TR-81-011

Relational Database Machines

By

Yang-Chang Hong

Institute of Information Science

Academia Sinica, Taipei, R.O.C.

This work is supported by National Science Council Grant

NSC70-0404-E001-02

December 1981

ABSTRACT

The limitations of conventional systems in supporting database functions have prompted the design and development of specialized computer hardware directly providing database applications. The functions of data searching, sorting, updating, security and integrity can, thus, be moved from the conventional processor to secondary storage, thereby reducing the data-transfer costs. The processing time is decreased by the content and/or context searches and the increased parallelism of processing. This paper surveys relational database machines, describes the general architecture and characteristics of database machines using cellular-logic concept, and addresses associative hardware techniques for implementing select and join operations. The limitations, issues and problems related to this type of machine are finally presented.

Content

Problems with relational DBMS on conventional systems

The relational model [7] has, more than any other data models during the past decade, attracted and held great interest of the database researchers and database management community. Its tabular representation of data is very suitable to ordinary users and it provides a powerful high-level nonprocedural data language for users to interact with a database and satisfies the requirements of data independence. The model, thus, is structurally and behaviorally far removed from the storage organization and primitive operators of existing computer hardware. This means that conventional computer hardware which is originally designed for supporting scientific applications is very difficult to efficiently support relational database applications. This is because multiple levels of sophisticated software are required to translate the high-level model and the language down to the low-level machine structures and codes. This can be seen from the two development projects [1,26].

On the other hand, in conventional systems data is stored in slow, large-capacity secondary storage devices. Rather limited processing capabilities are incorporated into these devices. Therefore, data stored in these devices must be staged in the main memory for processing. The data altered in the main memory must then be moved back to replace the original data in the secondary storage. This movement of data back and forth is not only very time consuming, but it often ties up the important resources of a computing system, such as communication lines, channels, and data buses. Software techniques (access methods, indexes, hash tables, etc.) can reduce

the amount of data to be staged, but they also introduce overhead and difficulty in the creation and maintenance of the indexes and tables. There is also a problem in keeping the contents of the indexes and tables consistent with the original files after data update, insertion, and deletion.

Hardware support for relational DBMS applications

The limitations of conventional systems prompted the design and development of specialized hardware directly implementing the relational algebra operators [8] (since they are received the most attention as a condidate for hardware support.) This specialized hardware approach mainly takes advantage of recent developments in hardware and memory technology. The approach alleviates the problem of moving data back and forth by distributing processsing logic over data. The data can then be searched and processed at the place where they are stored, and those data which are irrelevant to the search command can be filtered out by the processing logic associated with data. Processing efficiency is gained by content and/or context addressing of the data, and by parallel processing of the data in secondary storage devices, such as disks, CCD's, or magnetic-bubble memories. Furthermore, through the use of associative techniques (content and context searches, tagging and marking data, etc.) the data can be physically stored in a very "simple" structure —— i.e., no index or table/directory is needed, thereby decreasing the complexity of modifying the data.

Approaches of distributing processing logic over data

The most effective way of distribution is to associative logic with small amount of data. The maximizes the amount of parallel processing, thereby minimizing total execution time. In general, logic can be distributed over data in any of three ways : (1) by integrating both the logic and data on a single VLSI chip, (2) by associating the logic with the read/write mechanism of a disk track or a loop of electronic rotating store, such as CCD, MBM (magnetic-bubble memory), etc., and (3) by configurating distributed microprocessor-based architectures.

The first approach holds little promise for relational database applications, since MOS storage is too expensive for a large database typically involved. The second and third approaches, however, have been used to design hardware for select, join, and project for database applications. In the following we will survey relational database machines based on these approaches and then consider how each of these machine implements the select, project, and join operations.

The concept of associating logic with the read/write heads of rotating storage devices was first proposed by Slotnick [24] in the 1970's. Parker [22], Parhami [21], and Healy [11] refined this idea further. The same idea was first applied in database management applications by Copeland et al. [9], Lipovski [16], and Su and Lipovski [30]. Ozkarahan et al. [20], Lin et al. [17], McGregor et al. [19], and Shuster et al. [23] have also proposed ways to handle relational databases, while Edelberg [10], Chang [5], Chen [6], and Todd [32] have developed designs to exploit the new CCD and bubble

technologies. Su [27] has defined the class of cellular-logic (CL) devices which is more general than the "logic-per'track" devices originally conceived by Slotnick [24], in which logic is associated with each track of a fixed-head rotating device having no intercell communication facility. The general architecture and characteristics of CL devices are briefly reviewed in the following. (see [9,16,17,20,23,30] for a detailed description of several CL devices specifically designed for database applications.)

A CL device is controlled by a main frame computer (MFC) (see Figure 1), which translates the high-level queries or data manipulation statements into machine codes, transfers these codes to the CL device for execution, receives data transferred out of the device, and processes (or formats) and outputs the data to the user. A CL device consists of an array of cellular associative processors (or cells) which is dreiven in parallel by a controller. Each cellular associative processor is composed of a processing element P_i, and a circular (rotating) memory element M_i. Each P_i is a special purpose microprocessor (or simple logic) designed to perform direct operations on its associated M_i. Each M_i can be defined by a triplet (TG, DT, SR); TG is the storage space for tagging data items so that different data types such as character string, numberic value, delimiter, instruction, etc., can be distinguished; or different data elements, such as records, files, etc., can be delimited. DT is the storage space for the actual data as well as the program instructions. Data and instructions can be recorded in a fixed-word format or a free format with delimiters and tags to separate them. SR is the storage space for marking data items as a result of data searches. A status field is generally associated with each data item. This field con-

tains status bits which are used for marking the data item for output, indicating the item that satisfies a matching condition, marking the instruction for execution and the location for insertion, etc.

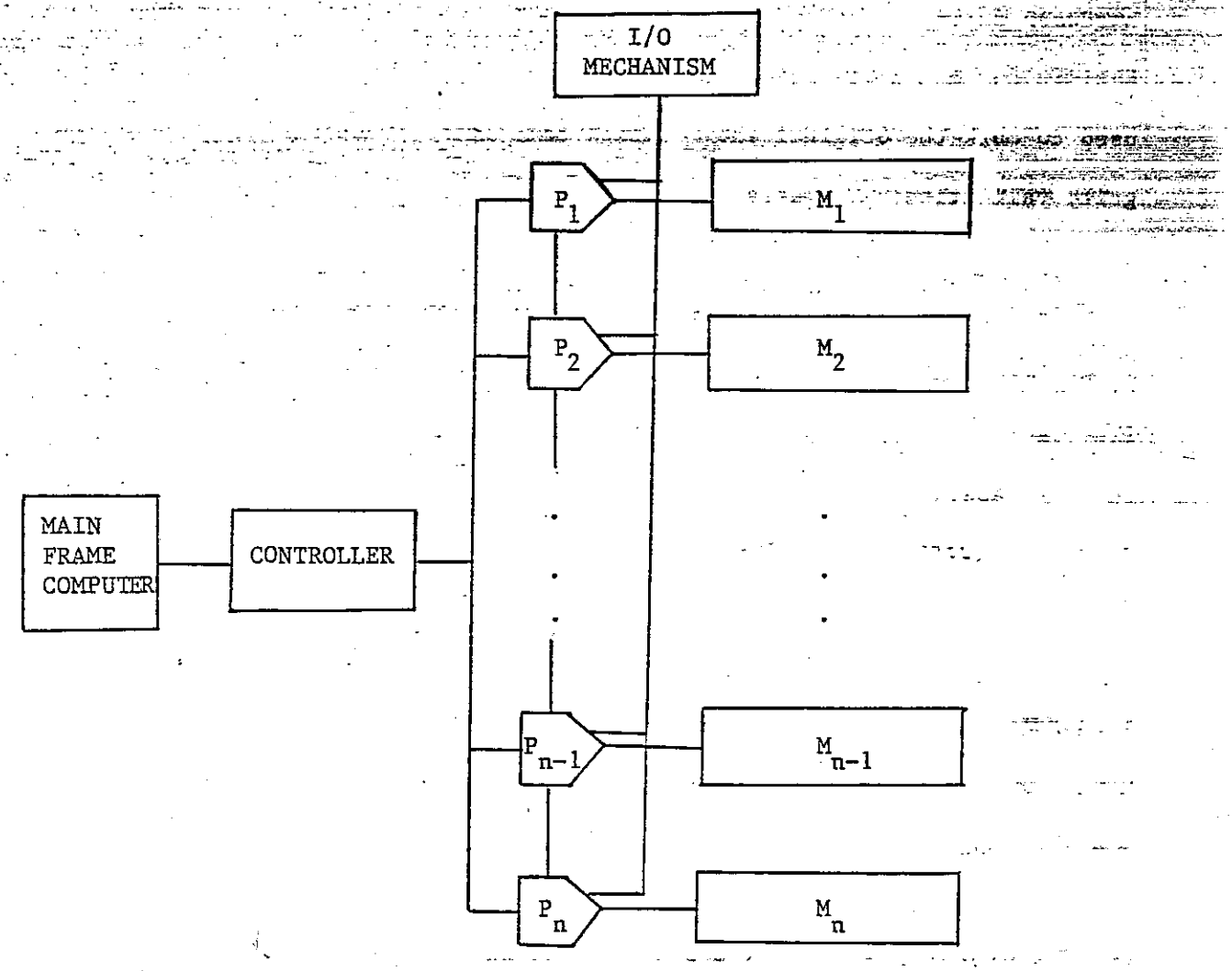


Figure 1. The architecture of cellular-logic devices

In Hong and Su [13,14], the status field is extended to include three additional status bits for database security and integrity purposes. one status bit, called the "no-access" bit, is used to mark the data item which is unauthorized for access. The data items with set no-access bits will be recognized by the processing element and will be bypassed by the hardware during search operations. Since the bypassed data items can never satisfy any search commands, they cannot be accessed. Two ssatus bits are used to mark the original and modified data items, respectively, for integrity validation. The detailed description of the use of these three bits is given in the papers mentioned above.

The tracks, or groups of tracks, of disks or drums have been used as M_i's. CCD shift registers, magnetic bublle memories, or any other delay line technologies are potential candidates for the circular memory. The contents of TG, DT, and SR in each M_i can be scanned repeatedly by P_i, since memory is circular.

There are two basic types of searches contained in the search mechanism. In the content search, the data value(s) to be searched is first loaded into the comparand registes of all P_i's. The data words stored in M_i's are fed, one at a time, into their associated buffer registers of P_i's. The mask control mechanisms of P_i's are set to select the subfield of the fed data words for matching against the comparands. All P_i's carry out the matching operation simultaneously. At the end of a complete scan of memory elements, the SR storage corresponding to data elements —— such as data items, records, or files —— that satisfy the matched conditions are marked. The marked data elements can be transferred to an output device,

or they can be used as context for the next content search. Context search allows data to be accessed by the context in which the data occur. It is accomplished by utilizing the search results, recorded in the SR of M_i, as the whole or part of the next search condition. Thus, only the data related to (or in the context of) the results of a previous search will be found. It is these capabilities of parallel searching, as well as accumulating search results for use in subsequent searches, that gives CL devices the power to handle complex Boolean search queries efficiently.

It is the parallel processing and the content and context search capabilities of these processing elements that distinguish the CL devices from the conventional rotating devices. Only the data which satisfy a search criterion are moved into the main memory for further processing. Thus, data are processed at the place where they are stored, and large quantities of data can be prefiltered out by the processing element. Furthermore, the content and context search capabilities provide more flexibility in data search. The process of calculation and maintaining data addresses before the data can be located is eliminated.

One feature that is uniquely possessed by CASSM is its programmability [28]. It allows the content and context addressing capabilities to be used to locate and activate programs stored in CASSM. In this system, programs can be activated or triggered when the data meet some high-level data conditions. This type of programming is called associative programming. It was used in Hong and Su [13,14] to enforce database integrity and security in CL devices.

The idea of basing the design of database hardware on a network of microprocessors dedicated to specific database functions was first developed

by Madnick in a system called INFOPLEX [18]. It organizes a memory and microprocessor hierarchy to exploit the parallelism inherent in concurrent accesses to a database. The Database Computer[3,4,15] is a more extensive design that first incorporates access control. The DBC has dedicated processors for access control, query and update interpretation, directory processing, etc. Similar to both INFOPLEX and DBC is the design for a distributed database machine proposed by Stonebraker [25]. It used conventional micro/mini computers to off-load database functions from a central host computer. Both the DBC and INFOPLEX support conventional access methods such as directories as well as associative logic-per-track processors.

We will devote most of our attention in this paper to the relational database machines using cellular-logic concept. However, it is important to note that the expolitation of these class of machines is not the only route possible in developing relational hardware. New approaches, such as LEECH [19] and CAFS [2], using a central processor architecture design can provide a better base for building relational systems.

In the following descriptions, we assume that the readers are familiar with relational data models and relational select, project, and join operations.

Relational data representation

As pointed above, one advantage of using associative techniques for data search is that it can simplify the storage representation of the data in secondary storage. The designers of the database machines have proposed

more simplified storage representations which allow the users to view the stored data in tabular (or close to tabular) form, but they differ greatly in organization. The storage representation determines the complexity of the processing logic, the number of revolutions (i.e., memory scans) needed to complete the query evaluation, the cost in space of storing a relation, the amount of contention for outputting qualified tuples. The discussions which follow are restricted to the storage representations of CASSM, RAP, and RARES. Representations of electronic rotating devices will not be given here. Interested readers can refer to the literature [5,6,10.32].

CASSM. In CASSM, data is laid out along the track or loop of the rotating storage device. Each track (or memory segment) contain a sequence of 40-bit word. In a 40-bit word, a 32-bit field is used for data or instruction, a 3-bit field for status, a 3-bit field for tags, a 1-bit field for parity, and a 1-bit field for internal use. Figure 2 presents a schematic description of different types of words and their formats. The tags are used to distinguish different types of words such as delimiter word (D), name-value word (N), instruction word (I), operand word (O), garbage word (GB), and end-of-file word (EOF). Status bits are used to mark the word for output (C-bit), to indicate the word that satisfies a match (M-bit), to mark the point for insertion (X-bit), to mark the instruction for execution (A-bit), and to mark the end of the operand list (F-bit). The formats of a 32-bit field for various types of words are different.

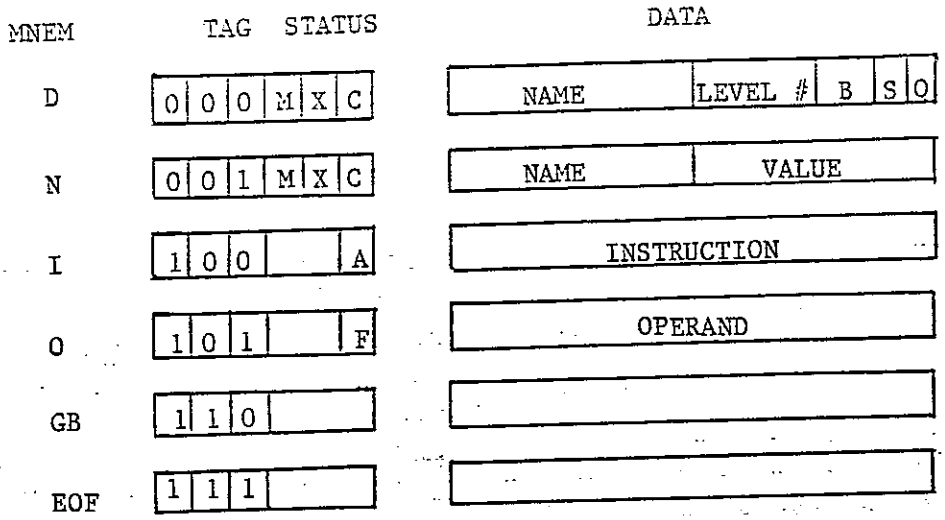


Figure 2. Word types and formats.

In CASSM, a relation is treated as a two-level tree. A tree is physically stored in a top-down and left-right order. Each node of a tree or subtree is delimited by a delimiter word which contains a level number, a coded value for a name, a 6-bit stack B, a qualification bit Q, and a specification bit S. The name is used for identifying the tree or subtree. It is usually the relation table or subtable name in an unnormalized relation, or the label assigned to a program segment. The bit stack is used to accumulate temporary processing results. The S- and Q-bits, repectively, mark the node as the place to store the search result and as a place in which the search should be conducted. In general, the Q-bits associated with a set of nodes are initially set to provide the context for a search. The search result (a logical "1", if a node satisfies the search condition) is

pushed onto the bit stacks of those nodes with S-bits that are set. This result can also be ANDed or ORed with the top bit of the stacks. (Other stack operations, such as complementing the top bit, pushing a bit, popping a bit, and exchanging the top two bits, are also available.) In evaluating a complex Boolean expression on a set of nodes (trees or subtrees), an atomic condition in the expression is first evaluated to set the top bit of the bit stacks of those nodes which have satisfied the condition. The second atomic condition is then evaluated to set another bit pattern which may be logically ANDed or ORed with the bit pattern set by the first atomic condition. It may also be pushed onto the corresponding bit stacks, depending on the logical operator used in the expression. The other atomic conditions can be evaluated in a similar way. The node satisfying a complex Boolean expression will finally have a logical "1" on the top of its bit stack. This generally means that the data in the node has satisfied a certain search condition and should be output to the user.

RAP. Like CASSM, RAP lays its data along the tracks of the rotaing storage device. Each tuple of a relation is stored in a fixed-length block. This length can vary from relation to relation, but within a relation all tuples must use the same amount of storage. Only one tuple of relation can be stored on a given track. Within a track tuples are stored one after the other (one per block), and the end of each block is marked by a delimiter. RAP's track format is illustrated in Figure 3.

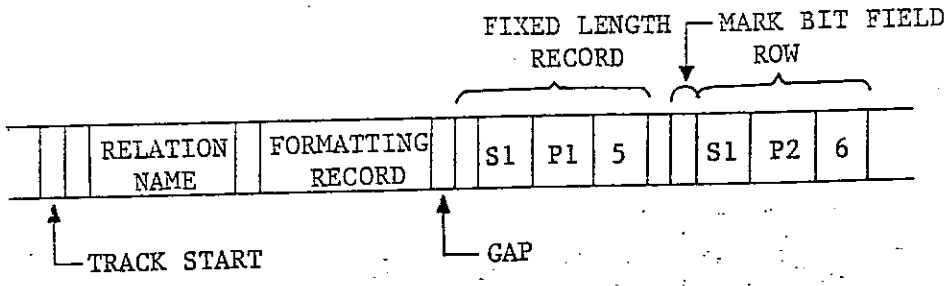


Figure 3. RAP data organization

The beginning of a track is indicated by a marker which is detected electronically. This marker also implies the physical end of a track. The first two blocks of a track contain the name of the relation stored on the track and the column names of the relation, respectively, and act as "header track." Each succeeding block contains the concatenated column values in a tuple. The order of column names determine the order of the values in each tuple. These concatenated values are preceded by a string of mark bits. All names (relation and columns) and values on the track are encoded as 32, 16, or 8-bit strings, each preceded by a 2-bit code indicating its length. There is an upper bound on the length of a RAP relation tuple which is determined by the length of the cell buffer (1024 bits). If a relation has too many tuples to be stored on one track, then several cell tracks are used. The hardware requires the relation and column names to be repeated once on each track of a relation.

A fixed length gap is required between every two blocks. The lengths of these gaps are proportional to the amount and speed of logic required between block operations.

RARES. Not like CASSM and RAP, RARES lays out relation tuples across tracks (i.e., along the radius of a disk) in byte-parallel fashion : the first byte of a column value is placed on a track; the second byte of the value is placed in the same position on the adjacent track, and so on. A byte-parallel organization means the processing of each relation tuple has to be completed with the amount of 8-bit shifting time. This amount of time is permissible for the logic available to process a tuple laid out along a radius, given the rotation time of the disk. It seems impossible for the current available logic to complete the process of a tuple in one-bit shifting time. This is why the decision to use a byte-parallel rather than a bit-parallel organization is made. Each set of tracks used to store a relation in this fashion is called a band. The number of tracks in the band can vary; the size of the band is determined by the length of a tuple. For relations with long length of tuples, more than one radius can be used to store a tuple. RARES's data organization is illustrated in Figure 4. This type of data organization is called an orthogonal layout. The decision to use an orthogonal layout rather than that used by CASSM or RAP is that orthogonal layout allows fewer tuples can come into contention for output. However, contention is still possible, so, like CASSM and RAP, RARES also needs an output arbiter.

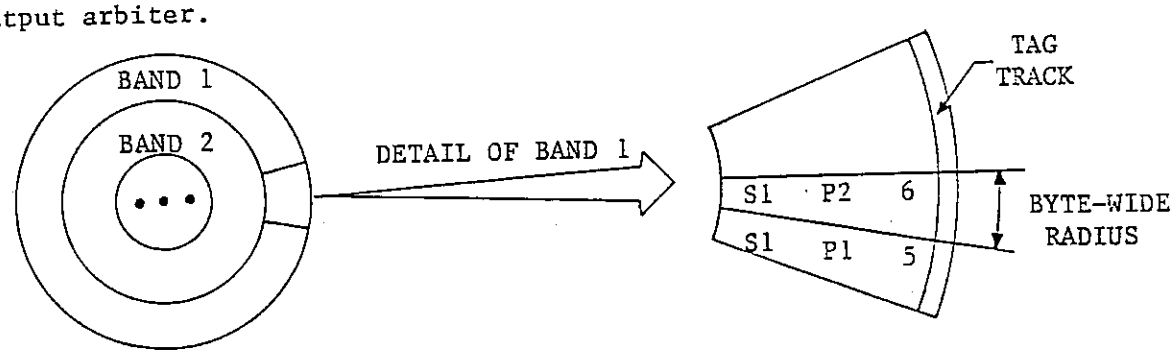


Figure 4. RARES data organization

Data searches

This section describes the data search performed by CASSM, RAP, and RARES.

CASSM. CASSM needs search logic for each track and uses a RAM for marking selected tuples. The search operation is best illustrated by means of a specific query. Figure 5 shows a relation SUPPLY with attributes S#, P#, and Q, storing in memory segment M_i. In Figure 5, only the delimiter words (D) and name-value pairs (N) are shown. The "D SUPPLY 0" is used as a delimiter word for the relation SUPPLY and " D SUPPLY 1" is used as a delimiter word for tuples, where 0 or 1 is a level number. File XY is the name of another file. (We will ignore the name and value decoding steps.) A query to select from the relation SUPPLY all supplier numbers with P# = P1 and Q > 5 would be translated in CASSM assembly program as follows :

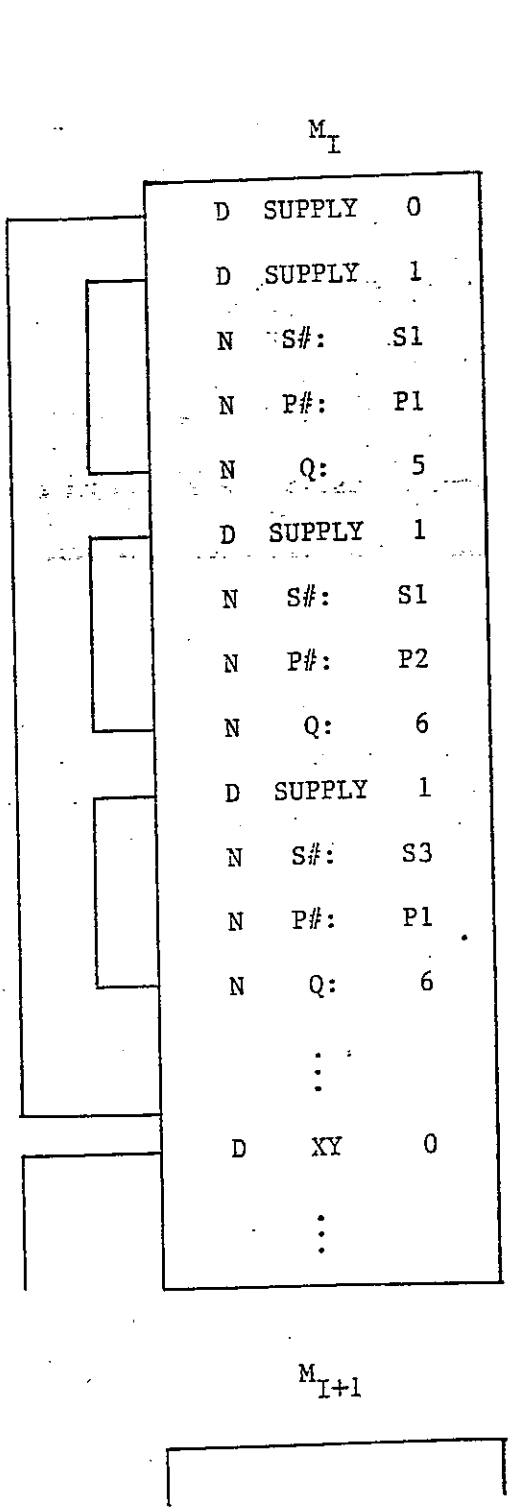
CASSM Program

```
T1:     DMK 1, SUPPLY
T2:     QSR =, N; PUB: IM(P#:P1)
T3:     QSR >, N; ANB; IM(Q:5)
T4:     FNSB   N; S#; ORS C
```

The program starts with a DMK (delimiter mark) instruction to set the S- and Q-bits of all delimiters that have a level equal to 1, and a name SUPPLY. The S- and Q-bits of all delimiter words "D SUPPLY 1" in Figure 5 are set as shown under T1. This instruction marks all tuples labelled 1,

SUPPLY of the SUPPLY relation as the location where search results are to be accumulated, and the next search is to be conducted. The T2 instruction QSR searches for P# = P1, specified by IM ——i.e., immediate operand clause —— in tuples whose Q-bits are set. The results of the search under these tuples are pushed (PUB subinstruction) on their associated bit stacks (B). The bit stack of each tuple (whose S-bit is set) is the place where the search result is stored. Accordingly, after the execution of this instruction only those tuples whose associated data have satisfied the search condition will have a logical "1" on the top of their associated bit stacks, as shown under T2 of Figure 5. The T3 instruction searches for Q > 5; the results of the search are ANDed (ANB subinstruction) with the top of their bit stacks (B) (as shown under T3). So far, only those tuples with asociated data that have satisfied the search condition will have a logical "1" on the top of their associated bit stacks (B), as shown under T3. Thus, the Boolean expression is processed and the search results are accumulated in the bit stacks.

The T4 instruction FNSB searches for name-value words in a tuple whose delimiter contains a logical "1" on its bit stack (B). If such words are found, the ORS C, the second part of T4, will logically OR a logical "1" into its status to set the collection bit (as shown under T4 of Figure 5). The collection bit C, when set, tells the hardeare that the word is to be output. After the execution of T4, the S# = S3 is marked for output. The marked data items will then wait for output by output mechanism.



	T1	T2	T3	T4
	SQBC	SQBC	SQBC	SQBC
D SUPPLY 0	0000	0000	0000	0000
D SUPPLY 1	1100	1110	1100	1100
N S#: S1	0	0	0	0
N P#: P1	0	0	0	0
N Q: 5	0	0	0	0
D SUPPLY 1	1100	1100	1100	1100
N S#: S1	0	0	0	0
N P#: P2	0	0	0	0
N Q: 6	0	0	0	0
D SUPPLY 1	1100	1110	1110	1110
N S#: S3	0	0	0	1
N P#: P1	0	0	0	0
N Q: 6	0	0	0	0
D XY 0	0000	0000	0000	0000

Figure 5. CASSM's data organization and an illustrated search

In CASSM, all marks to the fields of delimiter words (i.e., the B-stack or the S and Q bits) are made using the RAM available to the logic associated with each read/write head pair. For each delimiter word in the database it is assumed that there is a bit reserved in the RAM. If a delimiter is to be marked, its RAM bit is marked. In the next revolution, when the delimiter word is reached, the mark is placed in it before it reaches the rest of the processing logic for consideration by the next CASSM instruction. The RAM is also the main mechanism used to support the implicit join (to be described later).

RAP. RAP also needs search logic for each track. The matching is accomplished by rotating each tuple as it routes into the buffer of the search logic designed to hold an entire tuple. Instead of evaluating one atomic condition per revolution like CASSM, it can evaluate up to k atomic condition in a revolution. Thus, k ocmparators must be provided. The comparators in each logic operate in parallel. The results of these k independent evaluations are then fed into hardeare Boolean operation logic to complete the evaluation. If a selected tuple needs further processing, it must be marked. The mark field consisting of mark bits preceding each tuple is provided for this purpose.

RARES. Instead of needing search logic for each track like CASSM and RAP, RARES only needs search logic for each band. But because bands may vary for different relations, special logic is required to reassign the search logic to a new set of adjacent tracks whenever a band's width is redefined. The special logic used by RARES's designers is called "barrel switch [17]". RARES perform a Boolean select by matching one atomic condition per revolu-

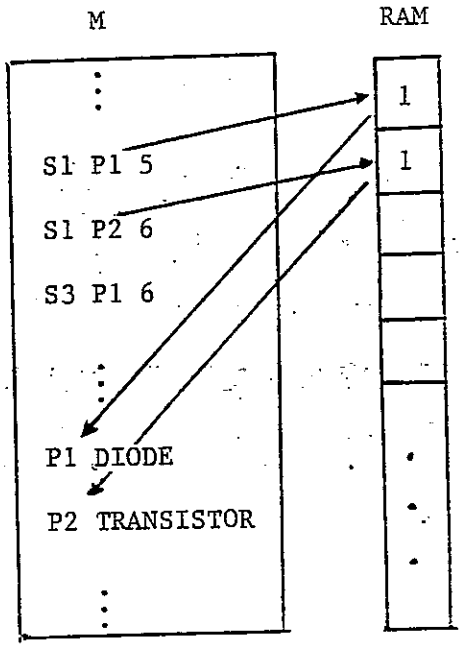
tion like CASSM. Within one revolution a radius of a tuple is selected for matching. If the match is successful but further matches are required on the tuple, a mark is placed in the response store associated with the search logic for the band. When previous matches have been performed on the tuple to which the redius belongs, the reponse store will contain a bit that indicates the accumulated result of these matches. The hardware combines the current result of this accumulated result. If the match is successful and complete, the search logic wait for a signal from the output arbiter. If the signal so idicates, the tuple is output to the buffer; otherwise it is marked for output on subsequent revolution.

Join algorithms

The design of associative hardware to implement the join operation has been to concentrate on a from called the "implicit join." This join does not require creating a new relation from the two original relations; instead the values of the columns being joined (called the join columns) in one relation are used to select tuples in the second relation that have those same values in their join clumns. Only RARES, LEECH [19], and CAFS [2] consider the problem of implementing the "explicit join" as contrasted to implicit join, i.e., it consists of taking the tuples in two relations and forming a new tuple in a third relation if their join columns have the same values. We shall describe algorithms used for implementing the implicit join in CASSM and RAP and then the explicit join in LEECH, CAFS, and RARES in the remainder of the section.

In CASSM, the implicit join of two relations is accomplished by transfroming the encoded values of the join columns from the selcted tuples in one relation to mark proper bits in the RAM [31]. Next the tuples of the second relation are matched on their join columns againinst the values in the RAM and marked if the match succeeds. This is illustrated by a specific query.

Consider that two relations SUPPLY with attributes S#, P#, Q and PART with P# and PD (part description) are logically stored in a single memory segment M one tuple after another (see Figure 6). A query is to list the part numbers (P#) and their descriptions (PD) that are supplied by S# = 'S1'. In first logical revolution (a logical revolution may take several revolution in CASSM), SUPPLY relation tuples are examined. For each tuple (S#, P#, Q) in SUPPLY, if S# = 'S1' then set RAM(P#) = 1. If we assume P1 is coded as 1 and P2 is coded as 2, then after this logical revolution, RAM(1) and RAM(2) are set to 1. This step is called forward pointer transfer. Next logical revolution (also called backward pointer transfer) is to examine the tuples of PART relation against the values in RAM. For each tuple (P#, PD) in PART, if RAM(P#) = 1 then output P# and PD. In this query, tuples (P1, DIODE) and (P2, TRANSISTOR) are satisfied and will wait their turn for output. Thus the number of revolutions required is fixed, independent of the number of tuples selected from the first relation. Similar idea is also used by CAFS to implement join [2].



. Forward Pointer Transfer

For each tuple (S#, P#, Q) in relation SUPPLY, if S# = 'S1' then set RAM (P#) = 1.

. Backward pointer transfer

For each tuple (P#, PD) in relation PART, if RAM (P#) = 1 then output P# and PD.

Figure 6. Implicit 'JOIN' in CASSM

RAP performs implicit join by taking the join column values of one relation and uses them as a disjunctive select condition for the second relation. Since it has k comarators per search logic, k column values from the first relation can be matched against the tuples of the second relation in each revolution. Thus, the number of revolutions required is $\lceil \frac{r}{k} \rceil$, where r is the number of tuples selected from the first relation and $\lceil x \rceil$ is the least integer greater than x. Obviously this number of revolutions required is dependent on r. No algorithms are provided by RARES for computing implicit join. The explicit join algorithms for LEECH, LAFS, and RARES are discussed as follows.

In LEECH machine, the tuples of each relation being joined are first scanned to produce a bit map of the values of their join columns. These bit maps are combined to produce a single filter for selecting tuples needed for the join. The relation being joined are then fed to the filter and the selected buples are processed by the front-end processor to form the concatenated tuples of the join. This algorithms uses the bit map to filter out the tuples irrelevant to the operation. The join thus is actually performed by conventional algorithms. This type of algorithms is most effective if the selected tuples to be joined are few.

CAFS scans the tuples in one relation to produce a filter in its bit-addressable memory (1-bit wide). The bit is addressed and then set by the encoded value of the join column of each tuple relevant to the join. (The possible values of the join columns are encoded by associating each with a unique address in the bit memory [2].) The relevant tuples are sent to the front-end processor to be held for computing the join. The second relation is then fed to the filter. The relevant tuples (i.e., those whose join

column values address set bits) are sent to the processor to compute the join. The only difference of this algorithm from that of the LEECH machine is the design of the filter. The join is still performed by conventional algorithms and will not be effective if the number of selected tuples being joined is large.

RARES provides a hardware-support algorithm for sorting the tuples of each relation being joined into buckets on their respective join columns. However, the tuples within each bucket are sorted in main memory by conventional algorithms, and the concatenation of tuples from the two sorted relations is accomplished by the general-purpose host. Obviously the explicit join in RARES is mainly performed by conventional algorithms.

Projection algorithms

The design of existing associative hardware fails to provide facilities for implementing projection, especially for multiple-column projection operations.(A projection operation consists of removing some of the columns of a relation and then removing any repetition.)

Summary and discussion

Relational database machines have been surveyed and the characteristics and operations of cellular-logic database machines have been described. There are limitations related to cellular-logic machines. In the case of handling very large database, say $10^{10} \sim 10^{15}$, the cost of building such type of

machine for holding and processing the data is prohibitively high. Their most likely use will be as staging devices. The data will reside in conventional mass memory and only needed portions will be staged into the database machine.

From the discussed above, we found that the ways of efficiently implementing the explicit join and projection are not available and have to be sought before these designs of database machines can move from prototype to product. Academia Sinica now has an on-going National Science Council sponsored project on the design and analysis of a hardware architecture for supporting full join and projection operations. The prelimilary results reveal that this architecture holds promise for providing direct support for join-dominating applications. Furthermore, it has been found that the same hardware used for explicit join can also be used for supporting the multiple column projection. The hardware architecture will be reported in a forthcoming report [12].

Mechanisms for controlling database access and integrity have been proposed and reported by Honag and Su [13,14]. In addition, this type of machine does not have facilities for other functions such as system recovery, error recovery, reliability, etc. The arithmetic processing capabilities of the search logic are still limited. No floating point arithmetic is allowed. Data in this type of machine can be search in parallel. However, the selected tuples cannot be transferred out in parallel. Perhaps the most important problem confronting this area is the lock of operating experience in real application.

REFERENCES

1. Astrahan, M. M. et al., "System R: Relational Approach to Database Management," ACM Trans. Database Syst., Vol.1, No.2, June 1976, pp.97-137.

2. Babb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM Trans. on Database Systems, Vol.4, No.1, Mar. 1979.

3. Banerjee, J., and D. K. Hsiao, "DBC — A Database Computer for Very Large Databases," IEEE Trans. on Computers, Vol.C-28, No.3, 1979.

4. Baum, R. I., J. Banerjee, and D. K. Hsiao, "Concepts and Capabilities for a Database Computer," ACM Trans. on Database Systems, Vol.3, No.4, Dec. 1978, pp.347-384.

5. Chang, H., "On Bubble Memories and Relational Data Base," Proc. 4th International Conf. Very Large Data Bases, West Berlin, 1978, pp.207-229.

6. Chen, T. C., V. W. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files," Proc. 4th International Conf. Very Large Data Bases, West Berlin, 1978, pp.312-315.

7. Code, E. F., "A Relational Model of Data for Large Shared Data Banks," ACM, Vol.13, No.6, June 1970, pp.377-387.

8. Code, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposia 6: Data Base Systems, Prentice-Hall, Englewood Cliff, N. J., May 1971, pp.65-98.

9. Copeland, G. P., G. J. Lipovski, and S. Y. W. Su, "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proc. 1st Annual Symposium on Computer Architecuture, Dec. 1973, pp.121-128.

10. Edelberg, M. and L. R. Schissler, "Intelligent Memory," Proc. 1976 NCC, Vol.45, AFIPS Press, Montvale, N. J., pp.393-400.

11. Healy, L. D., K. L. Doty and G. J. Lipovki, "The Architecture of a Context-Addressed Segment Sequential Storage," Proc. 1972 FJCC, Vol.41, Pt.II, AFIPS Press, Montvale, N. J., Pp.691-701.

12. Hong, Y. C., Lin, C. Z., and Lin, C. C., "Efficient Computing of Joins by Means of Specialized Hardware," submitted for publication.

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

14. Hong, Y. C., and S. Y. W. Su, "A Mechanism for Database Protection in Cellular-Logic Device," Paper under review for the IEEE Trans. on Software Eng.

15. Hsiao, D. K., K. Kanan, and D. S. Kerr, "Structure Memory Designs for a Database Computer," Proc. ACM 1977, Dec. 1977, pp.343-350.

16. Lipovski, G. J., "Architectural Features of CASSM: A Context Addressed segment Sequential Memory," in Proc. 5th Anny. Symp. on Computer Architecture, Palo Alto, CA., Apr. 1978, pp.31-38.

17. Lin, C. S., D. C. P. Smith, and J. M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," ACM Trans. Database Syst., Vol.1, No.1, Mar. 1976, pp.53-65.

18. Madnick, S. E., "INFOPLEX —— Hierarchical Decomposition of a Large Information Management System Using a Microporcessor Complex," Proc. 1975 NCC, Vol.44, AFIPS Press, Montvale, N. J., pp.581-586.

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

20. Ozkarahan, E. A., S. A. Schuster, and K.C. Smith, "RAP —— An Associative Processor for Data Base Management," Proc. 1975 NCC, Vol.45, AFIPS Press, Montvale, N. J., pp.379-387.

21. Parhami, B., "A Highly Parallel Computer System for Information Retrieval," Proc. 1972 FJCC, Vol.41, Pt. II, AFIPS Press, Montvale, N. J., pp.681-690.

22. Parker, J. L., "A Logic Per Track Device," Proc. IFIP Cong. 1971, North-Holland Pub. Co., Amsterdam, pp.TA4-146-TA4-150.

23. Schster, S. A., E. A. Ozkarahan, and K. C. Smith, "A Virtual Memory System for a Relational Associative Processor," Proc. 1976 NCC, Vol.45, AFIPS Press, Montvale, N. J., pp.855-862.

24. Slotnick, D. L., "Logic per Track Devices," Advances in Computers, Vol.10, J. Tou, ed., Academic Press, New York, 1970, pp.291-296.

25. Stonebraker, N., "A Distributed Data Base Machine," Electronics Research Laboratory, UC Berkeley Memorandum No. UCB/ERL M78/23.

26. Stonebraker, M. R., E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM Trans. Database Syst., Vol.1, No.3, Sept. 1976, pp.189-222.

27. Su. S. Y. W., "Cellular-Logic Devices: Concepts and Applications," Computer, 12, 3, March 1979, pp.11-25.

28. Su, S. Y. W., "Associative Programming in CASSM and Its Applications," Proc. 3rd Int'l Conf. on VLDB, Tokyo, Japan. Oct. 6-8, 1977, pp.213-228.

29. Su, S. Y. W. and A. Emam, "CASDAL: CASSM's Data Language," ACM Trans. Database Syst., Vol.3, No.1, Mar. 1978, pp.57-91.

30. Su, S. Y. W. and G. J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp.456-472.

31. Su, S. Y. W., L. H. Nguyen , A. Emam , and G. J. Lipovski, "The Architecture Features and Implementation Techniques of the Multi-cell CASSM," IEEE Trans. on Computers, C-28, 6, June 1979.

32. Todd, Stephen, "Hardware Design for High Level Databases," IBM United Kingdom Scientific Centre, Peterlee, TN 49, 12 pp.