QUERY EXECUTION IN DIRECT

by

David J. DeWitt

Computer Sciences Technical Report #343

December 1978

QUERY EXECUTION IN DIRECT

David J. DeWitt
Computer Sciences Department
University of Wisconsin

This research was partially supported by the National Science Foundation under grants MCS78-01721 and MCS77-08968 and the United States Army under contract #DAAG29-75-C-0024.

ABSTRACT

In this paper query organization, execution, and optimization in the database machine DIRECT are discussed. We demonstrate that the use of a monitor for each relation referenced by a query along with the use of the NEXT_PAGE construct permits the DIRECT back-end controller to assign a query to any number of processors for execution. Furthermore, these constructs also permit the controller to balance the load in the back-end by dynamically adjusting how many processors are assigned to each executing query.

We also identify the problem of relation fragmentation which occurs when a query is executed by several processors in parallel and develop a technique for estimating the optimal number of processors to compress a relation so that the execution time of the entire query is minimized. These results appear to be applicable to all database machines which employ parallel processing techniques to enhance query execution.

1.0 Introduction

This paper discusses query organization, execution, and optimization in DIRECT. DIRECT is a multiprocessor organization for supporting relational database management systems which is being implemented at the University of Wisconsin. DIRECT can support the simultaneous execution of relational algebra queries from different users in addition to parallel processing of single queries. Section 1.1 contains an overview of DIRECT's organization. More details can be found in [1,2].

One feature which differentiates DIRECT from the other database machines which have been proposed is its relational algebra query organization and execution. As will be demonstrated in Sections 2.0, 3.0, and 4.0, the unique structure of DIRECT queries results in the following two features. First, the number of processors assigned to execute a query can be dynamically determined based on the priority of the query, the type and number of relational algebra operations included in the query, and the size of the relations referenced. The second effect of our organization is that the number of processors assigned to a query can be increased or decreased <u>during</u> the execution of the query.

Section 5.0 presents a discussion of a problem which has been ignored by all previous database machine designers. This problem arises when more than one processor is used to select from a relation those tuples which satisfy a search condition. We call this problem "relation fragmentation". To illustrate we will use the "restriction" operator. Assume that the relation being "restricted" is divided into fixed size pages (tracks in RAP [3,4]) and that there is one processor per page (track). Then, when the search condition is applied by all processors in parallel to the relation, each processor will produce a subset of the new relation. This new relation will contain those tuples which satisfy the restriction. In DIRECT each processor will produce some fraction of a page of the new relation. In RAP, each processor will "mark" those tuples on its track which satisfy the search condition. (For a discusion of mark bits versus temporary relations see [1,2].) If this new relation represents the results of the query, this fragmentation is not a significant problem. <u>However</u>, if this new relation is to be used by a subsequent operator (such as a join) in the query, then the degree of relation fragmentation will have a significant impact on the performance of this operator. This performance degredation will occur because any subsequent operator which uses the fragmented relation as an operand will have to read all of the partially filled pages (tracks) in order to access all the tuples of the intermediate relation. RAP has no choice but to suffer the effect of this fragmentation and, as a consequence, RAP executes joins with about the same performance as a single conventional processor [5]. In Section 5.0, we introduce the concept of compression as a technique for reducing the degree of relation fragmentation. Then, we analyze the cost of compression in order to calculate the optimal number of processors to perform the compression so that the execution time of the entire query is minimized.

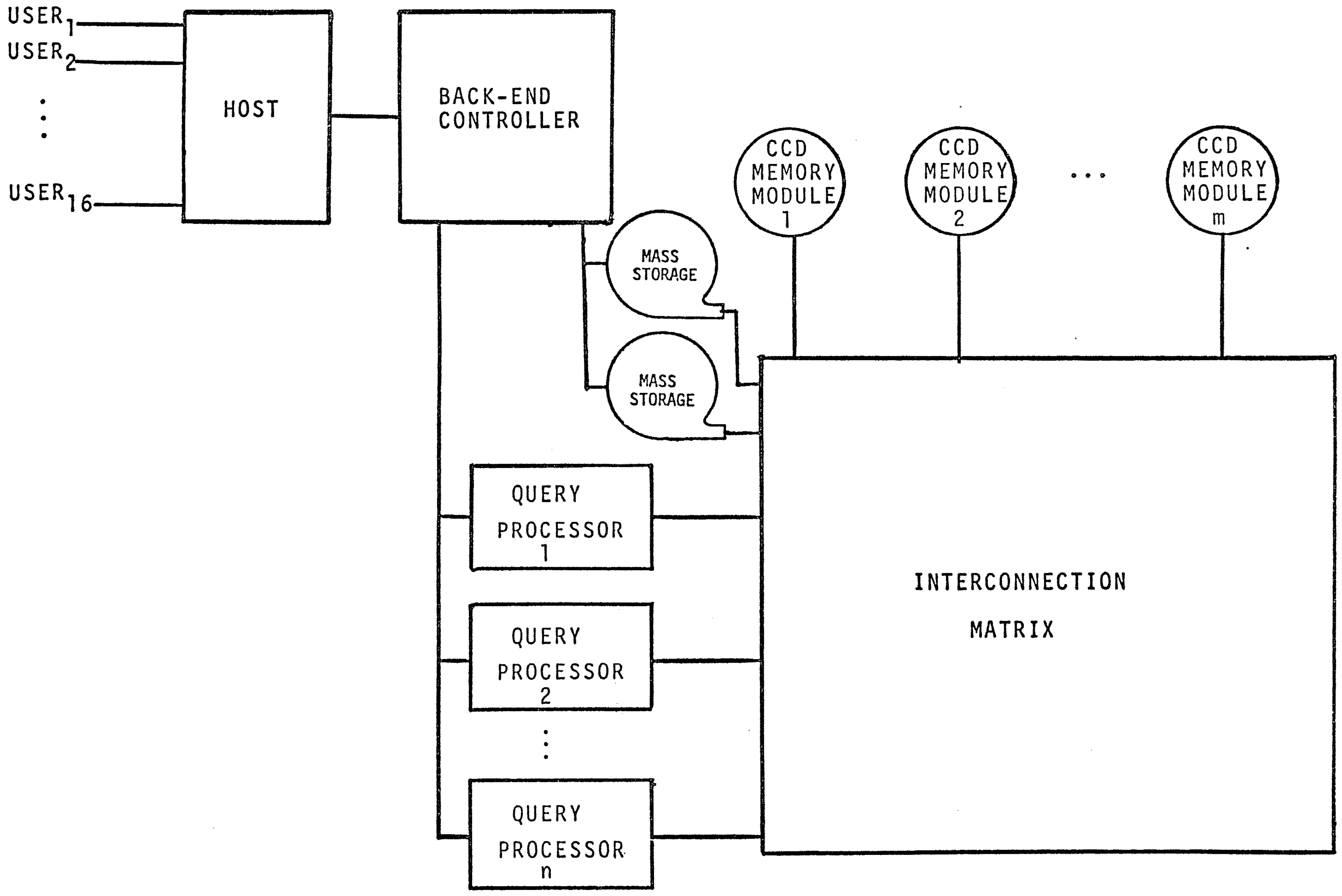
1.1 DIRECT Systems Architecture

DIRECT consists of six main components: a host processor, a back-end controller, a set of query processors, a set of CCD memory modules which are used as pseudo-associative memories, an interconnection matrix between the set of query processors and the set of CCD memory modules, and one or more mass storage devices. A diagram of these components and their interconnections can be found in Figure 1.1.

The host processor, a PDP 11/45 running the UNIX operating system, handles all communications with the users. A user who wishes to use DIRECT will log onto a modified version of INGRES [6] and then proceed in the normal manner. However, when the user wishes to execute a query, INGRES will compile the query into a tree of relational algebra operations called a "query packet" and then send it to the back-end for execution.

The back-end controller is a PDP 11/40. It is responsible for interacting with the host processor and controlling the query processors. After the back-end controller receives a query packet from the host, it determines the number of query processors that should be assigned to execute the packet. If the relations which are referenced by the query packet are not currently in the associative memory, the back-end controller will page portions of them in before distributing the query packet to the query processors selected.

Each query processor is a PDP 11/03 with 28K words memory. The function of each query processor is to execute query packets assigned by the back-end controller.



DIRECT SYSTEM ARCHITECTURE
Figure 1.1

Since DIRECT has a MIMD architecture, it is capable of supporting both intra and inter-query concurrency. To facilitate the support of intra-query concurrency, relations are divided into fixed size pages. Each query processor, assigned by the controller to execute a query packet, will "associatively" search a subset of each relation referenced in the packet. When a query processor finishes examining one page of a relation, it makes a request to the back-end controller for the address of the next page it should examine. Since several query processors, each executing the same query, can request the "next page" of the same relation simultaneously, the controller operations must be indivisible. This will insure that each of the query processors will be given a different page to examine. After receiving the address of the page from the controller, the query processor must be able to rapidly switch to that page. Details of an interconnection matrix which permits this can be found in [1,2].

To facilitate support of inter-query concurrency, the associative memory and interconnection matrix must permit two query processors, each executing different queries, to search the same page of a common relation simultaneously. By eliminating duplicate copies of a relation, we not only reduce memory requirements but, more importantly, we eliminate the problem of updating multiple copies of a relation without sacrificing performance.

2.0 Query Packet Format

The types of query packets which are received by the back-end controller can be divided into two classes based on information contained in the packet header. Class I contains those commands which manipulate system catalogues (e.g. CREATDB). Execution of these commands generally involves addition or deletion of information in one or more tables which are maintained by the back-end controller and does not involve any query processors. For more details the reader should see [1,2]. Class II contains those queries which will be executed by a collection of query processors. In this section, we will discuss the structure of this class of query packets.

Each Class II query packet is structured as a tree (see Figure 2.1). The leaf nodes of this tree correspond to relations referenced in the query. Each non-leaf node is either a relational algebra operator or an intermediate relation. The set of operators supported include the traditional operators such as JOIN, PROJECT, RESTRICT, UNION, INTERSECTION, and DIFFERENCE as well as aggregrate operators such as MAX, MIN, COUNT, etc.

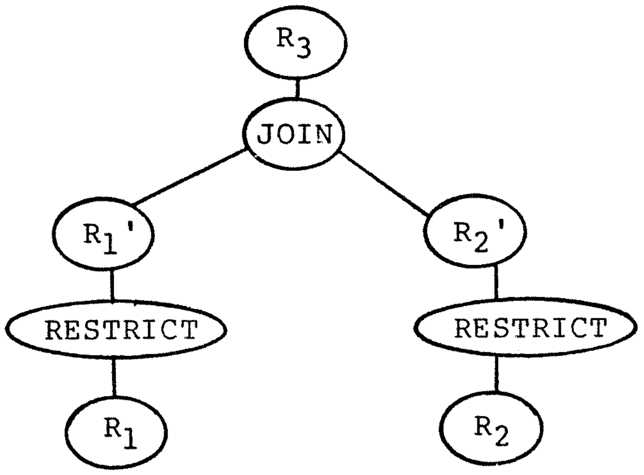


Figure 2.1

One of the unique features of DIRECT is the general structure which is common to all the relational algebra operators supported. This structure, as we will demonstrate in Section 4.1, permits a query packet to be assigned to any number of query processors <u>without</u> modifying the packet. <u>Furthermore</u>, this operator structure permits additional query processors to be dynamically assigned to execute a packet during execution of the packet by other query processors. We will conclude this section with an example of a simple INGRES query and the corresponding query packet. This example will be used in Section 4.1 to illustrate how our flexible query processor assignment scheme is implemented.

Given the SUPPLIER relation shown in Figure 2.2 and the INGRES query to find the names of all suppliers in N.Y.:

RETRIEVE (SUPPLIER.NAME) WHERE SUPPLIER.CITY = "N.Y."

Then, the compiled query packet for this query has the structure shown in Example 2.1. Note that the packet never asks for an explicit page of a relation. Instead, the next page of the relation is always requested. This structure, when combined with a monitor for every relation referenced by a packet, permits us to dynamically assign query processors to executing query packets (see Section 4.1).

SUPPLIER Relation

NUMBER	NAME	CITY
10	JONES	N.Y.
20	SMITH	CHICAGO
74	WHITE	DALLAS
101	RICE	N.Y.

Figure 2.2

CITYQPKT

```
LOCK(SUPPLIER,READ)
CREATE NYSUPPLIERS   /* create result relation */
DO FOREVER
BEGIN
     - ASK BACK-END CONTROLLER (BEC) FOR THE NEXT_PAGE  OF  RELA-
     TION SUPPLIER
     - WAIT FOR THE BEC TO RETURN THE PAGE FRAME NUMBER
     - IF THE BEC RETURNS "END OF RELATION" QUIT AND SIGNAL DONE
     OTHERWISE
          - READ NEXT PAGE OF RELATION SUPPLIER INTO LOCAL MEMORY
          FROM THE PAGE FRAME
          - EXAMINE ALL TUPLES READ IN
               - COPY EACH TUPLE THAT SATISFIES  THE  RESTRICTION
               SUPPLIER.CITY = "N.Y." INTO A LOCAL PAGE BUFFER
               - WHEN THE BUFFER IS FULL
                    - ASK  (BEC) FOR THE  NEXT_PAGE  OF  RELATION
                    NYSUPPLIERS
                    - WAIT FOR BEC TO RETURN A PF#
                    - WRITE BUFFER INTO PF#
END
UNLOCK(SUPPLIER)
```

Example 2.1

3.0 Query Processor Allocation

We are currently investigating two alternative approaches for allocating query processors to a query packet when the back-end controller decides to execute the packet. These approaches are termed the fixed allocation approach and the dataflow machine approach. In the fixed allocation approach the QPA process examines the packet and attempts to estimate an "optimal" query processor allocation for this packet. By optimal we mean that assignment of more than the optimal number of query processors to the packet will not decrease the execution time for the packet. For example, assume that the query packet joins relation A and relation B and that relation A is N pages long and relation B is M pages long. Then, the optimal query processor allocation for this packet is MAX(M,N). This allocation will require MIN(M,N) time units where a time unit is the time required to join one page of A with one page of B. In this example, the optimal allocation is truly optimal. However, consider the packet which first joins A and B and then joins the resulting relation with C. Since it is impossible to predict the size of A join B, it is impossible to determine exactly how many query processors should be assigned. We plan to investigate several heuristics for determining the "optimal" query processor allocation.

In the dataflow machine approach to query processor allocation, only simple relational algebra operations (JOINS, RESTRICTIONS, PROJECTIONS, etc.) are assigned to the query processors and not complete query packets. This eliminates the difficult task of estimating an "optimal" query processor allocation for an entire packet. Instead, an exact value for each step in the packet can be determined. In addition to increasing query processor utilization, this approach should also decrease page traffic in DIRECT. This should occur because as soon as a page of "A join B" is produced, another query processor can begin joining it with C. Hence, the likelihood that the page will be paged out is reduced. We are currently comparing this data flow machine approach with the standard QPA scheme to determine the effect of each on system throughput in DIRECT.

4.0 CCD Memory Management

The function of the CCD memory management process in the back-end controller is three fold:

- Respond to a NEXT_PAGE request from a query processor
- Respond to a GET_PAGE request from a query processor
- Schedule the movement of pages of relations between CCD memory page frames and mass storage as the result of NEXT_PAGE and GET_PAGE operations.

4.1 The NEXT_PAGE Operation

The form of a NEXT_PAGE request is:

NEXT_PAGE(QPKT$_i$,REL$_j$,QP$_k$)

This is a request from query processor k which is executing query packet i for the next page of relation j. The resulting action is for the BEC to send the page frame number which contains the next page of relation j to query processor k. A page fault can occur if the required page is not in some CCD page frame. Handling of page faults is the same for both the NEXT_PAGE and GET_PAGE operations and is discussed in Section 4.3.

Since a query packet can be assigned to any number of query processors, there must be a way to prevent two simultaneous NEXT_PAGE requests from different query processors executing the same packet from getting the same page. This is handled by the use of the Query Packet Task Table which is shown in Figure 4.1. The Query Packet Task Table has one entry for each instance of each relation referenced by each executing query. Associated with each entry is a monitor[7] which controls access to the table entry. Initially, the currency pointer for the relation is set to zero.

QUERY PACKET TASK TABLE

QPKT #	RELNAME	CURRENCY POINTER	POINTER TO PAGE TABLE FOR RELATION
CITYQPKT	SUPPLIER	i	
CITYQPKT	NYSUPPLIERS	j	

Figure 4.1

As an example, assume that query processor allocation initially assigns the CITYQPKT query in Example 2.1 to query processors QP5 and QP8. Assume that the order in which the monitor for the SUPPLIER relation receives requests is:

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_5)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_8)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_5)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_5)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_8)

Then, the pages of the SUPPLIER relation examined by QP_5 will be 1,3, and 4 and the pages of the SUPPLIER relation examined by QP_8 will be 2 and 5. Assume that after the last NEXT_PAGE request above, the QPA algorithm assigns an additional query processor (QP_1) to the CITYQPKT. Now there will be three query processors requesting pages from relation SUPPLIER. If the subsequent request stream is as follows:

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_1)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_5)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_1)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_8)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_5)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_1)

NEXT_PAGE(CITYQPKT,SUPPLIER,QP_8)

and the SUPPLIER relation consists of nine pages then QP1 will examine pages 6 and 8 and then will receive the "End of Relation" (EOR) message. QP5 will examine page 7 before receiving an EOR reply. QP8 will examine page 9 before it receives an EOR to a NEXT_PAGE request.

By having a monitor associated with each entry in the query packet task table and by using the NEXT_PAGE concept, we can dynamically assign additional query processors to a query packet that is already partially executed. In the case of a complex query such as that in Figure 2.1, it may be the case that the additional query processors are added after the other query processors have finished the restrict of R_1 and R_2. Our approach handles this situation correctly. When the newly assigned query processors attempt to restrict R_1 (or R_2), they will get "End of Relation" immediately and will therefore proceed to begin the join of R_1' with R_2'.

4.2 The GET-PAGE OPERATION

The form of the GET_PAGE Operation is:

GET_PAGE($QPKT_i$,REL_j,QP_k,$PAGE_m$)

It represents a request from a query processor for $PAGE_m$ of REL_j.

A query processor which is executing A join B will use this operator to retrieve every page in B for each page of A it examines. No monitor is needed to coordinate GET_PAGE requests.

4.3 PAGE FAULTS IN DIRECT

The third task of the CCD memory management process is to handle page faults by scheduling page transfers between CCD memory page frames and mass memory. A page fault occurs when a requested page is not in some CCD memory module. In DIRECT these page faults will be, to a large extent, avoidable by doing anticipatory paging. In the CITYQPKT (Example 2.1), for example, the controller knows that the entire SUPPLIER relation will be examined and hence the reference string of the CITYQPKT is known in advance. By using the currency pointer for the SUPPLIER relation in the query packet task table, the SUPPLIER relation page table, and the current query processor allocation, the controller can determine how far ahead it should attempt to be in order to insure that there will always be a page ready for each query processor which is executing the packet.

5.0 Relation Fragmentation and Compression

In this section we will discuss why the problem of relation fragmentation occurs, when a relation should be compressed, and how many query processors should be used to perform the compress operation. The query shown in Figure 2.1 will be used throughout this section to illustrate several points. In this query, relations R_1 and R_2 are "restricted" by boolean search conditions to form R_1' and R_2', respectively. Then, relations R_1' and R_2' are "joined" to form the result relation R_3.

In the following sections we will address the following questions:

- Why should a relation ever be compressed?
- Before performing the join operation, should relation R_1', R_2', or both R_1' and R_2' be compressed?
- How many query processors should be assigned to compress a relation so that the execution time of the complete query is minimized?

5.1 Relation Fragmentation

In order to explain the need for compression, consider the restriction of relation R_1 in Figure 2.1 to form relation R_1'. If R_1 contains 1000 pages and 1000 query processors are available to execute the restrict operator, then each query processor will examine one page of R_1 and will produce at most one page of R_1'. If 20% of the tuples of R_1 satisfy the search criterion, then each page of R_1' will, on the average, be 20% full. If only 500 query processors are used to evaluate the restrict operator then 500 pages, each of which is 40% full, will be produced. Note, however, that if one time unit is the amount of time required for one query processor to restrict one page of a relation, then the restrict operator will require one time unit if 1000 query processors are used and two time units if 500 processors are used. Thus, there is a tradeoff between the processing time of an operation and the degree of fragmentation of the relation produced.

As another example, assume that R_1 contains 1000 pages, that 25% of the tuples in R_1 satisfy the restriction, and that 100 query processors are assigned to perform the restriction. Then each query processor will examine (approximately) 10 pages of R_1 and will produce, on the average, 2.5 pages of R_1'. Therefore, before R_1' is compressed it will contain 300 pages. Two hundred of these pages will be full and 100 will be 50% full.

To proceed more formally we need the following definitions. Let QP be the number of query processors which are available to execute a query. For a restrict operator, let RF be the fraction of the tuples that satisfy the search criteria. Finally, for each relation, R_i, in the database, three pieces of information are maintained. S_i is the size of R_i in pages. QPF_i is the number of the S_i pages that are only partially filled. For those pages that are partially filled, F_i indicates how full each page is (on the average). For our previous example $S_1 = 1000$, $QPF_1 = 0$, $F_1 = 0$, $S_1' = 300$, $QPF_1' = 100$, and $F_1' = 0.5$.

Assume that QP query processors are assigned to restrict relation R_i with size S_i. If $P_{max} = MOD(S_i, QP)$ and $P_{min} = QP - P_{max}$, then P_{max} processors will each examine $C_{max} = \left\lfloor \frac{S_i}{QP} \right\rfloor + 1$ pages and P_{min} processors will each examine $C_{min} = \left\lfloor \frac{S_i}{QP} \right\rfloor$ pages of R_i. The size of the resulting relation R_i', S_i', will be:

$$P_{max} * \lceil C_{max} * RF_i \rceil + P_{min} * \lceil C_{min} * RF_i \rceil \text{ pages.}$$

QPF_i' will equal:

$$P_{max} * \lceil \text{fraction}(C_{max} * RF_i) \rceil + P_{min} * \lceil \text{fraction}(C_{min} * RF_i) \rceil$$

and F_i' will be equal to:

$$\frac{P_{max} * \text{fraction}(C_{max} * RF_i) + P_{min} * \text{fraction}(C_{min} * RF_i)}{QPF_i'}$$

Finally, if $QP > S_i$, then only S_i query processors will be used to perform the restriction.

5.2 The Compress Operator

The purpose of the compress operator is to eliminate relation fragmentation by compressing those pages of a relation which are only partially full. This can be accomplished by reading partially full pages and writing full ones.

If CP is the number of query processors used to compress relation R_i and T_{pr} is the time required to move a page between a query processor's local memory and the CCD memory then the time required to perform the compression is:

$$T_c = \left\lceil \frac{QPF_i}{CP} \right\rceil * T_{pr} + \left\lceil \left\lceil \frac{QPF_i}{CP} \right\rceil * F_i \right\rceil * T_{pr}$$

The first term represents the time required by the CP query processors to read the QPF_i partially filled pages and the second term represents the time required to write the compressed pages. The size of the compressed relation R_i' is:

$$S_i' = S_i - QPF_i + MOD(QPF_i, CP) * \left\lceil \left(\left\lfloor \frac{QPF_i}{CP} \right\rfloor + 1 \right) * F_i \right\rceil$$

$$+ (CP - MOD(QPF_i, CP)) * \left\lceil \left\lfloor \frac{QPF_i}{CP} \right\rfloor * F_i \right\rceil$$

As with the restriction operator, there is a time/fragmentation tradeoff associated with the compress operator. As the number of query processors employed increases, T_c, the time required to compress the relation, decreases but the value of S_i' increases since each of the CP query processors will terminate with a partially filled page. As we will demonstrate in Section 5.4, the optimal value of CP cannot be determined without examining how the compressed relation is used by subsequent operations in the query.

5.3 Employment of the Compress Operator

Consider the query shown in Figure 2.1. Since both R_1' and R_2' contain partially filled pages and since the execution time of the join is proportional to the product of the sizes of the two source relations, the obvious choice is to compress both relations before performing the join. The modified query tree representing this choice is shown in Figure 5.1. As we will now demonstrate, compressing R_1' will actually increase the execution time of the query.

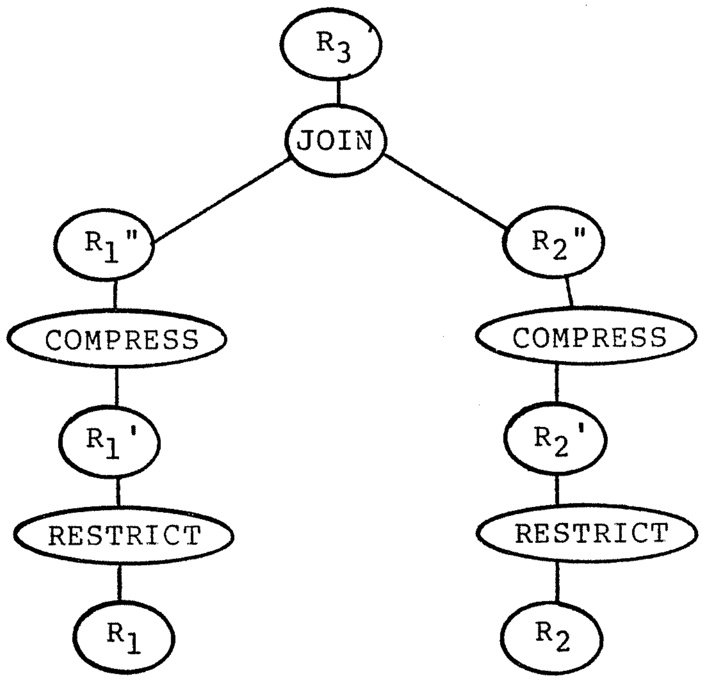


Figure 5.1

To understand why this is true, we must examine how DIRECT executes a join operation. If the number of query processors assigned to execute the join in Figure 5.1 is equal to the size of R_1" (i.e. QP = S_1") then each query processor will read one page of R_1". It will then attempt to join the tuples on its page of R_1" with <u>all</u> the tuples in R_2" by reading R_2" one page at a time.

There are several alternative reasons why compressing R_1' will increase the execution time of the join operation. If $QP \approx S_1'$, then compresssing R_1' will reduce the number of query processors which can be utilized to execute the join operation from S_1' to S_1". Furthermore, since compression does not reduce the number of tuples in a relation, each of the S_1" query processors will have to examine more tuples from relation R_1". Therefore, each query processor will produce a greater percentage of R_3. This will result in more page writes per query processor and hence increased processing time for the complete query.

If QP << S_1" (the size of the compressed relation), the above argument will no longer be valid. Rather, if R_1' is compressed, then since $S_1" < S_1'$ the time spent by each query processor reading pages of R_2" will be reduced:

$$\left\lceil \frac{S_1"}{QP} \right\rceil * S_2" * T_{pr} < \left\lceil \frac{S_1'}{QP} \right\rceil * S_2" * T_{pr}.$$

If CP query processors are assigned to compress R_1' before the join operation, then $\left\lceil \frac{QPF_1'}{CP} \right\rceil * T_{pr}$ seconds will be used to read the partially filled pages. Additional time will also be required to write the compressed pages and then reread the pages of R_1" to perform the join.

An alternative approach is to make the compression of R_1' an implicit component of the join. By having each of the QP query processors attempt to read enough partially filled pages of R_1' to fill its buffer before proceeding to read the first page of R_2'', the compression will, in effect, be performed in only

$$\left\lceil \frac{QPF_1'}{QP} \right\rceil * T_{pr} \text{ seconds.}$$

If $QP \approx S_1' \approx QPF_1'$, then each query processor will fail in its attempt to fill its buffer. It will, therefore, proceed to join a partially filled buffer containing tuples of R_1' with all the tuples in R_2''. By using this approach we achieve the best aspects of both approaches. We do not limit the potential concurrency when $QP \approx S_1'$ and yet we will achieve the performance improvement of compression when $QP << S_1''$.

While explicit compression of R_1' is not beneficial, compression of R_2' certainly will be. Since each query procesor will, for each page of R_1' it examines, read all of R_2', compression of R_2' will significantly reduce the number of page reads performed and hence the execution time of the query. There is, however, a tradeoff between the benefits obtained by doing compression versus the cost of performing the compression. In the following section, we will develop a method for choosing the value of CP so as to minimize the cost of performing the compression while maximizing the resulting benefits.

5.4 Calculation of the Optimal Value of CP

As discussed in Section 5.2, there is a tradeoff between minimizing the time required to perform a compression, T_c, and the size, S_i', of the compressed relation. As the value of CP increases, T_c will decrease but S_i' will increase. Since the compressed relation is subsequently used as an operand in a join (R_2'' in Figure 5.1), any increase in the value of S_i' will also increase the execution time of the join. In this section we will develop a formula for determining the value of CP such that the execution time of the compress and join operations is minimized.

In Section 5.2, expressions for T_c and S_i' were developed. The time to execute the join, T_j, is expressed by:

$$T_j = T_r + T_e + T_w$$

T_e represents the time required to join tuples of the two source relations and is independent of the number of pages in relations R_1' and R_2'' (S_1' and S_2'' respectively). The time required by the query processors to move pages of the result relation from local memory to a CCD memory module is T_w. T_w depends only the number of tuples produced by the join and QP, the number of query processors used to execute the join. T_r, the time spent reading pages of the two source relations can be expressed as:

$$T_r = \left\lceil \frac{S_1'}{QP} \right\rceil * T_{pr} + \left\lceil \frac{S_1'}{QP} \right\rceil * S_2'' * T_{pr}$$

Since T_e and T_w are independent of S_1' and S_2'' and since

$$\left\lceil \frac{S_1'}{QP} \right\rceil * T_{pr}$$

is not effected by the compress operator, T_j can be rewritten as:

$$T_j = \left\lceil \frac{S_1'}{QP} \right\rceil * S_2'' * T_{pr} + c$$

The time required to execute both the compress and the join, $T = T_c + T_j$, is thus:

$$T = \left\lceil \frac{QPF_2'}{CP} \right\rceil * T_{pr} + \left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil * T_{pr} + \left\lceil \frac{S_1'}{QP} \right\rceil * S_2'' * T_{pr} + c$$

Since the result of the compression is to reduce the size of relation R_2' from S_2' to S_2'', we can substitute the expression for

$S_2"$ developed in Section 5.2 and divide by T_{pr}. This yields:

$$T = \left\lceil \frac{QPF_2'}{CP} \right\rceil + \left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil$$
$$+ \left\lceil \frac{S_1'}{QP} \right\rceil * (S_2' - QPF_2' + MOD(QPF_2',CP) * \left\lceil (\left\lfloor \frac{QPF_2'}{CP} \right\rfloor + 1) * F_2' \right\rceil$$
$$+ (CP - MOD(QPF_2',CP)) * \left\lceil \left\lfloor \frac{QPF_2'}{CP} \right\rfloor * F_2' \right\rceil) + c_1$$

T can be simpified further by consolidating those terms which are independent of CP into c. This simplification yields:

$$T = \left\lceil \frac{QPF_2'}{CP} \right\rceil + \left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil$$
$$+ \left\lceil \frac{S_1'}{QP} \right\rceil * (MOD(QPF_2',CP) * \left\lceil (\left\lfloor \frac{QPF_2'}{CP} \right\rfloor + 1) * F_2' \right\rceil$$
$$+ (CP - MOD(QPF_2',CP)) * \left\lceil \left\lfloor \frac{QPF_2'}{CP} \right\rfloor * F_2' \right\rceil) + c_2$$

The obvious approach of differentiating T with respect to CP to find that value of CP which minimizes T will not work since this function is not continuous. We have plotted T, T_c (the first two terms of T), and T_{rj} (the third term of T) as functions of CP in Figures 5.2 and and 5.3 for two different sets of data.

As is illustrated by Figures 5.2 and 5.3, T_c is a nonincreasing step function of CP. Thus, $CP_j > CP_i$ implies $T_c(CP_j) \leq T_c(CP_i)$. T_{rj}, on the otherhand, remains relatively constant (with minor fluctuations) until a certain value of CP, CP_m, is reached. At $CP=CP_m$, T_{rj} begins to increase as a linear function of CP until CP becomes equal to QPF_2'. At this point, and for all values of CP > QPF_2', compression will have no effect since each of the CP processors will read exactly one page of R_2'.

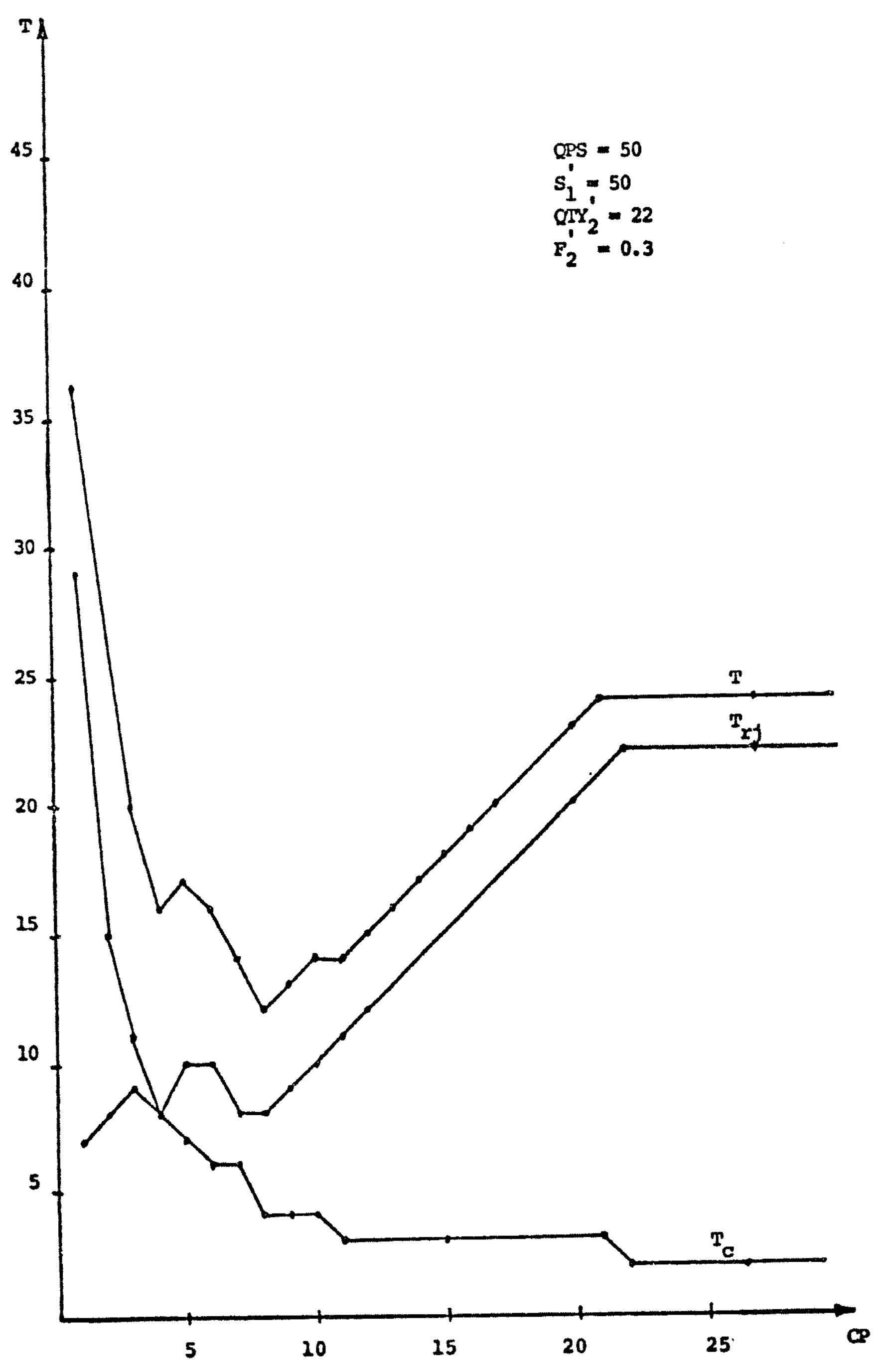


Figure 5.2

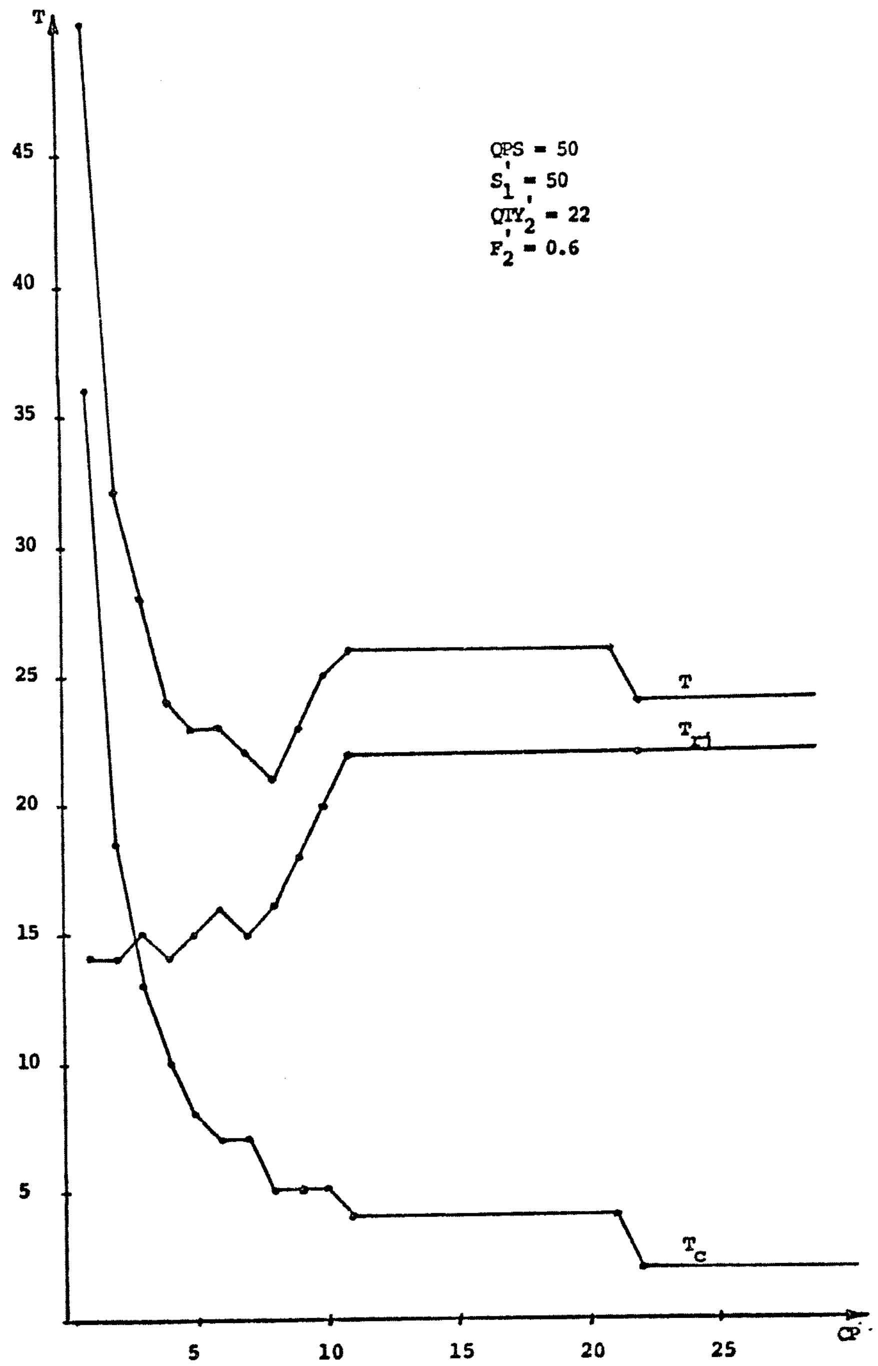


Figure 5.3

An obvious choice for that value of CP which may perhaps minimize T is CP_m. While values of CP less than CP_m may produce values of T_{rj} less than the value of T_{rj} at CP_m, the decrease in T_{rj} is more than offset by a large increase in T_c. We can determine a formula to calculate CP_m, by first observing what changes occur in T_{rj} at CP_m.

Starting with CP=1, as more and more query processors are used to compress a relation, the maximum number of pages of the compressed relation produced by any one processor decreases. For a given value of CP, this number is expressed by:

$$\left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil$$

As long as CP is less than by QPF_2', the minimum value of this expression is $\lceil 2 * F_2' \rceil$. Thus, when $F_2' \leq 0.5$, $\lceil 2 * F_2' \rceil$ is equal to 1. Otherwise, when $F_2' > 0.5$, it is equal to 2. The first value of CP such that

$$\left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil$$

equals 1 if $F_2' \leq .5$ or 2 if $F_2' > .5$ will be CP_m. As CP grows larger than CP_m, the maximum number of pages produced by any one processor will continue to equal to 1 (or 2). Thus, S_c (the size of the compressed relation) and hence T_{rj} will grow as a linear function of CP (with slope of 1 or 2 depending of the value of F_2') until CP becomes equal to QPF_2'. At this point S_c will become equal to QPF_2'.

Solving $\left\lceil \left\lceil \frac{QPF_2'}{CP} \right\rceil * F_2' \right\rceil = n$, n=1,2 for CP yields:

$$CP = \left\lceil \frac{QPF_2'}{\left\lfloor \frac{n}{F_2'} \right\rfloor} \right\rceil$$

If F_2' is less than 0.5, then evaluating this expression with n equal to 1 will yield the correct value of CP_m. Otherwise, it should be evaluated with n equal to 2.

For QPF_2' = 22and F_2' = 0.30, evaluation of the formula for CP yields CP = 8. As indicated in Figure 5.2, this is the correct value for CP_m and is also the value of CP that minimizes T. The expression for CP also gives the correct value for the example shown in Figure 5.4.

The expression for CP can also be used to determine whether compression of R_2' will be beneficial in terms of reducing the time to execute the join. For example, assume that QP=50, S_1'=50, QPF_2'=22, and F_2' = 0.826. Plotting T as a function of CP for this example indicates that the minimum value of T occurs when CP=22. Evaluation of the expression above yields CP_m=11. Since F_2' is greater than 0.5, each of the 11 processors will produce two pages of the compressed relation. Therefore, the size of the compressed relation will be 22 and hence it is not worthwhile to perform the compression.

CP_m will not, however, always give an accurate estimate of that value of CP which will minimize T. For example, if QP = 50, QPF_2' = 22, S_1' = 50, and F_2' = 0.0826, the value of CP_m which is computed is 2. The minimum value of T, however, occurs when CP = 4. This situation arises because T_c is the dominating term in T. If S_1' equaled 150 instead of 50, then CP_m would have indeed been the value of CP which minimized T.

Therefore, by evaluating $CP = \left\lceil \frac{QPF_2'}{\left\lfloor \frac{n}{F_2'} \right\rfloor} \right\rceil$ the optimal number of query processors to perform the compression can be estimated. If F_2' is less than or equal to 0.5, this expression should be evaluated with n equal to 1. Otherwise, it should be evaluated with n equal to 2. If $F_2' > 0.5$, then compression will be worthwhile if and only if $CP_m * 2$ is less than QPF_2'. While CP_m may not always be the value of CP which minimizes T, it appears to generally be a very good estimate of it. Furthermore, as is indicated by Figures 5.2 and 5.3, compression of R_2' using CP_m processors significantly reduces the execution time T when compared to the execution time of the query when R_2' is not compressed (i.e. the value of T when $CP = QPF_2'$).

6.0 Conclusions

In this paper we have discussed query organization, execution, and optimization in DIRECT. The use of a monitor for each relation referenced by a query along with the use of the NEXT_PAGE construct permits the DIRECT back-end controller to assign a query to any number of processors for execution. Furthermore, these constructs also permit the controller to balance the load in the back-end by dynamically adjusting how many processors are assigned to each executing query.

We have also identified the problem of relation fragmentation which occurs when queries are executed in parallel and developed a technique for estimating the optimal number of processors to compress a relation so that the execution time of the entire query is minimized. These results appear to be applicable to all database machines which employ parallel processing techniques to enhance query execution.

7.0 Acknowledgements

I would like to express my appreciation to Haran Boral, Raphael Finkel, and Ron Tischler for their helpful suggestions and criticisms of earlier versions of this paper.

REFERENCES

1. Dewitt,D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978.

2. Dewitt,D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," To appear in IEEE Transactions on Computers, Spring 1979. Also: Computer Sciences Technical Report #325, Univ. of Wisconsin, June 1978.

3. Ozkarahan, E.A.,Schuster,S.A., and K.C. Smith,"RAP - An associative processor for database management," Proceedings of the 1975 NCC,pp.379-386.

4. Schuster,S.A., Ozkarahan,E.A., and K.C. Smith, "A virtual memory system for a relational associative processor," Proceedings of the 1976 NCC, pp. 855-862.

5. Ozkarahan, E.A., Schuster, S.A., and Sevcik, "Performance of a Relational Associative Processor," ACM Transactions on Data Base Systems, Vol. 2, No. 2, June 1977, pp 175-195.

6. Stonebraker, M.R., Wong, E., and P. Kreps, "The design and implementation of INGRES," ACM Transactions on Data Base Systems, Vol. 1, No. 3, Sept 1976, pp.189-222.

7. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM Vol. 17,10, Oct. 1974, pp. 549-557.