TR-81-007

Efficient Computing of Joins by Means of Specialized Hardware

by

Yang-Chang Hong

This research is supported by the National Science Council under Grant NSC-70E-0404-E001-02

Institute of Information Science
Academia Sinica
Nankang, Taipei 115
Republic of China

December 1981

ABSTRACT —— A hardware architecture for supporting relational join operations is described. The architecture is intended as extending functional capabilities of existing relational associative database machines (RADM). It allows the rapid execution of join operations. The main features of this architecture are an RAM, which can not only rapidly remember or recall data but filter out irrelevant data of the join, and an array of queue servers which can form new tuples of the join in parallel. This architectural design emphasizes on much parallelism in the cross referencing, and thereby gives a significant performance improvement over existing join algorithms in RADMs, especially when a join-dominating application is involved. The resulting RADMs allow that the data search is performed by the search logic, while the join operations are carried out by this extended hardware.

1. Introduction

1.1 Problems with Existing RADMs

The limitations of conventional systems in supporting relational DBMs functions, as well as advances in processor and memory technologies, is prompting the design and development of directly associative hardware support for database applications [2,3,4,5,6,7,8,9,10,13,14,16]. Relational algebra operators have received the great attention as a candidate for the design. Currently, the design for implementing the join has been to concentrate on a form so called the "implicit join" in which the values of the columns being joined, called the join columns, from the selected tuples in one relation are transferred to select tuples in the second or the same relation that have the same those values in their join columns; it does not create a derived relation from the original relation(s).

The implicit join of two relations is executed in CASSM and CAFS [1,14] by first recording the join column values from the selected tuples in one relation in the single bit array store. Next the tuples of the second relation are matched on their join columns against the values in the store and marked if the match succeeds. RAP [10] implements the implicit join operation by storing k join column values of the first relation in k comparators of each cell as a disjunctive condition for the second relation in each pass. The implicit join in Chang's machine [3] is implemented by fetching the join column values, one by one, as search conditions for the second relation. The number of passes required in RAP and Chang's machine obviously depends on the number of values selected in the first relation. Existing relational associative database machines (RADM for short) fail to consider directly associative hardware support for "explicit joins"

(as contrasted to implicit joins) in which more data other than those in the join columns from the selected tuples in one relation are used to concatenate to the selected tuples in the second (or the same) relation that have the same join column values as those selected tuples of the first relation.

Most existing RADMs [2,9,10,11,12,13] are based on the parallel processing of the segmented sequential search and cannot efficiently support the explicit join operation,which often involves performing (possibly) a large amount of cross checking for forming new tuples, by means of the parallelism. Thus, they are not alone sufficient to make a high-performance database machine, especially when an application with join-dominating queries is involved. A new hardware architecture for efficiently implementing this type of join has to be sought to cope with the join-dominating database applications.

1.2 Recent Approaches for Joins

Several designs based on the host computer for the explicit join operation have been reported [1,8,9]. The LEECH and CAFS machines use a filter for selecting tuples needed for the join. The selected tuples are sent to the host to form the concatenated tuples of the join. The only difference between these two machines is the design of the filter. RARES provide a hardware-support algorithm for dividing the tuples of each relation being joined into buckets according to the value-intervals of their join columns. The tuples within each bucket are sorted in the main memory by conventional algorithms. The sorted buckets are used to produce the concatenated tuples of the join by the host. The approaches described above will not be very effective when the number of tuples being joined

is large. There has been proposed a totally hardware-support join based on pipeline searching and sorting engines for a data flow database computer [15]. Two heap trees of the two relations being joined are first constructed using the heap algorithm. The two heap trees are then converted into two binary trees before the tuples can be combined to form new tuples by the searching engine. One disadvantage of this approach is that a considerable amount of logic and memory is involved.

1.3 The Proposed Approach

This paper describes an on-going National Science Council sponsored project on the study of an efficient hardware-support algorithm for full joins. Our hardware architecture for joins is intended as a component of existing RADMs. It allows the rapid execution of join operations. The main feature of this architecture is an RAM extending the single bit array stores, as suggested by CASSM and CAFS for implicit joins, which allows to remember or recall more data than the bit store does. Through the use of the RAM, the join of two relations, in which the primary key or candidate key in one of the two relations is the join column, can be implemented as effectively as the implementation of implicit joins by the single bit array stores, thus, without actually creating a derived relation. For those joins which need to be implemented by creating a temporary relation, we suggest a memory bank for storing tuples in one of the relations being joined, as well as an array of queue servers for forming new tuples from the memory bank and the array of queues associated. The processing time is decreased in executing this type of join by increasing the parallelism of producing the concatenated tuples of the join in each server. This architectural design emphasizes on much parallelism in the cross referencing and thereby gives a significant performance improvement over existing join

algorithms in RADMs, especially when a join-dominating application is involved. The resulting RADMs allow that the data search is performed by the search logic, while the join operations are carried out by the extended hardware.

1.4 Organization of the Paper

The body of this paper is divided into three parts. In the first part the proposed hardware architecture is described. The second and third parts are concerned with the algorithms for computing the joins without and with the need for being implemented by creating a temporary relation, respectively. This is followed by a summary and the status of the project. A scheme optimizing the query expression evaluation based on the extended version of RADMs will be reported in the near future.

2. Hardware Architecture

Figure 1 shows the proposed hardware architecture which is intended as an extention of functional capabilities of existing RADMs, e.g., CASSM, RAP, REARES, DBC, etc. [2,9,10,13]. The architecture accepts a sequence of relational tuples or column values (in an encoded form) from the RADM in which data are stored in its memory segments and the content and context searches are performed in parallel by the logic associated with each segment. (Here data are assumed to be encodedly stored in the RADM; this is so assumed because of the fact that some RADMs claimed it is advantageous to deal with data in encoded form [10,13].) The command and control processor (CCP) receives the high-level data manipulation requests from the host computer; it translates them into commands for the RADM and the extended hardware (including the determination of the encoded values for each column value in the requests), distributes commands to the RADM and the extended hard-

ware for execution, receives and decodes the data transferred out of the RADM or the extended hardware, and outputs the data to the host computer.

The proposed hardware architecture consists of five major components —— IP, MB, RAM, S, and CP —— as illustrated in Figure 1 where

(1) IP is an input processor which accepts a sequence of tuples or column values in a coded form from the RADM and deposits them in queue Q. The queue Q acts as a buffer between the RADM and the extended hardware. There are two registers T and H and one flag F_Q in Q. The T- and H-registers are used to hold the locations of the last and first entries of Q. The flag F_Q is used to indicate whether the Q is full or not (which is set if full). The setting of the flag F_Q will notice the RADM to stop outputting tuples or column values to IP. The IP will start its processing whenever F_Q is reset.

(2) MB is a memory bank consisting of a set of memory modules M(i). It is designed to hold the tuples of one of the two relations being joined. In Figure 1, p memory modules are shown, where p is a design parameter.

(3) The RAM consists of two single bit array stores rA and rB and an array r of words. As shown in Figure 1, RAM is defined as RAM [0:M, 0:N]. The rA, rB, and r consisting of r' and r" are defined as shown in the figure. The rA and rB are addressed by encoded column values, which, like those in CASSM and CAFS, can hold immediate results of the join operations. Each word in r is also addressed by encoded column values. It can be used to store an encoded value, a counter, or a pointer (to be detailed later). The r is vertically divided into two arrays of words r' and r" with r' = RAM[0:M, 2:x] and r" = RAM [0:M, x+1:N] if it is treated as an array of pointer words. They are both addressed by encoded values. The use of the r is explained in the join algorithms of the next sections. The value x is determined by the number of memory modules used, while the value N is determined by the x and the size of each module. If

If MB consists of p modules and each module has q words, then the values $x = 1 + \lceil \log_2 p \rceil$ and $N = x + \lceil \log_2 q \rceil$, where $\lceil y \rceil$ is the least integer greater than or equal to y.

(4) S is a set of queue servers S_i, each of which has a queue Q_i associated with it. The array of queues Q_i is served to hold the incoming tuples of one of the two relations to be joined. Like the queue Q, associated with each queue Q_i are two registers T_i and H_i and one flag F_i. They serves the same functions as explained in (1). Each server S_i is designed to read data from its Q_i and the corresponding module M(i). Thus, there are as many servers S_i's or Q_i's as M(i)'s in the architecture. A buffer is provided for each S_i for holding the data either being output to the host computer or being stored back to the RADM for further processing. The transfer of the data temporarily stored in the buffer is accomplished by an output mechanism.

(5) CP is a central processor which fetches tuples or column values from Q and uses these data (or the extracted data from the tuples) as indexes to address the rA or rB for setting to 1 or 0, or testing for being 1 or 0, or to locate the desired words in r for many purposes to be detailed later. The CP also serves to allocate storage space in MB for storing the tuples of one of the two relations being joined. There are one T-register, one D-register, and a set of registers BR(i), $1 \leq i \leq p$, for being used during storage allocation. They are best explained when they are used.

3. Join Algorithms Using the RAM

This section shows how the store RAM performs the join operation. We shall first consider the implementation of implicit joins and then consider the implementation of explicit joins which can be implemented without creating a derived relation from the original relations.

3.1 Queries Involving the Implicit Join of Relations

Implementing an implicit join by the bit stores rA and rB is best explained by means of an example.

Example 1.

Print all the green items sold by the D1 department.

To answer this query, a simplified database with tables SALES and TYPE are assumed in Figure 2. This query can be implemented by various ways. One way is to apply the selection process to the table SALES to select the items sold by the D1 department. The selected items are then transferred to the table TYPE as a disjunctive condition to retrieve all the green items. The procedure using the single bit array store rA to implement the way just described is outlined below :

(1) Clear the single bit array store rA.

(2) Scan the table SALES by RADM and output the items sold by the D1 department to the input processor IP of the extended architecture. The items fed to IP are then queued in Q, which in turn are fetched and recorded in the single bit array store rA by central processor CP.

(3) Scan the table TYPE by RADM and output all the green items and store them in Q. Any item in Q is output to the host computer if the CP checks that it has been recorded in rA, i.e., it is an item sold by the D1 department.

Here we assume that the reader is familiar with the data search performed by the RADM. What is not made clear is the function of the single bit array store rA; how the CP records the items in rA and how it determines which green items are to be output to the host.

Recall that data are encodedly stored in the RADM. We believe that it is feasible to encode the values of the columns to be joined in such a way that they can be used as indexes to address the single bit array store. Each bit

position in the array can be made corresponding to an encoded value. Using this technique, the addressed bit can be set to 1 or 0, or tested for being 1 or 0. We give a real example to illustrate this technique.

Assume that BOLT is encoded as 0, i.e., <BOLT> = 0, and <CAM> = 1, <COG> = 2, <GEAR> = 3, <NUT> = 4, and <SCREW> = 5. At the end of step (2), the bit pattern of rA will be (0,1,0,1,0,..., 0). This pattern would record the list of items CAM and GEAR. In step (3), items <BOLT> and<GEAR> are selected and stored in Q for examination. Since rA(<BOLT>) = rA(0) = 0, BOLT is discarded. Similarly, rA(<GEAR>) = rA(3) = 1, <GEAR> is output to the host computer. Before <GEAR> is output, it is decoded by the encoding and decoding unit in CCP. Of course, values D1 and GREEN in the query have to be encoded by EDU before the query is executed. (We neglect the detailed encoding and decoding processes here.)

The discussion above assumes that all the encoded ITEM values are within the address space of rA. If not, they are divided into buckets; the values in the first bucket lie between 0 and $2^t - 1$; the values in the second bucket lie between 2^t and $2^{t+1} - 1$; and so forth, where t is the number of bits required in the address space. Each bucket is then evaluated by repeatedly applying the same procedure being described.

The idea of using the single bit array store to remember or recall data is the same as those used in CASSM and CAFS. CASSM uses a single bit array store per cell (consisting a memory element and a processing logic) and one logical single bit array store, consisting of the concatenation of single bit array stores of cells, addressable by each processing logic. To address a bit in the logical array store requires passing the bit address (i.e. the encoded value) from one logic to another. Moreover, only one cell is allowed to address the logical array store at one time. If two cells want to address the store simu-

taneously, one of the two cells must wait for the subsequent revolution. This means additional memory revolutions are required in addressing the bit array store. Our approach, like CAFS, uses a central processor CP to set or test a single bit array store, thereby eliminating memory addressing contention. Because of the use of an RADM, which acts as a filter, less data than CAFS are fed to the CP for setting or testing the bit array store.

Note that the single bit array store rB is used when the values selected from the second relation are transferred to select tuples in the third relation that have the same those values in their join columns. The stores rA and rB are alternately used if a chain of relations are involved in the implicit join operation.

3.2 Explicit Joins Without Creating a Derived Relation

We have mentioned in Section 1.3 that a certain type of explicit joins can be as rapidly executed as the implicit join by means of RAM, i.e., without actually creating a temporary relation. We uses two examples to illustrate their implementation in this section.

Example 2.

Find the names of the employees who make more than their department managers. The query is directed at the table

EMPLOYEE(NAME,SALARY,DEPT,MGR)

where the managers are also employees —— i.e., the values in the MGR column also appear in the NAME column.

One way to answer this query is to use the managers selected from the MGR column to select EMPLOYEE tuples where NAME = 'one of the selected managers.' Then, join the names and their salaries from the selected tuples with those EMPLOYEE tuples that have the same those names in their MGR column. Finally, scan the joined relation and output the employee names whose salaries are

greater than their managers. This method performs an implicit join followed an explicit join and a selection operation. It is insufficient to use single bit array stores to remember the manager's names and their salaries for being used to select those employee names who make more than their managers. Our approach records the manager's names in the single bit array store and uses their names as indexes to address the array r of words. The addressed words are then used for storing their salaries, respectively. By doing so, each qualified employee's name can be output by one implicit join followed one explicit join. We outline the procedure below :

(1) Clear the single bit array stores rA.

(2) Scan the EMPLOYEE table by RADM and output the entries in the MGR column and store them in Q. The entries stored in Q are then fetched and used to set the single bit array store rA by CP.

(3) Scan the EMPLOYEE table again and output the employee names and their salaries and stored them in Q. The pairs (<name>,<salary>) stored in Q are then fetched, one by one, to test if rA(<name>) is 1 or not. If rA(<name>) = 1, then the <salary> is stored in the corresponding word in r. i.e., r(<name>)← <salary>. Otherwise, discard the pair.

(4) Scan the EMPLOYEE table again and output the employee names, salaries, and managers and store them in Q. Each triple (<name>,<salary>,<manager>) stored in Q are then fetched to test if r(<manager>) < <salary> or not. If yes, output the <name>. Otherwise, discard the triple being held.

We notice that the encoded values in the SALARY column should have the same order as they originally have. This procedure combines one explicit join and one selection operation to a single process in which the managers' names and their salaries are recorded in RAM and each incoming EMPLOYEE tuple

is virtually concatenated to a proper entry in RAM so that the qualified employee names can be determined immediately. This technique can also be used to perform the join of two relations, which needs to form new tuples from the two original relations, if the primary key (or candidate key) of one of the two relations is the join column. We give an example to illustrate the implementation of this type of join.

Exmaple 3.

Join the tuples of the SALES table with those TYPE tuples having items which price is greater than 4P and output the DEPT, ITEM, and COLOR columns. This is a typical explicit join of two relations. It can be realized by the following procedure : Assume that ITEM is a primary key of TYPE. Thus, ITEM in table SALES is a foreign key.

(1) Clear rA.

(2) Scan the TYPE table and output the items and their color if their price is greater then 4P and store them in Q. Each pair (<item>,<color>) stored in Q is feteched and recored in RAM. That is, rA(<item>) ← 1 and r(<item>) ← <color>.

(3) Scan the SALES table and output the SALES tuples to Q. Each tuple (<department>,<item>) stored in Q is fetched to test if rA(<item>) = 1 or not. If yes, read r(<item>) and concatenate it to the tuple being held and output the new tuple (<department>,<item>,<color>) to the host computer. Otherwise, discard the tuple.(Note : if the resulting relation of the join is to be used for further processing, then each new tuple is stored back to the RADM.)

If the columns of DEPT, ITEM, COLOR, and PRICE are involved in the join of the tables SALES and TYPE, then it can be carried out by the following two sub-joins : one is the join of the tables SALES and $TYPE^{(1)}$ (ITEM, COLOR) via ITEM, denoting the resulting table as R1 = (DEPT, ITEM, COLOR), and the other the join of the tables R1 and $TYPE^{(2)}$ (IETM, PRICE) via ITEM. Each join can

follow the similar procedure being described above. In general, for two tables A and B with A = (A1, A2,..., Am) with key A1 and B = (B1, B2, ..., Bn) with key B1 being joined via Ai = B1, it can be carried out by (n - 1) sub-joins : the first one is the join of the table A and $B^{(1)}$(B1,B2) via Ai = B1, the second one A and $B^{(2)}$(B1,B3) via Ai = B1, ..., and the (n - 1)st one A and $B^{(n-1)}$(B1,Bn) via Ai = B1. And each join can be carried out by the above procedure. If the join column is not a primary key or a candidate key, then a derived relation has to be actually created in our approach.

4. Explicit Joins with the Need for Actually Creating Derived Relations

4.1 General Description

A join which needs to be implemented by actually creating a derived relation may make the implementation rather costly in time and storage. To reduce the storage cost, our design is to divide any "large" relations being joined into buckets, according to their join column values in such a way that the first bucket of the first relation is to join with the first bucket of the second relation; the second bucket of the first relation is to join with the second bucket of the second relation; and so forth. Two relations being joined are "large" if they satisfy one of the following conditions : (1) The range of the encoded values of the join column is over the size of the address space of RAM and (2) The smaller one of the two relations being joined cannot be fitted in the memory bank MB. The processing time is descreased by increasing the parallelism of computing the concatenated tuples of the join of buckets. This parallelism is achieved by further dividing the tuples of buckets being joined into sub-bucket, based on their join column values. The first sub-bucket of the first bucket is then joined with the first sub-bucket of the second bucket; the second sub-bucket of the first bucket is joined with the second sub-bucket

of the second bucket; and so forth. The join of the pairs of sub-buckets is done in parallel by the extended hardware. In logical effect, it is carried out in our approach by producing the concatenated tuples of the join from two "sorted" sub-buckets.

In our design, the division of the large relations being joined into buckets is relegated to the RADM, similar to RARES[9], so that less tuples are output to the extended hardware. The pairs of buckets are then sent to the extended hardware, one by one, for computing the join. In computing the join of two buckets, the extended hardware uses two single bit array stores rA and rB for filtering out the irrelevant tuples of the join since the join column values in one bucket may not appear in another. We will see that it is worthy of doing so, especially when a large number of irrelevant tuples are involved. The array r of words, except helping with one certain type of explicit joins on remembering or recalling data as described previously, can also help with another type of explicit joins on dividing tuples of each bucket into sub-buckets. For two buckets being joined, the sub-buckets of one bucket are first stored in the memory modules M(i) of the memory bank, one per module. Each incoming tuple of the second bucket is then stored in the corresponding queue Q_i; that is, the first queue Q_1 accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(1); the Q_2 accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(2); and so forth. This arrange permits each queue server S_i ($1 \leq i \leq p$) to produce the concatenated tuples of the join from its queue Q_i and the M(i) logically associated with it in parallel, without any memory addressing contention. What is not made clear here is how each S_i can know which tuples in M(i) are concatenated to the tuple being fetched from Q_i. This can be seen from the following algorithm.

4.2 Algorithm for Explicit Equi-Joins of Two Buckets

Let us denote the two buckets being joined as R_A and R_B of relations A and B with $A = (X_1, X_2, \ldots, X_u)$ and $B = (Y_1, Y_2, \ldots, Y_v)$, respectively, where $X_i (1 \leq i \leq u)$ and $Y_j (1 \leq j \leq v)$ are column names. Assume that columns X_a and Y_b are of the same underlying domain. Compute the join of buckets R_A and R_B over $(X_a = Y_b)$. The resulting table consists of the set of tuples t, where t is the concatenation of a tuple t' belong to R_A and a tuple t" belong to R_B and $x_a = y_b$ (x_a being the X_a-component of R_A and y_b being the Y_b-component of R_B). The algorithm for the join is outlined below :

(1) Initialization : Clear rA, rB, and r.

(2) Output X_a-components of R_A and set the rA and increment the corresponding counter words of r : Clear Registers T and H and the flag F_Q of Q. Scan the relation A by RADM and output the sequence of X_a-components x_a's of R_A (in encoded form) to IP. The IP accepts each component x_a and deposits it into Q. The x_a's in Q are then fetched, one at a time, by CP and used as indices to address the bits in rA and the corresponding counter words in r. The addressed bits of rA are set and the corresponding counter words are incremented by one. For example, if x_{ai} is fetched, then $rA(x_{ai}) \leftarrow 1$, and $r(x_{ai}) \leftarrow r(x_{ai}) + 1$. At the end of step(2), the word $r(x_{ai})$ contains a value indicating the number of R_A tuples with the component X_{ai} in their join columns. The discussions which follow use $<r(x)>$ to denote the contents or value of the word in r addressed by the component x.

(3) Output Y_b-components in R_B, set the rB, and allocate memory space in MB for R_A : Clear registers T and H and the flag F_Q of Q and BR(i), $1 \leq i \leq p$, and $D \leftarrow 1$. Scan the relation B by RADM and output the sequence of Y_b-components y_b's of R_B to IP. The IP accepts each y_b and deposits it into Q. The y_b's in Q are then fetched by CP and used to test if the corresponding bits in

rA and rB are set or not.

Cases : (i) if $rA(y_{bj}) = 0$, i.e., the y_{bj} does not appear in the join column of R_A, then ignore the component y_{bj}.

(ii) if $rA(y_{bj}) = 1$ and $rB(y_{bj}) = 0$, i.e., the y_{bj} is first encountered, then $rB(y_{bj}) \leftarrow 1$ and allocate memory space in MB for storing R_A tuples with $X_a = y_{bj}$.

The setting of $rB(y_{bj})$ will prevent the subsequent incoming $y_b = y_{bj}$ from re-allocating memory space in MB for those R_A tuples having $X_a = y_{bj}$. The memory allocation is done as follows : (Initially, registers BR(k), $1 \leqq k \leqq p$, are cleared and $D \leftarrow 1$, i.e., each BR(k) points to the starting address of k-th module and D points to the first memory module.)

(a) $T \leftarrow r(y_{bj})$, i.e., the value of word $r(y_{bj})$ is saved in T-register, which is a temporary register.

(b) $r'(y_{bj}) \leftarrow D$ and $r''(y_{bj}) \leftarrow BR(<D>)$, where <D>, the contents of D-register, is used to index one of BR(k), $1 \leqq k \leqq p$.

(Remember that r may be regarded as consisting of r' and r".)

(c) $BR(<D>) \leftarrow BR(<D>) + (T + 1)$ and $D \leftarrow (D + 1)$ module p, and if D=0, D←p. The former statement denotes that the current module will be allocated following the logical location $BR(<D>) + T + 1$ if it is to be allocated again. We add one extra logical word for each allocation (to be described in next step.) The later one indicates that the next allocation will be assigned to the module next to the current one.

The above three statements allocate a block of $(T+1)$ logical words (each can hold a tuple) in the module specified by D-register (before updating) for storing R_A tuples having the join column value equal to y_{bj}. For the case in which $rA(y_{bj}) = 1$ and $rB(y_{bj}) = 1$, this means that block allcation for R_A tuples with $X_a = y_{bj}$ has been done.

(4) Output R_A tuples and store the relevant tuples of the join in the

allocated memory : Clear registers T and H and the flag F_Q of Q. Scan the relation A by RADM and output the sequence of R_A tuples and stored in Q. The tuples stored in Q are then fetched by CP. The join column values x_a's of each fetched tuple are extracted and used as indices to address the corresponding bits in rB. The contents of each addressed bit are tested for being set or not.

Cases : (i) if $rB(x_{ai}) = 0$, i.e., the R_A tuple being processed by CP is irrelevant to the join since the join column value x_{ai} of the tuple does not appear in the join column of R_B, then ignore the tuple.

(ii) if $rB(x_{ai}) = 1$, then

① if $rA(x_{ai}) = 1$, then $rA(x_{ai}) \leftarrow 0$ and $MB(<r(x_{ai})>) = MB(<r'(x_{ai})> \cdot <r''(x_{ai})>) \leftarrow 1$, where the part $<r'(x_{ai})>$ specifies a particular memory module and $<r''(x_{ai})>$ specifies a particular logical word in the module specified.

② $T \leftarrow r(x_{ai}) + MB(<r(x_{ai})>)$ and $MB(<T>) \leftarrow$ the tuple being held by CP, and $MB(<r(x_{ai})>) \leftarrow MB(<r(x_{ai})>) + 1$.

From (ii), we know that each logical word in MB pointed by the contents of the word $r(x_{ai})$ is a word containing a value indicating the number of R_A tuples with $X_a = x_{ai}$ that have been stored. At the end of this step, the word will contain a value one larger than the number of R_A tuples with $X_a = x_{ai}$. This information is important to each server S_i where new tuples are formed.

(5) Output R_B tuples, deposit the relevant R_B tuples of the join into the proper queues Q_i, and produce the concatenated tuples of the join : Clear T and H registers and the flag F_Q of Q, and T_i and H_i registers and the F_i flag for $1 \leq i \leq p$. Scan the R_B tuples by RADM and output them to Q. The tuples in Q are fetched and their join column values y_b's are extracted by CP. The extracted y_b's are used to test if the corresponding bits in rB are set or not.

Cases : (i) if $rB(y_{bi}) = 0$, i.e., the tuple being held is irrelevant to the join, ignore the tuple.

(ii) if $rB(y_{bj}) = 1$, then fetch the $r(y_{bj})$ consisting of two fields $r'(y_{bj})$ and $r''(y_{bj})$, and deposit the concatenated R_B tuple, consisting of $r''(y_{bj})$ and the R_B tuple being held, into the queue specified by the word $r'(y_{bj})$. The $r''(y_{bj})$ holds an address pointing to the starting address of a block of R_A tuples to which the R_B tuple will be concatenate

Each server S_i will start its join of tuples from the Q_i and the corresponding module M(i) once Q_i is not empty (i.e., the contents of registers H_i and T_i in Q_i are not equal). After the join of two buckets is completed, the join of next bucket-pair follows and so forth, until all the bucket-pairs have been processed. Logically, we can say that each S_i produces the concatenated tuples of the join from two "sorted" such-buckets. The concatenated tuples of the join in each buffer will be either output to the host or stored back to the RADM for further processing.

So far, only a single join column is involved in the join operation. For joins of relations on multiple columns, the addressing bits or words of RAM using single encoded values has to be modified. One way is to associate the multiple join column values with a pre-compiled index, as suggested by E. Babb [1], whenever such a join is concerned. Another way is to dynamically encode the multiple join column values with a unique value.

4.3 An Illustration Example

This section shows how the above algorithm works by means of an example. Consider the same database as given in Figure2. As an example, we consider the equi-join of table SALES on column ITEM with table TYPE on column ITEM, although this join can be computed without creating a derived relation. We will follow the steps of the above algorithm to illustrate how this join is computed.

Step 1. Clear RAM, i.e., rA, rB, and r.

Step 2. Clear registers T and H and the flag F_Q of Q. Assume that the

sequence of ITEM-components of table SALES that are input to Q are <CAM>,<GEAR>, <CAM>,<NUT>,<CAM>, and <NUT>, as they appear in the SALES table of Figure 2. These components will be used as indexes to set the rA and update the corresponding counter words of r. At the end of this step, the bit pattern of rA will be (0,1,0,1,1,0...,0) and their corresponding values of the array of words r will be (0,3,0,1,2,0,...,0)(Figure 3(a)) — i.e., there are three SALES tuples with ITEM-component = <CAM>, one SALES tuple with ITEM-component = <GEAR>, and two SALES tuples with ITEM-component = <NUT>.

Step 3. Clear T, H, and F_Q in Q, and BR(i), $1 \leq i \leq p$, and set D to 1. Assume that the sequence of ITEM-components of table TYPE input to Q is <BOLT>, <CAM>,<COG>,<GEAR>,<NUT>, and <SCREW>. These values are used as indexes to test the rA bits for being 1 or 0. Since rA(<BOLT>) = rA(0) = 0, ignore the <BOLT>. Since rA(<CAM>) = rA(1) = 1 and rB(<CAM>) = rB(1) = 0 (initially, rB is cleared), set rB(1) = 1 and allocate memory space in MB for those SALES tuples with ITEM = <CAM>. Since r(<CAM>) = r(1) = 3, thus, 4 logical words must be allocated. The allocation will do the following :

(a) Save the contents of word r(<CAM>), now being 3, in T.

(b) Store the contents of D-register, now being 1, in r'(<CAM>) and the contents of BR(D) = BR(1), now being 0, in r"(<CAM>). The word r(<CAM>) = r(1) now is a pointer word pointing to the starting address of the first module. (In fact, the setting of rB bits can be used to distinguish pointer words from counter words.)

(c) (i) Increment BR(D) = BR(1) by 4 (= T + 1) so that if there is any memory allocation assigned to the first module, it will be allocated starting from the fifth logical word (i.e. logical address 4).

(ii) Increment the D-register by 1, indicating that next allocation, if any, will be assigned to the module next to the current one.

The third incoming value is <COG>. Since rA(<COG>) = rA(2) = 0, ignore the value. The same procedure is repeatedly applied to other values. At the end

of this step, the bit array store rB and r will be (0,1,0,1,1,0,...,0) and (0, 1·0,0,2·0,3·0,0,...,0)(Figure 3(b)), where r(1) = 1:0 = r'(1) · r"(1)(i.e., concatenation) and the contents of all registers in CP are shown in Figure 3(b).

Step 4. Clear rA and registers T and H and the flag F_Q of Q. Assume that the sequence of SALES tuples input to Q is the same as that of SALES tuples appearing in Figure 2. Any tuples with rB(x) = 1 (x being the ITEM-component of SALES) will be stored in the logical location in MB pointed by r(x). The first incoming tuple with rB(<CAM>) = 1 is stored in the logical location 1 of the first module M(1). (After this, the logical location 0 of M(1) has the value 2, which is initially set to 1 and incremented by 1 when a tuple is stored.). The second incoming tuple with rB(<GEAR>) = 1 is stored in the logical location 1 of M(2); the third incoming tuple with rB(<CAM>) = 1 is stored in the logical location 2 of M(1); and so forth, until the sixth incoming tuple which is stored in the logical location 2 of M(3). At the end of this step, the logical location 0 of M(1), M(2), and M(3) have the values 4,2, and 3, respectively. Figure 3(c) shows the contents of RAM and the first three modules M(1), M(2), M(3).

Step 5. Assume that the sequence of TYPE tuples input to Q is the same as that of TYPE tuples appearing in Figure 2. Any tuples with rB(y) = 0 (y being the ITEM-component of TYPE) are ignorant. Those tuples with rB(y) = 1 will be dispatched into the queues Q_i ($1 \leq i \leq p$) determined by r'(y). They are concatenated to the contents of r"(y) before dispatching into the proper queues. The first incoming tuple is ignored since rB(<BOLT>) = rB(0) = 0; the second one concatenated to the contents of r"(CAM) = r"(1) = 0 is dispatched into the first queue Q_1 since r'(<CAM>) = r'(1) = 1; the third tuple is ignored; the fourth one concatenated to the contents of r"(<GEAR>) = r"(3) = 0 is dispatched into Q_2 since r'(<GEAR>) = r'(3) = 2; the fifth one concatenated to the contents of r"(<NUT>) = r"(4) = 0 is dispatched into Q_3 since r'(<NUT>) = r(4) = 3; the sixth tuple will be ignored. Since each tuple dispatched is associated with a

pointer pointing to the beginning of a block of tuples to which the dispatched tuple is concatenated, each server S_i thus can produce the concatenated tuples of the join from each TYPE tuple in Q_i and the block of SALES tuples in M(i), without memory addressing contention problem.

5. Summary and Status of the Project

We have shown how relational join operations could be performed by a specialized hardware architecture. The architecture has an RAM store for intermediate results, which can be addressed either by encoded join column values or by precompiled (or -assigned) indexes. The algorithms of using the RAM to perform implicit join operations, as well as those explicit joins in which the join column is a primary key or a candidate key in one relation (i.e., a foreign key in another), are considered. The algorithm of using the RAM, plus a memory bank consisting of modules and an array of queue servers, to perform the other explicit joins is then detailed. It is designed to allow each queue server to operate on one and only one memory module, without memory conflicts. The architecture extends functional capabilities of existing RADMs and, thereby, gives a significant performance improvement over existing join algorithms used in RADMs, especially when a join-dominating database application is concerned. We believe this extention is adapted to current VLSI technology and has the important characteristics of being applicable with little or without modification to currently proposed RADMs' hardware.

We have developed a hardware simulator which is now being used to verify the accuracy of algorithms, study the design parameters, such as the length of Q, the number of queues Q_i, etc., as the functions of contents, features or characteristics, and the size of relations, and explore the architecture perofrmance and its bottleneck. We have also designed the projection algorithms using the same architecture with a little bit modification, which are not reported because of the limited space. A scheme optimizing query expressions executing on the resulting RADM is being investigated and will be reported in the future.

6. References

[1] Bobb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol.4, 1, March 1979, pp.1-29.

[2] Banerjee, J., and Hsiao, D. K., "DBC — A Database Computer for Very Large Databases," IEEE Trans. on Computers, Vol.C-28, 3, 1979.

[3] Chang, H., "On Bubble Memories and Relational Data Base," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.207-229.

[4] Chen, T. C., Lum, V. W., and Tung, C., "The Rebound Sorter : An Efficient Sort Engine for Large Files," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.312-315.

[5] Edelberg, M., and Schissler, L. R., "Intelligent Memory," Proc. 1976 NCC, Vol.45, AFIPS Press, Montuale, N. J., pp.691-701.

[6] Hong, Y. C., and Su, S. Y. W., "Associative Hardware and Software Techniques for Integrity Control," ACM TODS, Vol.6, 3, Sept. 1981, pp.416-440.

[7] Hong, Y. C., and Su, S. Y. W., "A Mechanism for Database Protection in Cellular-Logic Devices," Paper under review for the IEEE Trans. on Software Engineering.

[8] McGregor, D. R., Thomson, R. G., and Dawson, W. N., "High Performance for Database Systems," Systems for Large Databases, North-Holland Publishing Co., 1976, pp.103-116.

[9] Lin, C. S., Smith, D. C. P., and Smith, J. M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol.1, 1, March 1976, pp.53-65.

[10] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP — an Associative Processor for Database Management," Proc. 1975 NCC, Vol.44, AFIPS Press, Montvale, N. J., pp.379-387.

[11] Smith, D. C. P., and Smith, J. M., "Relational Database Machines," IEEE Computers, Vol.12, 3, March 1979, pp.28-37.

[12] Su, S. Y. W., "On Logic-Per-Track Devices : Concepts and Applications," IEEE Computers, Vol.12, 3, March 1979, pp.11-25.

[13] Su, S. Y. W., and Lipovski, G. J., "CASSM: A Cellular System for Very Large Databases," Proc. Int'l Conf. on VLDB, Sept. 1975, pp.456-472.

[14] Su, S. Y. W., Nguyen, L. H., Eman, A., and Lipovski, G. J., "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. on Computers, Vol. C-26, 6, June 1979, pp.430-445.

[15] Tanaka, Y., Nozaka, Y., and Masuyama, A., "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer," Proceedings of IFIP Congress 80, pp.427-432.

[16] Todd, Stephen, "Hardware Design for High Level Databases," IBM United Kingdom Scientific Center, Peterlee, TN 49.

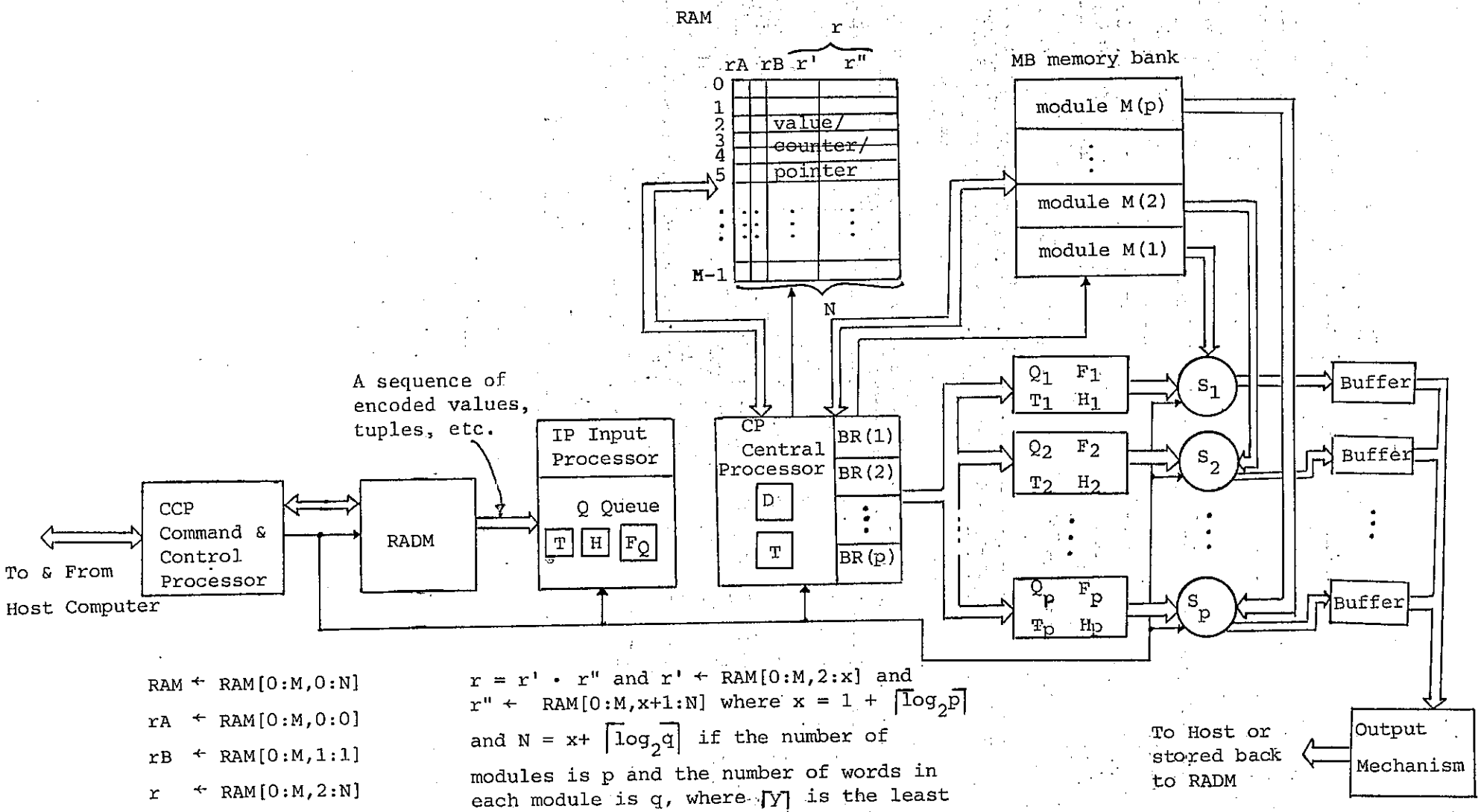

Figure 1. Hardware Architecture

SALES

DEPT	ITEM
<D1>	<CAM>
<D1>	<GEAR>
<D5>	<CAM>
<D5>	<NUT>
<D8>	<CAM>
<D10>	<NUT>

TYPE

ITEM	COLOR	PRICE
<BOLT>	<GREEN>	<5p>
<CAM>	<RED>	<2p>
<COG>	<RED>	<4p>
<GEAR>	<GREEN>	<4p>
<NUT>	<BLACK>	<8p>
<SCREW>	<YELLOW>	<7p>

Figure 2. A simplified database with two tables SALES and TYPE linked by ITEM.

RAM

	rA	rB	r
0	0		0
1	1		3
2	0		0
3	1		1
4	1		2
5	0		0
⋮	⋮	⋮	⋮
M-1	0		0

Figure 3(a)

RAM

	rA	rB	r'	r"
0	0	0	0	0
1	1	1	1	0
2	0	0	0	0
3	1	1	2	0
4	1	1	3	0
5	0	0	0	0
⋮	⋮	⋮	⋮	⋮
M-1	0	0	0	0

CP

D: 4

T: 2

BR(1)	4
BR(2)	2
BR(3)	3
BR(4)	0
⋮	⋮
BR(p)	0

Figure 3(b)

RAM

	rA	rB	r'	r"
0	0	0	0	0
1	0	1	1	0
2	0	0	0	0
3	0	1	2	0
4	0	1	3	0
5	0	0	0	0
⋮	⋮	⋮	⋮	⋮
M-1	0	0	0	0

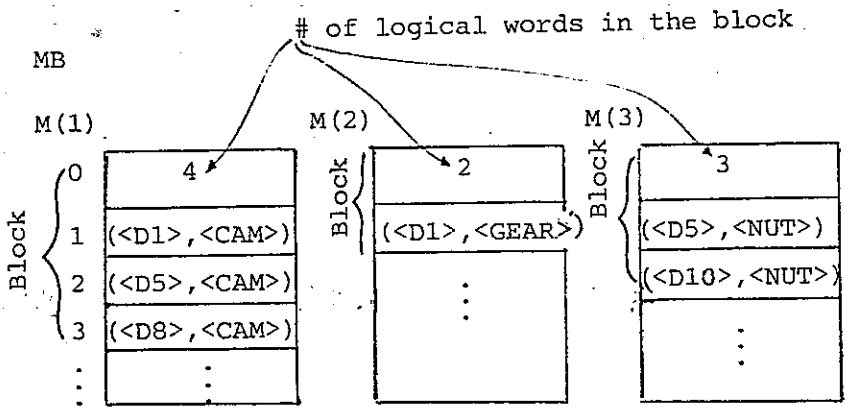

Figure 3(c)