a tree format. Leaf nodes in the tree represent operations that are executed on permanent relations in the database. Non-leaf nodes operate on temporary relations produced by their children nodes. Since all the operations require at most two input relations and always produce a single output relation the query tree is binary. In the remainder of this thesis we shall assume that the input to the database machine is a query tree in the form described above. An instruction to be executed by the database machine will generally correspond to a node in the tree.[3]

---

[2] Of the various designs surveyed in this chapter only DIRECT, INFOPLEX, HYPERTREE, and DBC (see Section 2.4 for the description of the last two architectures) can operate in MIMD mode. Of these, DIRECT is the only design that has been implemented and about which sufficient information exists to make a large number of observations.

[3] Note that this does not mean that an INGRES operator (such as append) will correspond to a single node, although this will generally be the case.

2.3.2.1. Overview of the Architecture

The design of DIRECT was undertaken after a critical look at other database machine research projects, particularly RAP. It was felt that RAP (which in the mid 70's was the most advanced and best known database machine design) suffered from a number of major shortcomings. One of these was its performance in the execution of complex operations such as the join (see Section 2.6.1). Another was the SIMD nature of its operation. Since in SIMD machines only a single instruction from a single program is executed at any given time instance, the number of transactions per time unit that such a database machine could support is limited. Another observation was that while a majority of the operations executed by the processing elements in a database machine involves searching text, other capabilities are sometimes needed (for example, arithmetic operations for aggregates). Thus, DIRECT was designed to employ general-purpose micro-processors that will operate in an MIMD mode. Finally, the known algorithms for complex operations require repeated scans of the data, thus a mechanism for doing this efficiently was required.

In a DIRECT configuration there is some number of processors (termed query processors) whose function is to execute operations, such as selection, join, and update on the database. These processors are controlled by a mini-

ıputer (termed back-end controller) which is responsible · distributing instructions and overseeing data transfers the processors. The database resides on some number of ıs storage devices (moving head disks). Each relation is ·anized as a vector of fixed size pages. A number of CCD ıory units serve as a distributed cache. The query pro-.sors and CCD devices are connected by a cross point tch that has two important capabilities: any number of ·cessors can read the same CCD device simultaneously; ., any two processors can read from any two CCD devices .currently. A sample configuration is shown in Figure .

The back-end controller initiates instructions as soon resources become available. Unlike associative disks, re the processors are physically bound to specific data, ECT must have some algorithm for allocating the proces-s (and CCD memories) to instructions (or data). Choice the specific algorithm used for resource allocation in ECT is discussed in Section 4.2. Initiating an instruc-n means sending the code to be executed to a number of cessors. The code consists of a loop in which the pro-sor:

requests a page of data to operate on from the back-end

then waits for a message telling it which CCD device to read the page from (or an instruction termination message)

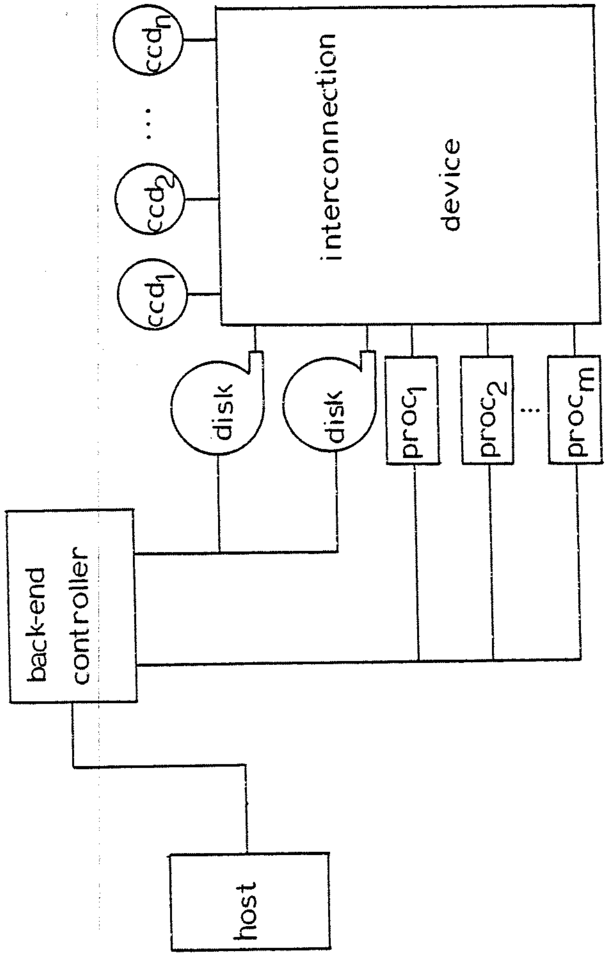

Figure 2.1: A DIRECT Configuration

(3) reads the page
(4) notifies the controller of the completion of the read
(5) and proceeds to execute the instruction.

In the event that a processor has output, it requests from the controller the address of an empty CCD device to which the output can be written.

The cache memory (CCD units) also serves as a temporary storage device for result pages of one operation that are to be used in a subsequent operation. This feature obviates the need for the mark bits used in a number of the associative disk machines. There are several reasons for using temporary relations rather than mark bits. First, because of the small and fixed number of mark bits per tuple, the number of concurrent operations executable on any relation is limited to one (i.e. read locks are exclusive).[4]

Second, temporary relations offer the possibility of executing a number of update operations on a single relation concurrently. Each update operation produces its own copy of the updated relation. Then, at commit time, the controller can find conflicts (by checking the page tables of the temporary relations against each other and against the original page table). In the event that no conflicts

[4] It should be pointed out that [25] briefly discusses the use of partitioned mark bits as a solution to this problem.

are found (each update operation modified a different page) a new page table incorporating all the modified pages can be created. Otherwise, the controller creates a new relation with some of the results and issues instructions to re-execute some of the updates on the new relation.

Third, output of the result can be executed more efficiently with the use of temporary relations since the data to be output is organized as a small number of large pages. If mark bits are employed the tuples to be sent to the host will be scattered over a large number of data pages. Furthermore, the original relation cannot be used by another, unrelated, operation until all the selected tuples have been transferred to the controller.

2.3.2.2. Algorithms for the Relational Operators

DIRECT executes selections similarly to the other systems described. The main differences are that the relation must first be brought into the cache, and the processors must request a data page from the controller each time they are done with their current page.

Joins are executed using the broadcast capabilities of the cross point switch. The larger of the two relations being joined is designated as the outer relation, the other is the inner relation. Each processor participating in the join receives one page of the outer relation. If the page

is not sorted on the joining attribute, the processor sorts it. Next, the pages of the inner relation, which are sorted on the joining attribute, are broadcast, one at a time, to all the processors with an outer page. Each processor joins its outer page with the incoming stream of inner pages. When a processor's output buffer fills, the processor first sorts it on the attribute that is to be used in the subsequent operation (if any) and then outputs it to an empty CCD device. In the event that the number of processors is smaller than the number of pages in the outer relation a number of passes is used.

Projections are executed in a similar manner. The outer and inner relations are one and the same. Rather than joining two pages each processor searches for duplicates and eliminates them as encountered (see Section 3.3.3 for a more precise description of the algorithm).

Aggregates are executed locally by the processors. The partial results are sent to the back-end controller, or alternatively to another processor via the cache, for final tallying. Aggregate functions are also executed locally. Each processor keeps aggregate information about every partition it sees. If the output buffer fills (i.e. the number of partitions is very large), the processor writes the buffer out and begins collecting the information anew for the remaining data. After this has been done for the entire relation, a second phase of the algorithm begins, in which the outputs of the processors are merged. A logical binary tree is used for the merging.

DeWitt [41] discusses the problem of relation fragmentation. When parallelism is used in the execution of the various relational operators, each processor may output a partially filled buffer for use in subsequent operations. The number of processors allocated to the succeeding operation will depend, to a degree, on the number of pages to be operated on. Presumably the page size is chosen so that the page can be efficiently operated on while a high degree of concurrency in processing the relation can be achieved. If at the end of an operation, the output relation consists of approximately the same number of pages as did the input relation, and each of the pages is only partially filled, then in following operations the machine resources will not be well utilized. To illustrate this point consider the following example.

Suppose that two relations are to be restricted and the results are to be joined with each other. Let one relation have 20 pages and the other 10. Assume that 20 processors are used and that the selection operations leave two temporary relations with 20 and 10 pages respectively. Let the average "fullness" of each page be .1 of its capacity. In this case, 20 processors will be allocated to the

join. Clearly, the amount of processing to be done by each of the 20 processors executing the join is minimal. However, the number of control messages, and the number of CCD memory devices used to execute this join will be the same as if the original relations were to be joined.

The use of a compression operator to be applied to one of the relations (the inner one) is proposed in [41]. Expressions are developed for picking the optimal number of processors to execute this operator. Also, an argument for compressing only the inner relation is presented.

2.4. Hybrid Machines

2.4.1. DBC

DBC [9,44] is the first database machine designed to incorporate both on-the-disk and off-the-disk processing capabilities. It was specifically designed to support very large databases (on the order of 10^{10} bytes). It consists of seven functionally different components. Of particular interest are the Mass Memory and Structure Memory components. The mass memory uses several moving head disks, with parallel readout capabilities, to store the database. The heads of the disks are connected, via a switch, to a number of processors which perform search operations. The structure memory is to be constructed out of one of the new memory technologies, such MBMs, CCDs, or EBAMs, and is used to hold an index. The index is different from indices used in conventional DBMSs in that it specifies cylinder addresses for predicates on relations. It is thus expected to be considerably smaller than typical indices (about 1% of the database size, perhaps less). In order to facilitate the use of the index, frequently accessed data, about which the index information exists, is clustered in as few cylinders as possible. The structure memory, then, is used to reduce the data space to be searched by the mass memory.

A query is sent to the controlling processor and passes through a number of stages which re-organize it in a form executable by the structure processor. The structure processor issues search queries to be executed by the mass memory. The output from the mass memory passes through a security filter and from there to a post processing unit for performance of the complex operations. Presently, the architecture of the post processing unit consists of a number of processors interconnected by a uni-directional ring with a single controlling processor that has a communication line to each processor [45]. In executing a complex operation, each processor receives a block of data and communicates some information about the data to the controller. The controller collates the information from all the processors, decides on the communication patterns between the processors necessary to execute the operation,

and notifies the processors. Communication between the processors (for data exchanges) is through the ring. One problem with all the algorithms used by the post processing unit is that it is assumed that the data to be operated on will fit in the memories of all the processors.

2.4.2. HYPERTREE

The HYPERTREE [46] machine is another hybrid architecture and is particularly interesting because of the way it was designed. Examination of existing database machines led to their classification according to the strategy used to interconnect their various components. The result was two classes: machines that used a simple one-to-one interconnection between processors and memories, such as RAP; and machines that used a complex many-to-many strategy, such as DIRECT. Each type of machine executed some operations efficiently. It was thus concluded that a database machine should possess both kinds of communication capabilities.

The performance of a number of different interconnection strategies was examined for the execution of the duplicate removal operation [47]. The various strategies were characterized in terms of their merits and demerits. An augmented physical binary tree structure was picked as the best. In this scheme, processors are organized as a binary tree, but with each node connected in some regular manner to one of its siblings. Leaf nodes are interconnected using the perfect shuffle structure [48] and are connected to disks (either fixed head disks or parallel read out moving head disks can be used). The leaf nodes are responsible for the execution of the simple search operations. Thus, they act as data filters to the higher level nodes which are responsible for executing the complex operations.

Joins are implemented using a parallelized version of the CAFS hashing algorithm. Each processor constructs its own bit vector in parallel with the others. The bit vector is assumed to contain about twice as many entries as there are unique values in the joining attribute. Parent nodes in the tree are responsible for ORing the bit vectors that their children produced. This procedure is executed on both relations. One node, presumably the root of the tree, receives the final two vectors, representing the two relations, and ANDs them to form a new vector. A copy of the new table is sent to each leaf node which rehashes its portion of the data to see if it has any tuples that might participate in the join. In the event that such tuples are found, they are sent to a prespecified processor where they are actually joined. The prespecified processors are picked in such a way that tuples that hash to the same value will be sent to the same processor.

Projections (the duplicate elimination part) are implemented using the binary tree or the perfect shuffle connections. Execution of aggregate operations has not been specified yet.

2.5. Custom Designed SISD Processors

In this section we describe some features of the IDM database machine which is commercially available from Britton-Lee Inc. Details about the IDM are very sketchy, making this discussion somewhat unreliable. IDM was designed to handle relational data management functions for the "mid range" user [16]. It can store databases of up to 32 Gbytes but is expected to be used for databases whose size is 100 Mbytes to 1 Gbytes. IDM can accommodate a transaction rate of between 100 and 1000 per minute.

One important component of the hardware is a custom designed pipelined processor that operates at 10 MIPs called the database accelerator. The accelerator was designed to execute specific data management subroutines. The major reason for designing this specialized piece of hardware was the observation that most of the execution time of a relational DBMS is typically spent in a very small portion of its code [16]. The IDM also uses information about the behavior of previously executed queries to cache frequently accessed data.

2.6. Performance Evaluation of Database Machines

As is the case with almost any area of research, data must be generated and evaluated before an underlying theory to explain various phenomena can emerge. Until recently database machine researchers concentrated on the design of different architectures. Only after a number of designs had been completed could researchers begin to evaluate their work and attempt to form a theory. In this section we present the results of two recent performance evaluation studies. While these studies have some interesting results, these are not sufficient to serve as an empirical base for a theory. In Chapter 6 we present some thoughts concerning the extension of these studies in a manner that will help researchers develop the needed theoretical base.

2.6.1. Comparison of RAP to a Uni-processor DBMS

An analytic performance comparison between RAP and an hypothetical uni-processor DBMS is described in [15]. A number of assumptions, some favorable to one design, and others to the other, are made. In particular, it is assumed that: the database is stored on a fixed head disk; the uni-processor DBMS has indices on every attribute; and, the result set is always at most 3% of the number of tuples in the relation. The second assumption clearly favors the conventional DBMS. The third assumption, on the other

hand, favors RAP, since it places an artificial constraint on the number of tuples to be retrieved. As we shall see in Section 4.3 RAP-like database machines perform poorly if a large number of tuples is to be retrieved.

It is shown that for the particular data used, RAP was between 3 and 60 times as fast as the DBMS for selection operations. When comparing the execution of updates, the performance differential was found to be even more dramatic, RAP is up to 5,000 times faster. This is due to the extra effort associated with updating the indices by the conventional DBMS. In general, the performance improvement grows as the number of records to be retrieved or updated is increased. For a join, with or without projection, RAP performs at approximately the same level as the DBMS does. The primary factor in determining the performance is the number obtained by dividing the number of unique values being joined into the total number of tuples in the relation. As this fraction decreases, RAP's performance degrades when compared to that of the DBMS. Thus, for a join on two relation keys, RAP will perform well; whereas for a join on two non-key attributes it will perform at about the same level or worse than the conventional DBMS.

The comparison clearly shows RAP's performance to be superior, except for the join operator. However, this study leaves several unanswered problems and questions. First, the assumed index includes an inverted list for each attribute. While this may seem favorable to the conventional DBMS, it is not. Generally, retrievals in a DBMS will be expressed in terms of a small number of the attributes so the additional inverted lists will seldom be used. However, the time to update a relation increases considerably in the presence of inverted lists, since each list must be updated and reorganized. We conjecture that decreasing the number of inverted lists will reduce the ratio of performance for updates significantly but have very little effect on the performance ratio for retrievals. Second, the number of qualifying tuples should be allowed to be higher than 3% of the number of tuples in the relation. Third, other operations, such as aggregates should have been studied. Fourth, each query was made up of a complex boolean expression that included several predicates. Such expressions are rarely used and can be handled more easily by RAP than by a conventional system. Finally, the cost of actually performing the join in RAP, i.e. collecting the marked tuples from both relations and constructing the result relation, should be incorporated into the cost of the join.

While it is clear that incorporating the above changes into the study will alter the results in favor of the DBMS,

we believe that RAP will still outperform the DBMS in the execution of selections and updates, but not for joins and duplicate elimination.

2.6.2. A Comparative Study of Database Machines

In [3] the performance of six database machine designs and the relational DBMS INGRES [43] is compared for a number of benchmark query programs. Since this is the first attempt at a comparative performance evaluation of database machines, we present a detailed discussion. Performance in this study was measured in the amount of work, expressed in units of time, each machine required to execute each query.

2.6.2.1. Physical Characteristics

The database machines covered were: Slotnick-type associative disks (i.e. very limited processing capabilities) which we shall refer to as AD, CAFS, CASSM, RAP (as described in [24]), DBC, and DIRECT. The architectures of all of these have been described in Sections 2.2-2.4. In order to avoid an "apples and oranges" comparison some of the architectures have been modified. Specifically, moving head disks were assumed to be the mass storage devices (adversely affecting the performance of AD and CASSM). A processing rate of 2-3 MIPs was derived as necessary for operating at the disk speed (based on IBM 3330 disk specifications). The number of processors used was 8 for DIRECT, 16 for RAP, and 19 for the other machines. Other parameters of importance were the host overhead and data processing costs, which were based on measurements of INGRES [10] and communication costs between the various machine components.

A number of other assumptions about the architectures and query processing on them have been made. An example is the processing of joins and aggregates on DBC. At the time that [3] was written, the specification of these operations was very sketchy. Thus, assumptions that at this time we know to be untrue, were made. Currently, DBC executes complex operations in the post processing unit part of the machine. The architecture of the post processing unit has been specified and the join algorithm has been described. However, this description lacks a number of crucial details. Also, specification of the projection and aggregate algorithms are still due. It should thus be recognized that the architectures evaluated in [3] were, and to a large degree still are, "moving targets".

2.6.2.2. Operational Characteristics

The performance of three benchmark programs was compared for all the machines and for a fast version of

INGRES.[1] The three queries benchmarked belong to the three classes of relational queries described in [10]. These are: overhead-intensive, for which the DBMS spends most of its time executing code not directly associated with the data; data-intensive, for which the DBMS has to scan large amounts of data; and, multi-relation which includes a multi-relation operations such as the join. Multi-relation queries are generally also data-intensive. The overhead intensive query is a simple selection. The data intensive query is an aggregate function.

An existing database was used for the performance measures of INGRES. For each of the queries, a number of observations were made that enabled evaluation of the performance of the queries on the database machines. These included: the number of result tuples, the number of pages in the relation(s) queried, and the distribution of the result data in the relation. For each query, best case and worst case equations were developed to describe the total amount of work and the expected response time on each machine. The definition of best case (and worst case) was different for each machine. For example, for INGRES best case meant that the relation was hashed (or indexed) and

---

[1] Currently INGRES is an interpreted system, and as such is quite slow. It is hypothesized that executing precompiled queries will lead to a decrease in the execution time of at least one half. Thus, the performance measurements used are one half of what was actually observed on the version of INGRES used at the time.

the the tuples to be retrieved resided on as few pages as possible, thus minimizing the amount of I/O to be performed. For RAP and DIRECT, on the other hand, best case means that the data to be scanned was resident in the cache at the time the instruction was initiated. This could happen either because the data was used in a previous instruction (or a concurrent instruction for DIRECT); or because a smart prefetching algorithm brought it in from the mass storage devices in anticipation of its use. The total amount of work was defined as the sum of the time spent in all the components of the machine. The response time was defined as the sum of all the component times that could not be overlapped.

2.6.2.3. Results

Because of the size of the benchmarked database only limited conclusions can be made about the relative performance of the machines (see our criticism below). For example, in no case was the difference in performance as much as an order of magnitude. It is our belief that had a "large" database (larger than 1 Gbyte) been used the results would have been more dramatic (and therefore more useful).

2.6.2.3.1. Overhead-Intensive Query

The simple selection query was applied to a relation of size 137 Kbytes. The results of the selection benchmark showed that AD, CAFS, and DBC had the best performance. RAP and DIRECT had the worst performance because of the need to bring the data into their caches before it could be scanned. INGRES and CASSM were about half way in between. In the event that the data already resided in the cache RAP and DIRECT had the best performance. These results show that, for the data examined, a conventional DBMS, or an AD device, best serve the needs of the selection operation. It should be noted that because of the small relation size only one cylinder was required for its storage, thus the use of moving head disks for the mass storage devices did not adversely affect the performance of AD or CASSM.

2.6.2.3.2. Multi-Relation Data-Intensive Query

This query included a selection operation on one of the relations and a join of the result of the selection with another relation. The join required performing two comparisons on each pair of tuples (i.e. two attributes had to match). The selection was to be applied to the smaller of the two relations which consisted of about 15 Kbytes. The other relation was considerably larger and had 1.43 Mbytes. Various auxiliary storage structures were assumed to exist for INGRES.

The results of this test are quite interesting and somewhat counter-intuitive. First, the best and worst times were very close for all the machines, except for INGRES and RAP. Second, as expected, DIRECT showed the best performance (for both cases). Third, AD which used moving head disks, and is the simplest associative disk type database machine (in the sense described in Section 2.2) exhibited the second best performance. Finally, both CAFS and CASSM, which use hashing to implement joins performed worse than all the other machines (except INGRES).

The poor performance of CAFS and CASSM was attributed to the fact that both machines could not perform the actual join of the tuples until the marking of candidate tuples in both relations was completed. AD, RAP, and DBC each implemented the join by issuing separate selection sub-queries on the second relation using the values obtained from the selection on the first relation. It was assumed that while one sub-query was executed by the processors, the controller would be collating the results of the previous sub-query and materializing the result tuples. However, RAP performed worse than the other two machines because it was sensitive to the second relation fitting in the cache. Since the second relation in this query did not fit into the cache RAP's performance suffered. It should be noted, though, that the equations developed for this query (and

the others) did not take into account the problem of bus contention between the processors. As will be shown in Section 4.3, contention has a significant impact on the performance of various associative disks.

The high performance level attained by AD and DBC is primarily attributed to the ability to overlap the execution of the searches with that of the actual join. DIRECT performed well because of its ability to use the cache efficiently (i.e. it is not necessary to fit the entire second relation into it).

2.6.2.3.3. Data-Intensive Query

The final query benchmarked contained an aggregate function applied to a relation of size 97 Kbytes. This particular query divided the relation into 17 distinct partitions. AD, CAFS, RAP, and DBC do not directly support the implementation of aggregates. Also the processing elements in these systems are not capable of performing arithmetic operations. Thus the algorithm used, was to retrieve the values on which the relation is to be partitioned into the controller, remove duplicates, and issue a number (in this case 17) of selection sub-queries. The results of each selection are tallied up by the controller.

Both the CASSM and DIRECT processing elements are capable of performing arithmetic operations. The DIRECT processors also have some temporary memory. Thus each processor keeps a running value for each partition for all the tuples it sees.

The results of the benchmarking of this query show that CASSM and DIRECT executed the query efficiently, primarily because of the arithmetic capabilities of their processing elements. DIRECT's performance was much better, though, because of its ability to completely parcel out the work among the processors. Each processor could proceed with the execution of the code independently of the others. Therefore a high degree of parallelism was attained. INGRES' performance did not significantly lag behind those of the other database machines.

2.6.2.3.4. Summary

Hawthorn and DeWitt's results are interesting and important for a number of reasons. First, it is shown that for databases with a large number of overhead-intensive queries a conventional uni-processor DBMS is probably the most cost effective. This may imply that a database machine like IDM is the best choice in this case. Second, for data-intensive queries, the additional complexity of some of the database machines was shown to be very helpful in attaining a good performance behavior.

However, there are a number of problems with the study. First, we note that the database used for the benchmarking was unrealistically small (12.3 Mbytes). Thus, the performance of the various machines under adverse load conditions was not measured. We conjecture that studying the sensitivity of the machines to variations in the database size will lead to a more precise characterization of the various machine performances and capabilities (see Section 4.2.4.4). For example, the importance of the index processor in DBC can be assessed. Also, the effect of the size of the cache for both RAP and DIRECT should be considered (in studying the behavior of RAP for the second query, it was shown that the size of the cache was a significant limiting factor in the possible execution improvement). Second, the mass storage devices were old. Similar equations should be developed for newer disks (such as the IBM 3380) [31] which have a much higher storage capacity and data transfer rates. Third, the performance of various components of the machines was not modeled. As will be shown in Section 4.2.4.5, the back-end controller in DIRECT is a bottleneck under certain conditions. Also, as will be shown in Section 4.3, the output bus of the various associative disks considerably slows down their execution time. Finally, it is not clear whether the values used for the host and controller overhead were accurate. We feel that the cost of controlling the various machines should be studied more closely.

The authors point out that although INGRES performed relatively well in the execution of the first query (the simple selection), associative disk type machines (AD, CAFS, RAP, and CASSM) did better. On the other hand, it was shown that data intensive queries require additional hardware in order to perform reasonably. Thus, Hawthorn and DeWitt [3] conclude by arguing that future database machines should possess both on-the-disk and off-the-disk processing capabilities, if they are to be used to enhance the performance of all operations in a relational DBMS.

2.7. Conclusion

In this chapter we have surveyed extensively the database machine literature. We considered numerous architectures, and classified them into four groups according to hardware organizations. In describing specific machines within each category, we provided an overview of the hardware organization as well as a description of the algorithms used to implement the various high-level relational algebra operators. We have also discussed the query processing strategies used by each machine-type.

We have also surveyed two comparative performance evaluation studies. These are the only studies of this kind in this field. The first study [15] compared the per-

formance of RAP, an associative disk-type database machine with that of an hypothetical uni-processor DBMS. The purpose of the study was to show that database machines that employ parallelism can provide a much higher performance level than can be attained by DBMSs running on conventional hardware. The second performance evaluation study presented [3], compared the performance of several database machine designs for three benchmarks. It was shown that no one machine was best. A number of useful features in database machines were identified and it was recommended that future database machines possess these features.

CHAPTER 3

METHOD AND ALGORITHMS

In this chapter we describe the method which we used to design our proposed database machine. We begin with a criticism of past database machine research in order to highlight the lack of methodology in previous design efforts. We then describe the goals of this thesis and the methods employed to attain it. The remainder of the chapter contains the description, and analysis, of the parallel algorithms to be employed in the architecture, and a summary of the features that the hardware must provide in order to implement the algorithms efficiently.

3.1. Criticism of Database Machine Research

In Chapter 2 we described a large number of database machine designs and classified them into four distinct groups. Of these, three groups employ some form of parallelism to speed up the execution of queries. The fourth group is of no interest to this research effort and was included for the sake of completeness. One reason for grouping the architectures in the manner done was to simplify their descriptions. Another, more important, reason is an attempt, in this section, to show that research leading to the design of most of the machines has been "archi-

tecture directed". By "architecture directed" research we mean that the architecture is developed without much consideration for some, perhaps even the majority, of the operations to be executed on it. Only after the architecture is specified, algorithms for the operations it must support are developed, using the available primitives.

A case in point is Slotnick's logic-per-track devices [13] from which all the associative disk database machines and, to some extent, DBC are derived. The basic design goal of Slotnick's organization was the efficient execution of the selection operation to select records which satisfy a certain criterion. This capability can be used to reduce the amount of data that need be transferred between the mass storage device and the computer requiring the data. It can also be used to reduce the amount of machine time required to process the data.

Once the concept of database machines emerged, a number of designs (most notably CASSM and RAP) used Slotnick's idea as the basic building block in the implementation of various operations. Other designs soon followed with various enhancements. For example, the data layout in RARES is organized to minimize contention for the output bus among the processors; and DBC uses moving head disks instead of the almost obsolete fixed head disks, in conjunction with indexing to reduce the number of cylinders to be searched. In fact, an entire class of the machines (which we have termed associative disks) has appeared which combines the processing capabilities of a controlling processor with those of the processing elements on the disk to implement a number of high-level operations.

As has been shown in Section 2.2, a number of the associative disks are intended to act as full fledged database machines supporting all the high-level operations required (generally, relational algebra operations). However, some machines, such as RARES, VERSO, and SURE, were designed to support only a subset of these operations (generally selections) and could function in conjunction with a conventional DBMS or within a multi-processor database machine.

An examination of the current literature yields some interesting observations. One of these is that the architectures of the machines in those projects that are still active (e.g. RAP and DBC) are significantly different from the original design specifications. This is due, in part, to analyses of the machines (or machine components) that revealed flaws which could be corrected (for example, the incorporation of buffers in the cell processors of both RAP and DBC). However, it is also due to the realization by the machine designers that in order to efficiently support some operations, the architectures had to be changed.

For example, the latest version of RAP [49] uses general-purpose micro-processors rather than specially designed logic to implement the cell processor. This enables the processing elements to process scalar aggregates in place instead of sending every tuple to the controller, as in the original design. Similarly, the architecture of the post processing unit in DBC, which is responsible for the execution of sorts, joins, and aggregates, has undergone several revisions [45,50,51].

DIRECT is another product of an "architecture directed" research. It was designed to operate on data off the disk because of the observation that joins could not be implemented efficiently on associative disk type database machines. Thus, certain architectural features (particularly the broadcasting cross point switch) were included in the machine design to facilitate efficient repeated scans of data. A beneficial side effect of the switch is that algorithms for other complex operations can also be implemented efficiently and with relatively little difficulty. This is the reason why re-designs of DIRECT have not been necessary. However, DIRECT is not a successful database machine design because it performs poorly for selections and scalar aggregates.

In designing the HYPERTREE machine Goodman [47,46] studied the operations of duplicate removal and join in detail. A number of interconnection schemes were considered, and each was characterized in terms of its advantages and disadvantages under various conditions. It was shown that the HYPERTREE structure had the most desirable features of those present in the other schemes with the least number of faults. This research represents a step in the direction that we advocate in this thesis. However, specification of algorithms for aggregates is still needed. It should be noted that the HYPERTREE machine was not included in Hawthorn and DeWitt's performance evaluation (presumably because of its recency). It is therefore inappropriate (and difficult) to make any statements about its expected performance relative to those machines compared.

We believe that the reason [3] concluded that there was no "best" database machine design is the same reason that a number of database machines require repeated redesigns, or perform poorly in some cases: lack of methodology in their design. We have so far shown that the research that led to a large number of original designs was "architecture directed" rather than "algorithm directed". By "algorithm directed" research we mean that algorithms for <u>all</u> of the high-level operations to be executed by the machine must be developed in the first stage of the research. Using these algorithms a specification for low-level hardware primitive operations that would support the efficient implementation of all of these algorithms can be generated. Only at this point should the researcher begin the design of the machine that would provide these primitives.

"Algorithm directed" computer architecture research cannot be used by all machine designers. In many cases there is very little information about the types of programs that will be run on the machine, or the range of types of these programs is so large so as to cause the machine design to be general-purpose. This, however, is not the case for relational database machines. Both the data structure and the types of operations are well-defined. Furthermore, the number of different operations that the machine must support is quite small (about ten). As a matter of fact, because this number is so small, not only should the machine designer consider the algorithms used for each operation; he should also study the structure of programs to see if any information about patterns of access to the data can be used to further tailor the design to meet the user needs.

So far we have concentrated on describing the non-architectural factors that must be considered prior to the machine design. Additional factors that affect the ultimate design include:

(1) Implementation considerations: Can existing technology be used? Is the design amenable to implementation

using current techniques (e.g. VLSI)?

(2) Expansibility issues: Can the machine be easily expanded according to the changing user needs?

(3) Bottlenecks: Is any one particular component of the architecture likely to become a bottleneck under either normal or abnormal operating conditions?

(4) Robustness: How crucial to the operation of the machine is each component?

In examining the various machine proposals in light of the above and other criteria, we find that several of the machines suffer from a number of serious flaws. For example [42], has shown that the controller in DIRECT does indeed become a bottleneck (see Section 4.2). It is also shown that similar problems will occur for most of the associative disk designs. Other problems with associative disks (bus contention) are described in [52] and [53] (see Section 4.3).

We close this section with the observation that of the large number of machines described in Chapter 2, only two: IDM and CAFS, are commercially available. We call the reader's attention to the fact that IDM uses no interprocessor parallelism, and CAFS employs some form of pipelining (although it has the capability of using a parallel readout disk). Of the remaining machines only DIRECT is actually running and supports all of the relational algebra operations.

3.2. Goals of Research

In this dissertation we would like to develop the design of a database machine using an "algorithmic approach". That is, we shall use the study of the algorithms to be employed and data access patterns of programs to specify a machine architecture. We shall also apply other, architectural, considerations, but these will be used in the later stages of the design. Our intention is to design as complete a database machine as possible. However, it is clear that the end product will be lacking in several respects due to the enormity of the task at hand and the relatively short time available. As a result of our interest in the study of the algorithms to be used, and our advocation of the algorithmic approach to the machine design, our treatment of the more hardware oriented and lower-level design decisions is incomplete.

Our intent is to design a database machine that can be constructed from existing hardware devices and support a large number of users. Because the execution of queries in SIMD mode only permits intra-instruction parallelism, we believe that SIMD-type machines cannot support a high volume of transactions. We have thus decided to organize the architecture so that it can operate in MIMD mode, where inter-instruction and inter-program parallelism can be used to enhance the performance of query programs in addition to intra-instruction parallelism.

The organization of the remainder of this chapter, and the next two chapters, reflects the design method described above. We begin with a description of the algorithms to be used. The algorithms, and their analyses, are drawn from [54]. In this paper several algorithms were presented for each relational algebra operator. Generally, for each operator there was one algorithm that used parallel sorting, and another that used broadcasting of pages from an unsorted relation. This implies that one could design a database machine with parallel sorting as a basic operation to be used in the implementation of algorithms for the various operators. Alternatively, a machine that provides an efficient broadcast capability can be developed.

The alternative algorithms for each operator are compared in [54]. The results of this comparison are inconclusive in the sense that neither type of algorithms proved to be better under all conditions. Therefore, we feel that it is reasonable to choose one class of algorithms and proceed from that point. In this thesis we have chosen the broadcasting approach. Our primary reason is simplicity of the control function. All three parallel sorting algorithms presented in [54] are quite complicated to control. The controller must maintain a large number of tables which are used to coordinate movements of pages between proces-

sors according to some ordering rules. Such algorithms require a large number of messages, and synchronization among the processors executing each operation. The broadcasting algorithms, on the other hand, can be implemented with a small number of messages. Also, in parallel with this effort, Friedland [55] is investigating the use of parallel sorting in database machines, particularly the cost of controlling the algorithms.

There are two other known classes of algorithms for database management: those that use hashing and those that use indexing. Parallel algorithms that use hashing have been developed for the join operation by Goodman [46] for the HYPERTREE machine. The Ohio State database machine group has been investigating the use of indexing in database machines. However, their study has been directed only towards their use for the selection operation. Additional work on the use of parallel index operations, particularly for the implementation of joins, has been done by Goodman [46]. It is expected that after further study of the other three classes of parallel algorithms for database management, a comprehensive comparative evaluation can be undertaken.

We continue the design process by examining data access patterns of representative query programs in Chapter 4. Hawthorn and DeWitt [3] have shown that queries should be processed directly on the disk if they contain operations that can be processed using a single scan of a relation by a uni-processor. Otherwise the queries should be processed off the disk. In Chapter 4, therefore, we examine problems associated with processing queries in these two types of organizations.

Since DIRECT is the only off-the-disk database machine about which sufficient detail has been published and with which we have had some implementation experiences we begin with an examination of processor allocation strategies for it. We use this study to gain some insight into the data access patterns exhibited by query programs using the algorithms described in the previous chapter. This information can be used to select an appropriate interconnection mechanism between the processors.

Next we compare three associative disk designs. None of the designs we examine exists - all of them are abstractions of associative disk design types representative of those described in Chapter 2. The purpose of this study is to quantify the relative performances of the three design types. Once such information is available it can be used, along with implementation and cost considerations, to select the appropriate associative disk to be used in the architecture.

3.3. Algorithms

The algorithms described in this section are for the relational algebra operators as supported by INGRES [43]. The operators covered are: select, project, join, aggregates (count, sum, average, etc.), aggregate functions, append, delete, and modify (see Appendix A for their description).

The algorithms presented below rely on a number of general points described below. First, relations are organized as collections of fixed size pages. The page size should be large enough so that it constitutes an efficient unit of transfer among the machine components, but at the same time it should be small enough so that a large number of processors, each examining a few (possibly one) pages, can participate in the execution of the operation. Second, each page in a given relation is sorted on some attribute (or group of attributes). That attribute should be the key, in the case that the relation is a permanent relation in the database; the attribute used in the subsequent operation if the relation is a temporary relation; or the entire tuple in some special cases. Therefore, it is the responsibility of each algorithm to sort individual pages (but not the entire relation) before they are output.

Third, the existence of a controlling processor which is the only processor with access to control information, such as page tables, is assumed. Finally, explicit messages are exchanged between the controller and processor for each "basic operation" (i.e. read a page and search it).

These same assumptions are also made in [54]. However, the presentation and analysis of the algorithms in [54] is based on a number of architectural specifications which we outline below. It is important to understand that these same algorithms can be implemented on <u>any</u> architecture. The same parameters would be used to characterize the algorithms. What would change from one architecture to another are the <u>values</u> (representing implementation costs) assigned to each parameter. In our presentation we have attempted to remove as many of these assumptions as possible.

The first architectural assumption made in [54] is that a three level memory hierarchy is used. The middle, cache, level[1] consists of several memory elements which are connected to a number of processors with an interconnection device whose capabilities are similar to DIRECT's cross point switch. That is, two processors can read (write) two separate memory elements simultaneously and, any number of processors can read the contents of a single memory

[1] The collection of the memories in all the processors forms the top level in the hierarchy, while the mass storage devices form the bottom level.

element. It should be noted though, that there are a number of ways of providing such a service, and that [54] does not concern itself with its implementation.

It is also assumed that there are no processor-to-processor interconnections, that is, all interprocessor communication is via the shared cache. Using this model, highly parameterized, precise equations, characterizing the performance of several algorithms for each operator are developed. In this presentation we describe some of the algorithms from [54] and analyze them. Our analysis is different, though, because it does not assume the parallel read capability granted by the cache organization and interconnection device. Rather, our formulas include an "I/O parallelism characteristic". This is a symbol that represents the amount of parallelism that an architecture allows in processing I/O operations. A value of 1 indicates that all I/O operations can proceed in parallel; whereas a value of p, the number of processors executing the operation, indicates that no parallelism is allowed.

We begin with a definition of the parameters used in the formulas. We then describe the update algorithms. We show that these operations maintain the sort order of individual pages. This is followed by the various retrieval algorithms (which cannot change the order of the tuples in the permanent relation pages). Throughout this discussion we have assumed the syntax and semantics of QUEL [56]. It should be noted that all of the algorithms operate with any number of processors for all relation sizes.

The parameters used in [54] measure three types of costs: I/O, processing, and communication. A number of basic tasks, common to all the algorithms (e.g. reading a page), have been identified. The execution time formula for each algorithm is expressed in terms of the costs of these basic steps. For different architectures, the parameters may have different values and may relate differently to each other; for example, the I/O cost may be more significant than the processing cost for some architectures, but not for others.

As indicated above data is moved and processed by page units. We assume that a full page contains k tuples; C is the cost of a simple operation such as comparing two attributes or performing an addition; and the cost of moving a tuple inside a page, is V time units. The I/O parallelism characteristic discussed above is indicated by P. We have chosen to represent fixed costs by capital letters. Other parameters (for example, the number of pages to be read) are represented by lower case letters. The basic tasks used in evaluating the performance of the algorithms are:

(i) I/O cost: A read request moves a page into a processor's memory. We denote its cost by C_r. The actual

value assigned to this parameter depends on the machine organization employed. In [54] a memory hierarchy is assumed. Thus the cost of a processor reading a page is made up of two components: the probability that the page is in the cache multiplied by the cost of reading from the cache; and the probability that the page is not in the cache multiplied by the cost of reading from the mass storage device. The cost of writing a page is denoted by C_w. A value can be assigned to this parameter in a similar manner to the read cost assignment.

(ii). Broadcast cost: This includes both the cost of sending the page and receiving it and is denoted by C_b.

(iii). Scan cost: If a page is to be scanned, the scan is implemented using a binary search since we expect that each page in the input to an operation is individually sorted on the attribute(s) used by the search.[2] The number of tuples in the page is assumed to be k. Thus, the scan cost C_{sc} is computed as:

$$C_{sc} = \log k * C$$

(iv). Merge cost: If two sorted pages are to be merged the number of tuples in each page is assumed to be k. Since all our operations both require and produce internally sorted pages, both pages will already be sorted. In the

---

2 A non-key selection requires a sequential search.

worst case, the number of comparisons required to perform the merge of two sorted lists of length k is 2k [57]. The number of tuples to be moved is the same. Thus, C_m, the cost of merging two pages is computed as:

$$C_m = 2k * (C + V)$$

(v). <u>Page re-organization cost</u>: There are two cases when a page must be reorganized to keep the tuples in sorted order. The first case occurs after the application of an update operation which modifies the attribute on which the page is sorted. We assume that the re-organization consists of both tuple comparisons and movements and expect that, on the average, half of the tuples in the page will be affected. As before, a page is assumed to have k tuples. We compute C_o, the re-organization cost as follows:

$$C_o = (k * (C + V)) / 2$$

The second case occurs when a buffer containing new tuples (e.g. the result of a projection or a page of an intermediate relation) is to be used in a subsequent operation. Since all our operations require internally sorted pages, the page must be sorted before it is written by the processor. We assume that the new page has k tuples (though in many cases this number may be smaller) and that, on the average, internal sorting of a page would require k log k comparisons and moves.[3] Thus, C_{so}, the cost to internally sort a page is:

$$C_{so} = k \log k * (C + V)$$

(vi). <u>Communication cost</u>: Since transfers of pages between processors are considered as I/O operations, only page request, reply, and assignment messages from the controlling processor to the other processors are considered for the cost of communication. When a processor wants to read or to write a page, it sends a request message to the controller specifying the relation name and the page number. The controller replies by sending to the processor the address of the page. Since it is important to include the cost of the request and reply messages in our definition of page read and write operations, we shall replace C_r by C_r+C(request message)+C(reply message). Therefore, the remaining communication cost of an algorithm can be measured by the number of control assignment messages sent. An example of these are messages necessary to allocate processors to an operation. Since the number of control messages is small compared to the number of I/O related messages, and since these messages are short (they contain only a few words of information), we are neglecting them in considering the costs of the algorithms.

---

[3] Actually, this is an upper bound which is seldom reached.

Table 3.1 summarizes the parameters described above.

3.3.1. Updates

Many of the retrieval algorithms presented in the following sections rely on the property that each page is sorted on some attribute or group of attributes. Permanent relation pages are sorted on the relation key. It follows then, that any update algorithm must keep the pages sorted. A second property that must be preserved is that no duplicates are introduced as a result of an update. We show that our algorithms do indeed preserve these properties. We shall also present an analysis of one algorithm's complexity.

We consider three update operations: delete, append, and modify. Each operation specifies a relation to be updated and a qualification clause indicating the tuples to

Table 3.1

Parameter	Description
k	Number of tuples per page
C	Time to compare two attributes
V	Time to move a tuple
C_r	Time to read a page
C_w	Time to write a page
C_{sc}	Time to scan a page
C_m	Time to merge two sorted pages
C_o	Time to reorganize a page
C_{so}	Time to sort a page
C_{msg}	Time to send a control message

be affected.

For example: Delete emp where emp.eno < 153.

However, there may be cases where the qualification criteria for an update operation is more complex than a simple selection. For example, suppose we wanted to delete all employees whose employee number is less than 153 and the department in which they work is not the toy department. In QUEL the query would be expressed as:

```
Delete emp where emp.eno < 153 and
   emp.dno = dept.dno and dept.name != "toy".
```

Here we have to restrict both the employee and department relations according to the selection criteria, perform the join, and then apply the delete operation to the employee relation, using the values produced by the join as the deletion criteria.

We term these two kinds of qualification clauses: simple and complex. A simple qualification is one that may be applied in a single scan of the relation. A complex qualification is one which requires us to perform some inter-relation operation(s), (e.g. join) in order to determine the tuples to be updated. The algorithms presented below handle both simple and complex updates.

For consistency reasons, we assume that updates are atomic operations. That is, an update either successfully terminates, or, in the event of a crash or abort, does not affect the stored database. One reason for aborting update

operations is the introduction of duplicates into a relation.

3.3.1.1. Delete

A deletion operation is, in effect, the negation of a selection. If the qualification is simple, no pre-processing is required. Each processor executing the deletion will examine a unique subset of the relation. Tuples satisfying the deletion criterion are removed from a page and it is compressed and flushed out to the buffer memory. The controller is informed of the size of the written page and stores it as a new page of the relation.

Complex deletes require a pre-processing step to determine the set of tuples to be removed. The set produced is a list of tuples which is broadcast to all the processors that have pages of the source relation. Each processor performs a modified merge of its source page with every broadcast page. The modified merge consists of deleting a tuple from the source relation page, if a key value in a broadcast page matches the tuple's key. As in simple deletes, modified pages are written out as new pages of the relation, replacing the corresponding source pages.

3.3.1.2. Append

A simple append is one in which a small number of tuples are to be appended to a relation. The simple append

begins with the controller deciding where to add the additional tuples, based on the density of the pages in the relation. The processors first search for duplicates of those tuples to be appended. If duplicates are found by any of the processors, the controller is informed, the operation aborted, and the relation restored to its pre-operation state. If no duplicates are found, tuples are added to the pages designated by the controller. A page chosen for appending will have to undergo re-organization to preserve its sort order.

Complex appends are executed in a similar manner to complex deletes. After the list of tuples to be appended has been generated, the processors search for duplicates using the modified merge described above. If the number of new tuples is small they are added to designated pages. Otherwise, the new pages are added to the relation's page table at the end of the operation.

3.3.1.3. Modify

There are two cases to consider for the modify operation. In the case that the modified attribute(s) does not contain the relation key (or part of it) we are assured that no duplicate tuples will be introduced into the relation as a result of the application of this operation. In this case, each processor executes the same code as the simple delete, applying the modification to matching tuples rather than deleting them. The same holds for a complex, non-key modify. Note that no page re-organization is required since the page is sorted on the relation key which is un-affected.

In the case that the query modifies some part of the key, the algorithm must check for duplicates. To do this we must have a list of the new key values, and check the source relation for duplicates using this list, before we apply the update. Our algorithm works in a similar manner to the algorithm for non-key modifies with one exception. When a tuple to be modified is found, the processor deletes that tuple from the source page, modifies it, and writes the modified tuple into a separate buffer. After all the pages of the relation have been scanned, each page containing modified tuples is sorted on the relation key. The new pages are then broadcast to all processors that contain source relation pages to check for duplicates. As in the other update operations, if duplicates are found the operation is aborted and the user notified. Otherwise, the new pages are added to the source relation page table. Note that in this case it must be ascertained that if duplicates exist among the modified tuples only one of these should be added to the relation. If more than one page of modified tuples exists this may require non-linear time in the number of pages.

As the update algorithms are all quite similar, we shall provide a performance analysis of only one of them. We chose to analyze the simple key modify since it is one of the more complicated algorithms, and it has elements that appear in all the others. The execution time of the simple key modify by p processors is given by the following formula:

$$T_p = (n/p) * (\underbrace{[T_1^1]}_{stage_1} + \underbrace{[T_1^2]}_{stage_2})$$

where:[4]

$$T_1^1 = P*C_r+C_{sc}+C_o+P*C_w+(j/k)*(C_{so}+P*C_w)$$

and

$$T_1^2 = P*C_r+1'*(C_b+C_m)$$

In the first stage each processor examines (n/p) source relation pages, looking for tuples matching the qualification ($C_r + C_{sc}$). We assume that on the average j such tuples exist in each source relation page. Each page containing qualifying tuples needs to be reorganized (C_o) and written out (C_w) after the matching tuples have been modified and moved to the buffer. Finally, the new tuples need to be sorted and written out ((j/k) * ($C_{so} + C_w$)).

---

[4] Note that we assume that in the event that P≠1, the reading of one page by one processor can be overlapped by the execution of "most" of the remainder of the operation by another processor. In fact, if a precise characterization of the time necessary to execute each sub-step shows this to be true, a much smaller number of processors could be used to attain the same performance level.

In the second stage, the processors search for the possible introduction of duplicates into the relation. Let l' denote the number of pages containing modified source tuples. Then each processor reads a page of the source relation and all of the l' pages. The processor performs the modified merge described above. Finally, if no duplicates are found, the l' new pages are added to the source relation page table.

We conclude this section with two observations. First, all the update algorithms operate in linear time. That is, given p processors, each algorithm would be executed by the p processors in n/p "basic" time units.[5] It should be noted that the basic time unit used in the algorithm for one operation may differ from that used by the algorithm for another.

Second, at no time will we experience page overflow problems as a result of the execution of any of these algorithms. This is clear for the delete operation. Both appends and modifies add new tuples to pages that are sparse (the controller is assumed to have such information about every page in the database) or form new pages. Periodic reorganizations of relations may have to be undertaken if too many pages become too sparse.

[5] The one possible case where this may not be true is for the key modify with more than one page of modified tuples and less processors than pages are available for the duplicate removal.

3.3.2. Selection

The algorithm we use for the selection operator is the crudest and the simplest: a scan of the relation. Each page is scanned using a binary search, if it sorted on the attribute being selected. Otherwise a sequential search is employed. It is clear that this algorithm (using the sequential search) can be implemented "on the fly" if the processor(s) executing it can keep up with the disk speed.

3.3.3. Project

The projection of a relation with domains d1,d2,...,dn on a subset of domains di,dj,...,dm requires the execution of two distinct operations. First the source relation must be reduced to a "vertical" sub-relation by discarding all domains other than di,dj,...,dm. Then, since discarding attributes may introduce duplicate tuples, these must be removed in order to produce a proper relation.

We assume that pages have already been reduced to a vertical form by the previous operation and there are no intra-page duplicates. Each processor reads one page. Let a processor be labeled according to the page number of the page it read (that is, the processor that read page i is known as P_i). Starting with P_p, and continuing with $P_{p-1},\ldots,P_2$, each processor, in turn, broadcasts its page and then exits. If processor P_j receives page i, then j<i. P_j compares the two pages and eliminates any duplicates found from <u>its</u> page. Note that P_j will not see page i if i<j. Consequently, it is guaranteed that only one copy of each tuple will remain in the relation (that copy will reside in the highest numbered page of all the pages that had a copy of it).

In the general case when the number of processors (p), is smaller than the number or pages (n), our algorithm works in a number of distinct phases. Each phase produces p projected pages and sees p less pages than the previous phase. In phase i there are (i-1)*p pages that have already been projected, p pages in the processors' memories, and n-(i*p) unprojected pages. Each phase begins with each of the p processors reading a page. Then the n-(i*p) unprojected pages are broadcast to the p processors for duplicate removal. After this step has completed, P_p broadcasts its page and exits. The remaining processors follow suit. The cost of phase i is thus:

$$P*C_r+(n-i*p)*(C_b+C_m)+(p-1)*(C_b+C_m+C_w)+P*C_w$$

If n = p*m, there are m phases and the total cost of the algorithm is:

$$m*P*C_r+m(m-1)p/2*(C_b+C_m)+m(p-1)*(C_b+C_m+C_w)+m*P*C_w$$

This may be rewritten as:

$$(n/p)PC_r+(n^2/2p+n/2-n/p)*(C_b+C_m)+(n-n/p+(n/p)P)*C_w$$

which is of the order of $n^2/2p$ operations. Note that if n is not an exact multiple of p, the last phase would use only n mod p processors and thus terminate faster.

A number of improvements to this algorithm are discussed in [54]. Namely, if a large number of duplicates is expected (this could happen when the key is projected out of the relation) parts of the relation can be compressed to reduce the number of pages in the relation. In particular, this can be done at the time that the p processors read their individual pages at the beginning of each phase.

3.3.4. Join

Given two relations R and T, the "smaller" relation (i.e. the one with fewer pages) is chosen as the inner relation, and the larger (say R) becomes the outer relation. The first step is for the processors to each read a different page of the outer relation. Next, all pages of the inner relation, T, are sequentially broadcast to the processors. As each page of T is received by a processor, it joins the page with its page from R. By joining two pages we mean the following: first the join is performed by merging, then the result page is sorted on the attribute of the subsequent operation (if there is one), and finally the result page is written out.

Let n and m be the sizes, in pages, of the relations R and T, and suppose $n \geq m$. Let p be the number of processors assigned to perform the join of R and T. S is the join selectivity factor and indicates the average number of pages produced by the join of a single page of R with a single page of T. If p = n, the execution time of this algorithm is:

```
T(read a page of R)
+ m*T(broadcast a page of T)
+ m*T(join 2 pages)
```

The number of result pages written depends on the join selectivity factor S defined by:

$$S = size(R\ join\ T)/(m*n)$$

If p<n, the same process must to be repeated n/p times yielding:

$$(n/p)*(P*C_r+m*(C_b+C_m+S*(C_{so}+P*C_w)))$$

In the case that the subsequent operation will use the same attribute the result pages need not be sorted.

3.3.5. Aggregate Operations

To compute a scalar aggregate, a processor maintains two fields: a count field and the aggregate value itself. The count field specifies the number of tuples contributing to the aggregate value and is used in averaging and initialization. When processing aggregate functions, a third field is also required to identify the partition (since a processor may be accumulating aggregate values for more than one partition at the same time). For aggregate functions, we want to account for the space required to maintain these fields ("result tuples") and that is the purpose of parameter 'r' below. In the following discussion, we assume these parameters:


```
n   # of pages in source relation
p   # of processors to process aggregate
m   for agg functions, # of partitions
r   for agg functions, # of result tuples per page
q   # of operations to apply for a simple
    qualification (if query has one), else 0
```

3.3.5.1. Scalar Aggregates

Scalar aggregates may be processed in a single pass over a relation. We use the obvious algorithm. Each processor computes an aggregate value for the pages it sees. When the pages are exhausted, we have p partial results and a single processor must combine them to produce the final value. If the aggregate operator is a "unique" operator, the source relation must first be projected on the agg_att so that duplicate tuples are eliminated. The cost of the algorithm is then:

```
Tsc_agg = T(exec qual)             (if complex qual)
        + T(project)               (if unique agg_op)
        + T(partial results)
        + T(combine p partials)
```

We are concerned with the time needed to produce and combine the partial results since the time required to execute the qualification and project the source relation have been covered previously.

T(partial results) =

$$(n/p)*(P*C_r+(q+1)*C_{sc})+C_{msg}$$

Each processor sees (n/p) pages. To process the page it must read it, apply a qualification to it (if simple) and update the partial result. Thus, each tuple requires a number of comparisons for the qualification plus an additional operation (e.g. add) to process the aggregate. The time to send the partial result is just the cost of a message. The processor which combines the partial results simply reads p messages and performs p arithmetic operations (note, the cost of the message is accounted for by the partial results formula). Thus, T(combine partials) = p*C.

3.3.5.2. Aggregate Functions

In this section we describe the steps necessary for the implementation of an algorithm that uses broadcasting for the processing of aggregate functions. Recall that we must consider two types of qualifications: an src_qual restricts the source relation to which the aggregate operation is applied; and a by_qual restricts the number of partitions. When an aggregate function contains an src_qual, any algorithm for processing the aggregate must begin by determining the set of desired partitions so that any partitions which are removed by applying the src_qual (e.g.

managers with zero counts, above) can be included in the result of the query.

Our algorithm works as follows. We begin by determining the set of desired partitions. If the query contains a by_qual (whether simple or complex), it is applied to the source relation in order to eliminate "unwanted" partitions. Then, the resulting relation (or the source relation if the relation did not contain a by_qual), is projected on the by_list attributes to determine the "names" of the desired partitions. The pages of the source relation are then broadcast to all processors and each processor computes the aggregate value for (m/p) partitions. If the number of partitions is greater than r (the number of result tuples per page), the source relation may have to be broadcast more than once. The cost of this algorithm (assuming no qualifications and non-unique aggregates) may be summarized as:

T(project by_list) + T(process partitions)

While processing partitions each processor sees every page of the source relation (n pages). Each tuple must be placed in the correct partition (depending on the number of passes over the source relation, there are either m/p or r possible partitions) and we assume that the partitions are sorted so a binary search may be used. When the broadcast is complete, the processor must write its result. Let b =

$\lceil (m/r)/p \rceil$ denote the number of complete broadcasts of the source relation. The cost to process partitions is:

$$T(\text{process partitions}) = b(n(C_b + (\log x)C_{sc}) + PC_w)$$
$$\text{where } x = \min(r, m/p)$$

If the query has a simple src_qual, it may be processed the same time as the aggregate is computed. This adds an additional q comparisons per tuple (see parameters defined above). In the event that the src_qual is a complex one an additional step is required to apply it to the source relation before we can begin processing the partitions. If a unique aggregate is specified, we must eliminate duplicates in the agg_att. This can be done along with the projection of the by_list step as follows. Each processor needs an additional buffer. In addition to the projection on the by_list each processor also performs a projection on the attributes specified in the by_list and the agg_att (treating them as a single value) and places the result in the additional buffer. This requires an additional comparison per tuple. In effect, the additional pages generated by this step replace those of the source relation in the partition processing phase of the algorithm.

This suggests an optimization to the original algorithm (regardless of whether the aggregate operation is an unique one or not). In the projection on the by_list phase

a vertical sub-relation of the original source relation can be formed which will contain only those attributes necessary for the remainder of the operation. The motivation of this, is the smaller number of bytes that have to be broadcast in the subsequent phase of the operation.

As can be seen, the performance of this algorithm depends on the complexity of the operation it has to perform. If we consider only the partition processing step then the time required to perform it is linear in the number of pages that need to be broadcast. Since the projection of the by_list can, at best, perform linearly in the number of pages in the source relation, the total cost of this algorithm is dominated by the time to do the projection.

3.3.6. Summary

In the previous sections we have described and analyzed the performance of the algorithms to be used in our machine. We next use these descriptions to specify what low-level primitives the architecture must provide.

In order to perform aggregates (whether scalar or function) each processor must have arithmetic processing capabilities. It must also have string handling operations since the majority of the search related operations deal with character type data. Each processor must also have sufficient memory to hold at least three (four for the optimization suggested for the aggregate function algorithm) pages of a relation at a time.

Each processor should be able to send a message (data page) to any other processor. Furthermore, the machine should be able to support a large number of these operations in parallel (i.e. we want the value of P, the I/O parallelism characteristic, to be as close as possible to 1). Also, each processor must have the capability of broadcasting a message to any number of other processors. Since we expect the machine to be executing a large number of instructions (i.e. modify, aggregate function) at any given time instance, the broadcast of a message should not block communication between processors executing other instructions.

CHAPTER 4

QUERY PROCESSING

4.1. Introduction

The results of [3] (see Section 2.6.2) show that in order to be successful a database machine must possess both "on-the-disk" and "off-the-disk" processing capabilities. DIRECT was the only database machine with off-the-disk capabilities that was examined in [3]. All the other database machines examined processed instructions directly on the disk. However, the machine organization of some of these was considerably different.

As indicated in Section 2.6.2, a large number of simplifying assumptions were made concerning all the architectures examined. In particular, no cost was assigned to the processor and CCD management function of the back-end controller in DIRECT, and the cost of outputting result tuples to the controller in the various on-the-disk machines was ignored. In this chapter we examine these two problems in detail. We begin with a look at processor allocation strategies for DIRECT. We show that one particular strategy is the most sensitive to the data access patterns exhibited by a number of benchmark query programs. We also show that this strategy suffers from a high control overhead. We then compare the performance of three associative disk organizations in order to determine the performance differentials which could be used to pick the most cost effective organization for future use.

4.2. Processor Allocation Strategy Study

In this section we discuss four processor allocation strategies that can be used by DIRECT. (The reader is referred to Section 2.3.2 for an overview and a sample organization of DIRECT.) We then describe a simulation which we used to evaluate them, and its results. This work originally appeared in [42]. The original purpose of this study was to find the most suitable policy for resource management in DIRECT. This problem did not arise in associative disk-type database machines because processors are physically associated with memory elements. In DIRECT, as in any MIMD computer, some policy for allocating processors and memory units to tasks is essential. Once the study was completed, we began investigations of a new architecture, described in the following chapter, which incorporates both on-the-disk and off-the-disk processing capabilities. We were able to use information about access patterns to the database by query programs during the design.

The algorithms used to execute the instructions were the ones described in Section 3.3. However, it should be noted that the query trees were restricted to have only

selection and join operations. As was seen in Section 3.3 the algorithms for the remaining operations (DIRECT has the capability of using any of the algorithms described in [54].) are similar to the algorithms for the selection and join operations.

4.2.1. The Four Strategies

4.2.1.1. SIMD Assignment

One of the original design objectives of DIRECT was to avoid the SIMD nature of previous database machines such as RAP and CASSM. We include an SIMD strategy, however, in order to obtain a measure of the performance differential between it and a number of MIMD strategies. In the SIMD assignment strategy, all processors are assigned to execute the same instruction from a single query simultaneously. When the current instruction terminates, the back-end controller assigns the next instruction from the same query packet to all processors. This continues until the packet has terminated at which point the controller selects the next query packet to execute.

4.2.1.2. Packet-Level Assignment

In this strategy, when the back-end controller decides to execute a query packet, it examines the query packet and attempts to estimate a priori the "optimal" number of processors to assign to it. The estimation heuristic uses the number and size of the source relations referenced by the operations, and the number and type of operators in the packet. Once this value has been computed it remains fixed throughout the execution of the query.

After the back-end controller estimates how many processors should be assigned to a packet, it examines the packet and selects an executable (enabled) instruction. An instruction is enabled when its input relation(s) exist. Clearly, if the query is in a tree format, all leaf nodes are immediately executable. A node higher up in the tree is enabled whenever all of its descendents have finished executing.

Let QPS represent an estimate of the "optimal" number of processors to be assigned to the query packet. If the instruction selected for execution is a selection, then the controller will assign $MIN(|S_i|,QPS)$ processors to the instruction where $|S_i|$ is the number of pages in S_i, the source relation to be restricted. If the operation is a join of relations S_i and S_j, then $MIN(MAX(|S_i|,|S_j|),QPS)$ processors are assigned. Selecting the larger of the two relations S_i and S_j as the outer relation means that, if $MAX(|S_i|,|S_j|) \leq QPS$, each processor will join one page of the outer relation with every page of the inner relation. This approach maximizes the degree of concurrency and hence minimizes execution time.

At this point, if all the processors assigned to the packet have not been utilized, the next executable instruction from the packet is initiated. This continues until either all the processors assigned to the packet are executing some instruction from it, or until no more executable instructions are available (their inputs have not been generated). If there are no more executable instructions, the available processors are placed on an idle list associated with the packet until an instruction is enabled. Idle processors are not assigned to another packet before execution of all the instructions in the current packet is completed.

A packet is initiated even if the number of available processors is less than the optimal number. When processors become free (another packet terminates) they are allocated to the sub-optimal packet. Only when all executing packets are optimal can a new packet be initiated. Thus, at most one sub-optimal packet is executing.

4.2.1.3. Instruction-Level Assignment

For this strategy, scheduling and processor assignment is performed on an instruction by instruction basis. The optimal number of processors assigned to an individual selection or join instruction is limited only by the total number of processors available. If the total number of processors available is MAXQPS, then for a selection

$$QPS = MIN (|S_i| , MAXQPS)$$

and for a join

$$QPS = MIN (MAX (|S_i| , |S_j|) , MAXQPS)$$

When a processor becomes idle, the back-end controller first attempts to assign the processor to any executing instruction which does not have its optimal number of processors. If no sub-optimal instructions exist, the processor is assigned to an enabled instruction from a query packet which is currently being executed. If there are no enabled instructions from the currently executing packets, then a new packet is initiated. If there are no packets awaiting execution, then the processor is placed on an idle list until a new packet arrives from the host or an instruction from an executing packet is enabled.

4.2.1.4. Data-flow Assignment

In this strategy a page of a relation is the basic unit which is used for scheduling decisions. This means that an instruction can be initiated as soon as at least one page of each participating relation exists. Assigning processors to operate based on the availability of pages rather than relations, offers the possibility of having a very flexible processor allocation strategy. Furthermore, it becomes possible to distribute processors across all nodes of the query tree and to pipeline pages of intermediate relations between them. This will reduce page traffic

between the CCD memory and the mass storage device(s) because after a page of an intermediate relation is produced by one processor it will be read by a processor executing the subsequent instruction.

The processing of queries in a data-flow manner is related to the idea of processing relational queries in a pipelined fashion which has been suggested by Smith and Chang [58] and Yao [59]. There are, however, two important differences between the two strategies. In the pipelined strategy, there will be at most one processor executing each node in the tree and therefore the concurrency obtained will be limited by the number of nodes in the query tree. With the data-flow strategy we can have any number of processors executing each node and can dynamically adjust which processors are executing which nodes in the query tree in order to maximize performance. The other major difference is that in the data-flow strategy we never need to wait for one node to completely finish before initiating the subsequent operator as has been suggested is necessary for pipelining [59].

One problem with the data-flow strategy is that at times decisions need to be made based on insufficient information. For example, when a join is initiated it is not known which relation should be the outer relation, if both relations are produced by a previous operation. The solution used is to pick the outer relation based on past statistics of similar operations on the relations in question. Such statistics have been termed selectivity factors and are reported to be used extensively in System R [60]. Although the collection and use of selectivity factors is not a subject of this research we note that various factors can be of help in deciding the cardinality of a result relation. One example is knowledge of whether a key is the attribute being operated on.

In observing a number of runs of earlier versions of the DIRECT simulation we found that for the broadcast algorithms there were two types of I/O operations: point-to-point transfers and broadcasts. The point-to-point transfers were required in the beginning of each operation to load each processor with its data portion. For example, for the join algorithm this meant loading pages from the outer relation into the processors' memories. Once this was done a broadcast phase would begin. For the projection operation this meant broadcasting each processor's page to the others. Thus, the number of I/O operations required by each operation should be linear in the size of the relations.

A problem that became apparent with time was that the actual number of I/O operations was approximately quadratic in the size of the relations for the so called complex

operations (join, projection). We realized that there was a conflict in using broadcasting on the one hand (which introduces artificial synchronization into the execution of queries but reduces the amount of I/O), and data-flow (which ideally should be entirely asynchronous) on the other hand.

Since data-flow is an advanced form of pipelining we compromised on the following pipelined method for processor assignment. Each processor executing an operation that is producing an outer relation was re-assigned to the subsequent instruction when its output buffer filled. This was done instead of flushing the output buffer to temporary storage and continuing the execution of the current operation. Thus the actual number of point-to-point transfers was reduced considerably. In fact, the only time during which such transfers were required was in the execution of leaf nodes in the query tree. The effect of this modified strategy is to introduce some amount of synchronization in the processing of individual instructions which leads to a reduction in the number of I/O operations due to a better use of the broadcast facility.

To implement this scheme, our scheduling algorithm viewed each query tree as a collection of groups of operations, rather than as a collection of operations. Each group consisted of a limb (or a chain) in the tree. A limb was defined to be a number of nodes, starting with a leaf, each of which produced the outer relation to be used in the subsequent operation. Clearly, for any tree with n levels there can be only a single limb with n nodes (although there may be none). If each limb was assigned a unique name, a new tree, whose nodes represented the named limbs, could be constructed showing the dependencies among the limbs. For an example see Figure 4.1. This dependency tree was used by the scheduler to decide the order of scheduling of the limbs. Care had to be taken that the leaf nodes in the dependency tree were initiated before any of their parent nodes began execution. This was necessary since processors were committed to the execution of a limb until its end (unless the number of its pages reduced between stages). If a leaf node (representing a limb producing an inner relation to be used by an operation in another limb) is not initiated before its parent in the tree is, it is possible to reach a deadlock situation if all the processors in the machine are assigned to other limbs in the tree which (directly or indirectly) depend on the leaf limb's output. Note, however, that if a sufficient number of processors is available, all the limbs can be initiated at approximately the same time (this is actually the ideal scheduling policy).

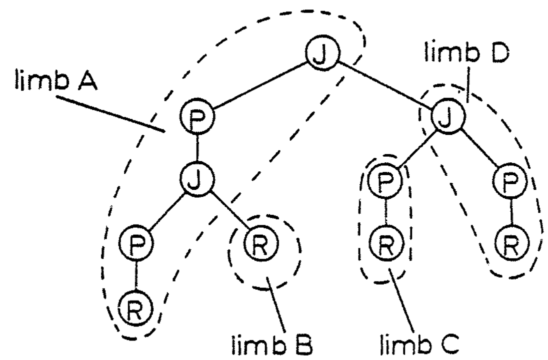

Figure 4.1a: A Sample Query Tree

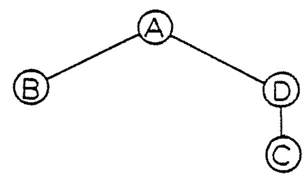

Figure 4.1b: Corresponding Precedence Tree

Earlier in this section we claimed that one of the advantages of the data-flow query processing strategy is that processors could be allocated to all the nodes in a query tree to achieve maximal benefit of the pipelining behavior of the strategy. Clearly, this is not the case in this modified strategy. What we can say, though, is that processors can be allocated to all nodes in the dependency tree (recall that each node represents a limb). The level of inter-instruction parallelism within a limb is not expected to be high - only 1 or 2 active instruction executions at a time. It is important to realize, though, that while it is no longer possible to have processors assigned to every node in the query tree, a substantial level of inter-instruction parallelism is attained through the inter-limb concurrency.

Another problem is that of relation fragmentation. Since operations are initiated before all of the data is present, the relation compression method of [41] cannot be used in the data-flow approach. Initial experimentation with this strategy showed that the effects of relation fragmentation can be so severe as to cause this strategy to perform worse than the other strategies described above. A dynamic compression scheme is employed. In this scheme each producer of inner relation pages attempts to merge its own output page with other uncompressed pages of the inner

relation which have not been read by any of the consuming processors. The compression results are almost never as good as in the other scheme because of the restriction that only those pages seen by no processors can be compressed.

4.2.2. Hardware Characteristics

4.2.2.1. Query Processors

The performance characteristics of a processor are based on the instruction execution times of a PDP LSI-11/03 [61]. There are three main operations which depend on these execution times:

(1) The time required to transfer a page between a CCD memory module and the local memory of a processor.

(2) The time required to determine if a tuple from a relation satisfies a selection criterion.

(3) The time required to join two tuples.

The time to transfer a page between a processor's main memory and a CCD memory module was assumed to be 33 ms based on an LSI-11/03 Q bus bandwidth of 500 Kbytes/second and a page size of 16 Kbytes.

For the purposes of the simulation it is assumed that each attribute in a relation is a character string and that if the tuple does not satisfy the selection criterion three tenths of the characters in the attribute are examined before a match failure occurs.[1] Also, on the average, three

[1] This assumption is not rooted in any fact. As far as we know such data is not available. The number we chose,

tenths of the characters of the joining attributes from both tuples are compared before a failure is determined. For both operations the entire attribute needs to be examined for the detection of a successful match.

4.2.2.2. CCD Memory Modules and Interconnection Matrix

The bandwidth of an individual memory module was assumed to be 2 Mbytes/second based on INTEL 2314 CCD chips. This implies that a 16 Kbyte page could be transferred into (from) a CCD memory module in 8.2 ms if the transfer rate was not limited by the LSI-11 Q bus or the disk transfer rate (see Section 4.2.2.3). Furthermore, it is assumed that the interconnection matrix does not impact CCD memory performance (see [5,62]).

4.2.2.3. Mass Storage Devices

Since DIRECT is a virtual memory machine, pages of relations which are not being referenced by an active query packet, are resident on one or more mass storage devices. When a page is initially referenced, it is loaded into a CCD memory module. IBM 3330 disks were used as the model for the mass storage devices (see [63] for specifications). The transfer time for a 16 Kbyte page is 20 ms. The time to seek N tracks is

as some others in the following few pages, seems "reasonable".

$$10 + N * 0.148 \text{ ms}$$

and the latency time is 8.4 ms. It is assumed that there are two disks available for relation storage and swapping.

4.2.3. Experiment Design

The database used to evaluate the four processor assignment strategies consists of 15 relations. The size (in pages) of each relation was randomly chosen from an exponential distribution with a mean size of 23 pages, minimum relation size of 1 page, and a maximum relation size of 100 pages. The tuple length of each relation was chosen from an exponential distribution with a mean size of 55 bytes, a minimum size of 10 bytes, and a maximum size of 100 bytes. The total database size is 5.5 megabytes. Appendix B contains detailed information on each relation.

For each selection the fraction of tuples which satisfy the selection condition was chosen from an exponential distribution with minimum of 0, maximum of 1, and mean of 0.125 (chosen to produce result relations with a reasonable size). For each join the fraction of tuples (of the product of the number of tuples in both pages) which satisfy the join qualification was selected based on exponential distribution with minimum of 0, maximum of 1, and a mean of 0.0035 (again chosen to produce reasonably sized result relations).

Six different sets of queries were chosen to evaluate the alternative processor allocation strategies. Classes I to IV each contain five query packets and correspond to a range of overhead-intensive queries (Class I) to multi-relation data-intensive queries (Class IV) [10]. Tests Mix I and Mix II each contain ten query packets and represent what we feel to be a reasonable mix of the different classes of queries. Table 4.1 summarizes the six different experiments.

We feel that Classes II and III contain the types of queries which are typically performed by users in accessing relational databases. This is why Mix I and Mix II include a high percentage of queries from these two classes. If views are supported by the relational database system, and queries are modified according to the view, it has been observed by the System R group [64] that it is not unusual for a modified query to contain five to seven join operations. Therefore, the results of Class IV may be as significant as the results of Mix I and Mix II, because Class IV contains a large number of joins in each query packet.

4.2.4. Simulation Results

4.2.4.1. Establishment of a CCD Memory Module to Processor Ratio

TABLE 4.1

Testname	Number of Queries	Description	Number of Source Pages Read by Test
Class I	5	Each with 1 S only	183
Class II	5	Each with 1 J & 2 S	250
Class III	5	2 Queries: 2 J & 3 S 3 Queries: 3 J & 4 S	393
Class IV	5	2 Queries: 4 J & 5 S 1 Query: 5 J & 6 S 1 Query: 6 J & 7 S 1 Query: 7 J & 8 S	529
Mix I	10	2 Queries from Class I 3 Queries from Class II 3 Queries from Class III 2 Queries from Class IV	624
Mix II	10	1 Query from Class I 4 Queries from Class II 4 Queries from Class III 1 Query from Class IV	644

S: Selection
J: Join

The first test performed served two functions. Its primary purpose was to establish an appropriate ratio of CCD memory modules to query processors. Once this value was established it would be used in all subsequent tests. The second function of this experiment was to determine how the performance of each processor assignment strategy is affected by this ratio. It was felt that this should provide some indication about how efficiently each strategy uses the CCD memory modules available.

To perform this experiment the number of processors available was fixed at 50 and Mix I was tested on all strategies for five different CCD memory sizes: 50, 100, 150, 200, and 250. Figure 4.2 contains the results of this experiment. The performance of the SIMD, packet-level, and instruction-level strategies continues to improve as the number of CCD memory modules increases. If the number of CCD memory modules is increased beyond 250, this trend continues until enough modules are present so that pages from source, intermediate, and final relations never have to be ejected to secondary memory. The data-flow strategy, on the other hand, is not significantly affected by the number of CCD memory modules present. This result seems to indicate that this strategy indeed succeeds at using pages from intermediate relations before they are paged out. This saves a write operation to mass storage followed by a read operation when the intermediate relation is subsequently accessed.

The reason that the packet-level strategy is less affected by an increase in the number of modules available than the instruction-level strategy seems to be that it is less flexible and hence has better locality properties than the instruction-level strategy. While this argument should also apply to the SIMD strategy the results indicate other-

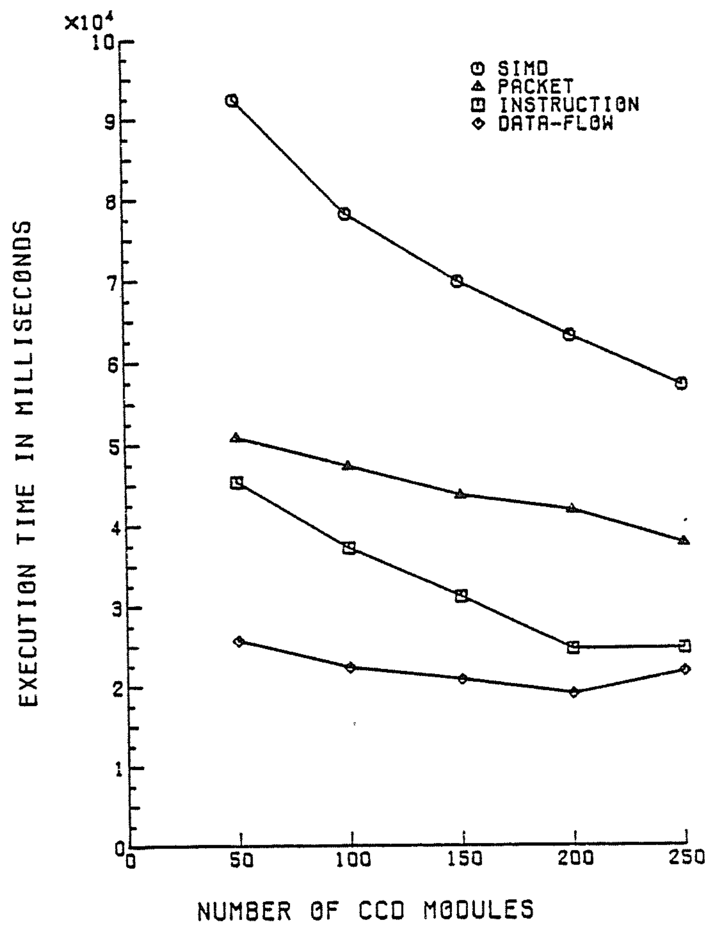

Figure 4.2: CCD to Processor Ratio

wise. Therefore, there may be another, yet undiscovered, reason why the packet-level strategy behaves this way.

Although not presented, similar results were obtained at several other levels of processors and for other tests. Thus, as a compromise between the thriftiness of the data-flow strategy and the greediness of the SIMD, packet-level, and instruction-level strategies, a CCD memory module to processor ratio of 2:1 was chosen. This ratio was used in all subsequent tests. Therefore, in the results presented below, if there are n processors in the configuration, 2*n CCD memory modules will be used.

4.2.4.2. Analysis of the Simulation Results

Using this 2:1 ratio, each of the six tests (Classes I through IV and Mixes I and II) was executed using each alternative strategy for a range of available processors from 10 to 100, in steps of 10. The results of these experiments are presented in Figures 4.3 through 4.8. The reader, when examining the graphs, should be aware that the schedule of packets and instructions for any given processor level is not necessarily an optimal one. At any given point in time, when the back-end controller makes a decision regarding which instruction a processor should be assigned to, or which page should be ejected from CCD memory, it chooses the "best" option. This local (immediate) optimization does not always produce an optimal

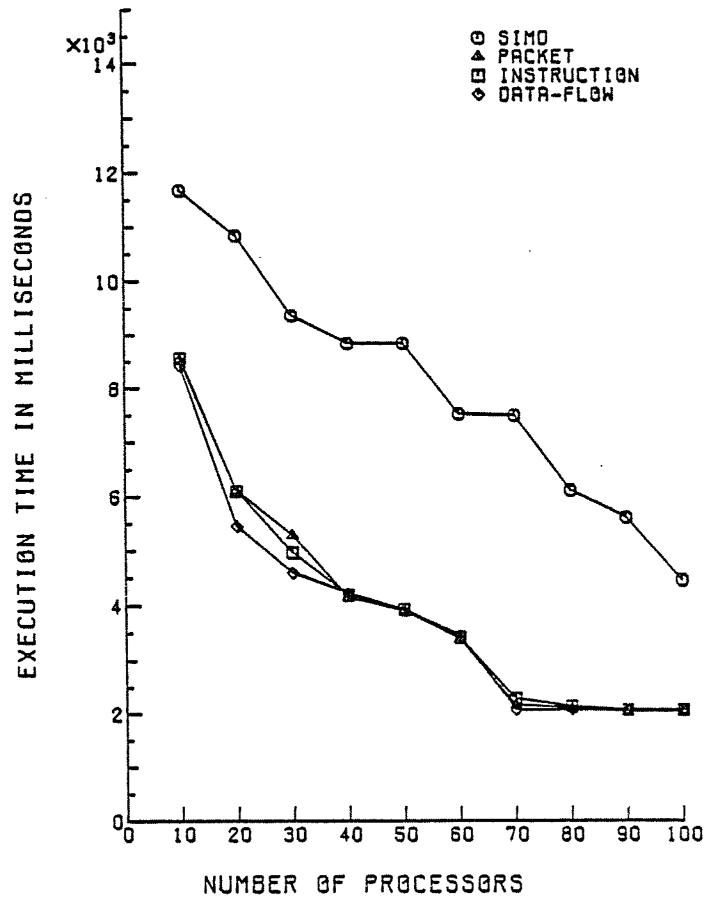

Figure 4.3: Class I

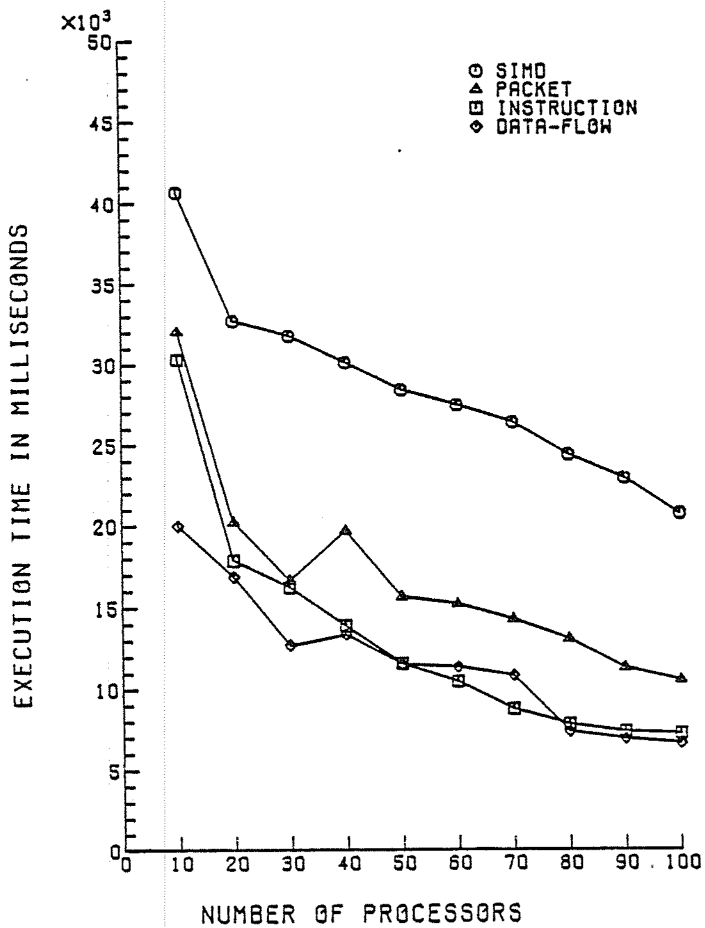

Figure 4.4: Class II

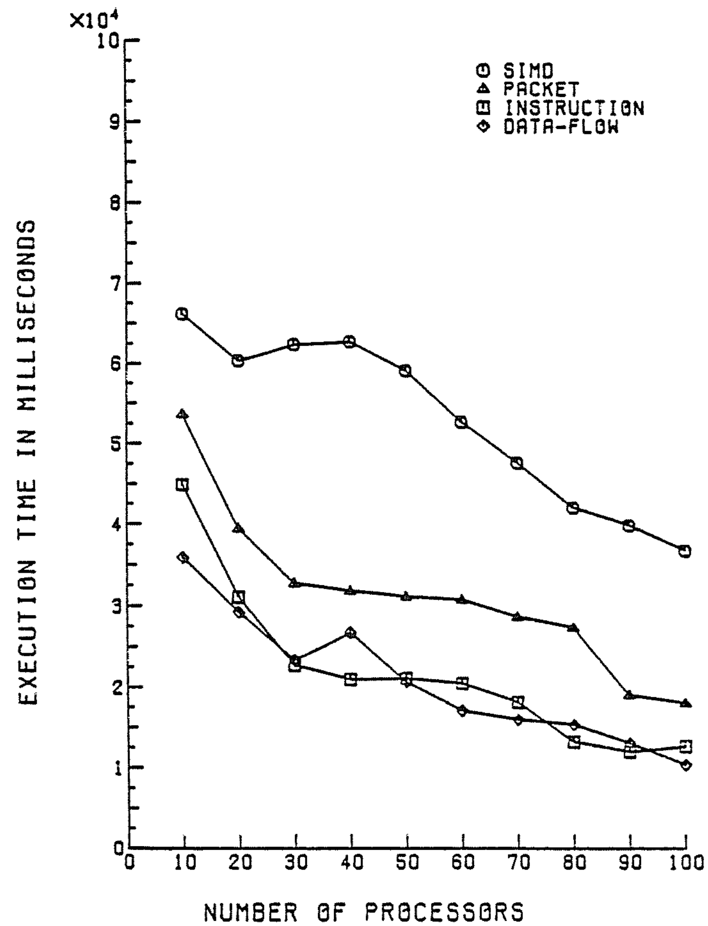

Figure 4.5: Class III

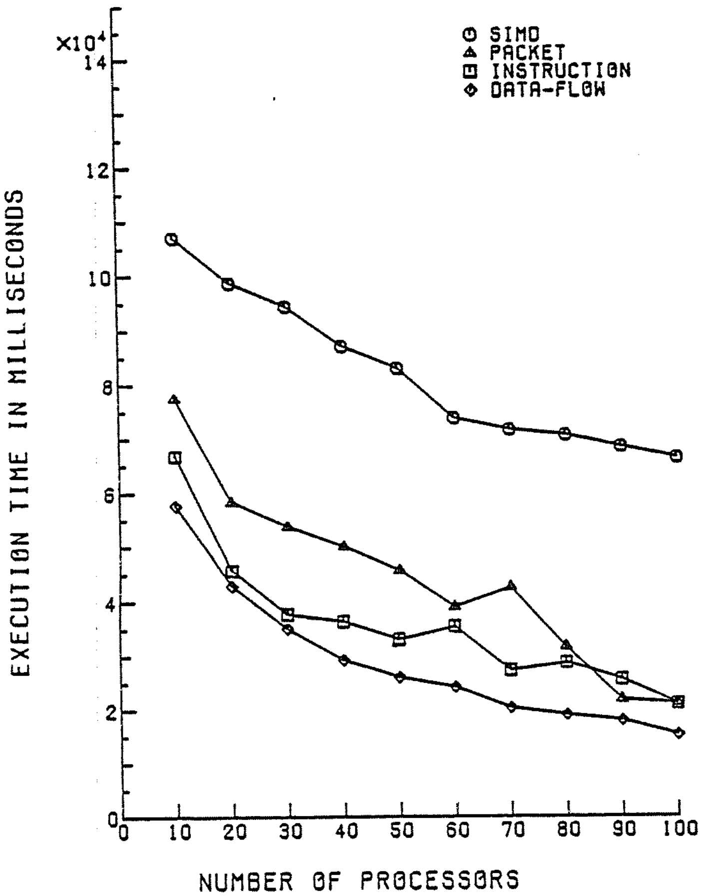

Figure 4.6: Class IV

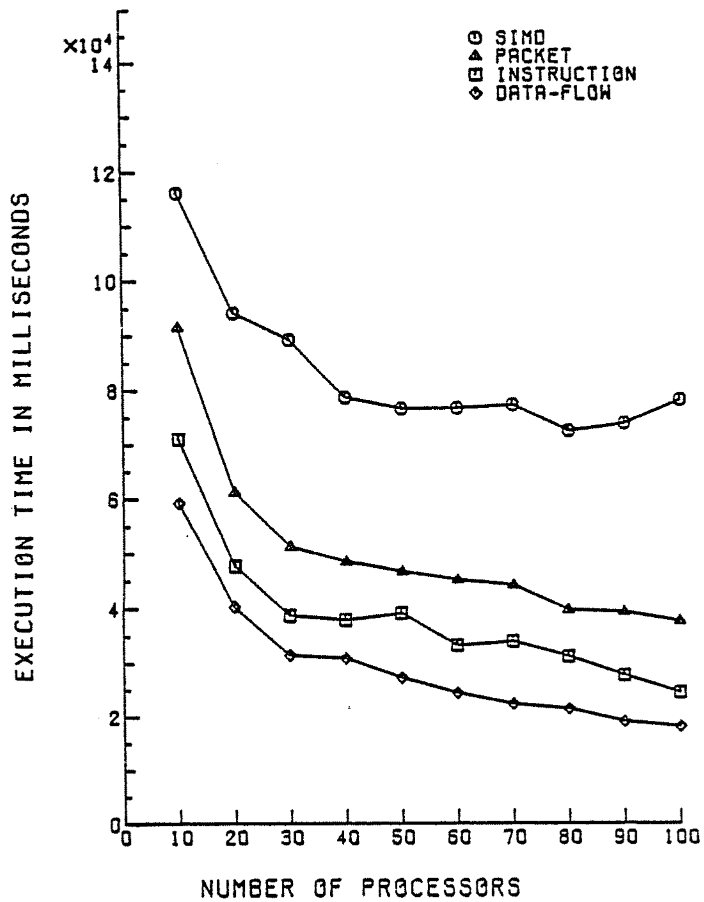

Figure 4.7: Mix I

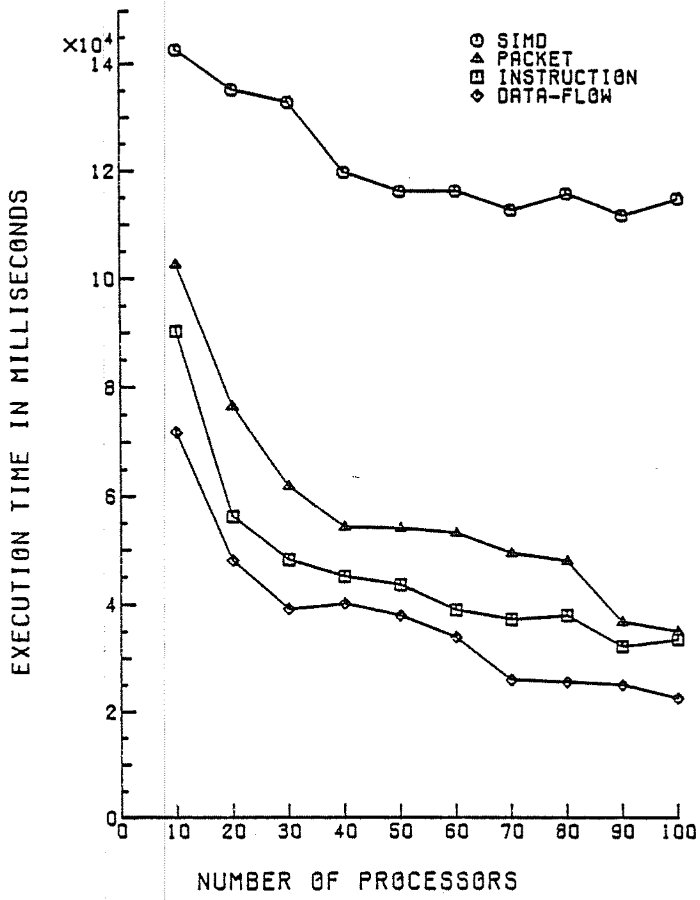

Figure 4.8: Mix II

schedule. Consequently, certain anomalies can occur in the results. For example, in Class II (Figure 4.4), as the number of processors available increases from 30 to 40, the execution time of the packet strategy increases instead of decreasing as expected. There is certainly some schedule of packets for this case in which the execution time either decreases or remains constant as the number of processors is increased from 30 to 40. It is simply the case that the back-end controller made a decision that, in the long run, turned out to be a bad decision.

One obvious result from these experiments is that the SIMD strategy always performs significantly poorer than all the other strategies. A surprising result illustrated by these tests is the relatively good performance of the packet-level strategy when compared with the instruction-level strategy. The execution time of Class I for both is identical because each query packet contains only one instruction. For Class II their performance is very similar. Class III is the only case where the instruction-level strategy is significantly better than the packet-level strategy. For Class IV, over the range of 40 to 90 processors, the packet-level strategy actually outperforms the instruction-level strategy. This apparently occurs because the packet-level strategy thrashes less. In the packet-level strategy, as a processor finishes executing an instruction it is either re-assigned to another instruction in the same packet, or is left idle until an instruction in the packet is enabled. Under identical conditions the instruction-level strategy may assign the free processor to an instruction from another packet, or even initiate a new packet. Executing this new instruction will probably result in pages from secondary memory replacing pages currently in the CCD memory. As a consequence, when an instruction from the original packet is finally enabled, its operands will most likely have been paged out. Finally, for Mixes I and II the performance of the instruction-level strategy is about 10% better than that of the packet-level strategy.

These tests clearly indicate the superiority of the data-flow strategy for processor allocation. In all the tests, ranging from overhead-intensive (Class I) to execution-intensive (Class IV), and the two mixes (Mix I and II), the data-flow strategy always performed significantly better than any of the other three strategies. If Mix I and Mix II are taken to be representative of typical query mixes, then, for a given number of processors and CCD memory modules, the data-flow strategy was approximately three times as fast as the SIMD strategy and about 1.3 and 1.7 times as fast as the instruction-level and packet-level strategies. Furthermore, when one examines the performance

of each strategy under heavy loads (less than 50 processors available), the data-flow strategy demonstrates an even greater performance improvement.

Initially the relative performance of the data-flow and instruction-level strategies on Mix I and Mix II was somewhat puzzling because the data-flow is only marginally better than the instruction-level for Classes II and III (which make up most of Mix I and Mix II). As an explanation we hypothesized that the data-flow strategy, because it is an advanced form of pipelining, tends to utilize the CCD cache more efficiently (this is borne out by the earlier experiment). Further reflection on the problem led to the observation that the queries in Class II are not as likely to benefit from the effects of the pipelining, since each query contains only three operators. However, the performance of Class III, in which some of the queries contained up to seven operators and consequently should have benefited from the pipelining characteristics of the data-flow strategy, seem to contradict the hypothesis. In a further attempt to verify the hypothesis we tried two experiments. The first was to increase the number of queries in Class III from five to ten (we also increased the number of relations in the database so that the effects of two queries referencing the same relation were minimized). For this variation of test Class III, the data-flow strategy was 11.6% faster than the instruction-level strategy (originally it was only 2.3% faster). This seems to indicate that the original Class III test did not generate sufficient CCD activity for the benefits of pipelining to appear. The second experiment conducted is described in the next section.

4.2.4.3. Effect of Swapping on Performance

The next experiment conducted was to determine the impact that swapping pages between CCD memory modules and secondary memory has on query execution time. It was felt that this was a very significant experiment because it has been argued that database machines which use paging will always be I/O bound [65].

To determine this effect the simulations for the data-flow and instruction-level strategies were modified so that the time to transfer a page between a CCD memory module and a disk is 0 ms. In this way it appears that the bandwidth of the channel and disk are infinite. Figures 4.9 and 4.10 present the results of this experiment.

For the instruction-level strategy (Figure 4.9), the improvement averaged over all processor levels is 39.5%. Thus swapping has the effect of decreasing system throughput slightly more than one third. While significant, the overhead of swapping is not as high as expected.

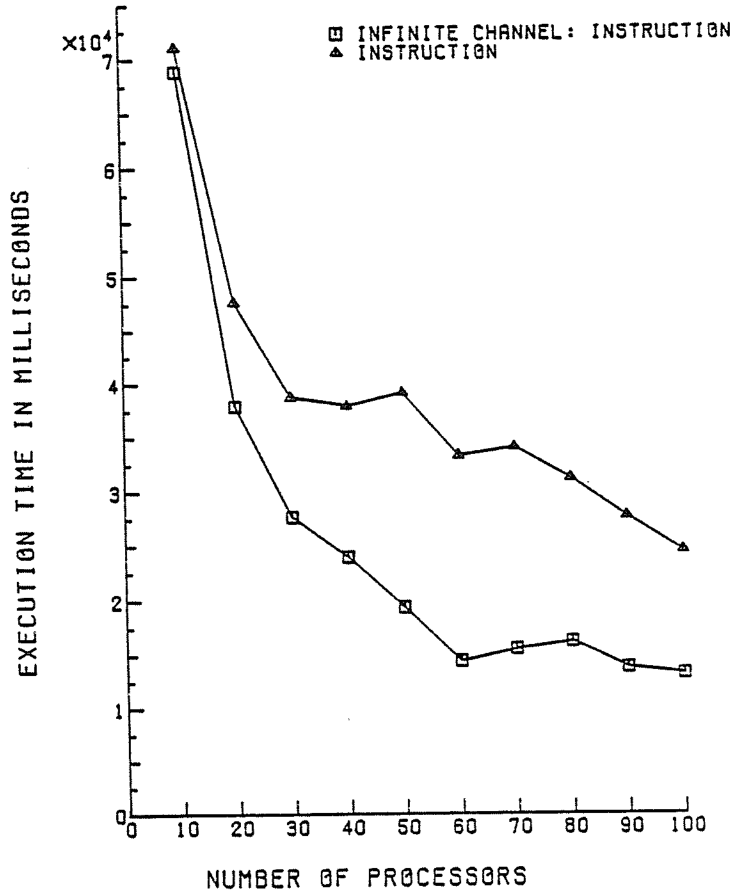

Figure 4.9: Infinite Channel

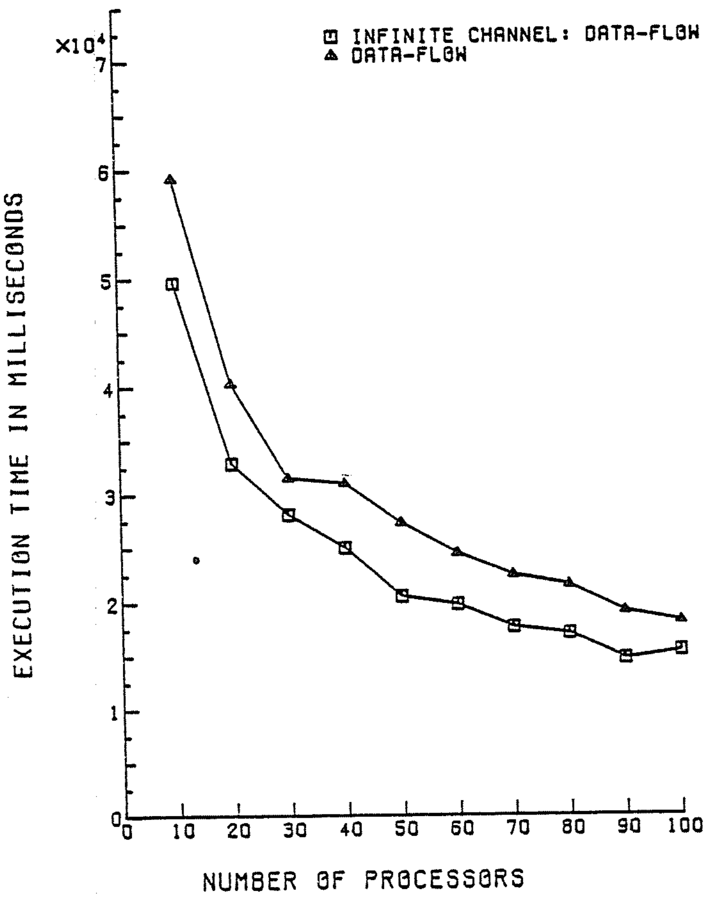

Figure 4.10: Infinite Channel

For the data-flow strategy the improvement averaged over all the processor levels is 18.4%. This figure clearly indicates that virtual memory database machines can be organized in such a way as to avoid being I/O bound.

The difference in improvement shown by the two strategies in the "infinite disk" case is 21%. This value is approximately the same as the difference between the instruction-level and data-flow strategies for tests Mix I and Mix II as shown in Figures 4.7 and 4.8, and seem to show that the cleverer use of the CCD cache by the data-flow strategy was the main reason for its superiority.

4.2.4.4. Effects of Database Size and Query Processor Speed

The last two experiments conducted attempted to measure the sensitivity of the results we have presented so far to the two parameters we consider the most important: database size and instruction execution time of the processor.

To determine the sensitivity of the instruction-level and data-flow strategies to the size of the database being accessed we modified test Mix I by doubling the size of each relation in the database. On the average, the data-flow strategy is 22.1% faster than the instruction-level strategy. Somewhat better results were obtained with other tests on this and other enlarged (in both the number of relations and the average number of pages per relation) databases. In general, as the database referenced increases in size, the percentage improvement of the data-flow strategy over the other strategies tends to increase for most tests. As stated earlier, we feel the reason for this increase is due to the improved CCD management exhibited by the data-flow strategy.

The second sensitivity experiment we performed was to double the speed of the processor by cutting in one half the execution time of each instruction and doubling the DMA transfer rate. For Mix I, on the average, the data-flow strategy was 35.5% faster than the instruction-level strategy (an increase of 11% over the difference between the two strategies for Mix I with the normal processor speeds). This result seems to again demonstrate the effect of pipelining in the data-flow strategy. However, compared to the results with normal processor speeds, the execution time decreased by only 21% for the data-flow strategy and only 6% for the instruction-level strategy. This implies that given the present hardware components performances, increasing the performance of one component (in this case the processor speed) does not mean that system performance will increase by the same factor.

4.2.4.5. Message Activity

While the back-end controller requirements for each strategy have not been modeled, the message activity of each strategy was. Each operator which is sent to a processor is counted as one message. In the SIMD strategy, the distribution of the operator to all processors is counted as only one message since it is assumed that the controller in such a database machine could broadcast control messages. Each relation page request executed by a processor is counted as three messages: one to the back-end controller to make the request, one to the processor from the controller containing the CCD memory module number, and another from the processor, to signal that it has read (written) the memory module so that the controller can update its tables accordingly. When "end-of-relation" is received on a page request (i.e. there are no more pages available), only two messages are exchanged.

Figure 4.11 shows the number of messages sent between the set of processors and the back-end controller for all the strategies running test Mix I. As is shown, this is one measure in which the data-flow strategy performs poorer than all the other strategies. There are a number of reasons for this result. First, there is only partial compression of intermediate relation pages at any time. Examination of the simulation results with the trace turned

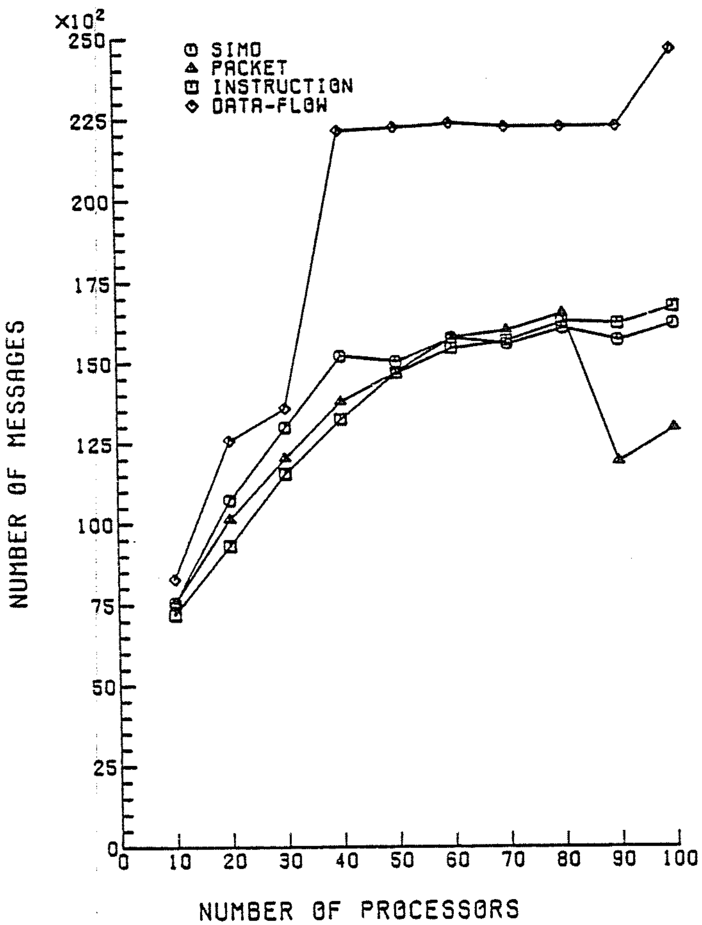

Figure 4.11: Messages

on, revealed that there were a number of cases where the dynamic compression scheme employed, had no effect. This happened when the first page of the inner relation was read by a processor executing the subsequent operation before additional pages of that relation were produced.[2] A second reason, which also affects the number of messages in the other strategies, is that no advantage is taken of the broadcast facility with regards to messages. When a processor needs a page, it requests the address of the CCD module in which the page resides from the back-end controller. The controller sends its reply only to the requesting processor (as opposed to broadcasting it). If two different processors that need to read the same page receive the replies to their requests at almost the same time, they will read the page almost simultaneously. However, they will both have to send request messages and receive individual reply messages from the back-end controller in order to achieve this.

4.2.5. Summary

In this section the superiority of a data-flow approach to processor scheduling in DIRECT was demonstrated. This was shown in a number of ways:

---

[2] Recall that the strategy used is to compress only pages that have not been read by any processors. Thus, once the first page is read, no compression can take place.

(1) A smaller number of CCD modules are needed for a given number of processors.

(2) In a data-flow organization the performance is always better than the other approaches. Almost always at least one and half times as good as the closest competitor.

(3) The traffic between the mass storage units and the CCD buffer is kept to a minimum.

(4) However, the number of messages exchanged between the processors and the controller is always higher.

Furthermore, these results were obtained under two assumptions favorable to the other approaches. These were the chosen CCD to processor ratio of 2:1, and the availability of two mass storage units, rather than one, for back up storage. Given the data-flow approach thriftiness and the other approaches' greediness in usage of CCD modules these are significant assumptions.

An important problem exposed by this research is the high level of message traffic activity. Regardless of the processor assignment strategy employed, the amount of message traffic which must be supported is very high. If 8,000 back-end controller instructions are required to process each of the 7,500 messages passed in executing Mix I for 10 processors, (a figure derived from UNIX pipe code efficiency) [66], then 60 million instructions will be executed just to process the messages. If each instruction takes one micro-second then 60 seconds will be required to process the messages (60 seconds are also required to execute the queries for the data-flow strategy). It is impor-

tant to notice that any decrease in query execution time by the use of additional processors may be offset by the increased time required to process messages in the back-end controller. For example, using the data-flow strategy and 50 processors, 27 seconds are required to execute the query and 178 seconds will be required by the back-end controller to process the messages.

There are three potential solutions for this problem. The simplest is to increase the page size. Increasing the page size by an order of magnitude should decrease message activity by a similar factor. However, there may also be a decrease in the maximal degree of concurrency possible. Another solution is to reduce the cost of processing an individual message by implementing message handling software in microcode on the back-end controller rather than reducing the volume of messages. This could help significantly. The ultimate solution, however, is to design a new architecture which would be tailored to the algorithms described in the previous chapter with distributed control. If at all possible, the total number of messages required to implement the algorithms should be reduced. In the following chapter we present a detailed description of such an architecture which is based, to some degree, on [67].

4.3. A Comparative Study of Associative Disk Implementations

In this section we examine three types of associative disk designs in detail. Our purpose to is to glean as much information as possible about each design type in order to enable an intelligent choice of an associative disk design for our database machine. This work has initially been reported in [52,53].

The three associative disk types examined are: processor-per-track machines (PPT) as exemplified by RAP [14], processor-per-head machines (PPH) with parallel readout disks as in DBC [9], and processor-per-disk machines (PPD).[3] In undertaking this study we believed that the machines could be classified according to such categories as cost and performance based on their type, rather than the particular technology or variation on organization used.

It is clear that under ideal conditions (e.g., an infinite bandwidth channel between the disk and the output device) PPT-type devices will be superior to the other designs. As an example consider a relation that occupies 5 cylinders, each with 20 recording surfaces. With an

[3] A PPT-type device that uses off-the-shelf bubble memory chips is also considered in [53]. Although these additional results are interesting, we do not include them here for reasons of brevity.

infinite bandwidth output channel, a simple selection operation in a PPT machine could be executed in a single revolution. A PPH machine would require 5 revolutions while the PPD machine would require 100 revolutions. Furthermore, both the PPH and the PPD machines will require additional time for the track-to-track seek times.

We feel that in order to obtain a realistic measure of the relative performances of these designs, one needs to consider a number of factors. One of these is the bandwidth of the channel connecting the associative disk to the host computer. Contention for the channel due to insufficient bandwidth may necessitate additional revolutions in order to completely process the data on the fly. Another factor is the availability of auxiliary information about the data. For example, DBC has the ability to restrict the number of cylinders to be searched through the use of indices and data clustering. A third factor is the processing capabilities of the processor associated with the disk. Space limitations on the read head of a fixed-head disk may force each processor in a PPT organization to have only a small amount of memory for temporary storage of selected tuples, further aggravating the delay due to channel contention.

4.3.1. Overview of the Three Organizations

In this section we present a very brief description of the three organizations examined. More detail is available in Section 2.2.1 and in [52,53].

The PPT organization we modeled is very similar to RAP as described in [14]. There are, however, a number of differences. First, our initial experiments do not use mark bits, although later we will show what their effect on performance is. Second, the processors are assumed to be able to compare only a single pair of values at a time. Third, each processor has available to it some number of buffers whose size is a multiple of the tuple length. The purpose of the buffers is to serve as temporary storage for selected data before it is output to the bus. The size of each buffer, and the number of buffers per processors are parameters that are varied in the simulation. A sample PPT organization is shown in Figure 4.12.

Our PPH organization is modeled on the Track Information Processors of the Mass Memory component of the DBC [44]. As with PPT, each processor has a number of temporary storage buffers for the same purpose. A sample PPH organization is shown in Figure 4.13.

In modeling PPD we did not have to consider a specific architecture. The processor is assumed to be able to keep up with the disk data transfer rate. Since in PPD there is

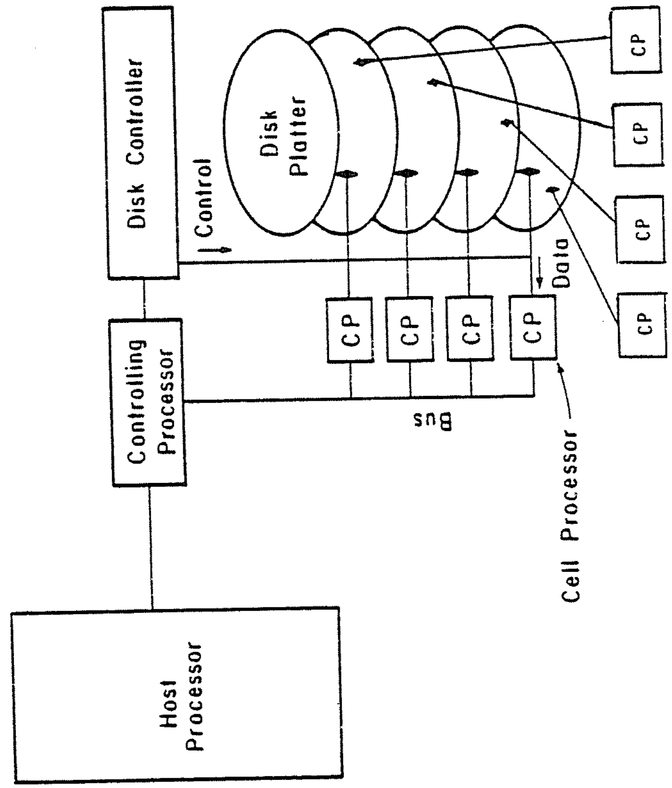

Figure 4.12: PPT with two tracks per platter

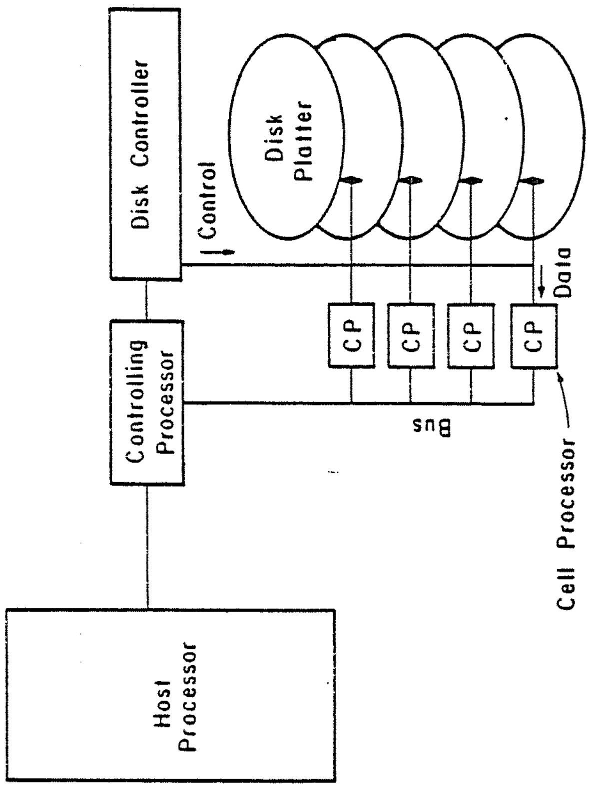

Figure 4.13: A PPH organization

no contention for a global resource (such as an output bus) it was not necessary to simulate its behavior. A sample PPD organization is shown in Figure 4.14.

4.3.2. Specifications of t'e Models

In this section we describe the physical and logical characteristics of the PPT, PPH, and PPD associative disks modeled.

4.3.2.1. Physical Characteristics

4.3.2.1.1. Mass Storage Device Specifications

The mass storage device employed in our models is based on the IBM 3330 disk drive [63]. This device has 404 cylinders with 19 tracks (recording surfaces) per cylinder. Each track holds 13,030 bytes. The rotational speed of this disk drive is one revolution every 16.7 ms. Head movement of the disk was modeled as two components: a time to start the head moving (10 ms) and a track-to-track movement time (0.10 ms). Thus, seeking from one cylinder to the next requires 10.1 ms and seeking 50 cylinders requires 15 ms.

4.3.2.1.2. Associative Disk Specifications

The PPH associative disk organization was modeled as a modified IBM 3330 disk drive with 19 processors (one per recording surface) and some number of output buffers per processor. In order to experiment with the effect of

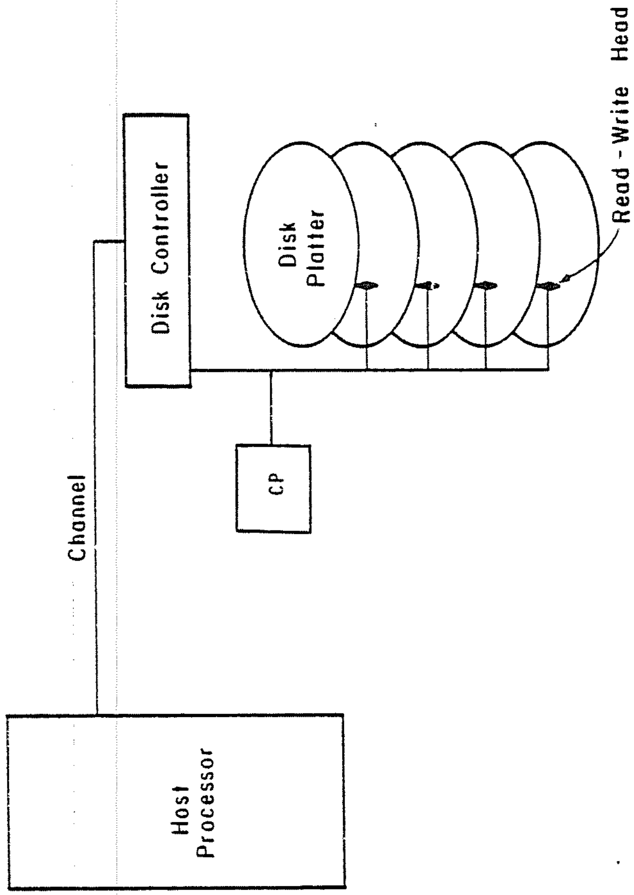

Figure 4.14: A PPD organization

output buffer size, the size of each output buffer was not fixed. Instead each was assumed to hold an integral number of tuples and was varied in different experiments.

Modeling the PPT associative disk organization was the most difficult. One choice would have been to assume that the PPT was implemented using a commercially available fixed-head disk drive such as the IBM 2305 Model 2 [63]. This device has 768 heads/tracks with a capacity of 14,660 bytes per track. Its rotational speed is 10 ms. This choice would have limited our experiments to relations with a maximum size of 5.4 Mbytes (which occupy only 22 cylinders of the 3330 moving head drive). Instead we decided to model the physical characteristics of the PPT design as a 3330 disk drive with one head for each of the 7676 tracks (404 cylinders * 19 tracks/cylinder) and some output buffers per head. While constructing such a device is probably out of the question, modeling the PPT associative disk this way enables us to establish a performance baseline by which the performance of the PPH and PPD organizations can be gauged.

The rotational speed for the PPT design was assumed to be 16.7 ms. While this value is somewhat higher than that of the 2305 Model 2 fixed head disk, it was chosen in order to avoid (in our minds at least) an "apples and oranges" comparison of the three approaches. If we had assumed a rotational speed of 10 ms then we would have had to make the processors in the PPT design approximately 50% faster (in order to process the same amount of data in two thirds the time).

Finally, the PPD associative disk organization was modeled as one IBM 3330 disk drive and one processor. As discussed earlier, the speed of the processor in all of the designs was assumed to be sufficient to permit processing selection operations at the speed at which data is delivered by the selected read head. For IBM 3330 disk drives this rate is approximately 800 Kbytes/second. Thus the processor has approximately 1.25 microseconds to examine each byte. Assuming that 3 instructions are required to examine a byte and that every byte must be examined, then the processor must be approximately a 2.4 MIP processor.

4.3.2.1.3. Output Channel Specifications

As discussed in Section 2.2, all cell processors were assumed to be connected to a single output channel for the transfer of selected tuples to the controlling processor. We assumed that this output channel operated independently and asynchronously from the cell processors. The bandwidth of this channel was assumed to be 2.0 Mbytes/second based on the maximum bandwidth of the VAX 11/780's Mass Bus Adapter. It should be noted that the output channel has to

be as fast as the disk data transfer rate, although it can be faster. The disk transfer rate determines the processor speed, while the output channel bandwidth affects the rate at which output buffers in the processors will be emptied (loading and unloading of the buffers are asynchronous operations).

The servicing of the cell processors by the output channel was modeled in two different ways: round robin and first come, first served. For the round robin service algorithm, we assumed that 1 micro-second was required for the output channel to poll the next cell processor to see whether it had a full output buffer to be transferred to the host.

Modeling the first come, first served servicing strategy required accounting for the overhead of arbitrating between two or more processors which attempt to acquire the output channel simultaneously. An implementation of this arbitration process would certainly be more complex and time consuming than having the output channel simply advance to the next processor. Therefore, we assumed that for this strategy 3 micro-seconds would be required to establish which requesting cell processor would be serviced next by the output channel.

4.3.2.2. Operational Characteristics

4.3.2.2.1. Source Relation Organization

For the PPD and PPH associative disks relations are stored in such a manner as to occupy the minimum number of cylinders possible. That is, tuples from a relation must first fill an entire track before a second track is used, then an entire cylinder, etc. In this way, the number of cylinders which must be searched to execute a selection operation on a relation is minimized and non-essential seek operations are eliminated. This organization is termed compressed. It is used for the PPD and PPH associative disks in all experiments conducted.

As first suggested by Sadowski and Schuster [68], concurrency can be maximized in the processing of a selection operation in a PPT associative disk if tuples from a relation are uniformly distributed across all tracks. This organization is termed horizontal and permits all cell processors to participate in every selection operation.[4] The horizontal organization was used for the PPT associative disk in all experiments conducted.

---

[4] Assuming that the relation has as many tuples as there are tracks.

4.3.2.2.2. Selected Tuple Distribution

A separate issue from the organization of the relations on the mass storage device is the distribution of the tuples which satisfy the selection criterion. For our experiments we considered two possible distributions: uniform and clustered. The uniform distribution implies that, on the average, the same number of result tuples are selected from every track that participates. However, if every cell processor in the PPH and PPT associative disks produced exactly the same number of tuples, then artificial contention for the output bus would occur. Therefore, the actual number of tuples selected from each track was determined by random selection from a normal distribution. Furthermore, the positions of the selected tuples within the track were randomly selected.

The selected tuples may form a clustered distribution in two cases which we term sorted and indexed. The sorted case occurs when a relation is sorted on an attribute, and that attribute is referenced in the selection criterion of the query (e.g. a relation corresponding to names in the phone book and the query: retrieve name="smith"). In this case a limited number of tracks will hold qualifying tuples but all tracks holding tuples from the relation must be examined. Furthermore, every track which contains qualifying tuples (except possibly the first and the last) will contain nothing but qualifying tuples from the source relation.[5] The second case of a clustered distribution of selected tuples occurs when there is a non-dense primary index (such as an ISAM index) on the attribute being qualified. As in the previous case, only a limited number of tracks will hold qualifying tuples. However, the existence of the index permits the search to be restricted to only those cylinders containing qualifying tuples. Since all processors in the PPT design are active simultaneously, these two cases of the clustered distribution are the same.

4.3.3. Experiments and Results

In this section the results of a number of experiments that we conducted are presented. We obtained our results from an event driven simulation written in Pascal and run on a VAX 11/780. As described in the previous sections, the models utilized were as realistic as possible. We ran the simulation using relation sizes of 10,000, 100,000, and 1,000,000 tuples. The tuple size was varied from 20 to 100 to 1,000 bytes.[6] We felt that these tuple lengths represented three realistic cases: a relation with 20 byte

[5] As a consequence of the horizontal data organization employed by PPT associative disks, tracks containing qualifying tuples will also contain tuples from other relations.

[6] We did not run a test for the case of 1,000,000 tuples each of size 1,000 bytes because the total relation size would have exceeded the storage capabilities of the IBM 3330 disk we were modeling.

tuples can represent an index; 100 byte tuples represent what we feel to be the "average" tuple size; Finally, 1,000 byte tuples can be found in relations describing personnel information in a corporate database. For all experiments performed, the data distribution was horizontal for the PPT design and compressed for the PPH and PPD designs.

4.3.3.1. Impact of Output Buffer Availability

The first set of experiments explored the impact of the number of output buffers available to each cell processor on the relative performance of the three associative disk designs. In each of these experiments a uniform distribution of selected tuples was assumed. Access to the output channel was done in a round-robin fashion. Tables 4.2 and 4.3 present the results of this set of experiments for the PPH and PPT organizations for a relation with 100,000 tuples of size 100 bytes and for queries with 3 different selectivity factor. A selectivity factor indicates the fraction of tuples from the relation which satisfy the selection criteria of the query. Similar results were observed for the other tests. It was not necessary to conduct this experiment for the PPD organization since it uses only a single processor, and thus there will be no contention for the output channel.

Initially we were puzzled by the results presented in Tables 4.2 and 4.3 as we had expected the performance of

Table 4.2
PPH - 19 Processors
100,000 Tuples of Size 100 bytes
Uniform Distribution of Selected Tuples

Output Buffers #	Size in Tuples	Execution Time in Revolutions Selectivity Factor of Query .0001	.005	.10
2	1	82	82	82
2	5	82	83	89
5	2	82	82	83
10	1	82	82	82
101	8	82	83	83

Table 4.3
PPT - 7676 Processors
100,000 Tuples of Size 100 bytes
Uniform Distribution of Selected Tuples

Output Buffers #	Size in Tuples	Execution Time in Revolutions Selectivity Factor of Query .0001	.005	.10
2	1	1	3	31
2	5	1	3	31
5	2	1	3	31
10	1	1	3	31

both designs to be significantly impacted by the number and

size of the output buffers available. Additional experiments and an analysis of the problem indicated that there are three primary factors which determine the execution time of a query: bandwidth of the output bus, the number of bytes to be transferred to the host, and the distribution of the selected tuples among the tracks. Consider, for example, the above experiment. For a selectivity factor of 0.1, the query will produce 10,000 one hundred byte tuples. For an output bus bandwidth of 2 Mbytes/second, 0.5 seconds are required to move the qualified tuples from the cell processors to the host. Ideally, in the PPT design the query should be executed in 1 revolution. However, a minimum of thirty revolutions is required just to transfer the selected tuples to the host regardless of the number or size of the output buffers. For the PPT design our calculations and experiments indicate that, until the bandwidth of the output bus reaches at least 60 Mbytes/second (the minimum bandwidth to transfer one million bytes in one revolution), having more than one output buffer per processor has little or no impact on performance. For the remaining experiments we have chosen to use 2 buffers of size 1 for the PPT design in order to permit some parallelism within a cell processor since there are cases where the blocking of processors does impact performance adversely.

The PPH design is unaffected by the number and size of the output buffers available for a completely different reason in this experiment. The relation being processed will occupy 41 cylinders of the disk. Hence the minimum execution time for the query (regardless of the selectivity factor) is one revolution for each seek operation plus one revolution for each cylinder.[7] Thus, the minimum execution time is 82 revolutions which is approximately the performance obtained for all tests presented in Table 4.2. Since 82 revolutions are required to process the query and only 30 are required to transfer the selected tuples over the output bus to the host, the bus is not a bottleneck for the PPH design. For the experiments presented in the following section we have used 2 buffers of size 1. In Section 4.3.3.3 we examine the impact of the number of buffers and their size on the performance of the PPT and PPH designs when the selected tuples are from a clustered distribution.

4.3.3.2. Comparison of the Three Organizations

The relative performance of each of the associative disk designs on selection operations with varying selectivity factors are shown in Tables 4.4-4.6 for a relation

---

7 Note that we do not assume the availability of positional sensing disks. Thus an entire revolution is required for each seek. A discussion of the effect of such devices on the performance of the PPH and PPD designs is included in Section 4.3.3.5.

with 100,000 tuples of size 20, 100, and 1000 bytes respectively. The values for the PPD organization were obtained by use of the following formula:

time = revs * 0.016666 seconds/revolution

where

revs = 1 + (19 * numcyls) + numcyls - 1

and where numcyls is the number of cylinders the relation occupies and 19 is the number of recording surfaces on the disk. The initial revolution is required for the seek to the first cylinder occupied by the relation. Nineteen

Table 4.4
100,000 Tuples of Size 20 bytes

Uniform Distribution of Selected Tuples

Selectivity Factor of Query	Execution Time in Seconds PPT	PPH	PPD
.0001	.008	.300	3.0
.0005	.008	.317	3.0
.001	.009	.367	3.0
.005	.013	.417	3.0
.01	.018	.383	3.0
.05	.066	.433	3.0
.1	.116	.433	3.0

PPT: 7676 processors each with 2 buffers of size 1
PPH: 19 processors each with 2 buffers of size 1
PPD: 1 processor

Table 4.5
100,000 Tuples of Size 100 bytes

Uniform Distribution of Selected Tuples

Selectivity Factor of Query	Execution Time in Seconds PPT	PPH	PPD
.0001	.011	1.37	13.6
.0005	.012	1.37	13.6
.001	.015	1.37	13.6
.005	.034	1.37	13.6
.01	.059	1.37	13.6
.05	.266	1.37	13.6
.1	.516	1.37	13.6

PPT: 7676 processors each with 2 buffers of size 1
PPH: 19 processors each with 2 buffers of size 1
PPD: 1 processor

revolutions are required for each cylinder. Finally, an additional revolution, to allow for the track to track seek time is required between cylinders.

Based on these experiments we have developed a number of conclusions regarding the performance of these three associative disk organizations. First, a lower bound on the PPH performance can be obtained from the PPD formula with the removal of the figure of 19 to reflect the parallel readout capability. Second, PPH generally performs at, or close to, the lower bound. Third, in general, for a

Table 4.6
100,000 Tuples of Size 1000 bytes

Uniform Distribution of Selected Tuples

Selectivity Factor of Query	Execution Time in Seconds PPT	PPH	PPD
.0001	.035	13.5	135
.0005	.041	13.5	135
.001	.066	13.5	135
.005	.261	13.5	135
.01	.510	13.5	135
.05	2.52	13.5	135
.1	5.02	14.0	135

PPT: 7676 processors each with 2 buffers of size 1
PPH: 19 processors each with 2 buffers of size 1
PPD: 1 processor

uniform distribution of selected tuples PPH will execute queries approximately 10 times faster than PPD since there are 20 revolutions for each cylinder in the PPD organization (1 for positioning and 19 for readout) and 2 in the PPH (1 for positioning and 1 for readout).

A fourth observation based on these results is that the performance of the PPT organization degrades linearly, more or less, as the selectivity factor increases.[8]

[8] Because of the expense of running our simulation we were not able to confirm this conjecture for higher selectivity factors.

Finally, in all the experiments conducted (Tables 4.4-4.6 present the results of only a few experiments) the PPT organization proved superior to the PPH organization which was better than the PPD. However, unlike the PPH machine, where contention for the channel did not seem to markedly degrade performance, the PPT organization suffers very heavily from this problem. We see that for small selectivity factors (.0001-.001) the PPT machine can complete the query in 2 or 3 revolutions whereas the PPH machine requires approximately twice the number of cylinders occupied by the relation. However, for large selectivity factors (.1) PPT is only 3 to 4 times as fast as PPH regardless of the relation size. We feel that this is remarkable considering the fact that the PPT design which was modeled had 404 times as many processors as the PPH design.

4.3.3.1. Impact of Clustering of Selected Tuples

As discussed in Section 4.3.2.2.2, the selected tuples can originate from a relatively limited number of tracks when either the relation is sorted on the attribute being qualified or a non-dense primary index exists on it. In this section we evaluate the performance of the three designs for these two cases.

Because the experiments on the impact of output buffer size and availability presented in Section 4.3.3.1 were conducted using a uniform distribution of selected tuples,

we began this set of experiments by re-examining the impact of output buffer size on the performance of the PPT and PPH designs. In addition we examined whether mark bits could be used as an alternative technique for enhancing the performance of the these two designs.

4.3.3.3.1. Impact of the Use of Mark bits and Output Buffer Availability

If mark bits are employed in a PPT or PPH design when a cell processor finds a qualifying tuple it sets the mark bit [14] associated with the tuple and attempts to place the tuple in one of its output buffers (whenever a marked tuple is placed in an output buffer, the mark bit is always turned off). By the end of the first revolution all qualifying tuples will have been marked. In subsequent revolutions (if they are necessary), whenever an output buffer becomes available, each cell processor will stuff the next marked tuple it finds. Recall that without mark bits, a blocked cell processor in the PPT or PPH design must wait an integral number of disk revolutions before it may resume (so that it continues precisely where it left off). Thus when mark bits are employed, once an output buffer becomes available the processor can resume outputting tuples without having to wait until it again reaches the position at which all output buffers were filled (saving at least 1/2 of a revolution on the average).

It is important to notice that our use of mark bits differs from the applications that have been suggested previously [14,20,37]. In earlier research, mark bits played an important part in processing entire queries (including joins, projections, etc.) directly on the disk through the use of multiple mark bits and multiple revolutions. In the experiments presented below, we are only concerned in evaluating the performance of associative disks when executing selections "on-the-fly". We use mark bits only as a technique for improving system performance by reducing the amount of processor idle time, and not as a means of increasing the disk processing capabilities.

In the tables below the impact of mark bits and larger output buffers on the performance of the PPH and PPT designs is presented. The results for the case when the relation is sorted on the attribute being qualified are presented in Tables 4.7 and 4.9 for the PPH and PPT designs, respectively. The impact on the PPH design when a non-dense primary index exists on the attribute being qualified is presented in Table 4.8.[9]

The results presented in Table 4.9 show that the use of mark bits or larger output buffers has little or no

[9] Recall that because of the horizontal tuple layout across cells in the PPT design the number of cells to be searched for the indexed case is the same as in the sorted case and thus the execution times are the same.

Table 4.7
PPH - 19 Processors
100,000 Tuples of Size 100 bytes
Clustered Distribution of Selected Tuples
Sorted Case

Output Buffers #	Size	Execution Time in Revolutions, Selectivity Factor of Query .0001	.005	.10
2	1	82	115	319
2	5	82	88	129
2	1 with mark bits	81	82	109

Table 4.8
PPH - 19 Processors
100,000 Tuples of Size 100 bytes
Clustered Distribution of Selected Tuples
Indexed Case

Output Buffers #	Size	Execution Time in Revolutions, Selectivity Factor of Query .0001	.005	.10
2	1	2	36	247
2	5	2	9	57
2	1 with mark bits	2	3	38

Table 4.9
PPT - 7676 Processors
100,000 Tuples of Size 100 bytes
Clustered Distribution of Selected Tuples

Output Buffers #	Size	Execution Time in Revolutions — Selectivity Factor of Query .0001	.005	.10
2	1	5	8	36
2	5	5	8	31
2	1 with mark bits	5	8	36

impact on the performance of the PPT design when the data selected is clustered on a few tracks. This occurs because the performance of the design is limited by the bandwidth of the output bus. However, the use of mark bits has a dramatic effect on the performance of the PPH design for both occurrences of clustered data. For a selectivity factor of 0.1, use of mark bits improves performance by as much as a factor of 3 for the sorted case and 6.5 for the index case. We therefore opt for their use in the PPH design in the experiments presented below.

4.3.3.3.2. Comparison of the Three Organizations

The performance of the three associative disk designs is presented below in Tables 4.10 (sorted case) and 4.11 (index case) for queries referencing a relation with 100,000 tuples of 100 bytes. One consequence of the

Table 4.10
100,000 Tuples of Size 100 bytes

Clustered Distribution of Selected Tuples
Sorted Case

Selectivity Factor of Query	Execution Time in Seconds PPT	PPH	PPD
.0001	.076	1.33	13.6
.0005	.109	1.33	13.6
.001	.110	1.33	13.6
.005	.128	1.35	13.6
.01	.150	1.38	13.6
.05	.342	1.57	13.6
.1	.592	1.80	13.6

selected data clustering test is that performance of the PPT machine further degrades due to output channel contention. The PPH machine suffers, to a lesser extent, from the same problem (despite the additional buffer space) in the sorted case. Finally, the PPD design is unaffected since there is no channel contention of any sort.

Examination of Table 4.11 (the index test) yields some interesting results. The first is, that both the PPH and PPD machines are able to capitalize on the availability of the index information. Second, the performance improvement in PPH and PPD is such that PPT is still better but not superior. Finally, PPD is almost as good as PPH. We feel

Table 4.11
100,000 Tuples of Size 100 bytes

Clustered Distribution of Selected Tuples
Indexed Case

Selectivity Factor of Query	Execution Time in Seconds PPT	PPH	PPD
.0001	.076	.018	.333
.0005	.109	.023	.333
.001	.110	.030	.333
.005	.128	.050	.333
.01	.150	.082	.333
.05	.342	.324	1.00
.1	.592	.632	1.67

PPT: 7676 processors each with 2 buffers of size 1
PPH: 19 processors each with 2 buffers of size 1 and mark bits
PPD: 1 processor

that this implies that machines that use indexing to reduce the search space, such as DBC, should utilize a PPD approach to the Mass Memory component since it is considerably cheaper and less complex while attaining almost the same performance level as that of PPH approach.

4.3.3.4. Impact of Output Channel Service Policy

The final set of experiments we conducted were to investigate the impact of the service strategy of the output channel. We modeled two strategies: round robin and

first-come-first-served. Our expectations that no significant difference would be observed in the PPH machine because of the small number of processors involved were confirmed. We felt that some performance improvement should take place in the PPT machine that uses the first-come-first-served service policy. However, no such improvement was found because the execution time is dominated by the time to output the tuples.

4.3.3.5. Summary and Critique

In this section we have presented a model for associative disks and simulation results of three different associative disk designs using this model. Our results show that in general, as expected, PPT outperformed the other two. In testing the effect of the amount of output data on the performance of each machine we found no effect on the performance of PPD, minimal effect on the PPH's performance, and significant degradation in PPT's performance. Furthermore, it was shown that PPT is insensitive to various data organizations on the disk (e.g. an index on the qualified attribute) while both PPH and PPD were able to utilize such access mechanisms to significantly reduce the amount of data space searched. This result (with respect to PPH) is not surprising and was used by the DBC designers in the design of the Mass Memory component of their machine [69]. However, what we find interesting is that

PPD performs almost as well as PPH when there is an index on the qualified attributed. While this may seem perplexing to the reader we wish to point out that although very few cylinders are actually searched, most of them will output large amounts of data causing channel contention (in the PPH case) to affect performance in a very adverse way.

This result leads to a number of conclusions about associative disks. First, the use of indexing (as in DBC) in combination with a PPH or PPD design will provide good performance. We feel that if a cost effectiveness study of these designs (with the presence of indices) was performed, PPD would emerge as best (PPH will probably be a close second). Second, if parallel readout disks are to be employed, then the best associative disk design is a SURE-like [30] PPD machine which employs indexing, since such a machine incorporates the parallel readout capability of the PPH design while avoiding its channel contention pitfalls. However, this approach requires a very high performance processor in order to keep up with the disk.[10] Finally, PPD machines (without indexing or parallel readout disks)

---

[10] The SURE project used a Siemens disk with 9 parallel read heads. If a SURE-like architecture is to be used in an IBM 3330 we estimate that the processor will have to operate at approximately 23 MIPs. While such processors are probably not within the realm of today's technology it should be noted that the processor will have a very simple instruction set (simplifying its organization). Also, the types of operations processed allow for a pipelined implementation.

provide a very cheap and simple way of filtering out undesirable data. There are numerous applications where such a feature can be utilized. One example is the Research Storage System (RSS) of System R which is responsible for eliminating undesired tuples from the data stream examined by higher levels [60].

Although not presented here, our study was extended to cover PPT organizations that employ off-the-shelf bubble memory chips. Although, these chips are very slow when compared with disks we were able to show that their performance was better than PPH for the clustered case and at times even better than PPT. The reason for this is the ability of the chips to start and stop rotation of bits (bubbles) at will. Thus, at the time that a processor is blocked it can stop the movement of its bubbles, and resume processing immediately after having one of its buffers emptied.

Our models have a number of shortcomings. The first is that they do not include the cost of using indices. We feel that a thorough study of the maintenance and access cost of indexing needs to be undertaken in order to confirm our statement concerning the relative performance of the three machines. Second, our model can be improved by incorporating positional sensing hardware in the disks. This feature would enable processors to begin scanning the data at any sector boundary on the disk instead of waiting for a specific bit position on a track. In our simulation we model the track-to-track seek time with the formula:

seektime = 10 + numtracks * .01

The value computed is then rounded up to the next multiple of the rotation time. With positional sensing disks this would not be necessary. The IBM 3330, which we modeled, has a rotation time of 16.7 ms. Thus, incorporation of this feature into the simulation means a net savings of about 6.6 ms per cylinder.

While the performance of the PPD design will indeed improve by almost 6.6 ms for each cylinder processed, the PPH design will not, in general improve as much. This is due to the (observed) fact that PPH is able to empty most of its full buffers during the additional rotation in between cylinders. Using positional sensing devices will cut down on the idle time in between cylinders and thus on the time the processors have to empty their buffers. The net effect, we feel, would be to still cut down on the search time but to a lesser degree than in PPD. It should be noted that this savings does not apply to PPT devices.

A final problem with our models is that the disk employed, the IBM 3330, is old. New disks, such as the IBM 3380 [31], have a much larger storage capacity due to higher storage density per track (47,476 bytes per track as

opposed to 13,030 bytes per track) and more cylinders per disk (more than twice as many as in the IBM 3330). We believe that such disks will tend to favor the PPD design because more bytes per track implies more tuples per track and consequently means more output channel contention.[11] Another reason for investigating the new disks is that they provide a small amount of storage space accessed by fixed heads. This space can be used to store the index. The IBM 3380 provides two cylinders with this capability (approximately 1.5 Mbytes of storage), this is about 0.25% of the total disk storage. An analysis of the storage requirements of indices is required before the use of the fixed head storage in the IBM 3380 disk can be assessed.

4.4. Conclusions

In this chapter we have considered two important issues necessary for the design of a database machine that incorporates both on-the-disk and off-the-disk processing capabilities. We first looked at the problem of processor assignment in DIRECT. We compared four different strategies and showed that a data-flow strategy outperformed the others because of its ability to adapt itself to the

---

[11] Another feature of the more modern disks is their higher speed data transfer rates, 3.0 Mbytes/second for the IBM 3380. Such high data rates place further constraints on the processor speed. For example, in PPH or PPD the processor would have to process instructions at a rate of 10 MIPs rather than 2.4.

data access patterns exhibited by the various benchmark query programs. We have also shown that all the strategies require such a large number of messages as to cause the back-end controller in DIRECT to become a bottleneck. The data-flow strategy was worse in that respect than the others.

In comparing the performance of DIRECT for the various processor allocation strategies we used some of the algorithms developed in the previous chapter. For these algorithms, there are two types of communications that an architecture must support: point-to-point transfers and broadcasts. However, using the data-flow processor allocation strategy we have shown that the majority of communications between processors (or processors and memories) is of the broadcast type. Thus, any architecture that is to use the algorithms of Chapter 3 and the data-flow processor allocation strategy must support an efficient broadcast facility. It is important to notice that this facility must allow for a multiple number of broadcasts simultaneously.

The second issue we considered was a comparison of the different associative disk types that have been proposed. Our intention was to evaluate these organizations under a number of realistic assumptions and obtain some information about their relative performance for a number of different

queries. Our results have shown that one of the simpler organizations, processor-per-head or processor-per-disk should be used.

CHAPTER 5

THE PROPOSED ARCHITECTURE

5.1. Introduction

In Chapter 3 we proposed and analyzed a number of parallel algorithms to be used in our architecture. Then in Chapter 4 we examined a number of different query processing strategies for DIRECT using these algorithms. We concluded that a data-flow strategy was superior because of its sensitivity to the data access patterns of the query programs. However, we also showed that this strategy requires a large number of messages between the processors and controller. In this chapter we shall use these results in the design of a new MIMD database machine which will support all the relational algebra operations as defined in INGRES [43] and described in Appendix A.

DIRECT certainly possesses the hardware and software capabilities required for the implementation of the algorithms described in Chapter 3 and supporting the data-flow query processing strategy. There are, however, numerous reasons why a new architecture should be designed rather than improving DIRECT.

First, Hawthorn and DeWitt [3] have shown that DIRECT, and in general, any machine that cannot process simple

instructions such as selections and scalar aggregates directly on the mass storage device, cannot support their efficient implementation at all. Second, in Chapter 4 we have shown that the number of messages required to process a join (this can be generalized to projections and aggregate functions) may cause the controller to become a bottleneck. In fact, we can show, analytically, that for a join the number of messages will be quadratic in the number of pages of the two relations. Thus, we see that it is insufficient to provide a broadcast mechanism that will reduce the number of I/O operations; some means of reducing the number of control messages is also required. One of the reasons for the large number of messages is that all inter-processor communication takes place through the shared memory cache which is controlled by the back-end controller. A final criticism, one that has appeared several times in the literature, concerns the cross point switch that connects the processors to the cache memory units. For example, Goodman [46] criticizes this switch for its cost and poor expansibility features.

What features should the new architecture possess in order to overcome these, and other difficulties? Clearly, for the efficient execution of simple selections and scalar aggregates some processing capabilities must be associated with the mass storage devices. Second, if the algorithms described in Chapter 3 are to be employed then the function of the controlling processor must be distributed. Third, the broadcasting capabilities of the cross point switch in DIRECT must be retained, although a different, cheaper and more easily expansible, implementation for the interconnection device must be found. Fourth, the machine (particularly the interconnection device) must permit MIMD activity in order to support a high number of transactions per minute. Finally, the data-flow processor allocation strategy described in Chapter 3 was shown to be the best and should be utilized.

The remainder of this chapter is organized as follows. We begin with a discussion of the logical machine organization. We present a rationale for assigning various control and non-control functions to component types and outline the actions that these components will perform. Next, we describe the physical organization which we propose. We then describe the various steps taken by the machine components during the execution of a particular query. We conclude with an overview of the implementation of the data integrity functions.

5.2. Logical Organization

5.2.1. Description

In [67] we presented a preliminary design for an MIMD database machine that employs a data-flow query processing strategy. In this design the controlling functions of the back-end controller in DIRECT were distributed. Two types of processors were designated for controlling the execution of instructions: a Master Controller (MC) and an Instruction Controller (IC). The MC is responsible for:

(1) Communication with host computers.
(2) Initiating instructions.
(3) Performing data integrity maintenance functions.
(4) Controlling resource allocation in the machine.

The IC is responsible for controlling the execution of an individual instruction. A third type of processor, an Instruction Processor (IP), is responsible for executing code as instructed by an IC. A database machine configuration of this type would consist of a single MC, several ICs, and a pool of IPs.

At instruction initiation time the MC picks an IC and assigns a number of IPs to it for the execution of the instruction. The IC is responsible for getting the data required for the instruction execution from mass storage or from other IC groups and allocating it to its IPs.

5.2.2. A Sample Instruction Execution

A join operation would be executed on this architecture in the following manner. First, the MC would pick an

IC to control the execution of the instruction. The IC will receive page tables and other descriptive information about the two relations to be joined. The MC will also attempt to allocate the "optimal" number of IPs to the IC. Next, pages of the outer relation (see Section 3.3.4 for a description of the algorithm) would be fetched from mass storage and distributed, one at a time, to the IPs.

After the outer relation pages have been distributed to the IPs the inner relation pages must be broadcast. Each IP sets up an "inner relation control" (IRC) vector which is used to monitor which pages of the inner relation it has seen. Initially the vector is empty. Upon receipt of an inner relation page an IP creates an entry for that page in its IRC. The IC also reads the page and stores it in its temporary storage area (see Section 5.3.2).

Should an IP fill its output buffer during the instruction execution it must flush it out. Since in this case the query packet consists of a single instruction (the join) the output page is sent to the IC where it is temporarily stored. At the end of the instruction execution the IC will collect and reorganize the result pages from all the IPs and pass them on to the MC. The MC will forward them to the host computer from which the query originated.

After the inner relation has been broadcast in its entirety, the IC solicits a status report from all its IPs. Each IP informs the IC whether it missed any inner relation pages. An IP can miss an inner relation page while preparing an output buffer for transmission to the IC (e.g. sorting it). In the event that some (possibly all) of the inner relation pages are needed (not necessarily by the same IP) they are rebroadcast by the IC.

After an IP has flushed its output buffer it informs the MC that it is ready for a new task assignment. Similarly, after the IC has finished reorganizing the output relation and sending it to its destination it also informs the MC that it is idle.

5.2.3. Comparison With DIRECT

The execution of a join on DIRECT is somewhat similar to the description above. First, outer relation pages are distributed to the IPs. Next, the inner relation pages are broadcast. Full output buffers are flushed out to temporary storage (a CCD module in DIRECT). There are, however, several important differences.

First, in DIRECT distribution of both outer and inner relation pages is on demand by the IPs. Therefore, the number of control messages that are required to control the execution of a join is quadratic in the size of the two relations. In our architecture, messages are exchanged only at prespecified points in the execution of the operation: at instruction set up time, outer relation pages distribution time, and at the end of inner relation pages distribution time. The number of messages required is linear in the size of the outer relation.

Second, the back-end controller of DIRECT is its only controlling processor. While this may not prove to be a problem when there is only a single instruction executing, we have shown that when a number of queries are active the back-end controller is a bottleneck (see Section 4.2.4.5). In the proposed architecture this cannot happen because the control of each instruction is overseen by a different IC. Also the number of messages required is linear in the size of the outer relation rather than quadratic in the size of both relations.

Third, although not described, the interconnection device used by the proposed architecture is considerably simpler, cheaper, and more easily expansible than the cross point switch of DIRECT. One advantage that the cross point switch has is that it allows a multiple number of point-to-point transfers to take place simultaneously. This cannot be done on broadcast buses. However, we shall subsequently show that such a feature is not of great importance in our organization.

5.2.4. Outline of Architecture

In this thesis we propose to adopt the processing hierarchy of the MC, IC, and IP. In addition to these components, there will be several associative disks (see Section 4.3). All communications with host computers will be handled by the MC which will also be responsible for instruction initiation and resource allocation. The associative disks will be responsible for the execution of simple selections, scalar aggregates, and under certain conditions, some of the update operations. Each IC and its allocated IPs will form an IC group for the duration of an instruction execution. The hardware must allow several IC groups to be active at the same time (i.e., the simple interconnection device used in the illustration above is insufficient). Processors in each IC group should be able to communicate with each other regardless of activity in other groups. In the next section we describe this hardware organization.

5.3. Physical Organization

There are three issues that must be addressed in this section: the interconnection between the ICs and IPs, the implementation of storage for temporary relations, and the organization of the associative disks. We begin with a discussion of the interconnection device to be used. We then argue against the use of some of the alternative interconnections that could be employed. Next we offer a solution to the temporary storage problem. We close this section with a discussion of the associative disk implementation.

5.3.1. Interconnection Device

The interconnection device must be able to support MIMD activity and communications of two types: point-to-point and broadcast. Recall that a side-effect of our use of limbs (see Section 4.2.1.4) for scheduling instructions for execution is that broadcasts are used much more frequently than point-to-point transfers.

Maglaris and Lissack [70] have suggested using a broadband, coaxial cable broadcast bus that uses frequency multiplexed, RF-modulated channels to allow for several simultaneous communications over a single bus. Each channel, operating at a different frequency, can support a data transmission rate compatible with the processor bus bandwidth, say 10 Mbps. Thus, several simultaneous communications can take place. A single, specially designated, channel, which we term the control channel, is used for coordinating activities on the machine. For example, processors that wish to establish a link must obtain a reserved channel through the use of the control channel. Once a reserved channel has been assigned to them they can switch frequency and proceed with their "session",

undisturbed, over their own reserved channel.

The transmission technology used is CATV. This technology can provide a transmission capacity of 400 Mbps. It is not clear at this point how many channels, each providing a communications medium of 1 to 10 Mbps, can be supported by this technology. Levy and Rothberg [71] claim that a transmission bandwidth of between 2.5 and 3 Hz is required per bit. Thus, only approximately 15 channels, each supporting a 10 Mbps transfer rate, can operate simultaneously.[1] Others, ([70], for example) allege that only 1 Hz per bit is required, leading to the figure of 40 channels.

In our architecture the MC will be responsible for all resource assignment. It will always monitor the control channel. In its idle state, any processor (IC, IP, or associative disk) will be listening to the control channel. Figure 5.1 shows the database machine in an idle state. All the processors (MC, ICs, IPs, and associative disks) are monitoring the control channel.[2] At the time that the MC decides to initiate the execution of an instruction (actually a limb) it picks an IC to control it and

---

[1] Alternatively, 150 1 Mbps channels can be supported. The actual bandwidth required will depend on a number of factors such as processor speed and can only be determined at a later stage of the design.

[2] Although not shown, disks are associated with each IC.

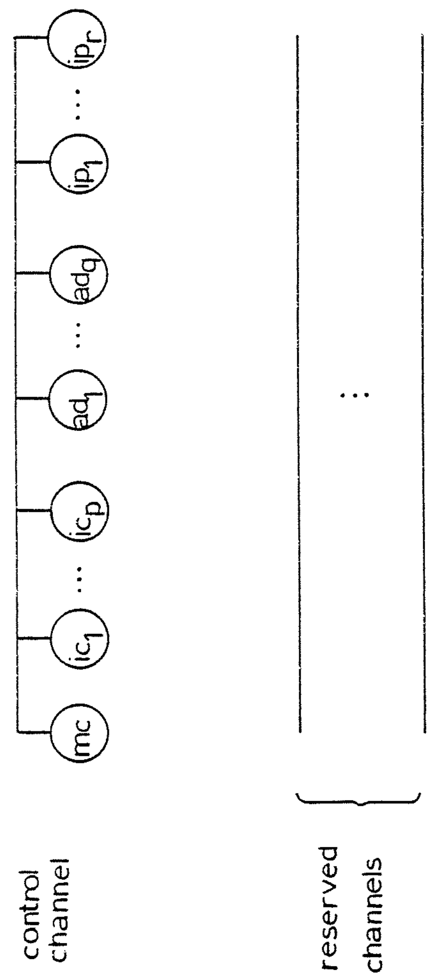

Figure 5.1: Machine in Idle State

allocates some IPs and a channel to the IC. The MC then sends the selection operation and the channel address to the associative disks that contain the data for this instruction. The processors switch frequencies to operate in the assigned channel (their operation will be described in subsequent sections). Figure 5.2 shows the machine with two active instructions, each utilizing a separate channel. Note that not all the machine resources are being utilized. At the time that a processor (IP, IC, or associative disk) terminates its current task it switches frequency back to the control channel, informs the MC that it is ready for a new task and waits for a new assignment message.

Since the interconnection we described above is ETHERNET-based [72] there are a number of problems that we must address, in particular, reliable delivery of messages. There are a number of different reasons why a message can be lost on an ETHERNET-like broadcast bus. We thus require that an acknowledgement of receipt (which may contain a reply) be sent by the receiver of any message. As will be seen later, in certain cases: broadcast of a stream of data pages, we relax this condition to allow for a single acknowledge message at the end of the stream.

What other interconnection devices can satisfy these requirements? One possibility is a switch such as a cross point switch [73] or a multi-stage network such as the

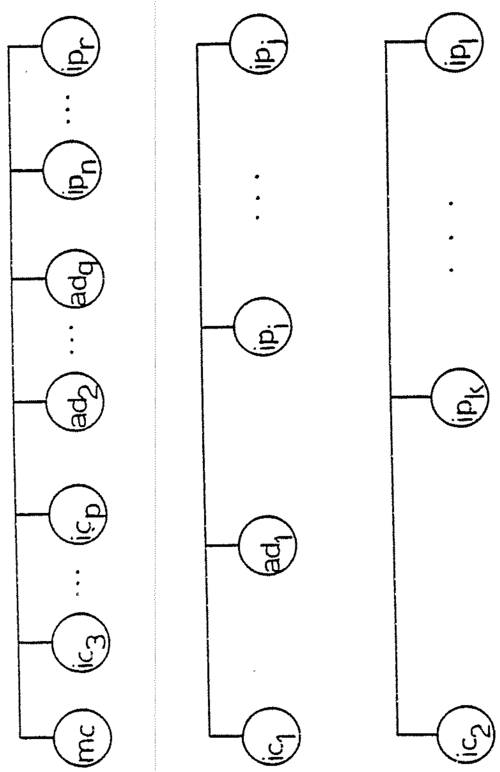

Figure 5.2: Machine Executing Two Instructions

banyan [74]. Such switches can be characterized as follows.[3] Two entities that wish to converse must establish a link. Links are physical and are obtained through requests to a central controller. Once a link has been established, data (message) exchanges proceed efficiently since, in effect, a hard-wired line exists between the two entities. Switches are expansible.

Although most of the properties described above are desirable from our point of view, we reject switches as viable interconnection devices because of the central control that is inherent in their operation. Our experiences with the DIRECT simulation have shown that the controller of a switch can easily become a bottleneck (see Section 4.2.4.5).

A second possible alternative is a high-speed ring. Rings possess several favorable features but have two major problems that make them unacceptable. The first problem is expansibility. As nodes are added to the ring the time required for a message to travel from one node to another increases linearly. A second, and more important problem, is that rings lack efficient broadcast capabilities.

---

[3] Our characterization is applied to all switches and as such is not always precise. For example, some types of switches are more easily expansible than others.

Taylor [75] suggests the implementation of a local network that will include features from both rings and local broadcast networks. The interconnection bus is assumed to be constructed from optic fibers; lasers will be used to transmit packets. To handle the high data transfer rates in the taps, optic fiber buffers are used to hold the data until the tap's logic can access it a bit at a time. Each tap has under its control a repeater whose purpose is to amplify the signal as it travels on the optic fiber. By judicious control of the repeater the tap can either read a message exclusively, in shared mode, or not at all. It is postulated that taps that can provide such capabilities for data transmission rates greater than 1 Gbit could be constructed in the middle to late 1980's.

Another feature of this design is that several transmissions can take place simultaneously using different frequencies. Thus, this interconnection device appears to be the most suitable for our needs. However, it requires what we term "exotic technology" which will most likely not be available "off the shelf" within the next year or two. We have therefore chosen to adopt the frequency multiplexed broadcast bus approach suggested in [70]. Should the technology described by Taylor in [75] become commercially available we may wish to modify our design.

5.3.2. Temporary Storage

Each IC must have some amount of memory for the storage of pages in anticipation of their use by the IPs. Two different uses for such a cache memory arise in this context. The first is for saving outer relation pages that could not be consumed because of an insufficient number of processors (see query execution example in Section 5.4). This problem can occur if an operation, say a join, which is generating the outer relation for the subsequent operation, has produced more pages than there are processors assigned to that operation.[4] The second, and more frequent, use is the saving of inner relation pages until it has been ascertained that every page has been seen by every processor that will need to see it.

The presentation and analysis of our algorithms in Chapter 3 reflects our belief that, in the general case, there will be an insufficient number of processors available for the execution of an operation. In such an event, the algorithms for the "complex" operations (join, project, aggregate functions) will require multiple passes over the data. In particular, the inner relation will have to be broadcast once for each pass. This means that the entire

---

[4] Recall that if an operation is producing the relation to be used as the outer relation in the subsequent operation then both operations are on the same limb in the query tree and the processors executing the child operation will be assigned to the parent operation.

inner relation will have to be stored in temporary storage for the duration of the instruction execution.

An important point to realize is that access to the inner relation pages will be sequential and in a well defined order. Thus, the temporary storage can be organized in an hierarchical manner. A small number of pages would be kept in a fast memory, from which they could be read with little or no latency. The remaining pages, those that will not be needed immediately, could be kept on a slower memory device and brought into the higher level memory in anticipation of their use. It is not clear how large and how fast the higher level memory need be. Some factors determining this are:

(1) The processing speed of the IPs
(2) The transfer rate of the interconnection device
(3) The size of pages

The secondary memory can be implemented using a small disk.

Disks with storage capacities of 40 and 60 Mbytes are available on the market for prices as low as $6,000. One problem that may arise is whether an IC will become a bottleneck in attempting to control its assigned IPs and the temporary storage memory. We shall address this problem in a subsequent section after the role of the various components in the machine has been presented in more detail.

5.3.3. Implementation of the Associative Disk

Databases will reside on several associative disks. In section 4.3 we described and compared three different approaches to associative disk design. The results show that the processor-per-track approach provides the best performance under nearly all conditions. Such organizations require either fixed head disks or one of the new memory technologies. Fixed head disks constitute an (almost) obsolete technology and provide a storage capacity which is a small fraction of the capacity of moving head disks. The new memory technologies that may someday replace disks, such as MBMs, possess several attractive features that lend themselves to their use in a processor-per-track organization. However, the cost per bit of such devices is still two to three orders of magnitude more expensive than it is for moving head disks [76]. Thus, it is not expected that these memory technologies will replace moving head disks in the near future. We have therefore chosen not to consider processor-per-track associative disks as a possibility for use in our architecture.

The comparison of processor-per-head and processor-per-disk organizations (see Section 4.3) showed that the processor-per-head organization is superior. However, it has also been shown that when indexing is used the performance differential is not as significant. As of now no comprehensive study of the use of indexing in a multiprocessor environment has been undertaken which shows their benefits and disadvantages.[5] We have thus not incorporated indexing into the present machine design. However, at a later time, when their use is more clearly understood, this may be a direction we would want to pursue. Therefore, the associative disk should be an easily replaceable part of the machine. We propose the following organization.

Each associative disk will be an independent unit. It will consist of the disk, its associated processing element(s), a controlling processor, and some amount of work space. The controlling processor will have access to information about the contents of the disk, such as relation and page tables. It will be responsible for communication with other components of the architecture, such as the MC and the various currently active ICs. The MC will send to it lists of tasks to be executed. These will be scheduled by the associative disk controller based on such criteria as the current cylinder being scanned. The temporary storage will be used to hold pages that need to be sent, and for results of intermediate computations as described later.

[5] It should be noted, though, that DBC and HYPERTREE use indexing, although in a limited sense.

If indexing[6] is to be incorporated into the associative disk it will be done internally and in a manner invisible to the remainder of the machine. We expect that additional memories and processors will be used for this purpose. These additional resources will also be controlled by the same associative disk controller. Thus, the only effect of such changes to the architecture, as seen by the other machine components, will be to receive data at a faster rate.

The operations to be supported by the associative disk are selections, scalar aggregates, simple updates (disallowing the introduction of duplicates) and the attribute elimination part of projections. By simple updates we mean update operations that define the tuples to be modified using a simple predicate (i.e. attribute = constant).

An associative disk's output consists of unsorted pages which may contain duplicates. Each page is prefixed with a header identifying itself by a number and its relation by name. These pages are placed on a channel specified by the MC at the time the instruction is assigned to the associative disk.

[6] In discussing indexing, we refer to their restricted use as in DBC. If indices are to be used in performing operations other than selections, the organization of the machine will most likely undergo major revisions.

5.4. Query Execution

Although the architecture was described in considerable detail in the previous section, additional information needs to be specified so that query execution on the machine can be better understood. In this section we describe the execution of one query in terms of the actions taken by each component type. Some of these actions are obvious and need little or no explanation. Other actions will require more exposition. It is felt that only by going through a query execution will all the details and the rationale for the actions taken by the components be understood.

The query we have chosen to describe consists of two selection operation followed by a projection on one of the result relations and a join. Its tree representation is shown in Figure 5.3a. The query consists of two limbs named A and B as shown in Figure 5.3b. Since limb B produces data that is used by an instruction in limb A it must be scheduled first. In Figures 5.4a-5.4c we show the execution of the projection instruction (limb B) by an IC group (the selection will be done by the associative disk on the fly).

We assume that the associative disk produces 5 output pages. The MC, however, allocates only 3 IPs to the execution of the instruction (perhaps it made a bad estimate of

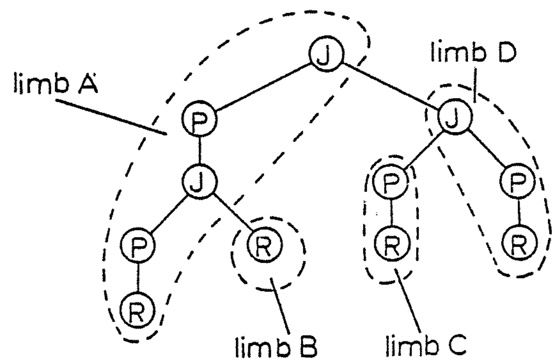

Figure 5.3: Query Tree for Example

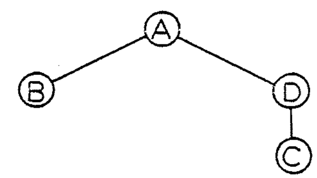

Figure 5.3b: Precedence Tree for Example

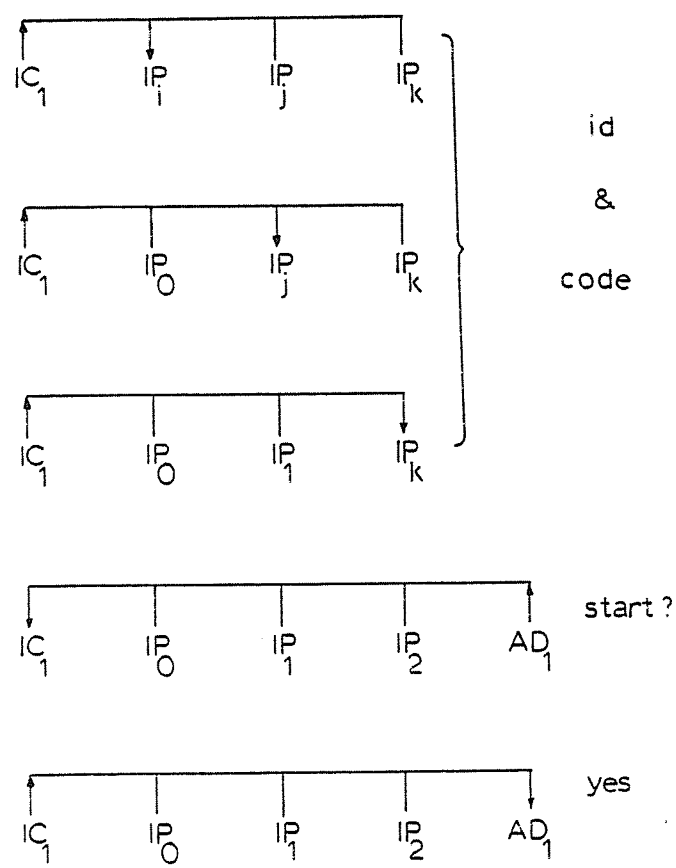

Figure 5.4a

the result relation size or it simply did not have a sufficient number of processors available). Therefore, two phases will be required to execute this instruction. The sequence of messages between the IC, IPs, and associative disk required to set up the instruction execution is shown in Figure 5.4a. Initially the IC sends a message to its IPs assigning them IP identifiers for the duration of the instruction execution. These identifiers will be increasing integers beginning with 0. The code to be executed is also included. In the event that this instruction produces data used by another instruction in another limb the channel frequency of the parent limb is also enclosed.[7]

In Figure 5.4b we show the message exchanges required to distribute the outer relation pages. Once all outer relation pages have been sent by the associative disk it can switch frequency to the control channel and receive its next task. The associative disk has no knowledge of the "names" of the IPs assigned to this instruction or even their number. It therefore broadcasts each page to all processors on the channel. Pages are identified with the relation name and a unique page number. These numbers are increasing integers beginning with 0. Every IP and the IC

[7] This information may not be available at limb initiation time. In such cases the IC will have to get it from the MC when the need for it arises. For the purposes of this description we shall assume that the IC has this information.

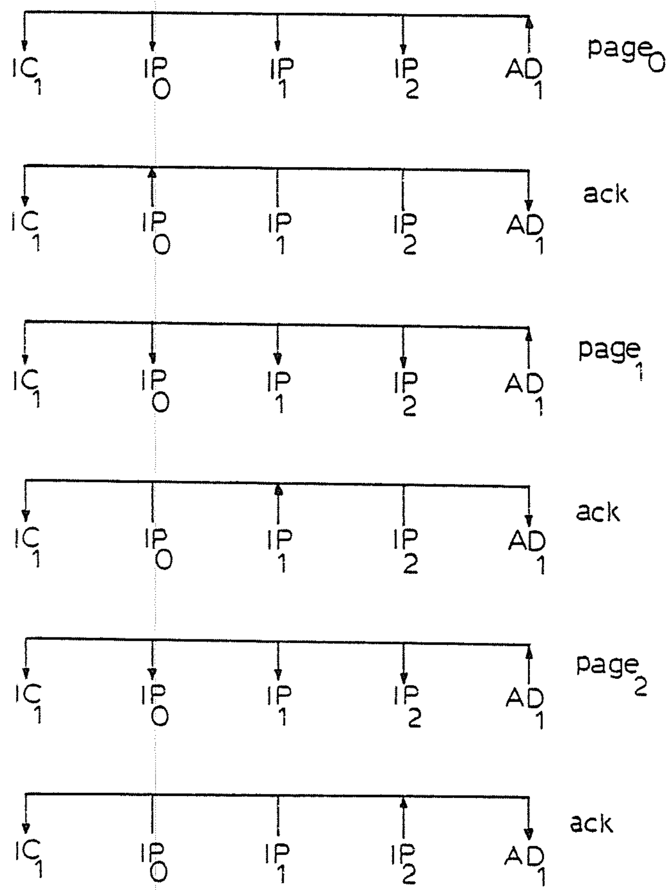

Figure 5.4b

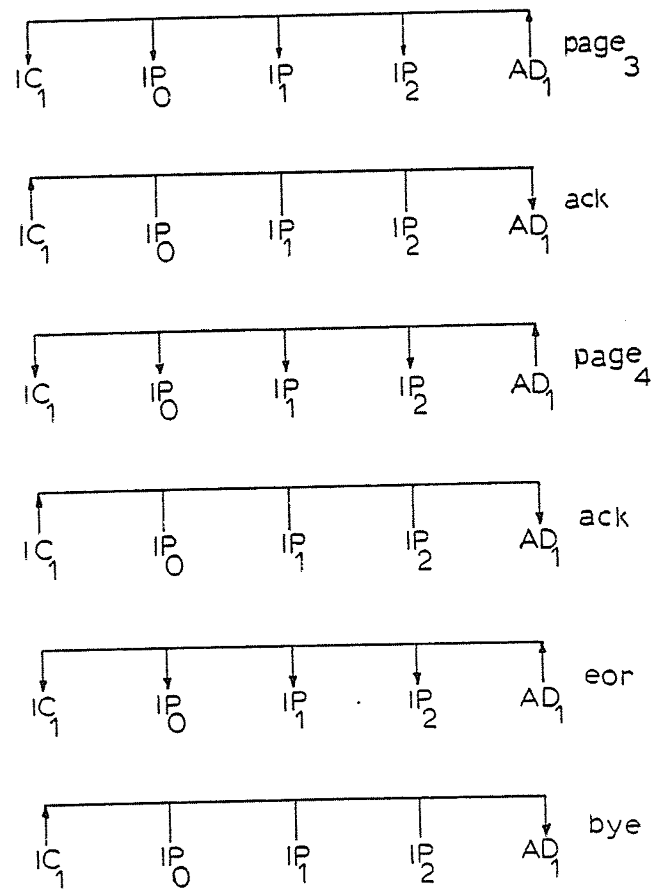

Figure 5.4b - continued

read each page broadcast by the disk. However, only IP_i keeps page $page_i$ and acknowledges its receipt with a message addressed to the disk controller and the IC. The IC, by examining the source of the acknowledgement and the page identifier can ensure that the correct IP read the page.

Since in our example the number of pages is greater than the number of IPs by 2 the IC will read the additional pages and acknowledge their receipt. The pages will be stored in the IC's temporary storage area and will be distributed to the IPs in subsequent phases of the execution of this instruction.

Figure 5.4c shows the sequence of messages required for the execution of the broadcast step of the algorithm. This step is initiated with a message from the IC to all the IPs describing what is to follow. In this case first the IC will broadcast pages 4 and 3. Each IP will read the pages (one at a time) and search for duplicates in its own page. In the event that any duplicates are found they are eliminated from the IP's own page (see Section 3.3.3 for the description of the algorithm). Next, IP_2 broadcasts its page. IP_1 and IP_0 repeat the preceding duplicate elimination step. IP_2's page is now purged of any duplicates and can be used in the subsequent operation. IP_2 obtains a page number for its output page from the IC (not shown in the figure) since this page is a part of a new relation.

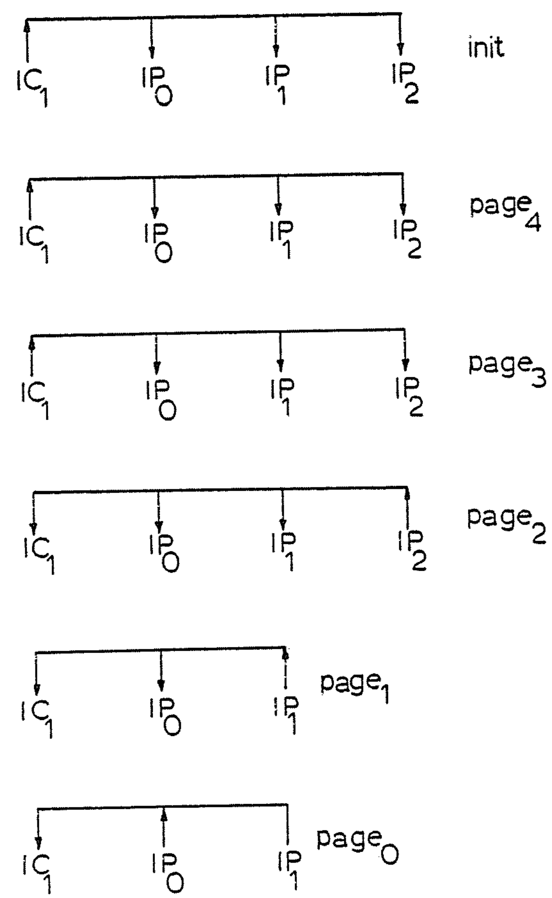

Figure 5.4c

The IP then switches frequency to limb A's channel, broadcasts its page, and waits for an acknowledgement. After the acknowledgement is received the IP has terminated its part of the limb execution and informs the MC that it is idle.

After eliminating duplicates between its page and $page_2$ IP_1 executes the same procedure. IP_0 is the last processor in the group and can therefore obtain a page identifier immediately after it has eliminated duplicates between its page and $page_1$. Both IP_1 and IP_0 return to the channel after broadcasting their page on limb A's channel for execution of the second phase of the algorithm. Since the steps taken by the various processors in this phase are the same as in the previous phase we will not show them in a figure or describe them.

It should be noted that at the time that an IP broadcasts its page to the other IPs for duplicate elimination the IC also reads that page. The page is stored in the IC's temporary storage. After all the IPs have finished executing the instruction the IC switches frequency to the subsequent instruction's channel and checks with its IC whether the correct number of pages have been received. If not, a backup copy of each page produced exists in the producing IC's temporary storage and can easily be retrieved.

In Figures 5.5a-5.5d we show the execution of limb A by a separate IC group. We stress the fact that the execution of this limb can proceed independently of and concurrently with the execution of limb B. We assume that a sufficient number of processors was allocated to IC_2 for the execution of limb A. We label the IPs executing limb A IP'_0, IP'_1, and IP'_2 in order to differentiate them from the IPs executing limb B.

Figure 5.5a shows the state of execution of limb A after the first two pages of the selected relation have distributed to IP'_0 and IP'_1 but before the third page had been sent to IP'_2. IP_2 (executing limb B) then broadcasts its output page. Since the IPs executing limb A are not ready for it only IC_2 will read it and acknowledge its receipt.

Figure 5.5b shows the state of execution after all the outer relation pages have been distributed. IP_1 (executing limb B) is shown broadcasting its page. At this time all the IPs executing limb A are ready to read it and do so. IC_2 also reads it and is the only processor to acknowledge receipt.

Figure 5.5c shows IC_1 sending an end of relation (eor) message to IC_2. This message includes the number of pages that IC_2 should have received. If the number received is smaller than the number that should have been received IC_1

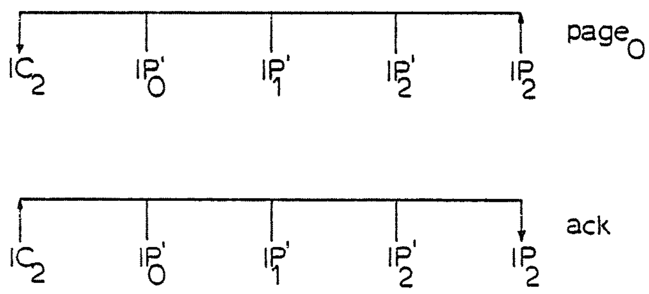

Figure 5.5a

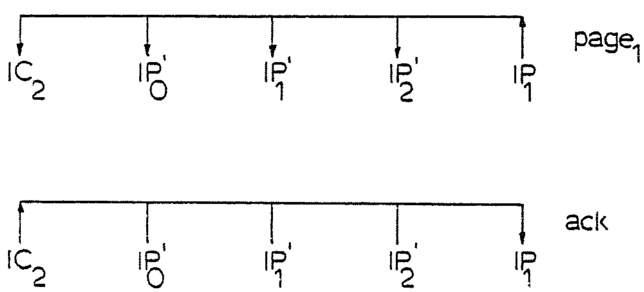

Figure 5.5b

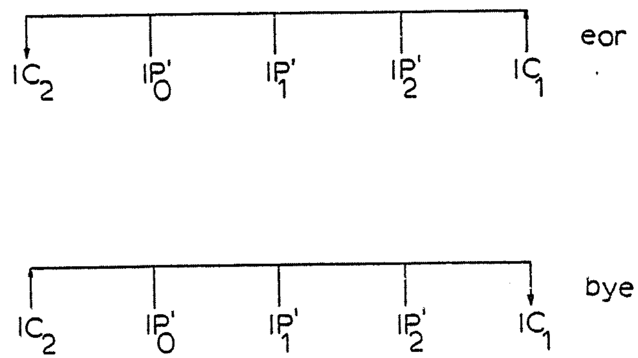

Figure 5.5c

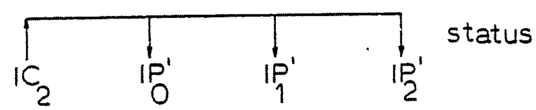

Figure 5.5d

can retrieve the missing pages from its temporary storage and send them to IC_2.

In Figure 5.5d we show IC_2 soliciting a status request from its allocated IPs. The request includes the number of inner relation pages that each IP should have received. Each IP will respond with a message indicating whether any page are missing. In our example, each IP will request $page_0$ since none of them were ready to read it at the time it was broadcast. Other pages may also be requested due to other problems.

At the end of this instruction execution IC_2 will collect the result page from the IPs, format the result relation, and send it to its destination.

5.5. Integrity Issues

Operating in an MIMD environment allows a large number of programs to access the same data "simultaneously". Therefore a number of mechanisms to guarantee data integrity are required. Specifically, there must a concurrency control policy and means for recovery from crashes of various components without affecting the stored database.[8] In this section we outline the mechanisms we propose

[8] The issue of providing stable storage on disks has been addressed in the literature (see [77] for an example), we ignore it in this discussion and focus on recovery from the failure of processors.

to use in our machine.

5.5.1. Concurrency Control

A large number of algorithms for concurrency control have been proposed in the literature. These cover a wide spectrum, from centralized locking algorithms to distributed non-locking ones. Our concurrency control algorithm need only apply to source relations in the database since temporary relations are created exclusively per packet. In our machine access to these relations is performed only by the associative disks. Therefore, placing the concurrency control function in the associative disk controller appears to be the most logical. Clearly, in the event that a query only references source relations that reside on a single disk the associative disk controller will have all the information necessary to make a decision concerning that query. However, it is certainly the case that queries will reference relations that reside on different disks,[9] thus our algorithm must also handle this more difficult case.

[9] This brings up an interesting research question: how should we place relations on disks? One reason for placing relations that frequently appear in the same query on one disk, is to simplify the concurrency control task. However, if these relations are placed on separate disks, a higher degree of parallelism can be attained in the execution of the queries than if they were placed on the same disk. In fact, the problem is to determine whether the cost of the additional parallelism inherent in distributing the data as much as possible is not offset by the cost of the additional overhead incurred in maintaining the integrity of the data. Clearly, some empirical knowledge

We therefore must provide a facility for communication between the associative disk controllers. At each point in time an associative disk will be serving at most one IC group. The only entity in the machine that knows what each associative disk is doing is the MC. Thus, any contact between two associative disk controllers requires assistance from the MC. Rather than having the second disk controller find out over which channel it can communicate with the first, we have chosen to implement the inter disk controller communication through a "mailbox" mechanism in the MC. This means that each disk controller will have to periodically check its mailbox in the MC.[10]

At transaction commit time the controlling IC will send a commit message to all the disk controllers that processed the leaf nodes in this transaction through their mailboxes in the MC. Since we require each associative disk controller to check its mailbox frequently we are guaranteed that each of the participating disks will process the commit message soon after it was sent. Each disk will perform conflict analysis locally and send its diagnosis to the MC. The MC will be responsible for doing the

concerning the rate of updates to the relations, the frequency of use in retrieval queries, the cost of messages etc. will have to be used in making this judgement.

[10] Recall that we require each associative disk controller to periodically check with the MC whether there are any ICs that need to write to it. Therefore checking the mailbox can be done at almost no additional cost.

final analysis and informing the disk controllers of its outcome. In the event that a deadlock situation is detected an abort message will be sent to the disks and the MC will have to restart the transaction.

The particular policies and algorithms used for the above are not discussed here for a number of reasons. First and foremost is that of a number of possible policies it is not clear which is best suited for what type of environment. No empirical comparisons of the various policies exist although Wilkinson [78] currently is studying a number of these for a local broadcast network. Second, choice of the algorithms and policy is independent of the architecture and thus is irrelevant to the main purpose of this thesis.

5.5.2. Recovery

In this section we consider the effects of the failure of five component types on the operation of the architecture. These are: IP, IC, associative disk controller, the local communication network, and the MC. It is clear the in the event that the MC or the local communication network fail the machine cannot continue operating. Failure of a number of IPs or ICs should not cause the machine to cease operations, although performance will suffer. Finally, failure of an associative disk controller in effect reduces the amount of data that can be accessed. Thus, only queries that use data on other disks can executed. Our description is informal and is only intended to show that at various points during the execution of the instruction some processor is able to detect the failure of another processor and thereby abort the execution of an instruction. Clearly, a more rigorous specification is required. However, our purpose at this stage is only to show that failure detection and recovery from certain failures is possible in the proposed architecture.

In general, failure detection leads to the abortion of the entire packet which contained the instruction executed with the faulty component. For retrieval packets (ones that do not contain any update instructions) this seems unnecessarily harsh. Clearly, a more elegant, though more expensive, alternative would be to attempt to reproduce only the data lost through the particular component failure.[11] This can be done if sufficient information concerning the source of the data in each page is kept. It should be noted that the amount of information per tuple grows for each additional operation. The actual cost of this approach needs to be investigated to determine whether the additional overhead is indeed offset by the savings accrued. Another possibility is to store some of the temporary relations at various checkpoints in the execution of a query. Then, at recovery time execution would have to be restarted only from the last checkpoint.

In the course of an update operation, new pages are written to disk but not incorporated into the relation until transaction commit time. At that point all the new pages replace the old pages in a single atomic operation. If an abort signal is received by the associative disk controller at any time during the execution of an update operation any new pages already written to disk (but not committed) are simply removed.

<u>5.5.2.1. IP failure</u>

We consider the effect of an IP failure at three different points during an instruction execution: while processing an outer relation page, an inner relation page, and an output relation page. We show that communication between processors at various points in the instruction execution act as checkpoints, used by the controlling IC to determine if any of its IPs have failed. During the outer relation pages distribution time each IP is responsible for acknowledging the receipt of a page. Since the IC is eavesdropping on the traffic over its channel it can assess whether any IP has failed. When inner relation pages are distributed, it is the IC's responsibility to acknowledge receipt. However, since after the entire inner relation

[11] This is particularly relevant in the event of a single IP failure.

has been broadcast each IP must respond to the IC request for a status report, this response can be used by the IC as a checkpoint.

Finally, if the IC group is producing an inner relation to be used in a subsequent operation, each IP must broadcast its output page to the IC group executing that instruction. Before doing so, each IP must obtain a unique identifier for its page from its controlling IC. We require that each producing IC send the number of pages produced to the consuming IC after all of these pages have been broadcast. The consuming IC checks this number against the number of pages received and can thus determine whether any IPs have failed between the time that they requested a page number and the time they sent their page.

5.5.2.2. IC failure

Detection of an IC failure can be performed by one of many different components in the machine depending on the time that the IC failed. As with the IP failure detection we specify a number of points in an instruction execution at which the IC is expected to communicate with some other machine component. We require an associative disk producing a relation to send a special end of relation message (which could be piggybacked on the last page sent) and the IC to respond to it. The IPs can detect failure if an end of a relation message (outer or inner) is heard and no

solicitation of a status report from the IC is received. Also, if the inner relation producer is another IC group the producing and consuming ICs must "shake hands" at the end of the first operation.

5.5.2.3. Associative Disk Controller

The associative disk controller must communicate with several IC groups and the MC. Since we require the controller to periodically check its mailbox in the MC this check can be used by the associative disk controller to inform the MC that it is operating successfully. The controller can also inform the MC at that point of any failure of one of the processors that is part of the associative disk complex.

The failure of the associative disk controller can have a more serious effect on the integrity of data than the failure of an IP or an IC because all the page tables of relations that are participating in active instructions reside in its memory. Of particular importance are the page tables of relations being updated. This problem has been addressed in a number of studies for both a centralized and distributed DBMS (see [79] for an extensive discussion). Generally, these techniques involve using a log to record all the write actions on the database and rolling a copy of the database out along with the log to a "safe" storage medium. After a crash the database could be restored to a state which is known to be correct (although it may not be the most recent correct state).

5.5.2.4. MC and communication mechanism

Integrity maintenance has been shown to be the function of the ICs (issuing commit and abort commands) and the associative disk controllers (implementing these commands). Although both the MC and the local network play an important role in relaying messages between the various components their failure will not affect the integrity of the data because of the local logs kept in each associative disk.

5.6. Summary

In this chapter we have presented our architecture. We began with a discussion of the rationale of adopting the MC-IC-IP control hierarchy. We then showed how the various components in the machine are interconnected. We illustrated the operation of the machine using a sample query and a detailed description of the steps taken by each component during the execution of each instruction. We concluded with a discussion of the actions necessary to maintain data integrity and an outline of a proposed implementation.

There are several unanswered questions and unclear points about the architecture. For example, can the

architecture be extended to allow for more than the number of processors supported by one cable? In fact, what is that number? The answer to the first question is a probable yes. We believe that multiple systems of the type described in this chapter can be interconnected in some manner. One possibility is the use of gateways [72]. Another possibility is a multi-bus structure such as the LENS [80].

The second question is more tricky to answer because it requires some quantitative information concerning the capabilities of the various components, the expected load on the system, etc. With the availability of such information a simulation of this architecture can be implemented which will answer this, and several other questions. For example, what should the ratio of ICs to IPs be? How should relations be distributed among the various disks?

CHAPTER 6

CONCLUSION

6.1. Summary of Work

In this dissertation we have presented the design of a relational database machine. This research is different from other work in a number of aspects though. Most important among these is the design methodology employed. We believe that past database machine research has been "architecture directed". The lack of understanding of the high-level operations and programs to be executed by the architecture has generally resulted in incomplete and/or partially inefficient machine organizations.

As an alternative we offer an "algorithm directed" approach. Essentially this is a top down design methodology. We advocate a careful study of the structure of all the algorithms to be employed by the machine as well as the structure of programs to be executed on it. Such a study can yield some qualitative and quantitative information that can be used by the computer architect in designing an architecture that is both complete and efficiently meets the user needs. We believe that this approach is viable for relational database machine design for a number of reasons.

First, the underlying data structure to which operations are applied as well as the operations themselves are well understood and not likely to change. Second, the number of high-level operations supported by a relational relational DBMS (and therefore database machine) is few. This enables the designer to search for algorithms that are both efficient and yet share a sufficient number of features in common to enable him to design an architecture with few primitive operations. Finally, nearly all programs expressed in the various high-level relational algebra languages can be compiled into the same format. Therefore the data access patterns to the database are known.

In Chapter 3 we discuss in detail the pros and cons of "architecture directed" and "algorithm directed" computer architecture research. We then go on to describe and analyze the algorithms to be employed in the architecture.

Algorithms used by a relational DBMS can generally be classified into four classes depending on the underlying primitive operation used: hashing, sorting, indexing, and nested loops. Algorithms in all of these classes can be generalized for use on a multi-processor. In this dissertation we chose to concentrate only on the parallel nested loops algorithms. Broadcasting is used to reduce the amount of inter-processor data communications. Parallel algorithms that employ hashing have been studied by Goodman [46] for use in the implementation of the join operation. Friedland [55] is currently studying the use of parallel sorting for the implementation of projections, joins, and aggregate functions.

Hawthorn and DeWitt [3] have shown that operations that can be implemented in linear time on a uni-processor are best implemented by database machines that can process queries on the fly as data is read off the disk. They have also shown that queries that contain other operations are best executed on database machines that process data off the disk. In Chapter 4, therefore, we studied problems related to the implementation of these two architecture types in an attempt to integrate them into a single organization.

We began with a study of alternative processor allocation strategies for DIRECT. We were able to conclude that a data-flow strategy yielded the best performance. We also showed that for all the strategies examined (even an SIMD strategy) the time required to process the control messages by the back-end controller exceeded the time required by the query processors to execute the queries.

Our second effort was targeted towards database machines that process queries directly on the disk (associative disks). Several associative disk designs have appeared in the literature, some differing only slightly,

others in major details. We felt that if such a processing capability was to be integrated into a database machine additional information about the organization of these designs and their relative performances was required.

We classified associative disks into three categories and implemented a simulation of three organizations representing these categories. We studied the behavior of the organizations for a benchmark of selection queries under different database size, result set size, and data distribution assumptions. We showed that a design based on a processor-per-track organization outperformed the other organizations. However, such a design is not deemed to be implementable. Also, if index information is available then both processor-per-head and processor-per-disk reach a performance level comparable with the processor-per-track design.

Finally, in Chapter 5, we described the proposed architecture. The machine organization consists of several instruction processors (IPs) controlled by a number of instruction controllers (ICs). Each IC has some number of IPs allocated to it for the duration of an instruction execution. Communication between an IC and its IPs is over a broadcast bus that uses broadband technology to support multiple channels, each operating at a different frequency. Assignment of instructions, IPs, and channels to ICs is performed by a single master controller (MC) processor.

The MC-IC-IP control hierarchy was chosen as a means for avoiding the control bottleneck that DIRECT suffers from. In this organization separate channels are used for separate tasks. Each IC need only handle messages pertaining to the instruction it controls. The MC, on the other hand, processes only resource allocation messages. An additional feature of the architecture and this control hierarchy is that the overall number of messages required to oversee the execution of any single instruction was reduced considerably from the number required by DIRECT. In particular, requests for specific pages by each IP were eliminated. The IC (representing the memory component) plays a passive role in page distribution. Each IP keeps track of those pages that it sees. Only at the end of the instruction execution do IPs inform their controlling IC whether they require additional pages.

The database resides on a number of mass storage devices. Each device has some number (possibly one) of processors which perform simple operations (such as selections and scalar aggregates) on the data as it is read off the disk. Each such associative disk has a controller associated with it. The controller communicates with the MC and the ICs as the need to do so arises (i.e. initiate a search operation). It is also responsible for overseeing the

operation of the processing elements on the disk, collecting their output, and sending it to the its destination.

Data integrity functions, such as concurrency control and crash recovery, are handled by the associative disk controllers with some assistance from the MC.

6.2. Contributions and Consequences of Research

We believe that the research reported on in this dissertation makes a number of contributions, particularly to the database machine area. By completing the algorithm-based design of a database machine we believe that we have made a step towards legitimizing the notion of "algorithm directed" computer architecture research.

Of particular interest is the fact that although the algorithms described in Chapter 3 are the same as those used by the architecture of Chapter 5 there are several differences between them. For example, our original description of the algorithms assumed the existence of a central controller. This controller assumed an active role in controlling the execution of an instruction. Explicit message exchanges were required between the controller and a processor executing an operation for each page seen by that processor. In our machine organization each instruction controller plays a passive, rather than an active, role in an instruction execution.

Another important point is that the description of the algorithms in Chapter 3 is independent of the inter-processor communication facility provided by the machine. These same algorithms can be, and in fact have been, implemented on DIRECT which uses shared memory for inter-processor communication. In our architecture direct processor to processor interconnections exist.

Our argument here is that the study of the algorithms can and should be divorced from architectural considerations as much as possible. Additional work, primarily in the design of more special purpose machines, is needed before "full" legitimization for the approach advocated here can be claimed.

In the process of arriving at our design we have provided a framework for comparing associative disk designs. Our comparison of three different associative disk types showed that the DBC approach of using moving head disks with indexing to cut down the search space is indeed successful. It was also shown that associative disks that employ a standard moving head disk (i.e. the processor-per-disk organization) also perform well if index information is available.

Our investigation of the application of data-flow machine techniques to database machines resulted in a number of interesting observations. First, we showed that

some form of data-flow scheduling can indeed improve performance. However, "pure" data-flow proved to be an unsuccessful strategy because of the high inter-processor communication cost incurred. In fact, we believe that this result may be of some interest to data-flow machine designers.

In essence the data-flow machine approach to the execution of our algorithms would be to unfold the nested loops structure entirely. This would enable the machine to execute the program employing the maximal degree of parallelism. This form of maximal loop unfolding is employed by nearly every data-flow machine design for the execution of a variety of programs. Recently, Gostelow and Thomas [81] presented the results of a simulation study of the Irvine data-flow machine. A number of different programs (matrix multiplication, fast fourier transform, and others) were simulated. It was shown that the high level of inter-processor communication dominated the execution time of the various programs simulated.

We believe that because two independent studies each using a different set of problems and simulating a different architecture arrived at the same results, these results are significant. Additional research into the tradeoffs between massive parallelism and minimization of communication costs is needed.

A final contribution of this research is that the proposed architecture can be constructed using off-the-shelf components. Judging from experiences with the implementation of DIRECT, which required a number of custom designed components, this is an important feature.

6.3. Future Work

There are several avenues of research to be explored based on this work. In this section we describe a number of these. We begin with a discussion of research in database machines and end with some points concerning data-flow machines.

One of the problems with the description of the architecture at this time is that it is vague at points. The difficulty is rooted in the fact that the machine organization is flexible. Its structure is intended to change over time to adapt to changing user needs. What is needed then is a tool that would "compile" a user community's profile into a specific configuration. Such a tool can most likely be designed. However, considerable experience with simulation of the machine is needed in order to understand the interactions and relationships between the component types. For example, such a simulation would lead to the determination of the speed of an individual channel, the page size to be used, and the ratio of ICs to IPs.

Two other purposes of such a simulation would be to gain some insights into possible performance bottlenecks and to compare the performance of this machine to others. This latter point is of particular interest. A considerable body of database machine literature has appeared. However, little of it is concerned with comparative performance evaluation. One exception is the work of DeWitt and Hawthorn [3,82]. Both of these papers take an analytic approach to performance evaluation. Although both papers raise some interesting issues and result in some concrete conclusions it is clear that their results are limited. It seems as though the next step in comparing the performances of several database machines must be simulation.

Such a simulation must be designed in such a manner as to encompass a large range of database machine architectures that employ different algorithms. The input parameters to the simulation must be rooted in the "real world", that is they must be based on observed executions of programs on existing databases. The reason for using such "real" data is for the evaluators of the machines to escape criticisms of the use of biased data. Hawthorn [4] has argued that transactions of different types will require different types of database machines. Therefore the data used must reflect a wide a range of transactions to determine the suitability of the various machines to different transaction types. However, data reflecting different views of the future transaction types must also be tested.

Another interesting area of research is the study of the use of indexing for the execution of entire queries. In such a system an index would be maintained for each attribute in the database. By executing a query on the index rather than the database less processing power (and communication and I/O overhead) would be required resulting in faster execution. However, the disadvantages of maintaining the index (both in terms of storage overhead and the increased complexity of update operations) as well as the cost to materialize result relations must be assessed.

We have stated earlier in this chapter that we believe that designers of data-flow machines should study the various performance tradeoffs in general purpose data-flow machines. Of particular interest is the effect of increased parallelism on the cost of communication. We intuit that the results of such a study will require us to investigate means of algorithm evaluation that would take into account features other than degree in parallelism when labeling an algorithm "good". Once such tools become available new algorithms for execution on data-flow machines (perhaps also for other parallel machines) can be designed.

APPENDIX A

In this appendix we briefly describe the relational algebra operations. We shall use INGRES [43] its query language QUEL [56] to illustrate some of the operations. Figure A.1 contains an instantiation of the Employee relation and will be used in our examples. We group the operations into two classes: retrieval and update.

The retrieval operations are selection, projection, join, and aggregates. A selection operation retrieves a horizontal subset of a relation based on a simple predicate applied to one or more of the attributes in the relation. For example:

```
retrieve (emp.all) where emp.name = "Smith"
         or emp.name = "Brown"
```

results in a new relation which contains the employee

Employee Relation:

Name	Dept	Task	Salary	Manager
Smith	Toys	Clerk	300.00	Johnson
Miller	Shoes	Buyer	650.00	Bergman
Jones	Books	Acct	550.00	Harris
Brown	Shoes	Clerk	400.00	Conners

Figure A.1: The Employee Relation

records of Smith and Brown.

A projection retrieves a vertical subset of a relation not allowing for duplicate tuples. For example

```
retrieve (emp.dept) where emp.dept = "Shoes"
```

will result in a new relation containing a single tuple made up of a single attribute (whose value will naturally be Shoes).

A join operation operates on two input relations. The predicate statement performs a relational operation (=, >, etc.) between an attribute in one relation and a compatible attribute in the other relation. A match will cause the "join" of the two tuples to occur. For example, suppose our database had an another relation in it containing some information about every department. In particular, the location of each department was specified. Then to find the location of all the employees in the company it is necessary to form the join of the two relations:

```
retrieve (emp.all , department.all) where
        emp.dept = department.dept
```

Clearly, there are many instances where one would like to combine the operations described above in a single query. For example, to find the location of manager Smith we would use the following statement:

```
retrieve (emp.manager , department.location) where
        emp.dept = department.dept and
        emp.manager = "Smith"
```


This query applies all three operations we have discussed so far. The clause

```
emp.manager = "Smith"
```

represents the application of the restriction operation.

```
emp.dept = department.dept
```

is a join between the two relations. Finally, retrieving only two attributes is an instantiation of a projection.

The general form of a retrieval operation is:

```
RETRIEVE (target_list) WHERE qual
```

where qual can be any number of the retrieval operations described above joined by ANDs and ORs.

In contrast with other relational operations join, project, etc., there is no commonly accepted set of aggregate operations among existing relational database systems. We thus discuss the operator in more detail than its predecessors. We distinguish between "scalar" aggregates and aggregate "functions". Scalar aggregates are aggregations (average, max, etc.) over an entire relation. Aggregate functions first divide a relation into non-intersecting partitions (based on some attribute value, e.g. sex) and then compute scalar aggregates on the individual partitions. Thus, given a source relation, scalar aggregates compute a single result while aggregate functions produce a set of results (i.e. a result relation). The two types of aggregates have the following form:

```
scalar:      agg_op ( agg_att where qual )
function:    agg_op ( agg_att by_list where by_qual )
                    where src_qual
by_list:     by att-1 by att-2 by ... by att-n
agg_op:      sum, avg, count, max, min, sumu,
                    avgu, countu
```

The agg_att is the attribute over which the aggregate is being computed. The aggregate operators (agg_op above) are self-explanatory except for those with the "u" suffix. The "u" denotes "unique" and implies that duplicates (tuples which match on the agg_att) will be eliminated before the aggregate is computed.

Qualifications may be added ("where qual") to compute an aggregate over a subset of tuples in a relation. For aggregate functions, the partitioning attributes are specified with the by_list. Note that relations may be partitioned on more than one attribute (e.g. partitioning employees by department and task within department). Also note that the result of an aggregate function may depend on qualifications outside the aggregate (src_qual) (this will be discussed in more detail later). In contrast, scalar aggregates are "self-contained" and are not affected by the rest of the query.

To understand why two different qualifications are required for an aggregate function consider the following example:

```
count (emp.name by emp.mgr) where emp.sal > 500
```

This query requests a count of the number of employees under each manager earning more than $500. However, even if a manager does not have any employees earning more than $500 (e.g. Johnson in Figure A.1), he should not be excluded from the list and his count should be set to 0. If we applied the qualification first and then computed the aggregate function on the result, we would miss those managers, since all their employees were removed by the qualification. As another example, consider:

```
count (emp.name by emp.mgr where emp.mgr!="Bergman")
        where emp.sal > 500
```

Clearly, in this case we want to include the count for all managers other than Bergman. Thus, we need to distinguish between restrictions on the source tuples and restrictions on the set of possible partitions. This is why we allow for two different types of qualifications in aggregate functions. Qualifications inside the aggregate (the "by_qual"), in addition to selecting a subset of the source relation, have the effect of eliminating unwanted partitions (e.g. manager Bergman above). While qualifications outside the aggregate (the "src_qual") primarily affect the source relation, they may have the undesirable side effect of removing desired partitions (e.g. managers for whom no employees earn more than $500) and we must correct for this.

The second group of operations are updates. Included in this group are delete, append, and modify. The syntax of the update operations is:

UPDATE relation WHERE qual.

qual can be a composition of any number of the retrieval operations (although typically it will consist of a single selection). In the case of an append operation, the values to be assigned to the various attributes in the relation can be formed by the qual clause or can be supplied by the user. A replace operation is used to replace the value of one or more attributes in a relation.

APPENDIX B

TEST DATABASE

RELATION	# PAGES	TUPLE WIDTH
1	27	41
2	36	41
3	40	57
4	34	20
5	28	99
6	5	63
7	12	39
8	7	11
9	8	26
10	20	41
11	38	38
12	40	37
13	40	42
14	10	14
15	9	77

BIBLIOGRAPHY

[1] R.H. Canaday, R.D. Harrison, E.L. Ivie, J.L. Ryder, and L.A. Wehr, "A Back-end Computer for Data Base Management," CACM 17, 10, (Oct. 1974).

[2] CODASYL Data Base Task Group, Report, ACM, New York (April 1971).

[3] P. Hawthorn and D.J. DeWitt, "Performance Evaluation of Database Machines," IEEE Transactions on Software Engineering, (To Appear 1981).

[4] P. Hawthorn, "The Effect of the Target Applications on the Design of Database Machines," Proc of the ACM SIGMOD 1981 International Conference of Management of Data, (May 1981).

[5] D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers c-28, 6, (June 1979).

[6] D.K. Hsiao, "Data Base Machines are Coming Data Base Machines are Coming," Computer 12, 3, (March 1979).

[7] D.C.P. Smith and J.M. Smith, "Relational Data Base Machines," Computer 12, 3, (March 1979).

[8] G.F. Coulouris, J.M. Evans, and R.W. Mitchell, "Towards content addressing in databases," Computer Journal 15, 2, (February 1972).

[9] J. Banerjee, R.I. Baum, and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," ACM TODS 3, 4, (December 1978).

[10] P. Hawthorn and M.R. Stonebraker, and P. Hawthorn, "Evaluation and Enhancement of the Performance of Relational Database Management Systems," Proc. of the ACM SIGMOD 1979 Int'l Conf. of Management of Data, (1979).

[11] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," CACM 13, 6, (June 1970).

[12] S.Y.W. Su, H. Chang, G. Copeland, P. Fisher, E. Lowenthal, and S. Schuster, "Database Machines and Some Issues on DBMS Standards," Proc. NCC 49, (1980).

[13] D.L. Slotnick , "Logic Per Track Device," in Advances in Computers, ed. F. Alt,Academic Press, NY (1970).

[14] E.A. Ozkarahan , S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," Proc NCC 45, AFIPS Press, (1975).

[15] E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, "Performance Evaluation of a Relational Associative Processor," ACM TODS 2, 2, (June 1977).

[16. R. Epstein and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," Proc NCC 49, AFIPS, (1980).

[17] J.L. Parker, "A Logic per Track retrieval System," IFIP Congress, (1971).

[18] N. Minsky, "Rotating Storage Devices as Partially Associative Memories," Proc FJCC, (1972).

[19] B. Parhami, "A Highly Parallel Computing System for Information Retrieval," Proc FJCC, (1972).

[20] S.Y.W. Su and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," Proc International Conference Very Large Data Bases, (September 1975).

[21] S.Y.W. Su, H.B. Nguyen, A. Emam, and G.J. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Transactions on Computers c-28, 6, (June 1979).

[22] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS 4 , 1, (March 1979).

[23] D.R. McGregor, R.G. Thomson, and W.N. Dawson, "High Performance Hardware for Database Systems," in Systems for Large Data Bases, North Holland (1976).

[24] E.A. Ozkarahan and K.C. Sevcik, "Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM TODS 2, 4, (December 1977).

[25] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan, and K.C. Smith, "RAP.2 - An Associative Processor for Databases and Its Applications," IEEE Transactions on Computers c-28, 6, (June 1979).

[26] S.C. Lin, D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS 1, 1, (March 1976).

[27] P.B. Berra and E. Oliver, "The Role of Associative Array Processors in Data Base Machine Architecture," Computer 12, 3, (March 1979).

[28] E. Oliver, "," in RELACS: An Associative Computer Architecture to Support a Relational Data Model, Syracuse University (1979).

[29] F. Bancilhon and M. Scholl, "Design of a Backend Processor for a Data Base Machine," Proc of the ACM SIGMOD 1980 International Conference of Management of Data, (May 1980).

[30] H.O. Leilich, G. Stiege, and H. Ch. Zeidler, "A Search Processor for Data Base Management Systems," Proc 4th Conference on Very Large Databases, (1978).

[31] IBM Corporation, "IBM 3380 Direct Access Storage description and User´s Guide," IBM Document GA26-1664-0, File No. S/370-07,4300-07 (1980).

[32] G. Stiege, Personal Communication to D.J. DeWitt.

[33] H. Chang and A. Nigam, "Major-Minor Loop Chips Adapted for Associative Search in Relational Data Base," IEEE Transactions on Magnetics mag-14 , 6, (Nov 1978).

[34] F. Chin and K.S. Fok, "Fast Sorting Algorithms on Uniform Ladders (Multiple Shift Register Loops) ," IEEE Transactions on Computers c-29, 6, (June 1980).

[35] K. Chung , A Study of Data Manipulation Algorithms in Magnetic Bubble Memories , Deptartment of Computer Science Univ of Illinois, Urbana-Champaign (June 1979) Ph.D. Dissertation.

[36] K.M. Chung, F. Luccio, and C.K. Wong, "On the Complexity of Permuting Records in Magnetic Bubble Memory Systems," IEEE Transactions on Computers c-29, 7, (July 1980).

[37] K.L. Doty, J.D. Greenblatt, and S.Y.W. Su, "Magnetic Bubble Memory Architectures for Supporting Associative Searching of Relational Databases," c-29, 11, (November 1980).

[38] J.W.S. Liu and M. Jino, "Intelligent Magnetic Bubble Memories and Their Applications in Data Base Management Systems," IEEE Transactions on Computers c-28, 12, (Dec 1979).

[39] P.K. White, "Bubble Memory Performance in System Design ," Proceedings of the Fall 1979 Compcon Conference, (1979).

[40] S.E. Madnick, "The INFOPLEX Database Computer: Concepts and Directions," Proc IEEE Computer Conference, (February 1979).

[41] D.J. DeWitt, "Query Execution in DIRECT," Proc. of the ACM SIGMOD 1979 Int´l Conf. of Management of Data, (May 1979).

[42] H. Boral and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," ACM TODS, (To Appear June 1981).

[43] M.R. Stonebraker, E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM TODS 1 , 3, (September 1976).

[44] J. Banerjee, D.K. Hsiao, and K. Kannan, J. Banerjee, and D.K. Hsiao, "Performance Study of a Database Machine in Supporting Relational Databases," Proc. Fourth International Conf. on VLDB. c-28, 6, (1979).

[45] D.K. Hsiao and J. Menon, D.K. Hsiao , and K. Kannan, "The Architecture of a Database Computer - Part II: The Design of Structure Memory and its Related Processors," Technical Report OSU--CISRC-TR-76-2, Computer and Information Science Research Center The Ohio State University, Columbus Ohio (July 1979).

[46] J.R. Goodman, , University of California, Berkeley (1980) Ph.D. Thesis.

[47] J.R. Goodman and A.M. Despain, "A study of the interconnection of multiple processors in a data base environment," Proc. 1980 International Conference on Parallel Processing, pp. 269-278 (August 1980).

[48] H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Transactions on Computers c-20, 2, pp. 153-161 (February 1971).

[49] K. Oflazer and E.A. Ozkarahan, "RAP.3: A Multi-Microprocessor Cell Architecture for the RAP Database Machine," Proc. of the Int'l Workshop on High-level Language Computer Architecture, (May 1980).

[50] D.K. Hsiao and J. Menon, "Parallel Record-Sorting Methods for Hardware Realization," Technical Report OSU-CISRC-TR-80-7, Computer and Information Science Research Center The Ohio State University (July 1980).

[51] D.K. Hsiao and J. Menon, "Design and Analysis of Relational Join Operations of a Database Computer," Technical Report OSU-CISRC-TR-80-8, Computer and Information Science Research Center The Ohio State University (September 1980).

[52] H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Performance Evaluation of Associative Disk Designs," The Sixth Workshop on Computer Architecture for Non-numeric Processing, (Submitted June 1981).

[53] H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Performance Evaluation of Four Associative Disk Designs," Information Systems, (Submitted March 1981).

[54] H. Boral, D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," ACM TODS, (Submitted October 1980).

[55] D. Friedland, , Computer Science Department, University of Wisconsin - Madison (1981) Ph.D. Thesis (In Preparation).

[56] K. Youssefi et. al., INGRES Version 6.0 Reference Manual.

[57] D.E. Knuth, The Art of Computer Programming Volume 1--Fundamental Algorithms (second edition), Addison-Wesley (1973).

[58] J.M. Smith and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," CACM 18, 10, (October 1975).

[59] S. Bing Yao, "Optimization of Query Evaluation Algorithms," ACM TODS 4, 2, (June 1979).

[60] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," Proc. of the ACM SIGMOD 1979 International Conference of Management of Data, (May 1979).

[61] Digital Equipment Corporation , Microcomputer Handbook. (1977).

[62] D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," Proc. of the 5th Annual Symposium on Computer Architecture, (April 1978).

[63] G.W. Gorsline , Computer Organization: Hardware / Software,, Prentice-Hall (1980).

[64] R.A. Lorie, Personal Communication to D. DeWitt.

[65] M.R. Stonebraker, Personal Communication to D.J. DeWitt.

[66] G. Chesson, Personal Communication to D. DeWitt.

[67] H. Boral and D.J. DeWitt, "Design Considerations for Data-flow Database Machines," Proc of the ACM SIGMOD 1980 International Conference of Management of Data, (May 1980).

[68] P.J. Sadowski and S.A. Schuster, "Exploiting Parallelism in a Relational Associative Processor," Fourth Workshop on Computer Arch. for Non-numeric Processing, (August 1978.).

[69] K. Kannan, "The Design of a Mass Memory for a Database Computer," Proc. of the Fifth Annual Symposium on Computer Architecture, (April 1978).

[70] B. Maglaris and T. Lissack, "An Integrated Broadband Local Network Architecture," Proc. 5th Annual Local Computer Networks Conference, (October 1980).

[71] W.A. Levy and M. Rothberg, "Coaxial Cable Finds a Home," Mini-Micro Systems 14, 3, (March 1981).

[72] R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed packet switching for local computer networks," CACM 19, 7, pp. 395-403 (July 1976).

[73] N. Pippenger, "On Crossbar Switching Networks," IEEE Transactions on Communications com-23, 6, (June 1975).

[74] L.R. Goke and G.J. Lipovski, "Banyan networks for partitioning multiprocessor systems," 1st Annual Symposium on Computer Architecture, pp. 21-28 (December 1973).

[75] H.F. Taylor, "Multi-processor bus architecture," Report ERC41015.4FR, Rockwell International (June 1980).

[76] C.S. Chi, "Higher Densities for Disk Memories," IEEE Spectrum 18, 3, (March 1981).

[77] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," CACM, Computer Science Lab., Xerox PARC, (Submitted 1979).

[78] W.K. Wilkinson, , Computer Science Department, University of Wisconsin - Madison (1981) Ph.D. Thesis (In Preparation).

[79] J.N. Gray, "Notes on Database Operating Systems," Report RJ2188, IBM, San Jose, California (1978).

[80] R.A. Finkel and M.H. Solomon, "The lens interconnection strategy," Proc. 14th Hawaii International Conference on System Sciences, (January 1981).

[81] Kim P. Gostelow and Robert E. Thomas, "Performance of a Simulated Dataflow Computer," IEEE Transactions on Computers C-29, 10, pp. 905-919 (October 1980).

[82] D.J. DeWitt and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures," Proc. VLDB-7, (September 1981).