DYNAMIC AND ORDER PRESERVING DATA PARTITIONING FOR DATABASE MACHINES

Esen A. OZKARAHAN **Mohamed OUKSEL**

Dept. of Computer Science, Arizona State University
Tempe, Arizona, 85287

ABSTRACT

The I/O bottleneck represents a major problem in architectures that have been proposed to implement hard database operations such as join and projection. It is recognized that solutions to this problem cannot be based on new database machine architectures alone if satisfactory performance goals are to be attained. A case in point is illustrated by the comparison of cellular/associative and in-stream pipeline based architectures. A methodology based on a global order preserving and dynamic partitioning is presented. The relevance of this approach to the solution of the I/O bottleneck problem is demonstrated through the efficient parallel processing of the join and projection operations. Finally, this methodology is incorporated into a specific database machine architecture; namely, the RAP.3 database machine. The partitioning strategy has been previously proven to be superior to the other known methods.

1.0 INTRODUCTION

Historically, the bottlenecks besetting the Von-Neumann architectures were primarily remedied by introducing parallelism and/or pipelining to support decentralized processing of both numeric and non-numeric data. However, the centralized processing nature of the architecture was not the only cause of bottlenecks. Indeed, other sources included the semantic gaps existing between the problems at hand and the architecture, which could not handle high level programming concepts, and the location based addressing. This latter problem resulted in the maintenance of various access paths to allow associative reference which could not be supported directly otherwise.

In the area of database management systems, the problems outlined above have finally led to development of database machines based on parallel architectures, the majority of which are associative. These machines provide high level data languages geared towards direct support of one or more data models, and because of their associativity, eliminate the access path problems. These features help close the semantic gap of the Von Neumann architecture.

The ongoing research in database machines has brought forth a variety of architectures and their performance assessments. The field has finally matured to realize the importance of the I/O bottleneck problem [Ozkarahan, 1983] which has so far plagued all the architectures which implement hard database operations such as the binary relational algebra operation of join and the unary operation projection. At first we have seen numerous proposals such as in-stream pipelined processing, multiprocessor architectures connected in a tree network which offered sublinear communication paths, and finally those based on the high density of VLSI. The VLSI designs found the rescue in "cramming" bits and/or circuits into a chip; and the higher the crammability the higher went the hopes for a solution.

After these efforts, the researchers have come to recognize that there is no simple architecture solution to the problem of mapping an arbitrary large memory space to a finite one (such as the memory of a database machine). In other words, we are realizing that efficient support of a virtual address space cannot come through the brute force approach of building larger and more clever architectures. Clearly, the problem lies elsewhere. Indeed, the effect of the I/O bottleneck problem can be reduced through the design of efficient partitioning of the data space in such a way as to exploit both clustering and parallel processing. In the rest of the paper we start first by comparing the basic database machine architectures and their relative performance in join. Then, we introduce a dynamic, order preserving partitioned file structure and show how one can exploit such an organization in database machines. Finally, we demonstrate how one can utilize this partitioning in database operations by demonstrating the processing of hard

operations of join and projection.

2.0 DATABASE MACHINE ARCHITECTURES

We can classify database machine architectures [Ozkarahan, 1985] into: a) cellular associative systems, b) multiprocessor based systems, c) systems with in-stream and/or pipeline processing, d) logic enhanced primary memory (VLSI) systems, and e) filters. While category (a) is typified by the RAP architecture, DIRECT [Dewitt, 1979] and DBC [Banerjee, Hsiao, and Kannan, 1979] are examples for category (b) (however, they belong to two different subcategories of tightly coupled homogeneous network of processors or functionally distributed processors, respectively). Examples for category (c) are DSDBM [Tanaka, 1984] and GRACE [Kitsuregawa, Tanaka, and Moto-Oka, 1984]. Systolic arrays [Kung and Lehmann, 1980] and highly concurrent tree machines are representative of category (d). Finally, filters are generally understood to be simple reduction (i.e., selection, may-be join filtering) processors that can be cheaply manufactured in a VLSI chip. ([Ozkarahan, 1985] can be consulted for the survey of database machines.) For our purposes here we will identify cellular/associative and in-stream and/or pipeline devices as the two extreme architectures and therefore make them the topic of our discussions. In the former, we have parallel processors (cells) that make up the database machine whereas in the latter the architecture is geared towards in-stream processing (e.g., searching, sorting) in pipeline. In the latter category although the sort and search engines can be connected in a network and operate concurrently, the entire purpose is to achieve I/O synchronous (e.g., as high as 3 Mbytes/sec channel speed) processing.

The argument in the in-stream architectures has been that they can bring down the join complexity of the nested-loop based architectures from $O(n^2)$, for n data items or tuples (i.e., relation cardinality) to a scale that is proportional to $n\log n$ by the virtue of sorting. At this point we can point out the following about join processing and database machines. Nested loop join processing and sort-merge based join (which can be the typical way how a join is processed in a uniprocessor) are the two distinct ways of performing join in database architectures. In a resident database, that is if the entire database can be kept in the database machine memory, the join complexity can be kept as low as O(n). This is due to associative/parallel search within the database machine. In nonresident databases however this complexity reaches $O(n^2)$ due to I/O bottleneck (i.e., the bottleneck caused by excessive and repetitive paging/staging of data into the database machine). This complexity cannot be blamed on the type of the database machine or the nested loop algorithm however. Despite the design of numerous architectures, the I/O bottleneck remains the major shortcoming. Before we talk about the sort-merge based join let us point out that the performance of join in the cellular/associative systems with resident database is superior to the sort-merge based join due to O(n) complexity of the former which benefits from parallelism and associativity. The sort-merge join, which is the typical way of doing a join in a uniprocessor, has a theoretical complexity of $c.n.\log n$ where the very large value of the constant c makes such a join infeasible. The in-stream and pipelined architectures help to alleviate this problem by utilizing a network of searchers and sorters that operate in pipeline. These architectures are also not immune to the I/O bottleneck problem because, at best, they have been demonstrated to keep pace with the channel speed. This speed is limited by the channel bandwidth, therefore, the performance of in-stream architectures is also limited unless further parallelism is exploited. We can quote the following relationships from [Ozkarahan, 1985]:

$$O\left(\frac{N}{B_{IO}} + t_u \cdot \log N\right) : O\left((t_{SCAN} + t_{IO}) \cdot \left\lceil \frac{N}{n} \right\rceil^2\right) \quad (1)$$

$$O\left\lceil \frac{\left(\frac{N}{B_{IO}} + t_u \cdot \log N\right)}{p} \right\rceil : O\left((t_{SCAN} + t_{IO}) \cdot \left\lceil \frac{N}{n} \right\rceil^2\right) \quad (2)$$

$$2 \cdot W_u \cdot S \cdot A_t \cdot R_{IO} \quad (3)$$

$$O\left\lceil \max\left\lceil \frac{N}{B_{IO} \cdot p}, \frac{t_u \cdot \log N}{p} \right\rceil \right\rceil : O\left\lceil \max\left\lceil t_{SCAN} \cdot \left\lceil \frac{N}{n} \right\rceil^2, t_{IO} \cdot \left\lceil \frac{N}{n} \right\rceil^2 \right\rceil \right\rceil \quad (4)$$

where
N = number of data items (i.e., tuples, relation cardinality)
n = number of processors (cells) in a cellular/associative device (i.e., database machine)
t_u = time to process one data item in a serial processor
t_{SCAN} = time to complete processing (i.e., one memory scan or cycle) in the cellular/associative device
t_{IO} = paging time to load data in the cellular/associative device
B_{IO} = channel bandwidth in the in-stream devices
p = number of parallel in-stream devices
R_{IO} = read/write time a unit of data, including seek time, in a secondary memory device such as disk
A_t = average number of attributes (on which fast access path structures are kept) updated in an operation
S = average number of values updated in an attribute update
W_u = proportion of updates with respect to overall transaction volume

In the above complexity expressions, expression (1) represents an in-stream architecture on the left and a nonresident cellular/associative architecture on the right for a join operation. As can be noticed, the cellular architecture is thrashing due to I/O bottleneck. The assumed join model is the nested-loop algorithm. In expression (2) we show the needed duplication of in-stream architecture to build parallelism so that the left side of the expression becomes less than or

equal to the cellular counterpart on the right. Expression (3) must be added to the previous complexity expressions if any one or both of the database machines represented by these expressions resort to access paths (as opposed to full associativity) in their operations. In other words for correct modeling of performance we should not ignore the expensive operation of access path maintenance. Expression (4) assumes architectural enhancement in the form of cache memory to provide staging to be overlapped with database machine processing. In this case, only the maximum of the overlapped quantities in both sides will determine the complexity.

As to the choice of architecture, the architect has to determine the cost of each architecture in view of the variables that will remain fixed in the above expressions and substituting alternatives for the remainder and computing the cost at the desired level of the performance relationship between the left and right sides.

In the following, we will present an order preserving dynamic space partitioning scheme that can enhance the in-stream and cellular architectures discussed above. More specifically, the partitioning to be presented will impact the values of p and $\frac{N}{n}$ for the respective architectures. The degree with which p or $\frac{N}{n}$ is effected will not be the same however. We will come back to this issue after we present our partitioning scheme.

3.0 DYNAMIC AND ORDER PRESERVING PARTITIONING

Various partitioning methodologies may be pursued to enhance the performance of database machines including semantic clustering, sorting, hash based filtering, hash based clustering, coarse indexing, in-stream filtering, file segmentation, and staging/paging that exploits locality of references [Ozkarahan, 1985]. Semantic clustering emphasizes the conceptual modeling task. At this level, the concern is about the choice of attributes composing a record type or the terms assigned to describe documents in unformatted databases. The other types of partitioning deal with the physical partitioning of the data space. Unlike other schemes proposed so far to deal with data partitioning the method presented and also referred to as *The Interpolation Based Grid File* [Ouksel, 1984a] is clearly a multidimensional partitioning structure in the strict sense which offers very efficient filtering, clustering, and global sorting capabilities. In varying degrees, it relates to extendible (dynamic) hashing, B-trees, dynamic multipaging, multidimensional B-trees, K-D-B trees, multi(or uni) dimensional linear and/or dynamic hashing, interpolation based index maintenance, and the grid file--refer to [Ouksel and Scheuermann, 1983] for references and detailed comparisons. However, the partitioning methodology described below offers significant advantages. Logically, it corresponds to a dynamically maintained and order preserving direct file organization. Some characteristics and advantages provided by this partitioning methodology are:(From this point on we shall refer to it as *DYOP Partitioning* which stands for Dynamic Order Preserving Partitioning.)

a) Search time is constant (typically two, one for the directory and one for the data file).
b) Directory and data file have identical structures and search characteristics.
c) No overflow buckets or chaining are needed.
d) Directory and data file partitions can be stored anywhere because their file addresses are kept in the directory (directories can expand into a multi level hierarchy).
e) No distributed free space needs to be kept in the directory and the data file.

The DYOP partitioning methodology is due to [Ouksel, 1985a], [Ouksel, 1985b]. The DYOP partitioning methodology is illustrated below by exhibiting the behavior of the data file and the directory through repeated insertions of data records.

3.1 DYOP Data File

The data file can be envisioned as an n-dimensional space where each dimension corresponds to an attribute A_i $0 \le i \le n-1$, n being the number of attributes (fields) in a tuple (record). This means that search space corresponding to the relation (file) will be the cartesian product of the domains underlying the attributes. For clarity we shall restrict our structure to the two dimensional case (i.e., $D_0 \times D_1$, not necessarily distinct, for the attributes A_0 and A_1). A record r will correspond to a vector $r = (v_0, v_1)$ in the search space where v_0, v_1 are the values taken from the respective domains D_0 and D_1. The data file is the set of partitions (or buckets) obtained through the repeated subdivision of the search space. Again, for simplicity, we shall assume a partition size of 2 records. Let us start with a data file containing only two records hence a single partition. This is shown in Figure 1(a) where the number of the only partition zero is indicated in the lower left corner. In the file the partitions are numbered in the order in which they are created. In the figure the axes correspond to domains and the superscripted domain variables (D_i^0) represent the current set of possible coordinates which are later used to build the directory. As will be shown later, the file system can be considered as a hierarchy of directories since the data file and the directory have the same structure. The data file is considered as the lowest level directory and thereby explaining the superscript zero.

Now if we insert a third record into the file an overflow will occur and the partition must be split along one of the dimensions (or inter-

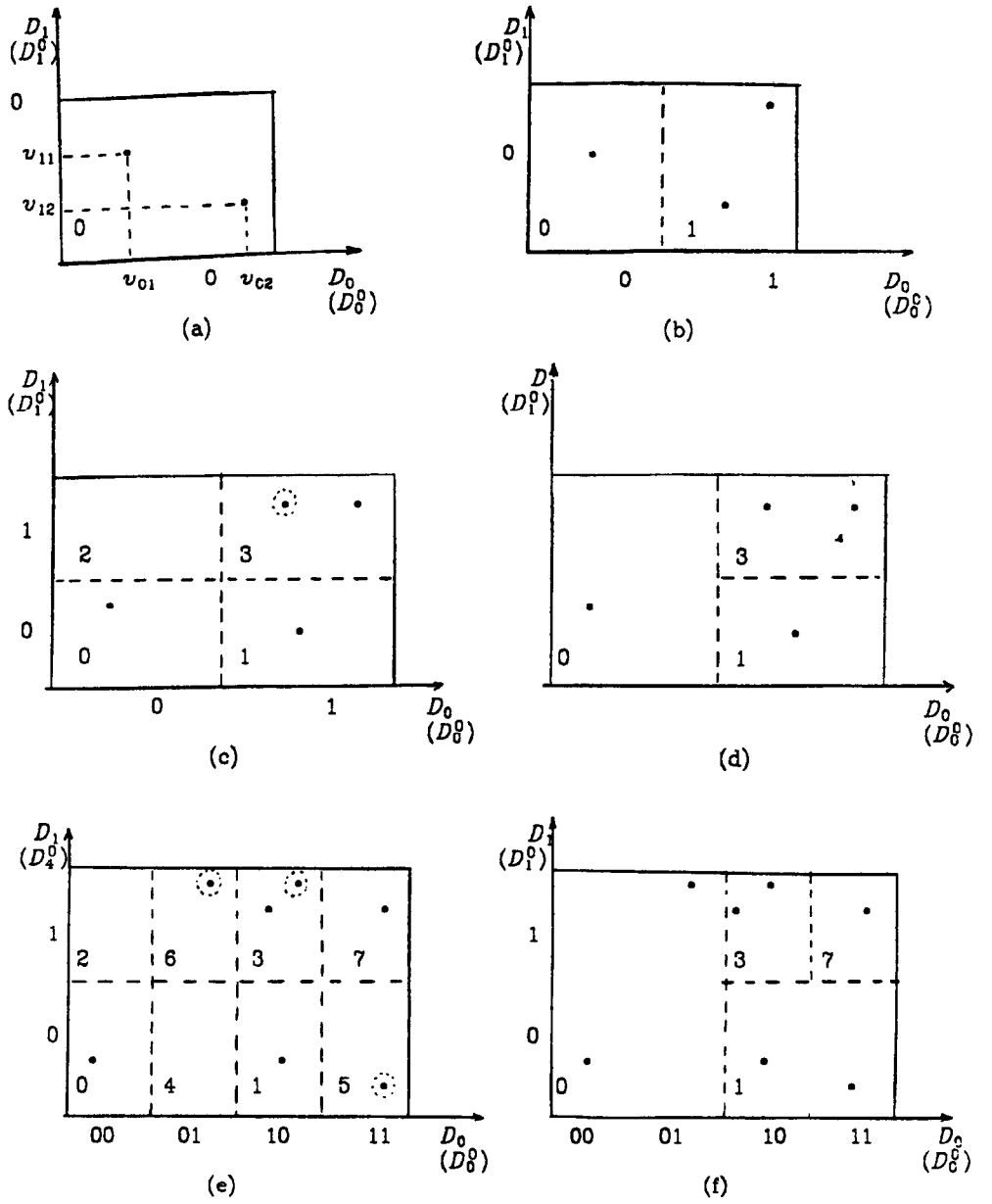


Figure 1. The data file

changeably, along one axis or coordinate). We shall adopt a cyclic order policy for choosing the splitting axis (others which permit a random choice are discussed in [Ouksel, 1983]). Hence, we will split partition #0 along D_0. This split will be made by halving the ordered range of D_0 values hence maintaining the linear order within each resulting half. At this point since a single partition corresponds to the entire range of domain D_0 the split will occur at $|D_0|/2$. Accordingly, all those records whose v_0 is less than $|D_0|/2$ will remain in partition #0 while all those $v_0 \geq |D_0|/2$ will be assigned to a new partition, partition #1. Figure 1(b) shows the split. The partitions are numbered 0 and 1 and so are the coordinates of the partitions (i.e., while they were both 0 in Figure 1(a), after the split, the D_0 subranges are numbered 0 and 1). As shown in Figure 1(c), another insertion into partition #1 will cause it to split, but this time along D_1 according to the cyclic order of split axes. We must emphasize here that a split of a given partition along a given axis triggers the implicit splits along the same axis of all the other partitions. This type of split is termed implicit because no physical splits occur. The purpose of this strategy is to provide logical uniformity that will permit the systematic numbering of partitions presented later. That is a unique mapping will exist between the coordinates of a partition and its assigned number. The split takes place in a linear order where all partitions are split in the order they are implicitly or explicitly created. Apart from their numbering however, there is no other effect on the implicitly split (i.e., unconcerned) partitions. They will only be identified as implicit partitions and stored in (i.e., physically

assigned to) a common explicit partition. This is referred to as the embedding of implicit partitions in explicit ones. While Figure 1(c) shows implicit splitting and linear numbering of implicit and explicit partitions, Figure 1(d) shows only the explicit partitions.

Consider adding a fifth record into partition #2. Partition #2 will still remain implicit because the insertion did not cause partition #0, in which partition #2 is embedded, to split. Another insertion into the same partition region will make partition #2 explicit. And this will happen without triggering another round of splits in the search space (i.e., it will only account for a previous split). This prevents unnecessary increase in the magnitude of coordinates which in turn keeps the directory simple and smaller. If we continue with insertions some partitions will require splitting, as is the case with partition #3 in Figure 1(e). In the cyclic order, this time the split occurs along D_0. However, the split will occur either at $|D_0|/4$ for those records whose $v_0 < |D_0|/2$ or at $3|D_0|/4$ for those records whose $v_0 \geq |D_0|/2$. Figure 1 (e) shows the linear partition splitting and numbering whereas Figure 1(f) shows only the explicit partitions. As can be seen in Figure 1(e), the split along a dimension is propagated to all the other partitions along the dimension even though only partition #3 has overflown. This creates the additional implicit partitions of 4, 5, and 6. Note also that the linear numbering of partition numbers is satisfied within the split axis as the major order and then within the other axis as the minor order (i.e., the order of partitions is 4, 5, and then 6).

In our partition assignment scheme we assign the lowest number to an explicit partition which may contain multiple implicit partitions (e.g., P_{i_2}, P_{i_3}, ..., P_{i_e}) so that the assignment can be shown as $P_{i_1} \leftarrow P_{i_2}, P_{i_3}, \ldots, P_{i_e}$ where $e \leq m$ and m is the total number of partitions in the data file. We say that the implicit partitions P_{i_2}, P_{i_3}, ..., P_{i_e} are embedded in the explicit partition P_{i_1}.

3.2 DYOP Directory

As we mentioned earlier D_0^0 and D_1^0 indicate the sets of coordinates along the axes of the data file. Accordingly, the vector $r^0=(d_0^0,d_1^0)$ represents the coordinates, in binary form, of partitions in the data file. For example, in Figure 1(f) (10,1) represents the two coordinates of partition #3 while the one associated with partition #0 is (00,0). The directory is made up of records which store coordinates of the data file partitions. The length of any d_i^0 is determined by the number of splits taken place along D_i^0. In the following we show the construction of the directory for the data file example illustrated above. As we already know the data and directory file structures are identical; however, since we can store more directory records in a partition (because directory records are small) the capacity of a directory partition will be large. But for clarity, we will assume it is 3. Accordingly, Figure 2(a) shows the initial make up of the directory. We see four points in the directory search space corresponding to the four explicit partitions in the data file. As can be noticed, the coordinates of the directory search space are labeled D_0^1 and D_1^1, superscript 1 indicating the first level directory (one higher than the data file at level 0). These coordinates constitute the basis for the 2nd level directory in the same way the first level directory is built from the data file. We stop building higher level directories when the number of partitions in the highest level directory is equal to one. Looking at Figure 2(a) we see that the directory partition #0 needs to be split because the partition size is 3. The split starts along the D_0^0 axis. The linear order in the range of the split axis must be preserved in the same way as in the data file. Therefore, when we split along D_0^0, those records whose coordinate (i.e., D_0^1) prefixes are a 1 (i.e., 10, 11 corresponding to higher values in the range) are moved to a new partition. This is illustrated in Figure 2(b) where the new partition is numbered 1.

According to our directory structure, the directory record (10, 1) which represents the data file partition #1 will itself be addressed as (1,0), i.e., directory partition #1, in the higher level directory. In general $r^h=(d_0^h, d_1^h, \ldots, d_{n-1}^h)$ represents the coordinate vector of the hth level where $0 \leq h \leq max h$ and $d_i^h=p(d_i^{h-1},l_i^h)$ for $0 \leq i \leq n$, where n is the number of dimensions, and

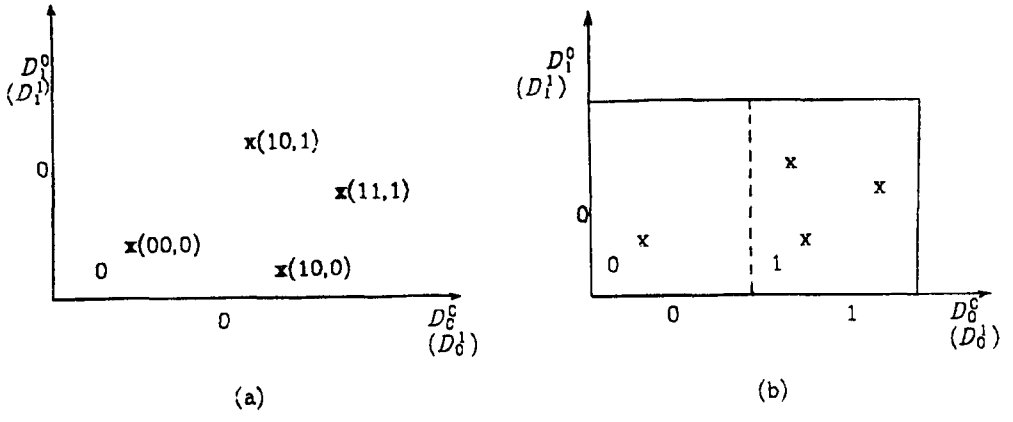


Figure 2. The directory

$1 \le l \le maxh$ where p(s,l) denotes the prefix of length l of binary string s. Note that no records are kept in the directory for the implicit data file partitions because as we will show later we can determine the explicit partitions embedding the implicit ones without additional directory accesses.

The DYOP partitioning scheme is inspired from the Interpolation Based Index Maintenance [Burkhard, 1983] or the Grid File [Nievergelt, Hinterberger, and Sevcik, 1984]. However, the structure that ultimately emerged is different. This is because DYOP combines the desirable properties of both and avoids their shortcomings. In the Interpolation-Based Index Maintenance no directory is needed and splits in the data file are delayed by adding overflow chains to file partitions. And when splitting becomes necessary, all partitions are split including those non-overflowing ones. In this scheme the space growth is worse than that of DYOP partitioning and furthermore overflow chaining deteriorates search time. In the grid file, the directory space requirement is at best a super-linear function and at worst an exponential function of the number of records. In DYOP however, the restriction to split only overflowing partitions guarantees a linear function. The differences between the DYOP partitioning and the other related organizations we have mentioned at the beginning are more fundamental and documented in detail in [Ouksel, 1983a]. It is important to note that unlike multikey structures such as K-D trees or K-D tries, DYOP is symmetric with respect to any of the attributes. Moreover, the directory structure has the same organizational properties as the data file; hence it also takes advantage of all the benefits.

3.3 Storage Addressing of DYOP Partitions

Due to the linear order of splitting of the implicit and explicit partitions in the DYOP files, we are able to deduce a mapping which allows a unique numbering of partitions. In the data file, a record r's ($r = (v_0, v_1, \ldots, v_{n-1})$) ith component v_i will have a relative position $x_i = v_i / |D_i|$ in the corresponding ordered set D_i. As we know, this record r will be represented in the directory by the coordinate vector $d^0 = (d_0^0, d_1^0, \ldots, d_{n-1}^0)$ which gives us the coordinates of the partition storing the data record r. These coordinates are given by :

$$d_i^0 = \left\lfloor x_i \cdot 2^{l_i^0} \right\rfloor \text{ for } 0 \le i \le n-1 \qquad (5)$$

To compute d_i^0, it suffices to know the number of splits l_i^0, occurred along the D_i^0 axis. This in turn can be determined as follows: Let l be the number of times the whole search space has been split. If the splitting order was cyclic then it can be expressed as $L^0 \cdot n + m$ implying that $l_i^0 = L^0 + 1$ splits occurred along axis i such that $0 \le i \le m$ and L^0 splits along axis i such that $m < i < n-1$. That is, at a given time we may not have completed the full round of splits along all the axes involved and some axes will remain unsplit for the current cycle. Let also d_{ij}^0 for $0 \le j \le l^i$ (where l^i also corresponds to the binary string length of the prefix of the ith coordinate) be the jth binary digit of d_i^0. Then the number of the partition storing the data record r can be obtained from :

$$M(r, l^0) = \sum_{i=0}^{n-1} 2^i \sum_{j=0}^{l_i^0 - 1} 2^{n(l_i^0 - 1 - j)} d_{ij}^0$$

$$\text{where } l_i^0 = \begin{cases} L^0 + 1 \text{ if } 0 \le i \le m \\ L^0 \text{ otherwise} \end{cases} \qquad (6)$$

The partition number M does not differentiate between explicit and implicit partitions. Because implicit partitions are not represented directly in the directory we must have a way of mapping coordinates of implicit partitions to the same directory partition representing the explicit partition in which the given implicit one is embedded. Only in this way can the directory search be resolved by a single access. This is accomplished by the use of the same M function. This time, it is used to find the directory partition which contains the coordinates of the explicit data file partition embedding the implicit target partitions. The following relationships have been shown to hold for the DYOP partitioning:

If Π represents a partition number then all the partition numbers for the partitions that may possibly contain partition Π is given by:

$$\Pi, \Pi - 2^{\alpha}, \cdots, \Pi - \sum_{s=0}^{\alpha} 2^s \Pi_s \quad \text{where } \alpha = \lfloor \log \Pi \rfloor \text{ and}$$

$\Pi_\alpha \Pi_{\alpha-1} \cdots \Pi_{\alpha_0}$ is the binary representation of Π

If we represent the coordinates of an implicit and explicit partition at the hth level directory, respectively, by r^{h_1} and r^{h_2}, then there exists and integer k such that:

$$p(r_i^{h_1}, l_i^{k+1}) = p(r_i^{h_2}, l_i^{k+1}) \quad \text{for } h < k < maxh \text{ and } 0 \le i \le n-1$$

This is a consequence of the previous relationship. The explicit partition number $M(r^{h_2}, l^h)$ is stored at the (h+1) th level directory whose partition number can also be obtained by r^{h_1}.

3.4 A Retrieval Example

Given a record $r = (v_0, v_1, \ldots, v_{n-1})$ and the number of times the entire search space has been split (i.e., $l = L.(n-1)+m$) we must determine the address a_0 where the record r is stored. To do that we must determine the following:

a) Compute r_0
b) Compute r^1, r^2, ..., r^{n-1}, stop when you obtain a directory consisting of a single partition (i.e., $r^i = (0,0)$) which corresponds to the top directory.
c) Search top directory partition whose number is M (r^{maxh-1}, l^{maxh-1}) in main memory to obtain the address a_{maxh-2} of the partition whose number is M (r^{maxh-2}, l^{maxh-2}) or the one it is embedded in.
d) Search lower level directories from k = maxh - 2 to 1. In each search, search the partition at the address a_k for the address associated with partition $M(r^{k-1}, l^{k-1})$ or the one it is embedded in.

The number of file accesses will be O(maxh-1).

Let us assume that we want to retrieve record r = (37500,10) and that D_0 = 50000, D_1=80, l^0=3, l^1=2, and l^2=0. Our search space is two dimensional, maxh=3, and the level 2 directory will be the top directory. And since l^2=0, meaning no split has occurred, the directory consists of one partition, it will be main memory resident. From v_0(37500) and v_1(10) we determine x=(37500/50000, 10/80) or (0.110 , 0.001) in the binary form. From this, by using expression (5) we determine r^0=(11,0). That is, because the zero-th level has been split 3 times then there will be two splits along D_0 and one along D_1 giving the respective powers of 2 in expression (5). Because the first level directory has been split twice i.e., once along each direction, we pick one digit from each of the coordinate prefixes of the lower directory to give r^1=(1,0) and similarly we find r^2=(0,0).

After this we determine the partition number $M(r^2,l^2)$ of the top level directory which comes out as M(0,0)=0. This number corresponds to a_2 in main memory. At this address we determine a_1, the address of the partition $M(r^1,l^1)$ or the partition embedding it. $M(r^1,l^1)$ evaluates to 1. In the lower level directory we search the partition at the address a_1 for the partition whose number is calculated to be $M(r^0,l^0)$=5, or the one it is embedded in. The address a_0 found at this point corresponds to the address of the data file partition in which the record r=(37500,10) is stored. The trees shown in Figure 3 abstract the splits that took place along the dimensions. In Figure 3(a) we see the splits in the lower directory and in Figure 3(b) we see the splits that took place in the data file. Each level in the trees corresponds to a split along an axis. If the splits, which preserve order, are labeled with 0 and 1 along the branches of an ordered binary tree, then the binary string formed by concatenating the branches along the path from the leaf associated with a partition to the root would give us partition number(s) we have determined by using M in expression (6). In the figures, those partitions are circled at the leaves and the paths are indicated in bold.

4.0 PARTITIONING FOR DATABASE MACHINES

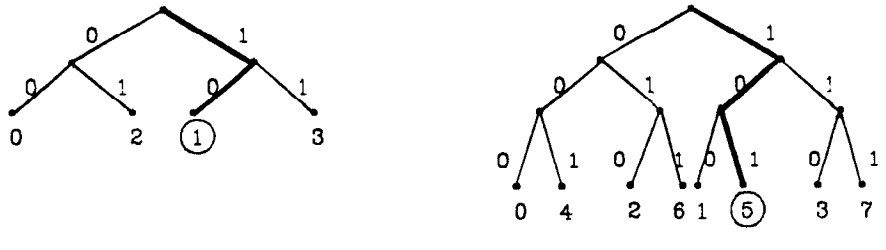
Although DYOP partitioning is used for direct addressed files, we can adapt it for space partitioning in database machines. The only parameter to vary will be the partition size which will be very large in the case of database machines. To give an example, consider RAP.3 database machine which is a cellular associative device. Each cell of RAP.3 machine has a cell memory capacity of 1 to 2 megabytes. Therefore each load of a 16 cell RAP.3 device will require database chunks of 16 to 32 megabytes. Each chunk will be a DYOP partition.

If we consider an in-stream architecture we need not make partition size that large. However, a small partition size will have adverse effect on data bandwidth provided by archive storage (e.g., disk(s)). In the case of a single disk drive, the larger the partition the higher will be the bandwidth because of continuous read out from contiguous disk locations. The smaller the partition the lower is the bandwidth because of frequently intervening disk seeks between partition accesses. To overcome that we may devise an interleaving and/or multiplexing scheme by using multiple disks. Therefore, the type of database machine architecture will determine the partitioning strategy; however, multiple disks will be preferable regardless of the type of architecture.

We can compare DYOP partitioning with the other similar methodologies. Specifically, if we take the partitioning by hash clustering used in the GRACE architecture the following comparisons can be made. First let us describe the GRACE method. Figure 4 shows this method.

As can be seen from the figure, in GRACE the data are not filtered but tagged with hash codes which scatter in space. Afterwards, compatible codes are gathered and distributed to various modules to absorb the nonuniform variation in the generation of hash codes.

In DYOP partitioning, because the global order is preserved the space reduction shown in Figure 4(a), which is a logical picture, is also preserved physically. This allows us to filter off the outer incompatible regions of the data space and therefore greatly alleviates the problems of insufficient bandwidth and/or memory space either directly or indirectly. More specifically, the $\left\lceil \frac{N}{n} \right\rceil^2$ complexity in expressions (1) through (4) will reduce down to $\left\lceil \frac{N}{n} \right\rceil$ due to the linearing effect of join space reduction and order preserving partitioning. In other words, because of order a partition will not be joined with incompatible partitions. We will show how to utilize DYOP partitioning in join



(a) (b)

Figure 3. Split directions and partition numbers

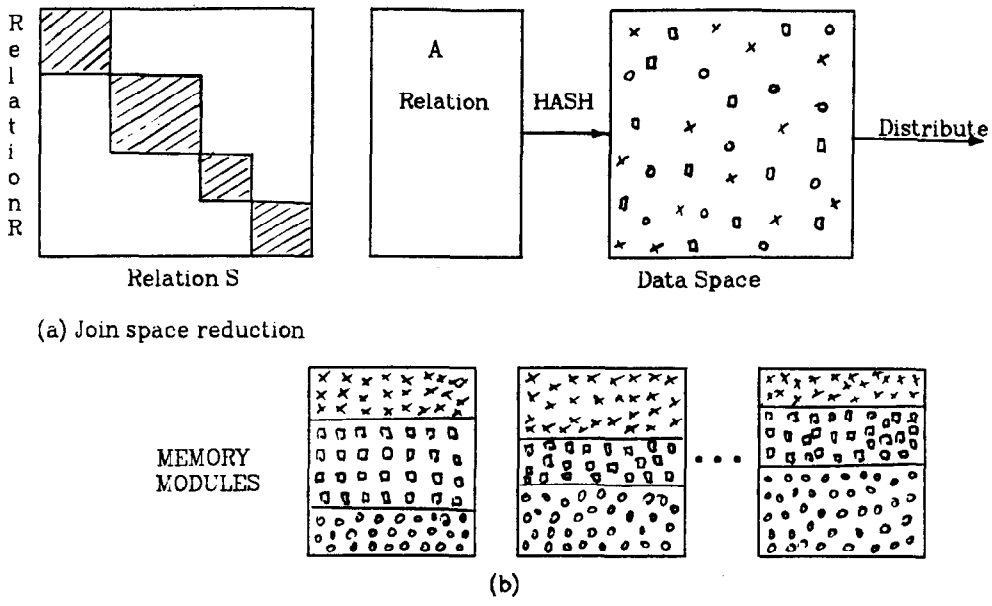


Figure 4. Hash based clustering in GRACE

and projection after we take a brief look at the RAP.3 architecture.

5.0 RAP.3 ARCHITECTURE

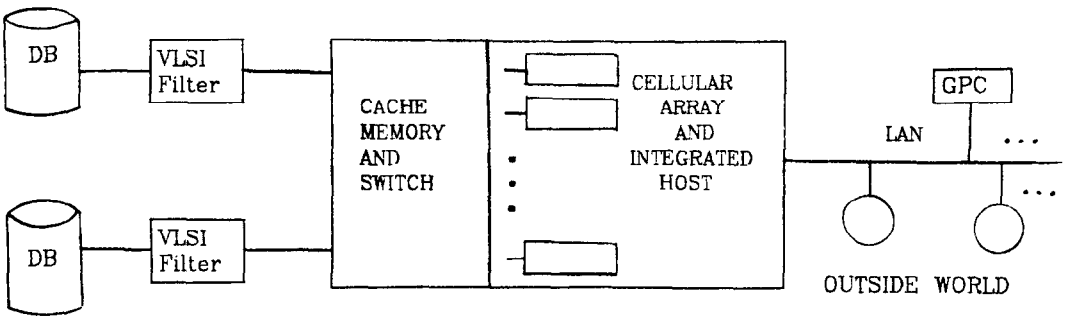
The RAP.3 database machine [Ozkarahan, 1982, 1985] has evolved from its predecessors which were called RAP.1 and RAP.2. Basically, RAP.3 adapted controllable memories (latest version of it used RAM's) to be used in the cell memory and mapped its cellular structure into a two dimensional parallelism. The result was to bring the parallelism of the cellular structure down to a modest value by compensating the decrease in cell parallelism with intra-cell parallelism. In other words, each cell is made up of parallel subcells where each subcell has the full functionality of a cell. This intra-cell parallelism enabled us to replace the specially hardwired logic with commercial microprocessors and firmware based query execution. The net effect was not a slow down, because of the two dimensional parallelism, but complete elimination of all possible rigidities and limitations imaginable with the RAP.1 or RAP.2 designs. Figure 5(a) shows the overall RAP.3 system architecture and Figure 5(b) shows the internals of a RAP.3 cell. The following concepts should be noted: The RAP.3 system does not believe in the "backend slave" concept, but rather believes in *GPN²C* (general purpose nonnumeric computer) operating among a network of computers. RAP.3 believes in indirect database search (because there is no alternative to it in real life) and therefore the need for space partitioning, in-stream filtering, overlapped (with processing) staging by the use of cache memories and processor-memory swap switch. That is, the cell processors must be able to switch between active and cache memories. While an active memory is processed, the cache must be staged in at the background and at the swapping point the roles of active and cache memories must be interchanged. In Figure 5(b), each subcell has enough local memory (8K bytes) to hold a tuple, the query code, and data such as the sorted contents of a batched transaction operands or source relation join attribute values in a semi-join (CROSS-MARK) operation. The DMA is a specially built hardware that eliminates I/O polling by the subcell microprocessors so that these processors either process their tuple contents or sit in a wait state until awakened by the DMA to continue processing (in actuality the wait state seldom occurs with the exception of start-up) The RAP.3 database machine has a universal instruction set and also "hardware macros" for some operations such as projection, cross-mark (semi-join) execution, and batched transaction processing. In the next section we will demonstrate the use of DYOP partitioning in conjunction with the RAP.3 architecture and its hardware macros for doing join and projection operations.

6.0 JOIN AND PROJECTION WITH PARTITIONING

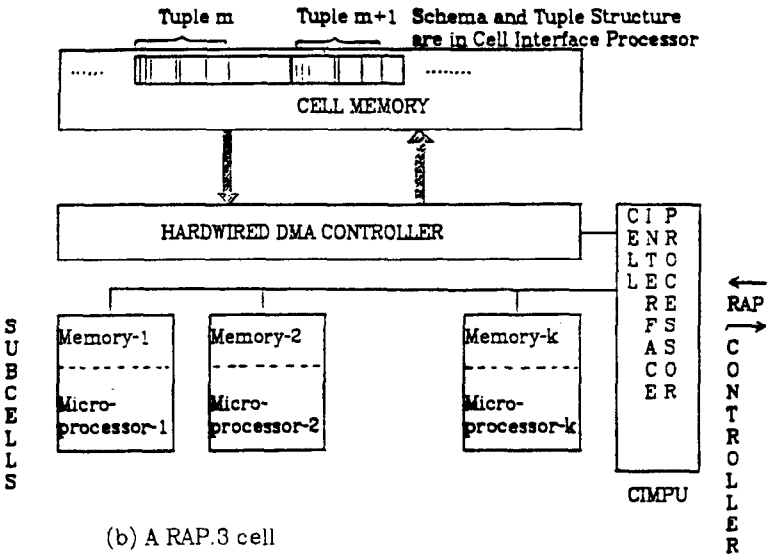
Below, we describe the use of DYOP to implement the join and projection operations. More elaborate algorithms concerning these binary relational database operations and others such as the inequality join and the m-way join are treated in [Ouksel, 1985b]. In the same study, the inherent parallel properties of the DYOP scheme is demonstrated through the design of parallel algorithms to execute these relational database operations independently of any specific architecture.

6.1 Equijoin

The join methodology we will describe here is general i.e., can be used both for semi-join and join. The difference comes in the way partitions are processed in the database machine. If we assume two relations R and S with cardinalities N and M to be joined and if b_0 represents partition size, then it is shown in [Ouksel, 1985a] that R and S will be mapped into approx-



(a) RAP.3 system architecture



(b) A RAP.3 cell

Figure 5. RAP.3 database machine

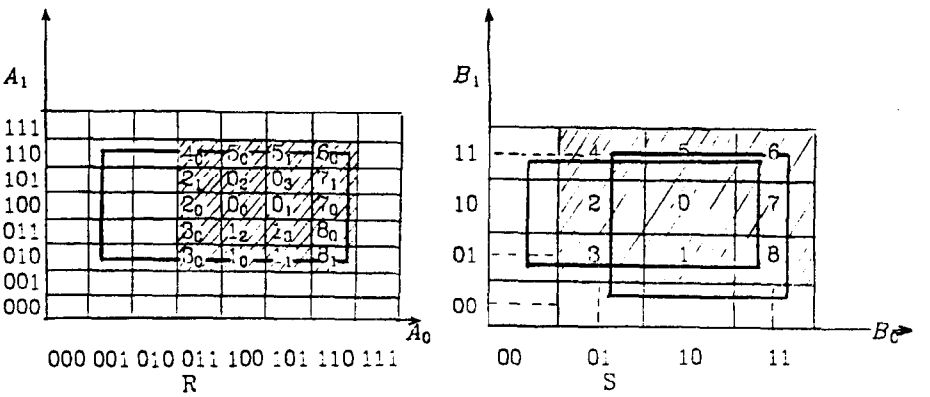


Figure 6. Join processing

imately E_R and E_S pages where

$$E_R = \frac{N}{b_0}\log e \quad \text{and} \quad E_S = \frac{M}{b_0}\log e \quad (e = 2.718)$$

Let us further assume that R and S were previously reduced by selection operations whose results are enclosed in the space delineated by the rectangles shown in Figure 6 for both R and S. The area enclosing the results of a selection is rectangular because the relations are assumed to be binary in this example and the key is composite. The area would have been a hypercube if the relations were composed of more than two attributes. In this example, both join and intersection are accomplished at once on the two attributes composing the relations. In the figure, rectangles in bold enclose the results of selections and the second narrow rectangle in the S relation indicates region of space that is compatible with that of the selected area in R. We should only compare partitions of compatible regions between the relations to do the join operation. In the following, we enumerate the possibilities in join processing.

CASE 1: A partition is fully contained in the selected region of the relation. An example to this is partition 0 in relation S and its corresponding (compatible) partitions O_0, O_1, O_2,

and O_3 in R. As can be noticed in the coordinates, partition 0 in S covers a larger region which is equivalent to the sum of the four partitions in R. The approach in join would be to:

a) Subdivide larger partition (here partition 0 in S) into smaller partitions that are equal in size to the size of partitions in the other relation with finer partition size (here R relation).

b) Send compatible partition pairs (i.e., one from R and one from S) to database machine to be joined. Here we assume that partition sizes are chosen such that they can both fit in the database machine memory. This join will be accomplished in one pass because DYOP partitioning preserves order so that compatible tuples will not be scattered in space.

c) Repeat the operation until all compatible partition pairs from the relations are sent to be joined in the database machine.

In (a) we mentioned subdividing partition 0 in S into $\{O_0^1, O_1^1, O_2^1, O_3^1\}$ to be joined with $\{O_0, O_1, O_2, O_3\}$ in R. Again the join will be between compatible pairs (i.e., $O_0^1 * O_0$, ..., $O_3^1 * O_3$). Here we will assume that the VLSI filter in the RAP.3 architecture will identify O_0^1 through O_3^1 in O in the data stream during staging by looking at the high order bits of attribute values and place them in their respective places in the cache memory prior to processing.

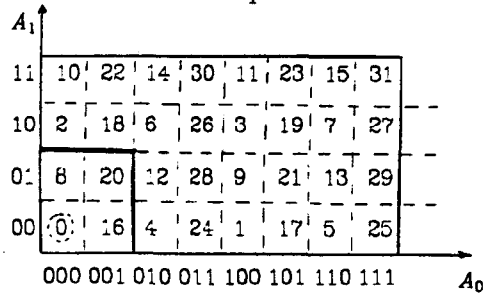
CASE 2: The partitions are not entirely contained in the selected regions of the relations. Example of this partition #1 in S which corresponds to partitions 1_0, 1_1, 1_2, and 1_3 in R. Here in both relations the tuples falling out of the selected regions in the partitions must first be filtered out before they are sent for join in the database machine. Again as indicated above, during staging, the VLSI filter can both decompose the larger partition #1 from S and filter out the irrelevant tuples from the decomposed partitions on the way to the cache, giving us first the set $\{1_0^1, 1_1^1, 1_2^1, 1_3^1\}$ and then $\{1_0^{11}, 1_1^{11}, 1_2^{11}, 1_3^{11}\}$ on the fly. The single primes indicate decomposition while double primes indicate subsequent filtering that are both accomplished by the VLSI filter. Therefore, the join will be between the pairs: $1_0 * 1_0^{11}$, $1_1 * 1_1^{11}$, $1_2 * 1_2^{11}$, and $1_3 * 1_3^{11}$.

At the end of each join dispatch, the RAP.3 machine will perform its parallel join according to its algorithm, described in the related references, which will not be repeated here. However, we will discuss the RAP.3 parallel projection macro following the projection example due to its natural compatibility with DYOP partitioning.

6.2 Projection

Projection operation requires sorted relation for efficient processing. It should be noted that while DYOP partitioning preserves global order among partitions in the multidimensional search space, data within a partition are not sorted. Also in cases where implicit partitions are embedded in an explicit partition the explicit partition will be unordered both within and between implicit partitions. Consider Figure 7. If we want values of A_0 ordered within A_1, then pulling out explicit partition #0, indicated by bold rectangle, will fall out of sequence because we should scan coordinate 00 of A_1 along A_0 first before we can pull coordinate 01 (i.e., implicit partitions 8, and 20). This means we need sorting within a partition whether or not it contains implicit partitions. There is no difference in the case of having implicit partitions, however, simply because an explicit partition's size is fixed (that is the partition has not grown enough to be split).

Let us show projection by referring to Figure 7. Projection on A_0 of R means that the duplicates must be searched in the following groupings:
{0,8,2,10}, {16,20,18,22}, ..., {25,29,27,31}
The addresses in each group are computed by using the two nested loops:



R

Figure 7. Projection of R(A_0, A_1) on A_0

one varying along axis A_0

the other varying along axis A_1 (within A_0)

Due to global order (i.e., intersection of groups is empty relative to A_0) projection can be processed in parallel among groups. Also, projection can be processed in parallel within a group if partitions are processed as single units. The dispatching of partitions of a group to the RAP.3 cells (or to the cache first) will be done by the VLSI filter by examining the higher order bits of the attribute values on the fly.

The following algorithm summarizes the RAP.3 projection hardware macro:

Algorithm Project

(a)

(1) Pointer sort (i.e., without moving tuples) the tuples in cell memory with respect to the attribute to be projected. This sort is performed by the CIMPU and takes place in parallel among the cells.

(2) Eliminate duplicates in the sorted list of (a.1) by marking the tuples with unique attribute values directly by the CIMPU in cell memory.

(b) c <-- # of relation cells

Repeat

(1) Pick the first unprocessed cell and read

into the controller the sorted attribute values taken out of the marked tuples in cell memory.

(2) The controller writes these values in cell interface processor memories of all the remaining cells that store the relation, simultaneously at each iteration.

(3) By a merge-like operation between the sorted attribute values in cell memory and the sorted values in cell interface processor memory, all remaining cells compute the set difference of *{current cell values}* - *{values input by the controller}* simultaneously. This is done by resetting the mark bits of the tuples containing values identical to those of the subtrahend.

(4) c <-- c-1

Until c=0

At the end of each iteration (1) through (4), the processed cell at step (1) contains unique attribute values within the relation. And at the end, all the cells are left with marked tuples corresponding to unique attribute values.

6.3 More General Join Cases

As indicated earlier, an n-degree relation would correspond to an n-dimensional hypercube in the DYOP search space. A join of the type $R[A_0 = B_1]S$ between relations $R(A_0,A_1,\ldots,A_{n-1})$ and $S(B_0,B_1,\ldots,B_{m-1})$ would be processed by partitions fetched in the value order, as shown in the foregoing. In other words, as in projection, for each partition along A_0(and B_1) before we get to a subsequent partition we would scan an $A_0(B_1)$ partition along the remaining n-1(m-1) coordinates. This can be interpreted as processing the join between two compatible subhypercubes of degree n-1(m-1) from the respective relations. Having a filter on the data path (or filters on multiple paths) enables us to dispatch variable sizes of data from the compatible subhypercubes as dictated by available processor (cell) memory in the cell partitions. Such an MIMD architecture consisting of cell clusters or partitions each processing a join dispatch is included in the RAP.3 architecture (Figure 5). This enables parallel join processing both within a join and among multiple joins. The model of join processing discussed here would benefit from m-way joins because the sizes of compatible subhypercubes would conveniently be reduced to the intersection of m data spaces.

7. CONCLUSION

We have argued that an efficient solution to the I/O bottleneck problem– which results in binary and projection operation complexity of $O(n^2)$ in relational databases--cannot be easily found if one reduces the problem to that of designing yet another architecture. The solution lies in efficient, dynamic, and order preserving data space partitioning techniques such as the DYOP. It has been shown that DYOP partitioning is superior to that of the GRACE architecture and can exploit parallelism. An example of compatible marriage between the DYOP partitioning and the RAP.3 architecture is demonstrated through the join and projection.

Once such a partitioning strategy is chosen we can concentrate on the choice of a specific database machine architecture based on the knowledge of our archival storage system (i.e., number of disks and channels). Because then we will able to know, in detail, our bandwidth and partition sizes (actually this design is an iterative process between the two ends). The knowledge of these parameters will enable us to choose the desired architecture. A guideline to weigh the alternatives is discussed at the beginning of our article.

REFERENCES

BANARJEE, J., HSIAO, D.K., KANNAN, K. [1979] *DBC-A Database Computer for Very Large Databases,"* **IEEE Transactions on Computers,** Vol. C-28, No. 6, pp. 414-429.

BURKHARD, W. A. [1983]. *Interpolation-Based Index Maintenance*, **Proc. of ACM SIGMOD-SIGACT Symposium**, pp. 76-85.

DEWITT, D. J., [1979]. *DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems*, **IEEE Transactions on Computers,** Vol. C-28, No. 6, pp. 395-406.

KITSUREGAWA, M., TANAKA, H., MOTO-OKA, T. [1983]. *Application of Hash to Database Machine and its Architecture*, **New Generation Computing,** Vol. 1, No. 1, pp. 63-74.

KUNG, H. T., LEHMAN, P. L. [1980]. *Systolic (VLSI) Arrays for Relational Database Operations* , **Proc. of ACM SIGMOD Conf.,** pp. 105-116.

NIEVERGELT, J., HINTERBERGER, J., SEVCIK, K. C. [1984]. *The Grid File: An Adaptable, Symmetric Multikey File Structure*, **ACM Transactions on Database Systems,** 9,No.1, pp.38- 71.

OUKSEL, M. [1983a]. *Order-Preserving Dynamic Hashing Schemes for Associative Searching in Database Systems*, Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern University, Illinois.

OUKSEL, M., SCHEUERMANN, P. [1983b]. *Storage Mappings for Multidimensional Linear Dynamic Hashing*, **Proc. of ACM SIGMOD-SIGACT Symposium**, pp. 90-105.

OUKSEL, M. [1985a]. *The Interpolation-Based Grid File* **Proc. of ACM SIGMOD-SIGACT Symposium,** pp. 20-27.

OUKSEL, M. [1985b]. *Data Structures and Parallel Algorithms for the Execution of Relational Database Operations*, in preparation.

OZKARAHAN, E. A. [1982]. *Implementations of the Relational Associative Processor (RAP) and its System Configurations*, Dept. of Computer Science, TR82-005, Arizona State University.

OZKARAHAN, E. A. [1983]. *Desirable Functionalities of Database Architectures*, **Proc. of IFIP83 World Congress,** pp. 357-362.

OZKARAHAN, E. [1985.11] **Database Machines and Database Management,** Prentice-Hall Inc., Englewood Cliffs, N. J.

TANAKA, Y. [1983]. *A Data-Stream Database Machine with Large Capacity*, in **Advanced Database Machine Architectures,** Ed. D. K. HSIAO, Prentice-Hall Inc., Englewood Cliffs, N. J., pp. 168-202.