PROCEEDINGS OF 1980 NATIONAL COMPUTER CONFERENCE, AFIPS PRESS.

DIALOG—A distributed processor organization for database machine

by BENJAMIN W. WAH and S. BING YAO

Purdue University
West Lafayette, Indiana

1. INTRODUCTION

The conventional physical storage mechanism of a computer system is usually comprised of a memory hierarchy that stores program and data. The requirement for high performance and low cost is achieved through a combination of memories of different speeds. By automatically managing the files so that the most frequently used files reside in fast storage, an overall speed comparable to the speed of the fastest memory can be achieved. However, with the applications of large databases, the maintenance of large files on a conventional memory hierarchy becomes increasingly difficult. Most database applications perform a small number of simple operations on a large amount of data. Usually only a small fraction of the data accessed is required by the application. It is more cost effective to perform database operations directly on the data in the secondary storage in order to avoid the transfer of unnecessary data across different levels of the memory hierarchy. The Database Machine (DBM) is the result of an architectural approach which distributes processing power closer to the devices on which data are stored and offloads database processing functions from the main computer [LAN79].

The feasibility of database machines has greatly improved with recent advances in the computer system device technology. The number of components per chip is approximately doubling each year and the CPU speed is growing exponentially. At the same time, the cost per unit of memory is decreasing and a wide variety of new storage devices, which include CCD memory, bubble memory, and the electron beam access memory, are becoming available. It is now possible to design inexpensive processing elements to be embedded within these new storage devices. Disk technologies have also been improved and it is possible to provide inexpensive secondary and archival storage to the computer system. On the other hand, the software costs and overheads are usually a deterrent factor for computer system development. This cost is becoming dominant in present data processing systems, and is expected to increase rapidly to an even higher percentage in the near future. The replacement of expensive database system software with custom built hardware is therefore a feasible and desirable trend in the future.

The database machine proposed so far can be divided into three types. The first type is the backend system which utilizes mini-computers to enhance the database processing of large host computers. The functions of the backend system include access validation, storage management, concurrency control and I/O control. However, the principle of the design is basically the same as a conventional database system. Large amounts of data not required by the database application are accessed and transferred to the mini-computers. The use of a backend machine is only a temporary method to extend the processing power of a large CPU.

The second type of database machine utilizes the single instruction multiple data stream (SIMD) principle. This concept is extended from backend machines in which the database processing functions are moved to a lower level. The characteristics of this design are that simpler, less costly processors that are dedicated to a small block of data are used. This concept, when applied to the storage cells (modules), is exemplified by the Logic-Per-Track device in which processing logic are duplicated for each track and the keys in different tracks are searched in parallel. Examples of this design are TapeDRUM [HOL56], Slotnick's Logic per Track Disk [SLO56], RAPID [PAR72], CASSM [LIP78, SU79], RAP [OZK77, SCH79], RARES [LIN76], DBC [BAU76, KER79, BAN79], and Chang's Major/Minor Loop Machine [CHA78]. In this type of design, the database workload must be heavy in order to keep the parallel resources fully utilized. For large databases, the amount of replicated processor elements may be prohibitively large. The degree of parallelism in this type of design is also limited by the number of read/write heads, i.e., the number of data streams that can be read in parallel. This concept results in an expensive memory device with parallel read/write heads. Moreover, the processors under most of these designs are quite general with limited amount of communications. Lastly, if the database machine is built on a disk, the processors must be extremely fast because many fast signal translations are needed in order to perform real time processing and disk marking.

If the replication goes further down to the bit level, an associative memory results. Associative memories are usually very expensive and, therefore, can be only shared by swapping in data when needed. This is exemplified by RELACS [BER79] in which STARAN is used as the associative

memory. However, this design experiences the usual problem of data swapping in a memory hierarchy, and there is no provision to select data before it is accessed and transferred to the associative memory. The throughput of the system is therefore quite limited. Further, I/O lines must be extensive in order to load the associative memory in parallel.

The last type of database machine design is based on the Multiple-Instruction-Multiple-Data System (MIMD) principle. In this architecture, which is exemplified by DIRECT [DEW79], the processing logics are separated from the storage modules and are interconnected with an array of CCD memories through a cross bar switch. This design offers more flexibility and better load balancing and allows the processors to be shared among the storage modules. Because of the fact that each processor can access multiple storage modules simultaneously, it is easier to perform database operations which require multiple files to be coupled, e.g. a multi-relation join. Further, expansion is easy and modular. However, this design suffers from the same disadvantage as the associative memory when the size of the CCD memory is not large enough, in which case excessive swapping will occur. Again, since the processing logic is removed from the storage device, large amounts of unnecessary data are accessed and transferred.

We note that the previous designs are built around a single type of storage device, e.g. disk, CCD, etc. Intramodule operations can be performed very efficiently because they do not utilize the I/O bus. However, inter-module operations often result in a bottleneck at the I/O bus. At any one instant, only one inter-module operation can be processed because the designs are essentially SIMD. In some designs, e.g. DIRECT, where the problem of I/O bus has been solved by using a simple cross bar switch, the operations are expensive because data have to be transferred from the mass storage to the CCD storage modules. Only when a sizable amount of operations are performed on the file transferred would the transfer be cost effective.

In this paper, we propose a design of a backend database machine, DIALOG(DIstributed Associative LOGic database machine) which addresses some of the problems mentioned above. We want to design intelligent but simple processing logic so that they can be replicated on the storage modules. Algorithms such as select, project and join will be implemented in hardware so that they can be processed very efficiently. These processors work directly on the storage devices, so that the amount of data transfer is kept at a minimum. A network is proposed which provides a uniform medium to connect heterogeneous memory devices together.

The DIALOG database machine is designed with the following design goals in mind. First, the system should be extendable and be able to support very large on-line databases in the future. Second, the design must have high performance and the cost should be low by replicating a few simple devices. Third, the system should use existing memory technologies in the design such that it can be implemented now, and the design should be able to evolve as new memory technologies are available in the future. Fourth, the system should accommodate heterogeneous storage devices such that files with different workload and sizes can be stored in the devices with appropriate speeds and sizes. Fifth, the system should implement low level operations (such as select and join) and facilitate higher level query optimization. Finally the system should be able to interface with the host computer in multiple data models (such as relational, hierarchical, and network data models [COM76]).

This paper is divided into five sections. Section 2 provides the overall architecture of the system. It also illustrates how an inter- or intra-module operation is performed. Section 3 describes the architecture of the data module. The functions to be performed in a module are partitioned into the select processor, the associative processor, the join processor and the communication processor. Section 4 presents an analysis on the buffer size required. Section 5 provides some discussions of this system and compares this system against other systems. Lastly, Section 6 gives some concluding remarks.

2. SYSTEM ARCHITECTURE

The secondary storage, where the bulk of the database is stored, is usually made up of multiple types of storage devices like disks and tape drives. As the storage technology advances, devices such as CCD memories, bubble memories, EBAM's, etc. could become part of the storage system. Each of these storage types has different capacities and speeds. Since many database operations require data stored in multiple files, it is often necessary to access these files simultaneously. One approach is to transfer all the required data for an operation to a uniform storage device such as CCD memories, which are connected to a set of processors, before processing it. However, there is a significant overhead associated with this data transfer, especially when the original storage device has no selection capability. Further, if the intermediate device is not large enough to accommodate the entire file, costly multiple passes have to be made.

An alternative approach is to send the required data directly from the storage device on which the file is stored to a second storage device on which the file is to be processed. The time to stage these files to an intermediate storage device can be saved. If the storage devices are enhanced by sufficiently powerful processing elements, the results of database operations can be output at a very high speed. In order to facilitate the transfer of data, a communication network must be designed to connect the heterogeneous devices together.

In this section, we describe the general architecture for the latter approach to database machine design. The design of the network will be examined and the processing capability of the machine will be discussed.

A. General architecture

The general architecture of the system is shown in Figure 1. Data in the system is stored in the *data modules*. Each

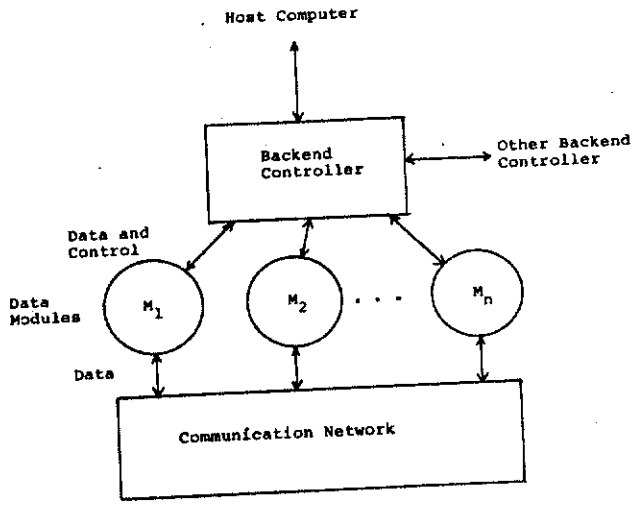

Figure 1—System architecture of a cluster in DIALOG.

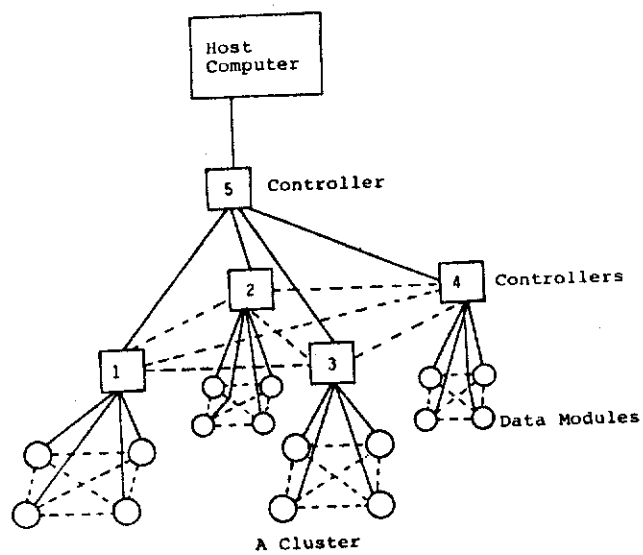

Figure 2—An example of the hierarchical network in DIALOG.

data module is consisted of a storage device and an associative processor. Data modules are connected together by an interconnection network. Only data can flow across this network. The cluster of data modules are connected directly with a backend controller which allows both data and control communication. The backend controller provides communication between the cluster of data modules and the host computer. Queries expressed in a high level representation are sent from the host to the backend controller and query responses are returned to the host.

The major functions of the backend controller of a cluster include pre-processing and optimizing the queries, looking up system directory, establishing links between data modules, initiating and scheduling operations within each data module, receiving and buffering output from data modules, managing the sharing of resources, and initiating rollback and recovery as system components fail. All of the above functions will be implemented in software.

Depending on the number of data modules in the database machine, the interconnection network may become quite complex. It may be necessary to group data modules into multiple clusters. The backend controllers of these clusters are connected using a network of the same design as that developed for interconnecting the data modules within a cluster. A higher level backend controller is used to coordinate the controllers of all the clusters.

An example of this hierarchical network is shown in Figure 2. We note that the communication between data modules of different clusters is achieved by sending data through the backend controllers and a high level communication link. The highest level backend controller assumes the responsibility of communicating with the host computer. The operation of the network in each level of the hierarchy behaves in the same fashion as the network in the lowest level which will be discussed later.

B. Processing capabilities

Although DIALOG is designed to support multiple data models, only the database operations defined on the relational data model [COD70] are discussed in the present paper. The relational operations currently included in the design are: select, project, join, union, and cartesian product.

Operations that require only one file, such as selecting records that satisfy a given predicate and projecting on certain attributes are usually resolved within one data module. Database operations that require the cross-referencing of files stored in two data modules (such as join and cartesian product) are performed by sending all the required records to one of the data modules and then processed there. In the case when the two files to be processed reside in the same data module, one of the files can be retrieved into the input buffer. The processing continues as if the file in the input buffer were received from a different module.

Database operations that access multiple files in several data modules are decomposed into a set of two-module operations which can be processed either in sequence or in parallel. The decomposition and scheduling of database operations and the routing of files are determined by the backend controller.

C. Network design

The design of the interconnection network has an important implication on the throughput of the system. A ring network is the simplest form of communication network, but it could cause a lot of contention. If the conflict is resolved by a centralized controller, the controller may become a bottleneck of the system. If the conflict is resolved in a dis-

tributed fashion, the software for the communication protocol may be complicated. It is therefore desirable to have a network that is conflict free and requires a small amount of software development.

The simplest conflict-free network is the fully connected network, although this network may seem very expensive because the number of links grows as n^2 where n is the number of data modules. However, these links are merely serial lines governed by a central clock, and their costs are minimal. An implementation of the link between two data modules is shown in Figure 3. The link can be established by the controller which sends a command to the communication processing modules of the two data modules. The communication processors at the two data modules then set the multiplexor and the demultiplexor and the link is established. The links can only be broken by a command from the controller. The communication processor is responsible for managing the buffer pool, detecting and correcting errors in the data, informing the controller if a non-recoverable error is detected, and signaling the source of a data transfer to stop when the buffer is full. Since the communication between two data modules is governed by a central clock, any type of memory device can be connected to the network. The design can also be modified to broadcast data to several modules simultaneously.

With this network design, global data transfer is very easy. Referring to Figure 2, suppose one of the data modules in cluster 1 wants to send data to a data module in cluster 3. Since there is no direct path of communication between these two data modules, data from cluster 1 will be sent first to its controller. Since there is a direct path of communication between controllers of clusters 1 and 3, this path is established by controller 5, the controller in one level higher than controllers 1 and 3. The file transfer can be carried out without the intervention of controller 5. Of course, this approach can result in excessive overhead for the lower level controllers when inter-cluster transfers are large. However, by carefully allocating the relations to the clusters so that the inter-cluster communications are minimal, this technique is still a cost effective approach.

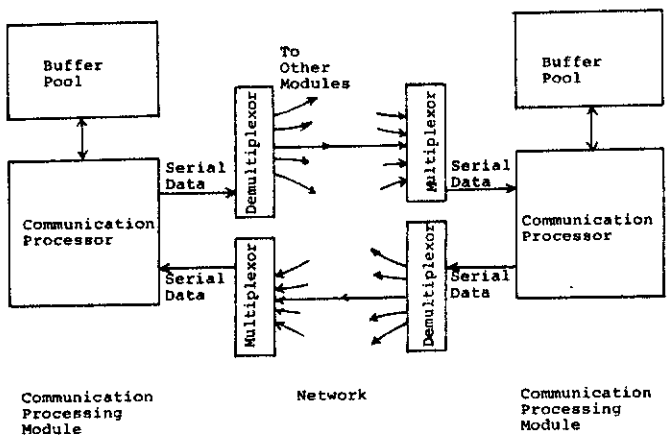

Figure 3—A communication link between two data modules.

3. DATA MODULE DESIGN

In this section, we describe the internal design of the data module. The function of the data module is to store the data and to process relational queries directed on the stored data. The design should allow both inter- and intramodule queries to be performed efficiently. Furthermore, the design should be modular and functional so that current microprocessor and VLSI technologies can be used in the design.

A. General data module architecture

The general architecture of a module is shown in Figure 4. There are five basic sub-modules of the system, namely: the physical storage device which contains the database; the selection processing module which processes projection and selection operations; the associative processing module which compares the output from selection processing with the stored search keys and passes successful matches to the join processing module which then produces the join; and lastly, the communication processing module which manages the buffer pool and communicates with other data modules in the network. We describe each of these sub-modules here.

B. Data allocation in the physical storage device

The physical storage device is made up of multiple circulating loops of data with a single read/write head for each loop. The loops may be simple which can model devices like CCD memories. It may be more complicated such as the major/minor loop of a bubble memory, the LARAM organization of the CCD memory, or a cylinder of a disk. The access time distribution for accessing a piece of data within the loop consists of two components, the time to switch to read/write a particular loop and the time to shift the data in the loop to the read/write mechanism. Different memory devices have different time delays for each component. In the case of a single loop, data can only be read out serially. However, in the case of a multi-loop organization, data may be read out one loop at a time with electronic switching between the loops. This is characterized by a cylinder of a disk or a LARAM organization of CCD memory. Data may also be read out serially from all the loops simultaneously. This type of memory is characterized by a multi-chip bubble or CCD memory and the Parallel-Transfer-Disk [AMP78]. However, in some implementations, it may not be possible to synchronize all the parallel outputs on the device (e.g. a cell may be bad in one of the loops and unless all the corresponding cells in other loops are marked bad, the outputs will be out of synchronization). It is therefore not practical to organize the database such that a tuple of a relation is read out bit parallel but word serially. The organization of the database in a multi-loop device is therefore chosen to be bit-serial and word-parallel, that is, multiple tuples will be available simultaneously at the read heads. In this case, since we assume that the selection processing module can

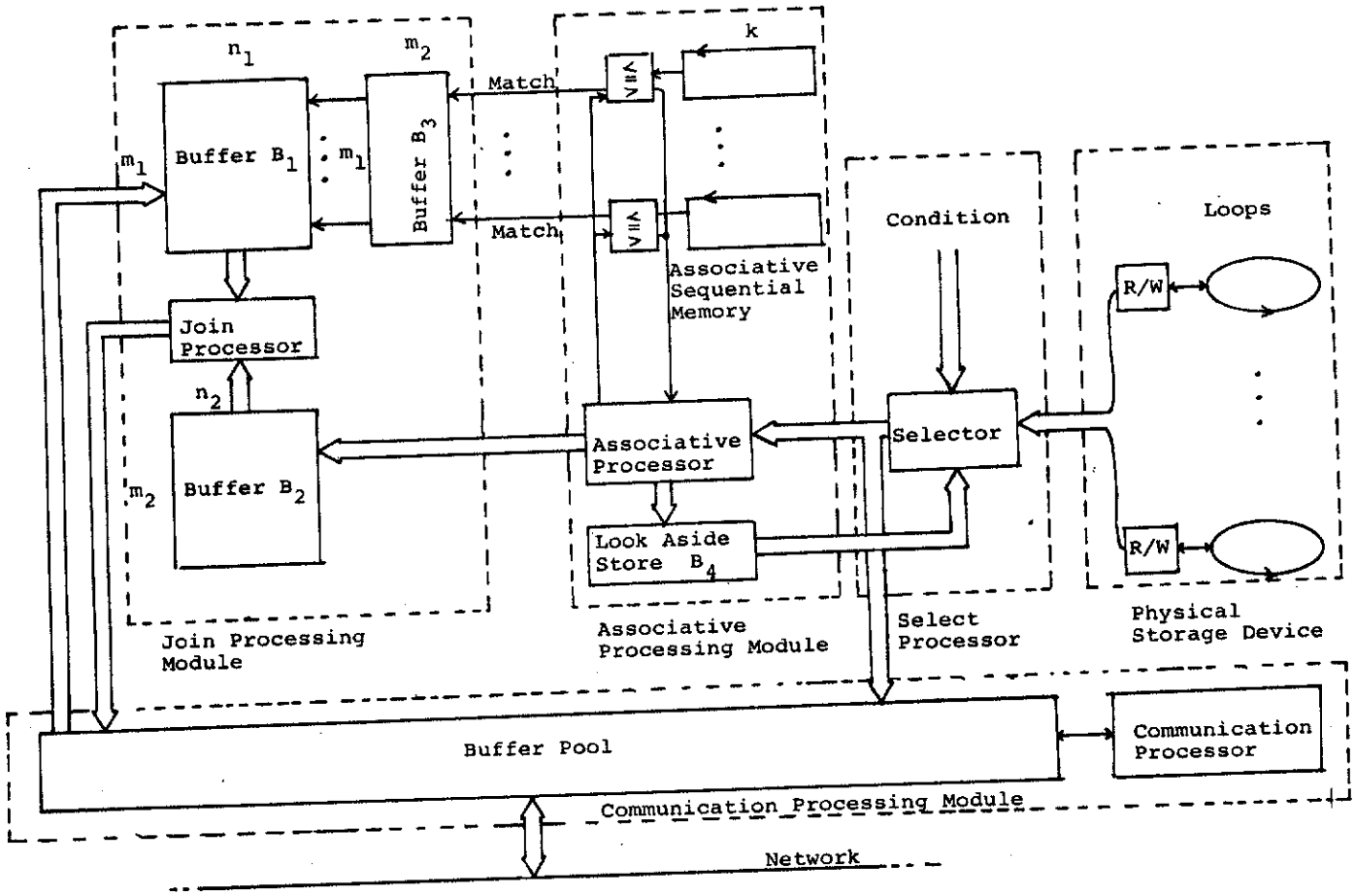

Figure 4—Architecture of a data module.

process one tuple at a time, a buffer is added between the physical storage device and the selection processing module so that all the parallel tuples are sequenced sequentially. Of course, the speeds of the other processing modules have to be increased from a single loop model correspondingly. With the use of a buffer, a multi-loop device behaves like a single loop device. We conclude that a single loop model is sufficient for our design.

C. Selection processing

In selection processing, tuples are examined in the database one at a time and only those tuples satisfying certain predicate will be output. This output can be sent to the buffer pool or it can be sent to the associative processing module. When the output is sent to the buffer pool, it may be sent to another data module in the system for join processing or it may be used to join with another relation on the same data module. Projection is an operation similar to selection. However, we assume that duplicates will not be removed in the data module because of the limited buffer capability.

D. Join processing

In performing a join on this system, both the associative processing module and the join processing module will be used. The buffer data allocations are as follows (see Figure 4). Buffer B_1 contains m_1 tuples, each of n_1 bytes. The contents of B_1 is filled from the buffer pool and they represent part of the tuples of a relation coming from the same or a different data module in the system. These tuples are used to join with the tuples coming from the output of the selection processing module. The size n_1 is chosen so that it can accommodate the largest tuple used. The size of m_1 will be determined in Section 4.

The associative sequential memory in the associative processing module is made up of m_1 circulating loops of size k. It contains the keys of the join domain of the corresponding tuples in buffer B_1. It is a one to one correspondence and the keys of the ith tuple in B_1 is contained in the ith loop of the associative sequential memory. The size k is chosen so that the loop can accommodate the largest key used. For smaller keys, multiple copies may be made in the same loop to allow faster access.

The associative processing module compares the serial output tuple from the selection processing module associatively against all the keys in the associative sequential memory. If at least one match occurs, it indicates that a join is possible between this tuple and the corresponding tuple in B_1. This tuple is put in buffer B_2 and it will be joined with the corresponding tuples in B_1 when all the previous joins are performed. B_2 contains a queue of m_2 tuples, each of n_2 bytes that can be used to join with keys in B_1. The size of n_2 is chosen so that it can accommodate the largest tuple used (n_1 and n_2 are probably chosen to be identical). The size of m_2 is determined in Section 4.

As the match is done in the associative processor and a successfully matched tuple is added to B_2, a word is written into buffer B_3. B_3 is an m_1 by m_2 bit matrix. A column in B_3 represents the set of tuples in B_1 which match with the corresponding tuples in B_2, where a 1 indicates a match and a 0 indicates a no-match.

The example in Figure 5 illustrates the functions of B_1, B_2, B_3 and the associative sequential memory where m_1, m_2 are chosen to be 2 and n_1 and n_2 are chosen to be 12. In this example, suppose the following join is to be performed.

RETRIEVE (A.city, B.pname): A.$p\#$ = B.$p\#$

B_1 contains part of the tuples of relation A, and B_2 contains part of the tuples of relation B which are to be joined with the tuples in B_1. For the left column in B_3, it indicates that the first tuple in B_2 can be joined with the first tuple in B_1. Similarly, the right column in B_3 indicates that the second tuple in B_2 can be joined with the first tuple in B_1. The loops in the associative sequential memory contain the keys of the corresponding tuples in B_1. Note that in our example, the $p\#$ domains in B_1 and B_2 are never output from the join processor and they are not used by the join processor to determine which of the tuples in B_1 and B_2 are to be joined (because B_3 provides this information). These two domains therefore do not have to be included; however, they are shown here for illustration.

The join processor performs the join between the tuples in B_1 and B_2 by picking up the first tuple in B_2 and joining it with all the tuples with matched keys in B_1, before proceeding to the next tuple in B_2. The results of the join processor are routed to the buffer pool.

E. *Associative sequential memory*

The design of this associative sequential memory is based on the design by Ramamoorthy, Turner and Wah [RAM78,

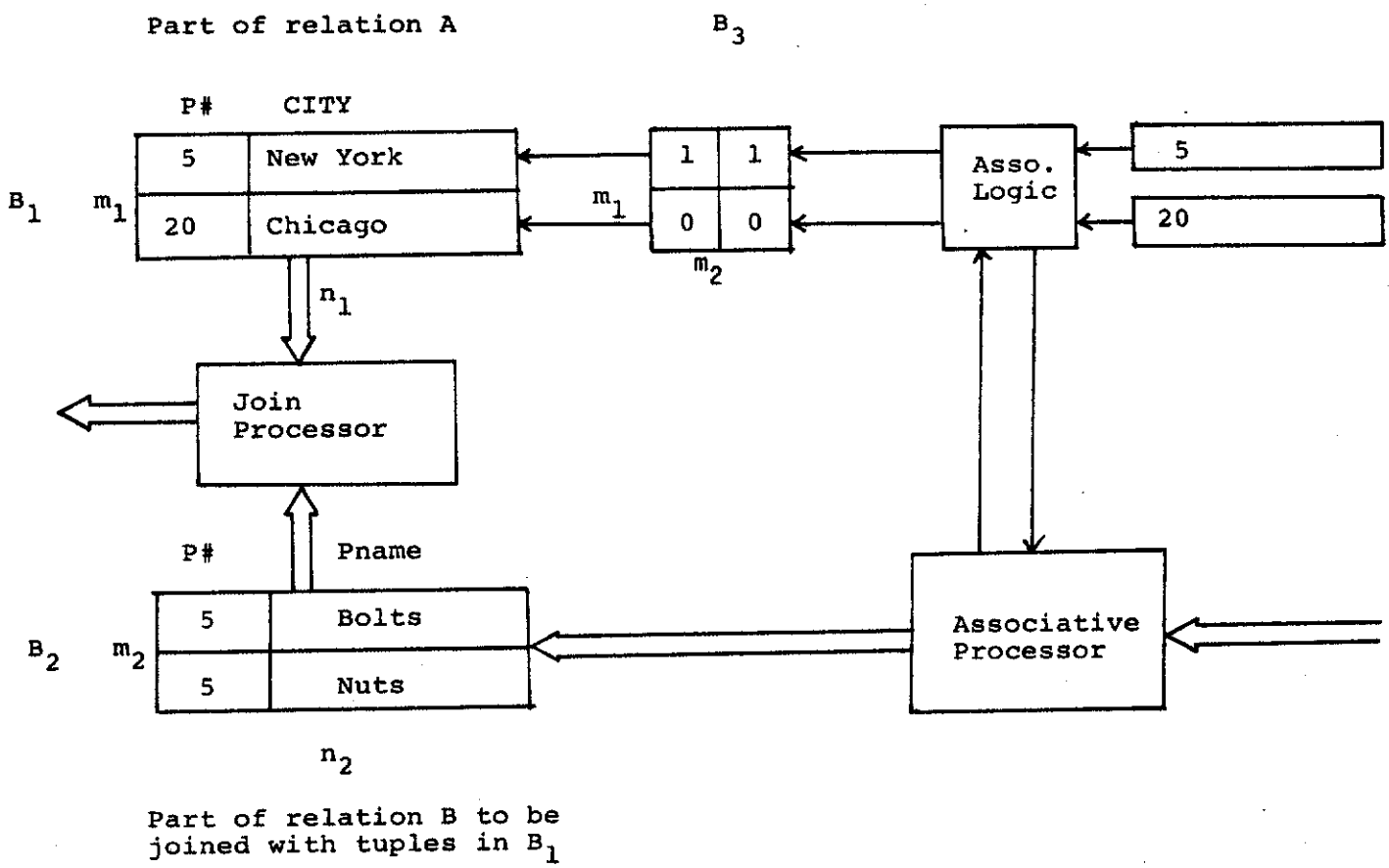

Figure 5—Example to illustrate the function of buffers B_1, B_2, B_3 and the Associative Sequential Memory.

WAH79]. The sequential memory is made up of multiple loops of circulating bits shifting in synchronism that may be CCD memory or bubble memory. In this design, m_1 keys are stored in the memory, with one key occupying each loop. Associative logic capable of performing equality, threshold and proximity searches are put at the read/write head of each loop. The architecture of this design is shown in Figure 6. The detailed cell logic is shown in Figure 7. During a clock period, a bit-slice of these m_1 keys is shifted out from the sequential memory and is compared associatively with the output from the selection processing module. The enable signals are stored in temporary flip flops. As the bit-slices are shifted out, most significant bit first, the bit-slice, together with the stored enable signals, generate a new set of enable signals which are stored back into the temporary flip flops. There are three significant improvements in this design as compared with conventional logic-per-track device. First, the additional logic for each loop is very small and therefore the cost increase is minimal. Further, the cell logic is simple enough to be implemented on the same chip as the memory elements with only a minimal cost increase. Second, there are no communication lines between two adjacent loops and therefore the number of loops can be modularly expanded. Lastly, the number of loops is not governed by the number of read/write heads on the physical storage device. Previous database machine designs assume that the number of associative logic is replicated for each head of a disk which therefore limits the maximum degrees of parallelism. In our design, the number of heads on the disk is not a factor in the number of loops used in the associative sequential memory. The degree of parallelism can therefore be improved significantly.

One problem that exists with this design is to choose the loop size of the associative sequential memory so that the

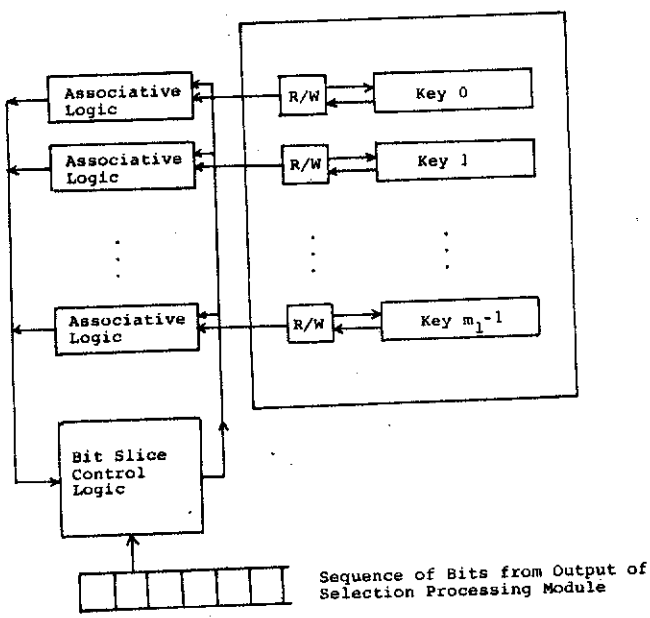

Figure 6—Associative Sequential Memory [RAM78, WAH79].

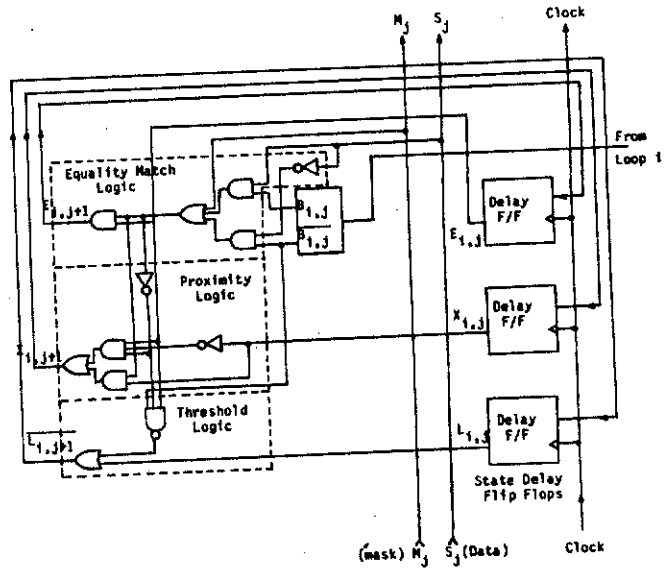

Figure 7—Associative logic for Associative Sequential Memory with equality, greater than, less than and proximity searches [RAM78, WAH79].

largest key can fit into the loop. However, if the loop size is too large and the time it takes for the loop to make one complete revolution is longer than the time it takes for a tuple to be shifted out, the associative sequential memory may not be able to catch up with the output rate of the selection processing module. In this case, one simple solution is to duplicate the keys in the loop so that the time to reach the key is always shorter than the time to shift out a tuple. Buffers may also be placed between the associative processing module and the selection processing module to smooth out the irregularities. Since the output of the associative processing module goes to the join processing module and there is a finite probability that B_2 is full, the associative processor may not be able to continue to process the tuples once B_2 is full. In this case, a lookaside buffer, B_4, is used to store the missed tuples. This buffer is a queue with each element being a tuple identifier. The size of B_4 is chosen so that it would accommodate all the missed tuple identifiers.

F. Join processor

The join processor uses buffers B_1, B_2, and B_3 to produce the join outputs. One characteristic about B_2 and B_3 is that they are variable size queues. This feature may be implemented by a hardware or a software linked list. Further, accesses to each column of B_3 must be made in parallel. The join processor uses one column of B_3 to find out which tuples in B_1 are to be joined with a tuple in B_2. This is a conventional multiple match resolution problem. We assume that a sequential search is made to find all the 1's in a column of B_3. However, tree circuits [FOS68] or associative memory [RAM78, WAH79] can be used to do the multiple match resolution.

We have discussed in this section the detailed design of

the data module. Finally, when the tuples in B_1 have been matched with all the tuples in the physical storage device (e.g. a cylinder, a loop, etc.), the contents of B_1 and the associative sequential memory are switched to a new set. As data are moved into the buffers, they are also moved in parallel into buffers B_1 and the associative sequential memory. The processing of a join is therefore pipelined and the throughput of the system can be greatly enhanced. The throughput of the system is a function of the sizes of B_1, B_2, B_3, the tuple size and the bit rate (the rate at which tuples are selected out from the physical storage device). We present in the next section an analysis for the sizes of B_1 and B_2 if the join outputs are to be made in disk transfer rate.

4. APPROXIMATE ANALYSIS OF BUFFER SIZE

In this section, we perform an analysis on the size of the buffers. The sizes of B_1, B_2 and B_3 are critical factors in our design. We first establish an upper bound on the average queue length in B_2 and B_3 given the size of B_1 and the probability that a match occurs between the contents of B_1 and a tuple output from the selection processing module. We further assume that B_2 and B_3 are very large. Using the upper bound on the queue size of B_1, we can establish a size for B_2 and B_3 so that, under most circumstances, B_2 and B_3 will not be full. The approximations are then compared with the simulation results.

Let:

P = probability that a tuple of relation A in B_1 will be joined with a tuple of relation B;
λ_a = length of each tuple of relation A in B_1;
λ_b = length of each tuple of relation B in B_2;
λ_a' = length of each tuple of relation A after restriction;
r = transfer rate of physical storage device;
r_0 = desired join output rate;
s_1 = number of tuples in B_1 (= size of buffer B_1, m_1);
s_2 = number of tuples in B_2 (size of buffer B_2 is assumed to be very large);
$\overline{s_2} = E(S_2)$.

When the join processor picks up a tuple in B_2, there is at least one match with a tuple in B_1 (otherwise, the associative processor would not have passed this tuple to B_2). The time to produce all the corresponding joins depends on the number of matches in B_1. For i equal matches ($i = 1,2,\ldots,s_1$) with a probability of $\binom{s_1}{i} p^i(1-p)^{(s_1-i)}/1-(1-p)^{s_1}$, the time to produce all the joins are $(\lambda_a' + \lambda_b)*i/r_0$. On the other hand, the inter-arrival time distribution is geometrically distributed, with a probability of $(1-p)^{s_1(i-1)}*(1-(1-p)^{s_1})$ for an interarrival time of $(i*1_a)/r$ ($i = 1,2,\ldots$). This system is in effect a GI/G/1 queue. The queue is shown in Figure 8. We use Marshall's inequality [MAR68] to find the average delay time that a tuple spends in B_2, d.

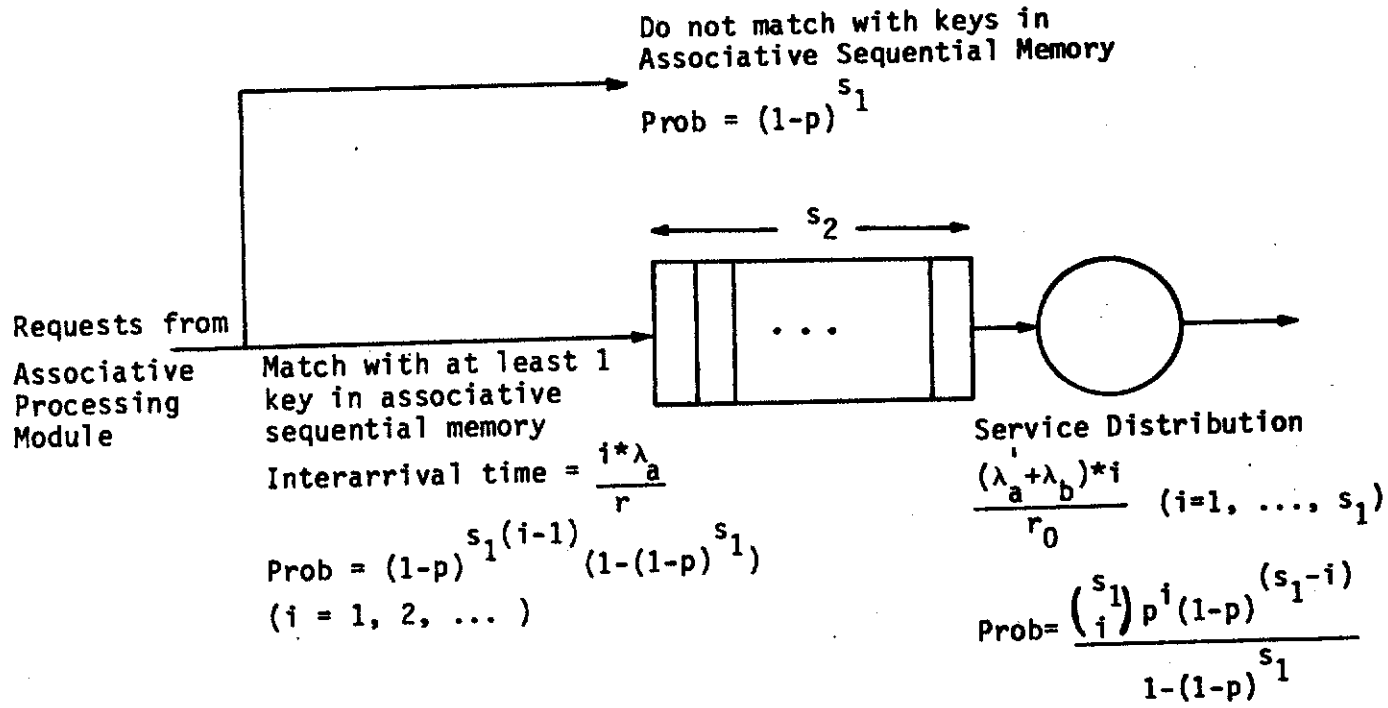

Figure 8—Queue to model operation of associative processing module and join processing module.

$$d \leq \frac{\lambda(\sigma_a^2+\sigma_g^2)}{2(1-\rho)} \qquad \text{(Marshall's Inequality)}$$

where $1/\lambda$ and σ_a^2 are the mean and the variance of the interarrival time, $1/\mu$ and σ_g^2 are the mean and the variance of the service time and $\rho=\lambda/\mu$. After evaluating d, we can further apply Little's Formula [LIT61] to find a bound on $\bar{s}_2(\bar{s}_2=\lambda d)$

$$\bar{s}_2 \leq \frac{\left[\bar{p}+\left(\frac{\lambda_a'+\lambda_b}{\lambda_a}\right)^2\left(\frac{r}{r_0}\right)^2 s_1 p(1-\bar{p})\left[p(s_1-1)+1-\frac{s_1 p}{(1-\bar{p})}\right]\right]}{2\left[1-\frac{r}{r_0}\left(\frac{\lambda_a'+\lambda_b}{\lambda_a}\right)s_1 p\right]}$$

where $\bar{p}=(1-p)^{s_1}$. Assuming that $\lambda_a=\lambda_b=\lambda_a'$, $r=r_0$, we have,

$$s_2 \leq \frac{\bar{p}+4s_1p(1-\bar{p})\left[p(s_1-1)+1-\frac{s_1p}{(1-\bar{p})}\right]}{2[1-2s_1p]}$$

Simulations are also carried out on the queue in Figure 8. The simulation program is written in ASPOL and is run with 2000 requests for each pair of p and s_1.

Table I shows the difference between the simulation results and the estimated bounds for a sample of values of s_2 for various values of s_1 and p. It is seen that the estimations, although not accurate, produce a close enough bound for s_2. We have also indicated in Table I a column that indicates the value of s_2 such that Pr(queue size $<s_2$)$\leq$0.95. These values can be used to determine a fixed bound on s_2. For example, with $p=0.01$, which means that there is a chance of 1 percent for a particular tuple in relation B to be joined with a tuple in relation A in B_1, s_1 and s_2 can be chosen to be 50 and 36 respectively. For these values of s_1 and s_2, the utilization of the server (join processor) is 0.994, which means that the join output is indeed coming out at the disk transfer rate. However, the choice of s_1 and s_2 is not arbitrary. If s_1 is set to be too large, the time to switch the contents of B_1 is large which induces large overheads on the system. On the other hand, if s_1 is too small, then there may not be enough requests in B_2 and therefore the join processor may be idle most of the time. It is therefore necessary to choose s_1 as small as possible while giving a reasonably high utilization for the join processor. The choice is driven by the value of p which is dependent on the number of tuples in the relations and the number of distinct join keys in the two relations. For large relations, this is usually small. Table I shows some simulation results with p set to 0.001. It is seen that with s_1 set to 160, the utilization of the server (join processor) is 0.317 and s_2 can be set at 1. This means that the join processor will be idle 69 percent of the time and the output rate is 31 percent of the disk transfer rate. With s_1 set to be so large, there is a tremendous switching overhead in the system. In order to remedy this, multiple sets of B_1 and associative sequential memories can be implemented in each data module. The join processing operations on one set of B_1 can be overlapped with the loading of another set of B_1. The switching overhead on B_1 is therefore overlapped with the join processing. This is feasible because the communication processor in each data module can be designed to send and receive from different data modules so that as the join processor is producing outputs to be sent to one data module, the communication processor can be receiving tuples from another data module to load a different set of B_1. A final problem occurs when p is extremely small. In this case, B_1 has to be chosen to be excessively large in order to attain a reasonable join output rate. In this case, it may be better to send different sets of tuples of relation A to a set of spare data modules, each with a small buffer memory. Relation B from this data module is then broadcast simultaneously to this set of spare data modules where the join can be performed in parallel. This approach provides a solution for an extremely small value of p without constraining on the size of B_1. Buffers B_1 can be kept relatively small at each data module. The use of spare data modules can also be applied for load balancing. This will be discussed in the next section.

TABLE I.—Sample of Values of s_2 for Various Values of s_1 and p

		Simulation				Estimation
s_1	p	mean s_2	std. dev s_2	Pr(queue size$<s_2$) $\leq$ 0.95	Server Utilization	upper bound of s_2
5	0.01	0.003	0.055	0 (0.9970)	0.099	0.53
	0.05	0.152	0.445	0 (0.8737)	0.477	0.80
	0.10	10.538	7.784	24 (0.9440)	0.976	∞
	0.20	∞	∞	∞ (1.0)	1.0	∞
10	0.01	0.015	0.122	0 (0.9850)	0.191	0.57
	0.02	0.068	0.266	0 (0.9352)	0.377	0.68
	0.03	0.264	0.620	1 (0.9480)	0.570	0.97
	0.04	0.925	1.313	3 (0.9407)	0.773	1.89
	0.05	5.572	6.495	20 (0.9460)	0.978	∞
	0.07	∞	∞	∞ (1.0)	1.0	∞
15	0.01	0.032	0.183	0 (0.9698)	0.280	0.62
	0.02	0.252	0.550	1 (0.9570)	0.588	0.98
	0.03	3.066	3.484	10 (0.9588)	0.880	3.80
	0.04	∞	∞	∞ (1.0)	1.0	∞
25	0.01	0.177	0.523	0 (0.8641)	0.486	0.80
	0.02	29.993	12.346	46 (0.9534)	0.996	∞
	0.03	∞	∞	∞ (1.0)	1.0	∞
40	0.01	0.890	1.265	3 (0.9500)	0.770	1.91
	0.02	∞	∞	∞ (1.0)	1.0	∞
50	0.01	22.332	10.911	36 (0.9500)	0.994	∞
	0.02	∞	∞	∞ (1.0)	1.0	∞
75	0.001	0.007	0.082	0 (0.9933)	0.141	0.55
100	0.001	0.013	0.115	0 (0.9865)	0.192	0.57
120	0.001	0.019	0.136	0 (0.9816)	0.229	0.59
140	0.001	0.028	0.169	0 (0.9729)	0.278	0.61
160	0.001	0.046	0.225	0 (0.9578)	0.317	0.63

4. DISCUSSION

The design we have proposed in this paper uses both associative and distributed processing to enhance the system performance. Associative processing is used so that the data can be preprocessed and only the necessary data are selected. The amount of communications is therefore reduced. The associative processors designed are also very simple so

that they can be implemented easily using VLSI technologies. On the other hand, distributed processing is used in the design of the system and therefore many useful techniques in distributed databases can be applied to enhance the performance. These techniques include query processing, file placement and migration, rollback and recovery, etc. We discuss several of these issues here.

A. Query processing

Since the system functions as a distributed database, it is possible to optimize the sequence of the processing of queries so that some optimization criteria, such as minimum total time, can be satisfied. There are many techniques developed, e.g. [WON76, WON77, EPS78, HEV79], which can be used to optimize the processing of queries here. The processing of queries using this kind of data flow analysis is new in this design. Previously, the intermediate result of a query usually had to be stored in a temporary file before it could be used, e.g. DIRECT. By allowing intermediate results to be piped to their destinations, higher throughput can be achieved.

B. File placement and migration

Because each data module can read only one file at a time, query processing will be sequentialized if all the queries access files on the same data module. By allocating the files on the system so that related files are allocated to the same cluster and files that need to be processed together are allocated to different data modules, conflicts can be kept to a minimum and maximum utilization of resources can be attained. There are many file placement techniques developed, e.g. [CHU69, CAS72, MAH76, MOR77, WAH79] and can be applied to allocate the files on the system. The problem is easier to solve than the general file placement problem because duplicate copies are usually not needed in the system. Further, as the access characteristics change, the files can be reallocated dynamically.

C. Concurrency control

Although there may exist multiple copies of a file on the system, we do not allow multiple copies to exist on the same type of memory device. Therefore, multiple copies of a file may exist in devices of different speeds and the copy on the fastest device is considered to be the primary copy. All the other copies are considered as secondary copies and no write throughs are done during updates. With this restriction, the problem on concurrent updates is solved easily because the processor at each data module provides a secure gateway to the data on the device.

D. Load balancing

Although our design does not require staging to transfer the files to a fast memory before the processing, it is not able to utilize multiple processors to process a join in parallel as is done in DIRECT. However, we can provide some spare data modules on the system, each of which has a small buffer memory. Instead of sending tuples directly from one data module to another, different sets of tuples of one relation are sent to the spare data modules, and the tuples from the other relation are broadcast to the spare data modules. The join is therefore performed in parallel in the spare data modules. In this sense, this resembles DIRECT's design. However, there are two significant differences. First, data sent to the spare modules have been reduced in size by the selection processing modules of the two data modules and therefore their sizes are reduced. Second, the distributed system approach allows queries to be processed very efficiently because data can continuously be piped to other data modules as they are produced. A detailed evaluation of this system will be presented in a future paper.

6. CONCLUSION

In this paper, we have proposed a database machine design which facilitates query processing and parallel processing. Since the system is hierarchically designed, it can easily be extended to a very large database in the future. Communications at higher levels can be reduced by carefully allocating related files to individual clusters so that most of the communications are intra-cluster. The logic we have designed in the associative processing module is very simple and can be extended modularly. This is different from the conventional approach in which the degree of associative processing is limited by the number of read heads. Further, we have designed the system based on the assumption that current memory technologies are used, e.g. movable arm disk, CCD, etc. This means that the system can be built now. However, we have made no restriction on the structure of the memory which is assumed to be a single loop of data. If, in the future, data can be stored inexpensively in parallel transfer disks, the data can be assumed to be buffered so that a serial stream of access is always achieved. Since the network provides a homogeneous medium for data transfer, heterogeneous memory devices can always be connected together. This advantage also facilitates future evolution of the system. Lastly, our design implements both low level database operations and high level query optimization. The network approach is very versatile and allows high level operations to be implemented.

Our proposed design is given in a functional form. Although individual modules are separated, they do not have to be implemented in individual hardware. In fact, most of these functions can be built in either hardware and software. A detailed evaluation of the system will be shown in a future paper.

BIBLIOGRAPHY

[AMP78] Ampex Corporation, PTD-9300, Parallel Transfer Disk Drive, Redwood City, CA. 1978.

[BAN79] Banerjee, J., Hsiao, D. K., and Kannon, K., "DBC—A Data Base Computer for Very Large Data Bases," *IEEE Trans. on Computers*, Vol. C-28, No. 6, June 1979.

[BAU76] Baum, R. I. and Hsiao, D. K., "Data Base Computers—A Step towards Data Utilities," *IEEE Trans. on Computers*, Vol. C-25, No. 12, Dec. 1976.

[BER79] Berra, P. B. and Oliver, E., "The Role of Associative Array Processors in Data Base Machine Architecture," *IEEE Computer*, March 1979.

[CAS72] Casey, R. G., "Allocation of Copies of a File in an Information Network," AFIPS, SJCC, 1972.

[CHA78] Chang, H., "On Bubble Memories and Relational Data Base," 4th Int'l Conf. on Very Large Data Bases, Berlin, Sept. 1978.

[CHU69] Chu, W. W., "Multiple File Allocation in a Multiple Computer System," *IEEE Trans. on Comp.*, Vol. C-18, No. 10, Oct. 1969.

[COD70] Codd, E. F., "A Relational Model of Data for Large Shared Data Bases," *CACM*, Vol. 13, No. 6, June 1970.

[COM76] Special Issue in Data Base Management Models, *Computing Surveys*, Vol. 8, No. 1, March 1976.

[DEW79] DeWitt, D. J., "DIRECT—A Multiprocessor Organization for Supporting Relational Data Base Management Systems," *IEEE Trans. on Computers*, Vol. C-28, No. 6, June 1979.

[EPS78] Epstein, et al., "Distributed Query Processing in a Relational Data Base System," Report No. UCB/ERL M78/18, Electronics Research Laboratory, University of California, Berkeley, CA., 1978.

[HEV79] Hevner, A. G. and Yao, S. B., "Query Processing in Distributed Data Bases," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 3, May 1979.

[HOL56] Hollander, G. L., "Quasi-Random Access Memory Systems," *AFIPS Conf. Proc.*, EJCC, 1956.

[KER79] Kerr, D. S., "Data Base Machine with Large Content Addressable Blocks and Structural Information Processors," *Computer*, Vol. 12, No. 3, March 1979.

[LAN79] Langdon, Jr., G. G., "Data Base Machine, An Introduction," *IEEE Trans. on Computers*, Vol. C-28, No. 6, June 1979.

[LIN76] Lin, C. S., et al., "The Design of a Rotating Associative Memory for Relational Data Base Applications," *ACM Trans. on Data Base Systems*, Vol. 1, No. 1.

[LIP78] Lipovski, G. J., "Architectural Features of CASSM: A Context Addressed Segment Sequential Memory," *Proc. 5th Ann. Symp. on Comp. Arch.*, ACM-SIGARCH.

[LIT61] Little, J. D. C., "A Proof of the Queuing Formula, $L = \lambda w$," *Operations Research*, 9, 1961.

[MAH76] Mahmoud, S. and Riordon, J. S., "Optimal Allocation of Resources in Distributed Information Networks," *ACM Trans. on Data Base Systems*, Vol. 1, No. 1, March 1976.

[MAR68] Marshall, K. T., "Some Relationships between the Distributions of Waiting Time, Idle Time and Input/output Time in the GI/G/1 Queue," *SIAM Journal of App. Math.*, 16, 1968.

[MOR77] Morgan, H. L. and Levin, K. D., "Optimal Program and Data Locations in Computer Networks," *CACM*, Vol. 20, No. 5, May 1977.

[OZK77] Ozkarahan, E. A., et al., "Performance Evaluation of a Relational Associative Processor," *ACM Trans. on Data Base Systems*, Vol. 2, No. 2, June 1977.

[PAR72] Parhami, B., "A Highly Parallel Computing System for Information Retrieval," *AFIPS Conf. Proc.*, 1972, FJCC, Vol. 41, part II.

[RAM78] Ramamoorthy, C. V., Turner, J. L., and Wah, B. W., "A Design of a Cellular Associative Memory for Ordered Retrieval," *IEEE Trans. on Computers*, Vol. C-27, No. 9, Sept. 1978.

[SCH79] Schuster, S. A., et al., "RAP.2—An Associative Processor for Data Base and its Applications," *IEEE Trans. on Comp.*, Vol. C-28, No. 6, June 1979.

[SLO70] Slotnick, D. L., "Logic Per Track Devices," *Advances in Computers*, Academic Press, 1970.

[SU79] Su, S. Y. W., et al., "The Architectural Features and Implementation Techniques of the Multi-cell CASSM," *IEEE Trans. on Computers*, Vol. C-28, No. 6, June 1979.

[WAH79] Wah, B. W., "A Systematic Approach to the Management of Data in Distributed Data Bases," Ph.D. Dissertation, University of California, Berkeley, 1979.

[WON76] Wong, E. and Youssefi, K., "Decomposition—A Strategy for Query Processing," *ACM Trans. on Data-bases*, Vol. 1, No. 3, Sept. 1976.

[WON77] Wong, E., "Restructuring Dispersed Data from SDD-1: A System for Distributed Data Bases," Comp. Corp of America Tech. Rep. CCA-77-03, 1977.