Multiprocessor Algorithms for Relational-Database Operators on Hypercube Systems

Ophir Frieder
George Mason University

As databases expand and applications become more diverse, demands on computational engines supporting database processing increase. With appropriate algorithms, commercially available hypercube systems can meet the demands.

Exploiting parallelism in database processing has been a research goal since the early 1970s. Originally, special-purpose architectures were developed to provide the computational and input/output bandwidth needed for database processing. More recently, many researchers have relied on commercial multiprocessor and local area networked systems by such vendors as Sequent, Tandem, BBN, DEC, and various hypercube system manufacturers to improve database processing performance.

This tutorial focuses on hypercube interconnected architectures as a computational engine for relational-database processing. Like other architectures with distributed memory and resources ("shared-nothing" architectures[1]), hypercube systems can support the high I/O bandwidth required for database processing. However, unlike the other architectures, hypercubes are scalable to thousands of nodes. For example, NCube Corporation currently manufactures hypercubes comprising up to 8,192 nodes. These engines can provide large-scale concurrency for both interquery and intraquery processing[2] and are well suited for such computationally intensive processing as protocol verification using database technology.[3]

After reviewing hypercube systems, this tutorial briefly highlights several implementations of the many currently available hypercube systems and comments on their potential performance in evaluating relational database operators. All algorithms assume that the relevant data are memory resident.

The hypercube multicomputer

A hypercube graph is an n-dimensional Boolean cube Q_n defined as a cross product of the complete graph K_2 and the $(n-1)$-dimensional Boolean cube Q_{n-1}, with $Q_1 = K_2$. In an architecture based on hypercube interconnection, each node is connected

0018-9162/90/1100-0013$01.00 © 1990 IEEE

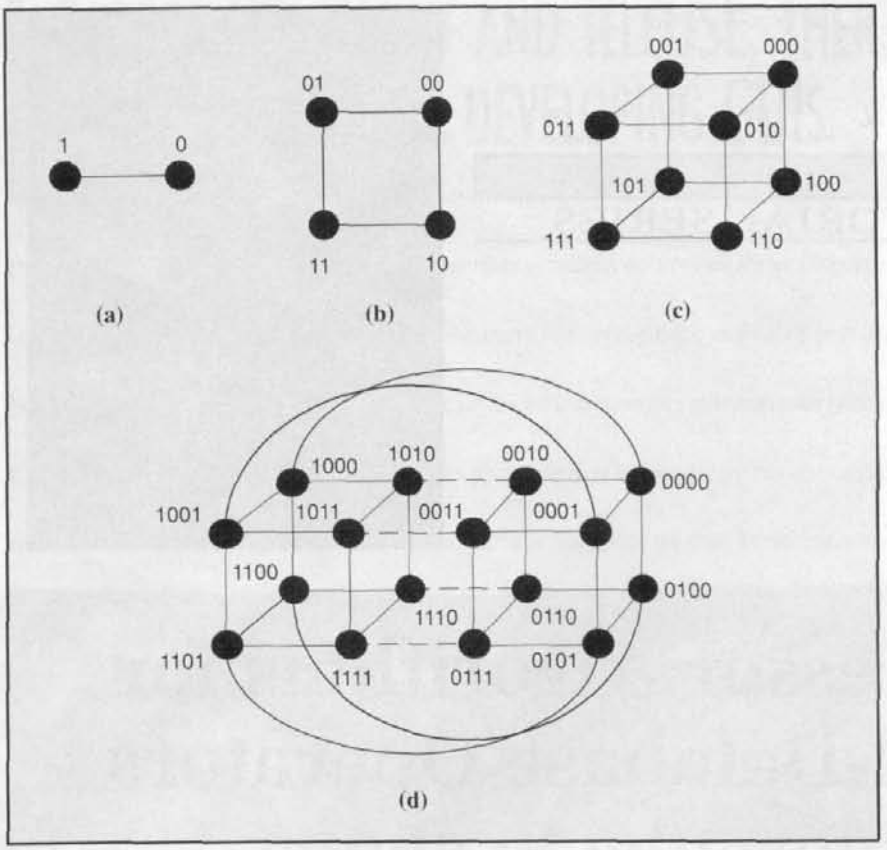


Figure 1. Various hypercube configurations: (a) one-dimensional, (b) two-dimensional, (c) three-dimensional, (d) four-dimensional.

(or adjacent) to each of its $n = \log_2 N$ neighbors, where N is the number of nodes. For example, in a four-dimensional cube Q_4, node 0000 is adjacent to nodes 0001, 0010, 0100, and 1000. Figures 1a through 1d illustrate the communication paths of one-dimensional (two-node), two-dimensional (four-node), three-dimensional (eight-node), and four-dimensional (16-node) hypercube systems. Note that each system consists of $N = 2^n$ nodes, with n being the cubical dimension of the system. We assume that the node address bits are numbered 0 to n-1, with the leftmost bit (bit 0) being the most significant. Existing hypercube machines include Caltech's Cosmic Cube,[4] Intel's iPSC/2, and NCube's NCube/10.

Nodes communicate by sending messages in packets. Packet size varies, but protocol imposes a maximum. Packets, which in database processing contain tuples, can be sent between any two nodes in the system, possibly being routed through intermediate nodes. In current hypercube systems, typical internode communication times are on the order of microseconds.[5]

Synchronization among nodes can be achieved either through hardware support or strictly through software. In software synchronization, a receiving node "blocks" until a message arrives. Thus, the arrival of a message synchronizes the two nodes. This blocking send/receive technique can be generalized to synchronize all nodes within the system. A possible synchronization algorithm for hypercube architectures is based on a message-sending ordering technique called recursive halving, which I will describe in a later section.

Other hypercube systems use global hardware lines to synchronize the processors. A global wired-AND or wired-OR line is connected to a number of nodes, and each node has a local input. The global line value is the logical AND'ing or OR'ing of the local inputs. Synchronization is maintained by monitoring the changes in the global line value. For example, if a "global AND" line is used, all nodes maintain their local input value at false throughout the execution of their local task. A global line value of true implies that all nodes terminated local execution. Similarly, wired-OR lines can be substituted for wired-AND lines by reversing the local line values. The algorithms presented in this article assume the availability of global synchronization lines. These lines need not be supported by hardware but may be provided logically by software.

Other scalable "shared-nothing" architectures include mesh-based multicomputers and ring-based local area networks. Each hypercube node requires more communication ports than the nodes making up meshes and rings, but the hypercube interconnection significantly reduces the maximal communications diameter as compared with meshes and rings. The maximal communications diameter in a hypercube comprising N nodes is $\log_2 N$ as compared with $N/2$ and $\sqrt{N}$ in similarly sized rings and meshes, respectively.

Hypercube architectures as database engines

Database processing requires vast I/O and data-access bandwidth and significant computational resources. Hypercube systems provide all three. By horizontally partitioning relations (see sidebar on pp. 16-17), across a parallel I/O structure like the disk-per-node I/O subsystem in the Intel iPSC/2, a relation can be read or written in parallel, if appropriate synchronization primitives are available. Thus, currently available hypercube systems can remedy the known I/O bottleneck of database processing.

The degree of I/O parallelism depends on the partitioning scheme and the data values relevant to the query. Common horizontal partitioning techniques include random, round-robin, hash-based, range-based, and user-specified distribution of tuples across the sites. Both the hash-based and the range-based partitioning approaches allocate the tuples according to the hashed or actual value of a set of specified attributes. A round-robin partitioning scheme evenly distributes the tuples across the sites.

The Gamma Project[6] is a hypercube database engine initially developed for a ring-based multicomputer. The system currently runs on an Intel iPSC/2 hypercube comprising 32 80386-based nodes with a disk drive per node. Gamma exploits the available parallel I/O capability by horizontally partitioning the relations across the disk drives. The relational-database operator algorithms presented in this tutorial differ from those developed for the Gamma project[7] in that they are not independent from interconnection

topology, and hence are actually optimized for the hypercube interconnection topology.

Database performance is enhanced whenever the variance in the data distribution across the various processing sites is low. Poorly distributed relations result from individual database operators that favor a particular value, for example, a selection. Independent of the data organization, data skew is likely to result during the processing of some user queries. For example, consider the pathological case of a 50-processor system on which a population database is partitioned according to states: A range-based horizontal partitioning scheme based on the STATE attribute is used, resulting in each processor accessing data for only a single state. A query interested in information about only the state of Michigan yields data at only a single processor. Without redistributing the data, 49 of 50 processors remain idle for the duration of the query.

Dynamic data redistribution (on-the-fly data reorganization) preceding each multiscan operator in a query tree has been proposed to guarantee a near-even workload across the processors. A *multiscan operator* is any operator in which the processing of an individual tuple involves comparing its attribute value(s) against other tuples. For example, the Select is not a multiscan operator since the relevance of the tuple is independent of any other tuples. However, both the Join and Project are multiscan operators because the relevance of each tuple depends on the values of other tuples: unique value in the case of Project, and a similar value in the other joining relation in the case of Join (see sidebar).

Baru and Frieder[8] have demonstrated the reduction in time resulting from executing a nested-loop Join on a hypercube instead of a special-purpose, bus-based architecture. The study also demonstrated that data redistribution on a hypercube system as part of the Join can be achieved with low overhead. A 16-node hypercube system using dynamic data redistribution achieved roughly a 15- to 80-percent reduction in the processing times of various Join computations. The exact savings depended on the degree of skew in the data distribution.

As databases expand in size and applications using databases become more diverse, the computational engines supporting database processing must likewise continue to improve. Currently, hypercube systems comprising more than 8,000 processors are available commercially.

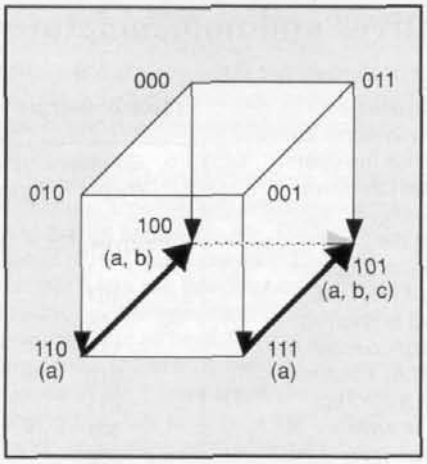


Figure 2. Aggregation example.

Thus, for at least the near future, hypercube systems meet the computational demands of database processing.

Hypercube database algorithms: Uniscan operations

A query tree is a partially ordered sequence of operators initiating at the leaves, with the result of the query obtained on termination of root execution. The output relation of the child operator is the input relation of the parent operator. The relations accessed by the leaf nodes are called base relations and typically are physically stored in the database. All computed relations, except for the final output, are called *intermediate relations*.

The relational operators can be viewed as being in one of two categories: *uniscan* and *multiscan* operators. Both Join and Project are examples of a multiscan operators since each tuple, in turn, is compared against a set of tuples. However, both Select and the aggregation operators are uniscan operators because the processing of each tuple is independent of the processing of any other tuple.

The Select operator. In optimized query trees, the Select operators are typically located near the leaf levels of the query tree. Thus, in a selection on a horizontally partitioned relation, it is common to assume a *uniform* distribution of tuples across the nodes. As Select is a uniscan operator, each tuple can be processed independently. Hence, computing a Select in parallel requires each node to read its resident tuple set and, for each tuple read, compare the tuple attribute value against the desired value. If the node detects a match, it keeps the tuple.

Aggregation operators. Scalar aggregation operators are an extension to the relational algebra and include such uniscan operators as Max, Min, Count, and Average. In parallel systems, aggregation is generally performed in two phases. In the first phase, each node computes its local aggregate value. Tuples are accessed as described for the Select operator. In the second phase, the global aggregate value is computed by combining all the local values at the final destination or target node.

On a hypercube the global aggregation phase, phase 2, takes n steps and is based on a common technique called recursive halving. As the communications diameter of the hypercube is n, n steps represent the minimal number of steps required for the scalar aggregation, since relevant data can reside on any node in the cube. In the kth step, $k=0$ to $n-1$, nodes whose leftmost k address bits equal the leftmost k bits of the target address receive the intermediate aggregate value from the nodes that differ in address from the target in the kth bit.

Figure 2 illustrates an eight-node cube where node 5 (101) is the target node. Target node designation is specified in the query. In step a ($k = 0$), each of the nodes 0 (000), 1 (001), 2 (010), and 3 (011) sends its value to nodes 4 (100), 5 (101), 6 (110), and 7 (111), respectively. The receiving nodes, indicated with the letter a, compute the new aggregate values. These new values are used in the second step, step b ($k = 1$), where nodes 6 (110) and 7 (111) send their values to nodes 4 (100) and 5 (101), respectively. Receiving nodes are marked by the letter b. Once again, the receiving nodes compute the new values. Finally, in step c ($k = 2$), node 4 (100) sends its value to node 5 (101), the target node marked by the letter c, which computes the final result.

Sometimes the user may wish to aggregate values by categories. For example, in a population database for the United States, a user may want the average age of the population in each state. In such cases, the aggregation is performed by repartitioning the data according to attribute values (cat-

(Continued on p. 18)

Relational database primitives and nomenclature

Many database-management systems are available today. A majority of the most recently developed systems are based on the relational-database model, in which the operators available to the user operate on relational structures. An attribute is any symbol from a finite set $\mathfrak{L} = \{A_0, A_1, A_2, ..., A_n\}$. A relation $\Re$ on the set $\mathfrak{L}$ is a subset of the Cartesian product of dom $(A_0) \times$ dom $(A_1) \times$ dom $(A_2) \times ... \times$ dom (A_n), where dom (A_i) is the domain of A_i. $R[A_0\ A_1\ A_2\ ...\ A_n]$ represents $\Re$ on the set $\{A_0, A_1, A_2, ..., A_n\}$ and is referred to as the schema of $\Re$. In $R[A_0\ A_1\ A_2\ ...\ A_n]$, each column A_i is called an *attribute* of R, and is denoted as $R.A_i$. Each row of R, namely a *tuple*, is designated by $<a_0, a_1, a_2, ..., a_n>$, where $a_i \in$ dom (A_i). The value of attribute A_i of tuple $x \in R$ is denoted as $x[A_i]$. Similarly, if tuple $x \in R$, then $x[W]$ is the value of the attributes of attribute set W in tuple x.

For illustration, assume the relations shown in Table A. Relation EHW has three attributes — EHW.Employee_No., EHW.Height, and EHW.Weight — while relation EA has only two attributes — EA.Employee_No. and EA.Age. Each relation consists of 16 tuples.

Three of the more common operators in the relational model include the Select, Project, and Join. These three operators are formally defined as follows:

Select – The selection on R[XYZ], denoted as $\sigma_{A=a}$ (R), is defined by

$$\sigma_{A=a}(R) = \{x \mid x[A] = a,\ x \in R\},$$

where A is an attribute of R.

Project – The projection on R[XYZ], denoted as π_A (R), is defined by

$$\pi_A(R) = \{x[A] \mid x \in R\},$$

where A is a set of attributes of R.

Join – The Join of two relations R[XYZ] and S[VWX], denoted as

R[XYZ] |×| S[VWX], is defined by
R[XYZ] |×| S[VWX] = $\{x \mid x[VWX] \in S$ and $x[XYZ] \in R\}$,

where V, W, X, Y, and Z are a disjoint set of attributes. If no common joining attributes exists, the Join of R and S, is the Cartesian product of R and S.

Using the above relations,

List all the employees who are 72 inches tall.

E72W [Employee_No. Height Weight]
= $\sigma_{EHW.Height = 72}$ (EHW) =

Selecton example:

Employee_No.	Height	Weight
101	72	195
303	72	180
801	72	187

Table A. Sample relations EHW and EA.

Employee Height and Weight Relation (EHW)			Employee Age Relation (EA)	
Employee_No.	Height	Weight	Employee_No.	Age
101	72	195	101	31
106	69	141	106	26
115	70	182	115	40
210	64	108	210	25
211	74	185	211	45
301	68	172	301	37
302	71	201	302	52
303	72	180	303	34
304	70	165	304	43
454	62	180	454	35
531	64	125	531	29
640	73	212	640	32
801	72	187	801	55
802	71	198	802	33
803	73	170	803	28
804	67	210	804	34

Enumerate all unique heights of the employees.
UH [Height] = $\pi_{EHW.Height}$ (EHW) =

Projection example:

Height
62
64
67
68
69
70
71
72
73
74

Find the weight and age of all the employees who are 72 inches tall.

E72W [Employee_No. Height Weight] |×| EA [Employee_No. Age] =

A Join example:

Employee_No.	Height	Weight	Age
101	72	195	31
303	72	180	34
801	72	187	55

In the project operator, only one <72> tuple exists. That is, the duplicate <72> tuples generated were eliminated.

A common extension to the relational database is the aggregation operation. The operator, denoted by $agg_{f(X)}(R)$, computes a global aggregate function on attribute set X of the relation R. Ex-

amples of an aggregate function, $f(x)$, include Sum, Max, Min, and Average. For example, the total sum of the EHW.Height is

$$agg_{sum(EHW.Height)}(EHW) = 1,112.$$

Note that the above is not equivalent to the sum of $\pi_{P.B}(P)$,

$$agg_{sum}(\pi_{EHW.Height}(EHW)) = 690.$$

Additional operators such as Theta-Join, Union, Difference, and Renaming have also been incorporated in many existing database-management systems. For a complete treatise on the relational-database model, see D. Maier, *The Theory of Relational Databases* (Computer Science Press, Rockville, Md., 1983).

Database Partitioning

In distributed and parallel database systems, two common relation-distribution schemes are horizontal and vertical partitioning. A relation $R[A_0\ A_1\ A_2\ ...\ A_n]$ is horizontally partitioned across multiple sites S_i, $0 \le i \le k$, denoted $H_{i,R[A_0\ A_1\ A_2\ ...\ A_n]}$, $0 \le i \le k$, if $x \in H_{i,R[A_0\ A_1\ A_2\ ...\ A_n]} \rightarrow x \in R[A_0\ A_1\ A_2\ ...\ A_n]$ and $U_i\ (H_{i,R[A_0\ A_1\ A_2\ ...\ A_n]}) = R[A_0\ A_1\ A_2\ ...\ A_n]$. Informally, the tuples of R are partitioned so each site contains a possibly empty subset of the tuples of the original relation. A relation $R[A_0, A_1, A_2, ..., A_n]$ is vertically partitioned across multiple sites S_i, $0 \le i \le k$, denoted $V_{i,R[X_i]}$, $0 \le i \le k$, if $\{A_0, A_1, A_2, ..., A_n\} \supset X_i$, $\pi_{X_i}(R) = V_{i,R[X_i]}$, and $V_{0,R[X_0]} |\times| V_{1,R[X_1]} |\times| ... |\times| V_{k,R[X_k]} = R[A_0\ A_1\ A_2\ ...\ A_n]$. Simply stated, vertical partitioning distributes the relation by attributes so each site contains a number of attributes from the base relation. The Join of the partial relations at the individual sites yields the original relation: The Join composition is lossless.

The data-partitioning scheme significantly affects the total query-processing time. If an application requires that at each site only tuples whose attribute values fall within a specified range are present, but that the entire contents of the resident tuples be available, then horizontal partitioning should be used. Horizontally partitioning the relation reduces the volume of data that must be processed at each site without incurring any interprocessor communication. If vertical partitioning is used, additional Joins would be required to obtain the nonresident attribute values. The additional Joins result in both added interprocessor communication and increased computation.

If an application requires at each site only a subset of the tuple's attributes, then vertical partitioning should be used. Vertical partitioning of the relation reduces the volume of resident data and makes unnecessary the projection of the desired attributes from the original relation. Horizontal partitioning requires a projection on the union of all the local data sets and results in additional interprocess communication.

To see the advantages of each relation-partitioning scheme, assume that a company has an Employee_No., Height, Weight (EHW) database as in Table A and that it has one geographically distant division. Further assume that the employee numbers of all employees at site 1 range from 100 to 499, whereas all employees at site 2 have employee numbers ranging from 500 to 899. As all the information concerning each employee is required at the employee's local site, horizontal and not vertical partitioning of the EHW relation is appropriate. Table B shows a logical horizontal partitioning of the EHW.

Now consider the hypothetical situation that at each site there are freight elevators with maximum weight limitations and storage rooms with low ceilings. For each employee, only weight or height, respectively, is needed to determine whether he or she can join the people or cargo already in the elevator or enter the room. In this case, each local database should be vertically partitioned, as shown inTable C.

This simple example demonstrates that a composition of both horizontal and vertical partitioning can reduce the processing demands.

Table B. A logical horizontal partitioning of the EHW.

$H_{1,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Height	Weight
101	72	195
106	69	141
115	70	182
210	64	108
211	74	185
301	68	172
302	71	201
303	72	180
304	70	165
454	62	180

$H_{2,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Height	Weight
531	64	125
640	73	212
801	72	187
802	71	198
803	73	170
804	67	210

Table C. A logical vertical partitioning of the horizontal partitioning of EHW.

$V_1,H_{1,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Height
101	72
106	69
115	70
210	64
211	74
301	68
302	71
303	72
304	70
454	62

$V_2,H_{1,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Weight
101	195
106	141
115	182
210	108
211	185
301	172
302	201
303	180
304	165
454	180

$V_1,H_{2,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Height
531	64
640	73
801	72
802	71
803	73
804	67

$V_2,H_{2,EHW[Employee\_No.\ Height\ Weight]}$

Employee_No.	Weight
531	125
640	212
801	187
802	198
803	170
804	210

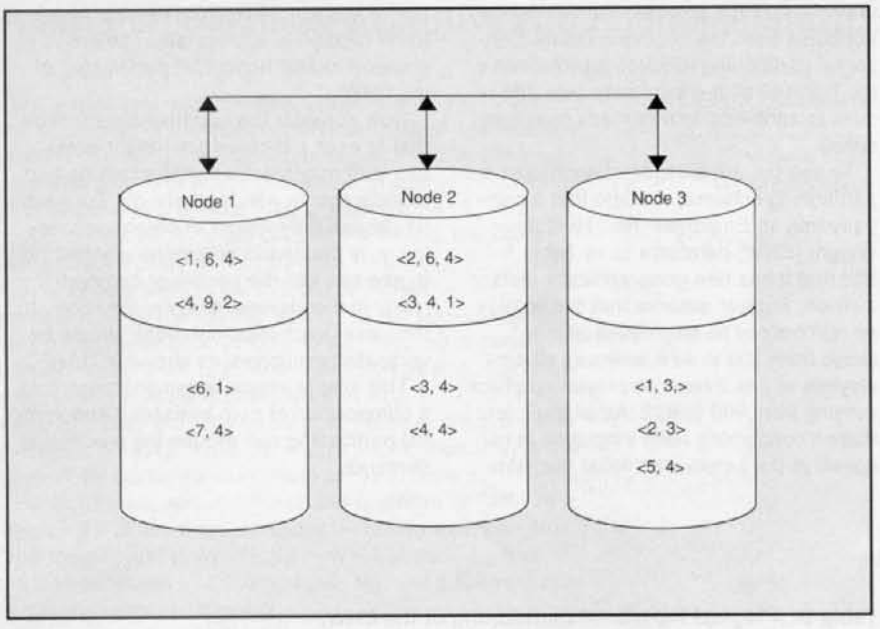


Figure 3. Database partitioning for a multiprocessor Join example.

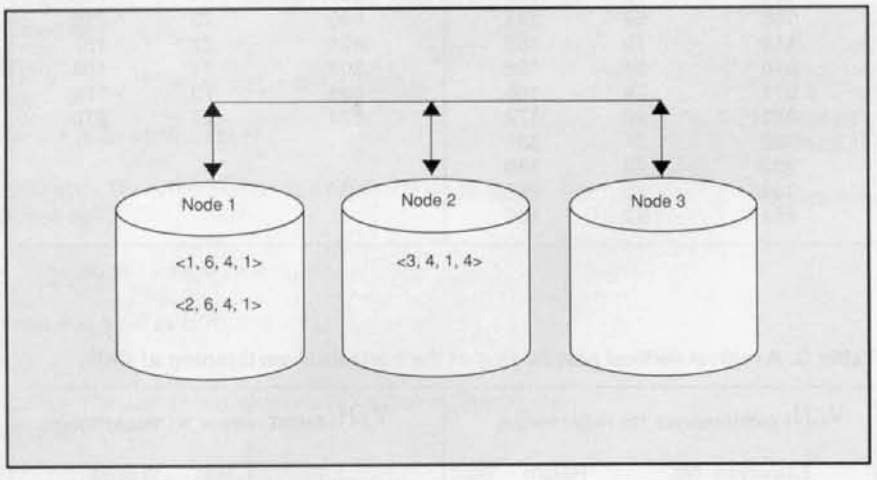


Figure 4. Broadcast-based Join result.

egories), with each node computing the aggregate for the resident categories. I discuss data redistribution according to attribute values in later sections.

Multiscan database operators

Multiprocessor implementations of the multiscan database operators can be classified into two main categories: broadcast-based and bucket-based. The first category, broadcast-based, requires that each node, in turn, broadcast its portion of the smaller relation, R_1 (in the case of a two-relation operator), or the relation at hand (unirelational operator), to all the system nodes. All nodes receive the broadcast message (tuple set) and perform the appropriate local computation, which involves the resident tuples and the received packet.

Bucket-based solutions include implementations that rely on sorting and/or hashing techniques. (Numerous extensions and modifications to the basic bucket approach described here appear in the literature.[7,9,10]) We characterize any operator-processing approach as bucket-based if the approach partitions all the data elements involved in the operation into buckets according to their attribute values. Each bucket corresponds to a range of attribute values, and only tuples consisting of attribute values within the given range reside in the bucket.

This approach has an advantage over the broadcast-based approach: Only tuples that are likely to match are compared with one another. However, this approach suffers from the need to repartition both relations for a multirelational operator. Furthermore, when one relation is significantly larger than the other, the communication requirements of the bucket approach result in greater processing time than a broadcast-based Join. The time involved in partitioning both relations instead of only the smaller relation is greater than the savings obtained by eliminating redundant comparisons.

An example illustrates both approaches. Consider a Join of two relations P and T on attribute B, R[ABCD] = P[ABC] |x| T[BD], on a three-node multiprocessor where the relations are partitioned as shown in Figure 3. In the figure, P is shown as the top relation. Node 1 contains four tuples, two of P and two of T. Node 2 contains two P tuples and two T tuples, while node 3 has only three T tuples. Clearly the P relation consists of fewer tuples; hence, we refer to it as the smaller relation.

A broadcast-based Join algorithm proceeds as follows. Since P is the smaller relation, each node in turn broadcasts each tuple of P to all other nodes. Thus, node 1 will broadcast <1, 6, 4> and <4, 9, 2>, node 2 will broadcast <2, 6, 4> and <3, 4, 1>, and node 3 will not broadcast any tuples. End-of-tuple transmission is indicated by a default-value broadcast. All nodes monitor the transmission and compute the local Join of the broadcasted tuple value and their local T tuples. Figure 4 illustrates the resulting relation.

Continuing with the example, initially the maximal possible joining attribute(s) range is computed. This range consists of actual attribute values and not necessarily the domain of the joining attribute(s). For example, the range of B of the "combined" relation, R, is 1 to 9, designated attr-ran(R.B) = [1,9]. However, dom(B) may comprise a much greater range, say the set of natural numbers. A tighter bound on maximal attr-ran(R.B) is the intersection of the regions attr-ran(P.B) = [4,9] and attr-ran(T.B) = [1,7], namely maximal attr-ran(B) = [4,7].

Once the attr-ran[R.B] is computed, the corresponding disjoint attribute range is assigned to each processor. Assigning each processor an attribute range is accomplished

either statically or dynamically. In a system using static partitioning, the range attr-ran(B) is partitioned equally across the processors. Such a partitioning scheme results in a skewed processor workload whenever the distribution of attribute values is biased.

A dynamic partitioning scheme nullifies the effects of a skewed tuple distribution. Randomly sampling a small number of the joining tuples as part of the previous operation provides a crude histogram of the data distribution and hence of the processor workload distribution. Using this information, the range attr-ran(R.B) is partitioned to yield a near-optimal workload partitioning of the Join processing across the processors. However, like all dynamic load-balancing algorithms, dynamic sampling introduces some overhead. The precise observed reduction in processing time depends on the skewness of the data and various machine-specific parameters, for example, communications overhead.

The example presented here uses static partitioning. Nodes 1, 2, and 3 are assigned B attribute value ranges 4-5, 6, and 7, respectively. All tuples of both relations are redistributed on the basis of their B attribute value to reside in the proper bucket. When the redistribution terminates, all processors compute the local Join of their P and T tuples. Figure 5 illustrates the resulting tuple distribution for the bucket Join method. Comparison with Figure 4 shows that both schemes produce the identical relation; however, the relation-partitioning differs.

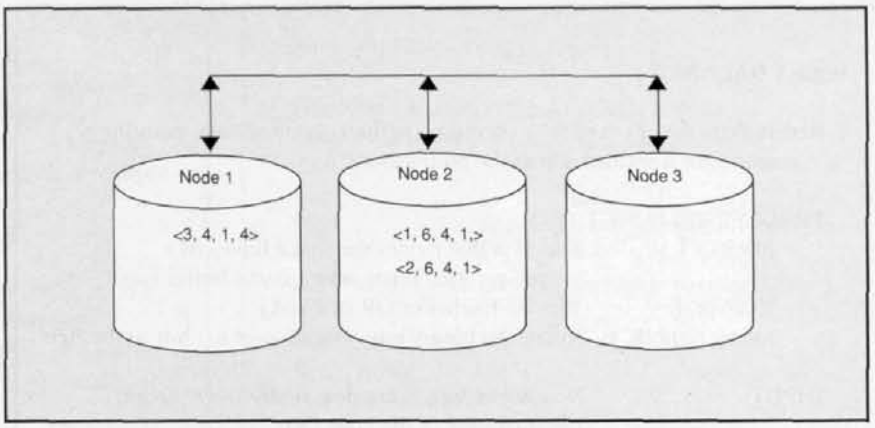


Figure 5. Bucket-based Join result.

The Join operator: A broadcast-based solution

Because broadcast-based solutions reduce communication costs (transmit less data) at the expense of redundant comparisons, ensuring full processor use is vital. Baru and Frieder[8] describe a three-stage, broadcast-based Join that guarantees a nearly even processor workload. First, a dynamic data-redistribution algorithm referred to as *balancing* ensures an even distribution of input tuples. Incorporating the balancing step results in a minimal increase in communication costs but roughly a 35-percent reduction in the overall Join processing time. Once they are evenly distributed, the tuples are sorted.

In the second stage, the Relation Compaction and Replication (RCR) stage replicates the smaller relation R_1, originally stored in a cube of dimension n, so it is replicated in each of the two equal-sized, dimension n–1, logical-cube partitions of the original cube. This primitive's goal is to increase the number of tuples from R_1 stored at each node until the volume of R_1 tuples present at each node is the size of one packet, or until R_1 has been fully replicated at each node. It ensures that packets used in the cycling phase are as full as possible, and that the packet-formation overhead per tuple is minimized for the cycling primitive.

Finally, the cycling primitive sends the tuples of the smaller relation around in a ring. Local Joins of the resident tuples and the pipelined circulated relation are performed simultaneously at each node. Often the design of each node incorporates dual buffers, so the broadcast of packet i+1 can overlap the local computation of packet i.

Assuming multiple independent communications processors, the balancing of two relations simultaneously proceeds as follows. Initially, the local tuple counts of the relations R_1 and R_2 are computed. During each step j ($0 \le j \le n-1$), the nodes whose addresses differ in the jth bit exchange their local-relation R_1 tuple count. The node with the greater number of tuples (if any) then sends the difference of the average tuple count and its own tuple count of tuples to its paired neighbor. Simultaneously, the nodes whose addresses differ in the $((j+1) \bmod n)$th bit balance R_2. The local tuple counts for R_1 and for R_2 are then recomputed, and the next step is initiated. After n steps, all nodes contain roughly the same number of R_1 and R_2 tuples. A more complex balancing algorithm that incorporates the sending-node number can guarantee a difference of one tuple per node, per relation. Once balancing is complete, both relations are sorted locally.

Figure 6 provides a simplified pseudocode description of the single-relation balancing algorithm. Several instructions require explanation. The *send(addr, data)* and *receive(addr, data)* send or receive from nodes whose address is *addr* the tuples designated by the set *data*. *Rotate_right(num, count)* rotates *num*'s binary representation *count* bits to the right. Similarly the *rotate_left(num, count)* instruction in the RCR primitive (Figure 7) rotates the *num*'s binary representation *count* bits to the left. For example, using a four-bit representation of 1 (0001), rotate_left (1, 1) = 2, rotate_right (1, 1) = 8, and rotate_left (1, 2) = rotate_right (1, 2) = 4. Finally, Global AND (*GAND*) is the global synchronization line value obtained by AND'ing all the local synchronization line values. The *ABS*, *XOR*, and *CEIL* instructions are the mathematical absolute value, exclusive OR, and ceiling functions, respectively. The node address bits are numbered 0 to n–1.

The second stage of the broadcast Join, relation compaction and replication, proceeds as follows. Initially the local tuple count of the smaller relation, say R_1, is computed. During each step j ($0 \le j \le n-1$), the nodes whose addresses differ in the jth bit exchange their local R_1 tuple count. RCR is possible if the combined volume (in bytes) of R_1 tuples in each pairing of nodes does not exceed the maximal size of a single packet. When possible, all nodes transmit their tuples to their paired neighbors. While maintaining a sorted order, the local and received tuples are merged, and the new

```
begin { BALANCE }

{ Redistribute data dynamically according to the volume of data, ensuring a
    nearly even workload across the processors.

  PREDEFINED FUNCTIONS:
      FIRST ( k )          Function that moves the first k tuples of a
                             prespecified relation to a stated buffer (send_set)
      XOR (k, j)           Bitwise Exclusive OR of k and j
      rotate_right (k, j)  Rotate the binary representation of k, j bits to the right

  INPUT:                   Number of locally resident tuples (own_count)
                           Local address of the node (X)

  PRIOR TO EXECUTION:      Relation randomly distributed across nodes

  EXECUTION TERMINATION:   Evenly distributed relation across nodes   }

  for j := 0 to n – 1 do begin
    send ( XOR( X, rotate_right ( 1, j+1 ) ), own_count );
    receive ( XOR( X, rotate_right ( 1, j+1 ) ), neighbor_count );

{ Simultaneously equate the tuples across each node pair.      }
    if ABS ( own_count - neighbor_count ) > 1 then begin
        avr := CEIL ( [ own_count + neighbor_count ] / 2 );

        if own_count > neighbor_count then begin
          send_set := FIRST ( own_count – avr );
          send ( XOR( X, rotate_right ( 1, j+1 ) ), send_set );

          loc_tuple_set := loc_tuple_set - sent_set;
          own_count := own_count - | send_set |;
        end
        else begin
          receive ( XOR( X, rotate_right ( 1, j+1 ) ), receive_set );

          loc_tuple_set := loc_tuple_set + receive_set;
          own_count := own_count + | receive_set |;
        end
        end

  end { for }
end { BALANCE }
```

Figure 6. Data-balancing pseudocode.

tuple count is computed. RCR continues until either n RCR steps have occurred or the single-packet limitation is exceeded. Each successful RCR step results in the duplication of the smaller relation. After j successful RCR steps, a copy of R_1 is resident in each of $2^j(n-j)$ dimensional hypercubes. Figure 7 shows a pseudocode description of the RCR stage.

The final stage, called *cycling*, creates a Hamiltonian cycle within each logical cube partition generated by the RCR primitive, and pipelines the data packets throughout the cycle. Only those address bits not used in the RCR step are considered when forming the cycle. All nodes within each formed cycle have the identical bit values in the *iteration* rightmost address bits. A Hamiltonian cycle can be dynamically generated with reflexive Gray codes, as shown in Figure 8. Executing the pseudocode results in a reflexive Gray code ring in Send_Array. Given any node number, say stored at Send_Array [i], its left and right ring neighbors are the values at Send_Array [$i-1$] and Send_Array [$i+1$], respectively.

We now work through an example. Consider two relations partitioned over an eight-node hypercube, $N = 8$ and $n = 3$. Figure 9 shows the initial distribution of the 24-tuple relation R_2 (upper portion of each bin) and 16-tuple relation R_1 (lower portion of each bin). Initially, the balancing stage is used, resulting in a uniform distribution of both R_2 and R_1. As shown in Figure 10, in steps 1, 2, and 3 a maximum of two, one, and one tuples, respectively, are transferred in a neighbor pairing. The figure illustrates two relations being balanced simultaneously. Assuming a maximum packet limitation of four tuples — because there are two R_1 tuples after balancing at each node — a single RCR step is possible. After the RCR step, two full copies of the smaller relation exist. The first copy of R_1 is partitioned over nodes 0 (000), 2 (010), 4 (100), and 6 (110), while the second copy spans nodes 1 (001), 3 (011), 5 (101), and 7 (111). Note that a sorted order is maintained. Finally, a Hamiltonian cycle is generated in each of the smaller cubes — (000→ 010 → 110 → 100 → 000) and (001 → 011 → 111 → 101 → 001) — and the compacted smaller relation is circulated in both cubes simultaneously. Figure 11 gives the resulting relation.

The Join operator: A bucket-based solution

Nearest neighbor pairing algorithms — for example, the balancing algorithm presented in Figure 6 — can implement bucket-based Joins. Initially all nodes sort their local tuple set of both relations based on the joining attribute values. In bucket-based Join algorithms, instead of computing the average number of tuples between each pair of nodes, at every node-pairing step j ($0 \le j \le n-1$), each node partitions its tuple set into two. The first set S_1 consists of all tuples whose attribute values are assigned to hypercube nodes whose jth bit address is 0. All remaining tuples are placed in S_2. With a sorted tuple set, partitioning the tuples into two sets only requires finding the boundary condition.

During each step j ($0 \le j \le n-1$), nodes

```
begin { RELATION COMPACTION REPLICATION }

{ Replicates a relation partitioned across the nodes of
  Q_n into each Q_{n-1}. Via replication, the number of
  nodes required for the later stage, cycling, is halved.
  Replication continues until packet limitation is reached.

  PREDEFINED FUNCTIONS:
    FIRST ( k )          Function that moves the first k tuples
                           of a prespecified relation to a
                           stated buffer (send_set)
    XOR (k, j)           Bitwise Exclusive OR of k and j
    SYNCH (GAND)         Global AND'ing of all local
                           feeds (GAND_loc)
    rotate_left (k, j)   Rotate the binary representation
                           of k, j bits to the left

  INPUT:                 Local address of the node (X)

  PRIOR TO
  EXECUTION:             Relation distributed across all
                           nodes in Q_n

  EXECUTION
  TERMINATION:           2^iteration copies of the relation, each
                         partitioned over a disjoint Q_{n-iteration} }
  GAND_loc := true;
  iteration := 0;

  SYNCH( GAND );

  While GAND and ( iteration ≤ n – 1 ) do begin

  { Inform neighbor of own count }
    f_send ( XOR( X, rotate_left ( 1, iteration ) ),
                       own_count );
    f_receive ( XOR( X, rotate_left ( 1, iteration ) ),
                       neighbor_count );

  { Determine if compaction / replication is possible     }
                       if (own_count + neighbor_count) >
                       MAX_TUPLES_IN_PACKET
                       then GAND_loc := false;

    SYNCH( GAND );

    if GAND then begin { step is possible }

      send ( XOR( X, rotate_left ( 1, iteration ) ),
        tuple_set );
      receive ( XOR( X, rotate_left ( 1, iteration ) ),
        receive_set );

      iteration := iteration + 1;
      tuple_set := tuple_set ∪ receive_set;
      own_count := own_count + | receive_set |;
        end
      end { step possible }
  end { while }
end { RELATION COMPACTION REPLICATION }
```

Figure 7. Pseudocode for relation compaction replication.

```
begin { GENERATE CYCLE }

{ Generate a routing table that yields a reflexive
  Gray code in each of the Q_{n-iteration} that resulted
  from the RCR step.

  INPUT:          Cube dimension (dimension_cube)

  OUTPUT:         Reflexive Gray code routing table }

  Send_Array [ 0 ] := 0;
  Send_Array [ 1 ] := 1;
  curr_num_nodes := 2;

  For dimen := 2 to dimension_cube do begin

    curr_num_nodes := curr_num_nodes × 2;
    j := -1 ;

    For i := ( curr_num_nodes / 2 ) to
      curr_num_nodes – 1 do begin

      j := j + 2;
      Send_Array [ i ] := Send_Array [ i –j ]
          + curr_num_nodes / 2;
    end;
  end;
end; { GENERATE CYCLE }
```

Figure 8. Cycle-generation pseudocode.

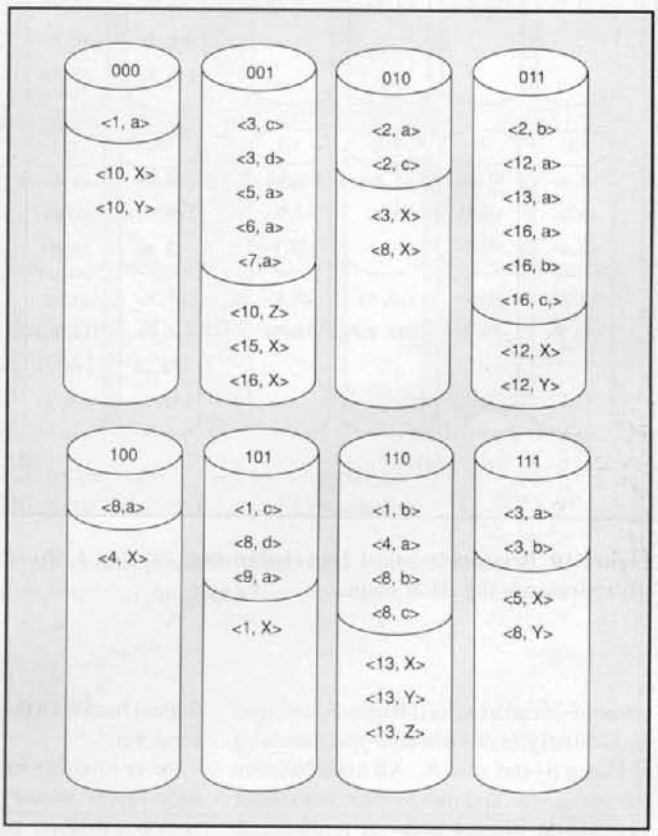


Figure 9. Initial data distribution.

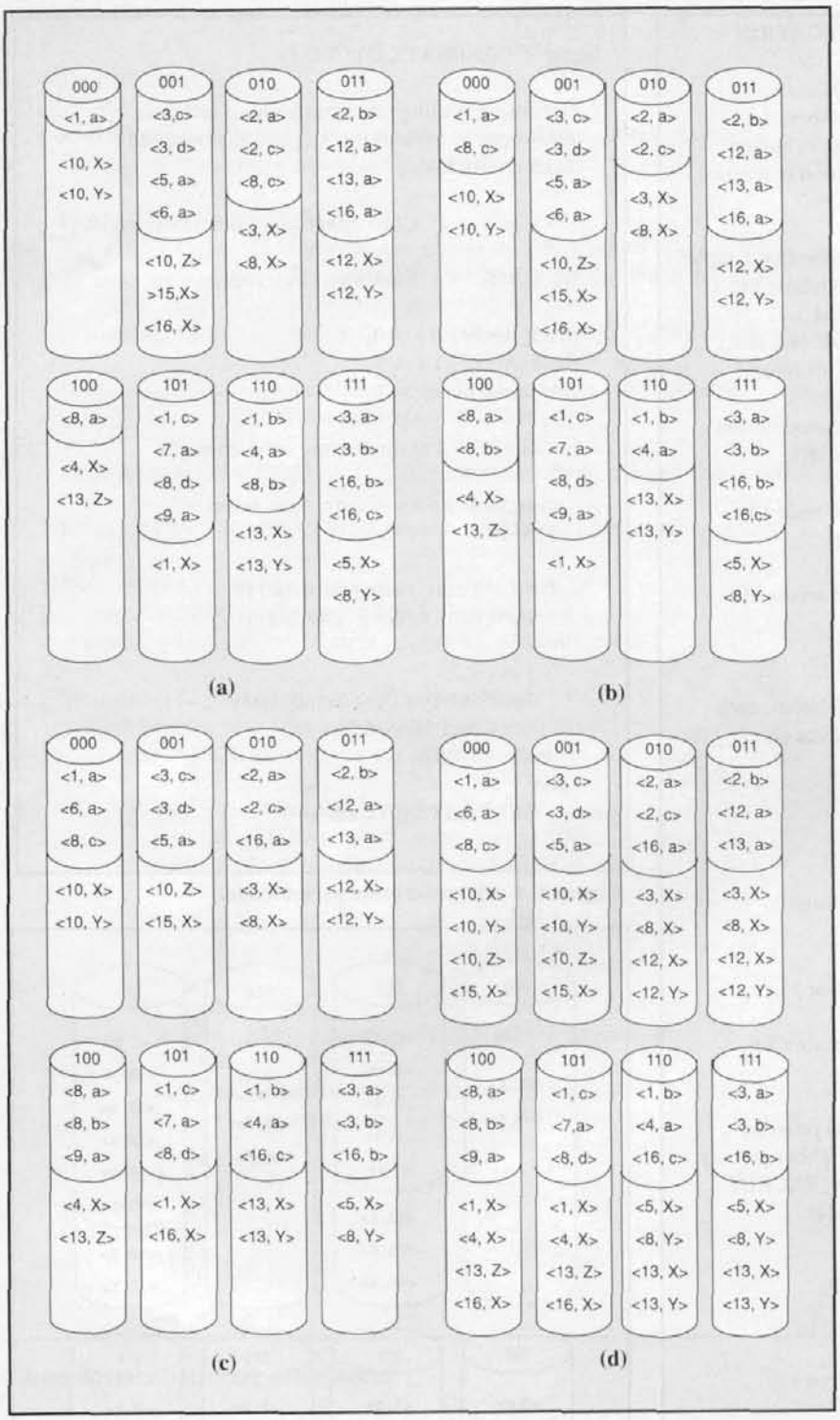


Figure 10. Broadcast-based Join (balancing: (a) step 1, (b) step 2, and (c) step 3). (d) represents the RCR stage.

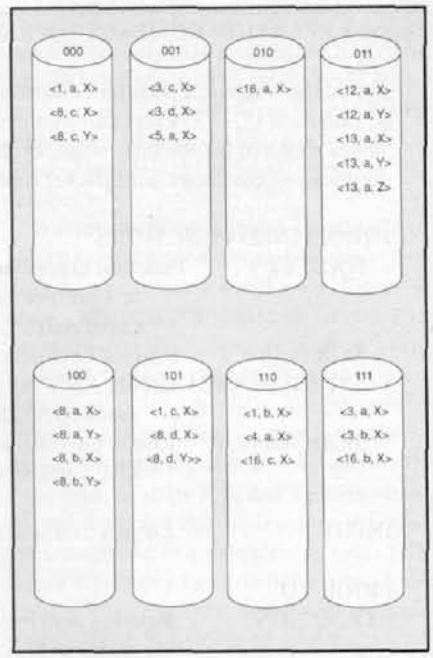


Figure 11. Final distribution of broadcast Join.

whose $n-j$th address bit is 0 keep S_1 and send S_2. Similarly, nodes whose $n-j$th address bit is 1 keep S_2 and send S_1. All nodes receive the sent tuples and merge them with those kept tuples. Sorted order is maintained. The GET_BUCKET(a, b) primitive places logical buckets a through b, inclusive, into send_set.

After n pairing steps, each node contains only tuples whose joining attributes fall within a disjoint subrange of the joining domain. Once both relations are redistributed, the local Join computation proceeds in parallel at each node.

Figure 12 shows a pseudocode description of the bucket data redistribution for a single relation. As with the balancing operation, both relations can be redistributed in parallel, with the one relation using link j at step j, while the second uses link $(j+1)$ mod n at step j. In this algorithm, however, either the portion of the algorithm that selects the buckets to exchange during each node-pairing step or the attribute-range-to-bucket-number assignment must be modified in the code that simultaneously repartitions the second relation. Also, in both the balancing and the bucket redistribution algorithms, simultaneous redistribution of both relations does not require separate links. If the two relations can be separated efficiently, both relations can be routed together on the same link. However, given independent communications processors and buffers, the use of multiple links generally reduces the data-transfer time. (Omiecinski and Tien describe a similar repartitioning algorithm that uses hashing to redistribute the relations.[10])

With the example in Figure 8, using a bucket Join algorithm that simultaneously redistributes both relations results in the following: In each of the three steps, a maximum of 5, 4, and 3 tuples, respectively, are routed in any internode communica-

tion. Note the significant difference between the maximal tuple transfer in the partitioning algorithm compared with the tuple-balancing algorithm. Figure 13 shows the result relation of the Join computation. As required, both Join approaches result in the same relation; however, the final distribution varies.

Characterizing performance potential

Here, I present experimental timings obtained from a portable hypercube-based database system. The system implementation issues are beyond the scope of this tutorial but can be found in an upcoming publication.[11] The evaluation focuses on only the two Join approaches presented here and includes a comparison of the timings obtained when the balancing step in the broadcast Join is omitted. Only timings for the Join operator are provided since the processing time of the Join greatly exceeds that of other common relational operators, making it the *computational* performance bottleneck in database processing.

I emphasize computational to differentiate between the I/O and CPU demands associated with database processing. The parallel I/O demands are commonly handled through parallelization of the Select operator, because the Select operators typically precede all other database operators in optimized query trees. No interprocessor communication exists in the Select operator, so performance measurements are not provided.

The timings presented were obtained using macros developed at Argonne National Laboratories[12] executing on an Encore Multimax running the Umax 4.2 operating system. These macros provided a fully portable, simulated, distributed-memory environment, allowing for system portability and independence from the hardware limitations of a particular vendor. The ability to vary hardware parameters comes at the expense of simulative overhead. Although the assumed hypercube architecture was simulated, all the database algorithms actually executed as though they were running on an actual hypercube. The Multimax configuration consisted of 16 processors, enabling each logical hypercube node to execute on a dedicated node in the Multimax. As the total system memory comprised 128 Mbytes, the entire database was memory resident.

```
begin    { BUCKET }

  { Redistributes data by attribute(s) values. Each bucket has a nonoverlapping
    subset of the global attribute domain. The union of the attribute value range
    of all the buckets is the global domain. Assumed is that all local tuples are
    already partitioned into local buckets and must only be routed to the
    appropriate processor.

    PREDEFINED FUNCTIONS:

        GET_BUCKET (k , j)        Function that moves all the tuples assigned
                                    to buckets k through j inclusive to
                                    a stated buffer (send_set)
        XOR (k, j)                Bitwise Exclusive OR of k and j
        rotate_right (k, j)       Rotate the binary representation of k, j bits
                                    to the right
        merge ( Y, buckets )      Mark the corresponding bucket for each
                                    tuple in buffer Y and move to the
                                    appropriate memory location

    INPUT:                        Local address of the node (X)

    PRIOR TO EXECUTION:           Relation randomly distributed across nodes

    EXECUTION TERMINATION:        Tuples distributed across nodes according
                                    to attribute value      }

    first := 0;
    last := N - 1;

    for j := 0 to n – 1 do begin

  { Select buckets to send      }
      if X[j] = 0 then begin

            send_set := GET_BUCKET ( (first + last + 1)/2, last );
            last := (first + last - 1)/2;
          end
          else begin

            send_set := GET_BUCKET ( first, (first + last - 1)/2 );
            first := (first + last + 1)/2;
          end
        end

      send ( XOR( X, rotate_right ( 1, j+1 ) ), send_set );
      receive ( XOR( X, rotate_right ( 1, j+1 ) ), receive_set );

  { Place received tuples into appropriate local bins      }
      merge ( receive_set, local_buckets );

    end { for }
  end { BUCKET }
```

Figure 12. Bucket-based partitioning pseudocode.

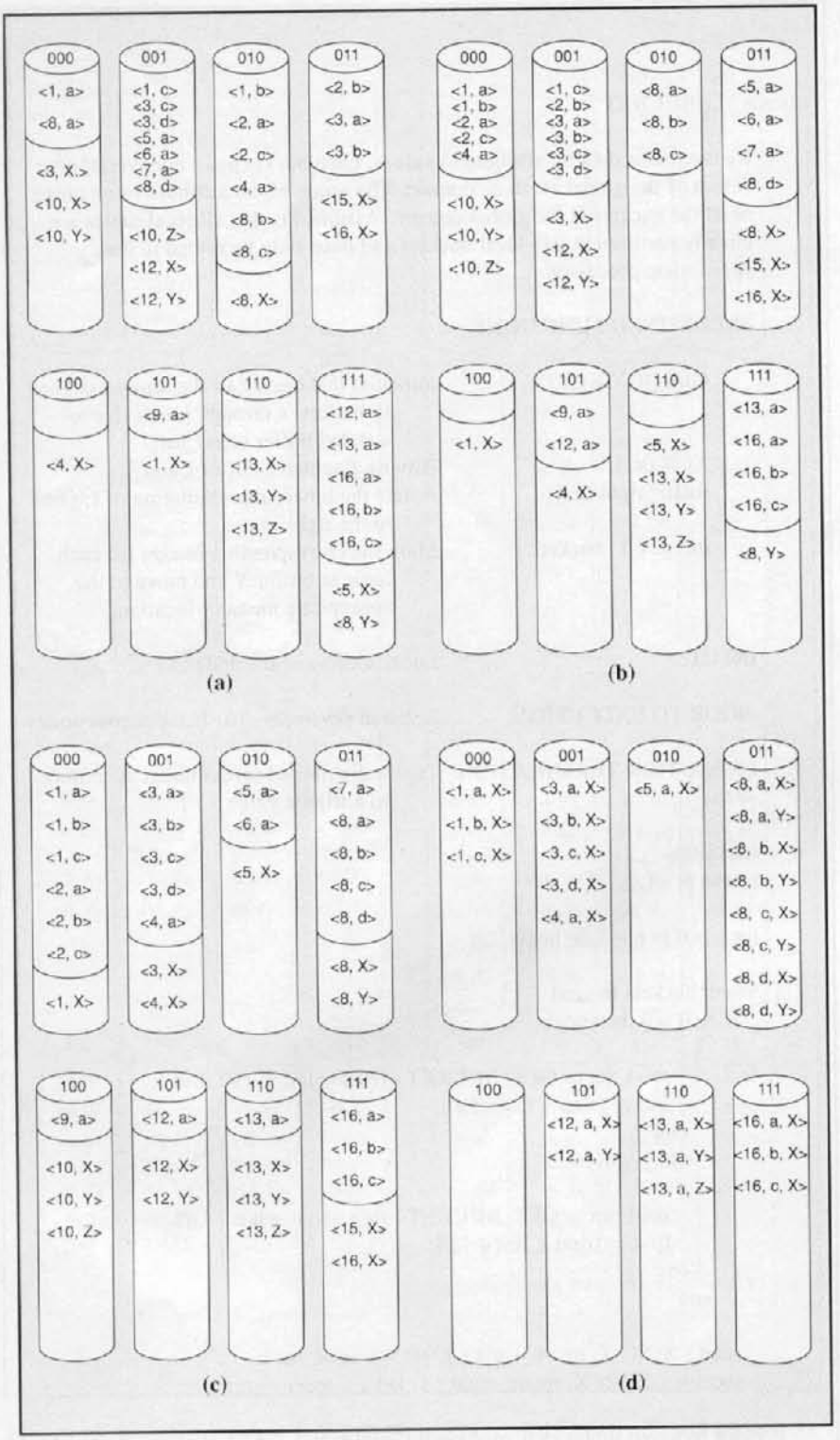


Figure 13. Bucket-based Join example: (a) step 1, (b) step 2, and (c) step 3. Part (d) represents the final result (bucket).

The emulative overhead significantly affects the actual timings observed in terms of absolute time. Hence, these timings should not be used for comparison against other systems, but only as a tool for comparing the two presented Join approaches. Because we are interested only in seeing the similarities and differences between these Join approaches, runs of a standard benchmark data set are not presented. Instead, a synthetic database that better illustrates the behavior of the presented approaches was constructed.

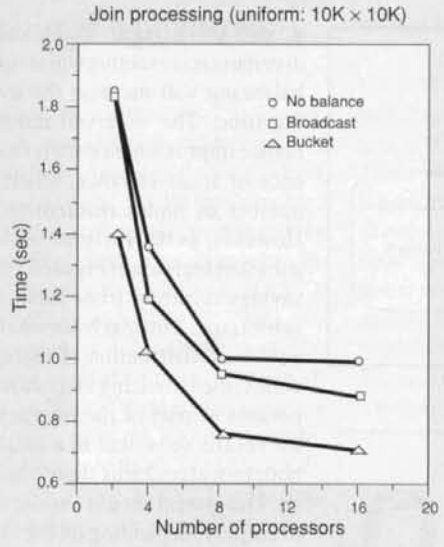
The database comprised relations of four different sizes: 1K, 10K, 100K, 1M tuples. Each tuple had 24 bytes. Relations were randomly generated using a uniform-attribute value distribution. The tuples of each relation were horizontally partitioned across the nodes using either a uniform (marked as "uniform" on the graph) or a bimodal skewed distribution (marked as "skewed" on the graph).

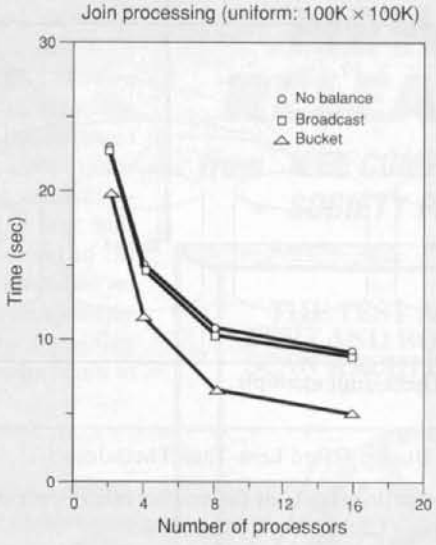
Results for multiple-size Joins are presented: 10K × 10K (uniform), 100K × 100K (uniform), 100K × 100K (skewed), 10K × 100K (uniform), 1K × 1M (uniform), and 1K × 1M (skewed). Each Join algorithm was run on four different hypercube configurations: two, four, eight, and 16 nodes. By definition, the smallest hypercube is of dimension 1, so single-node runs are not included in the timing presentation.

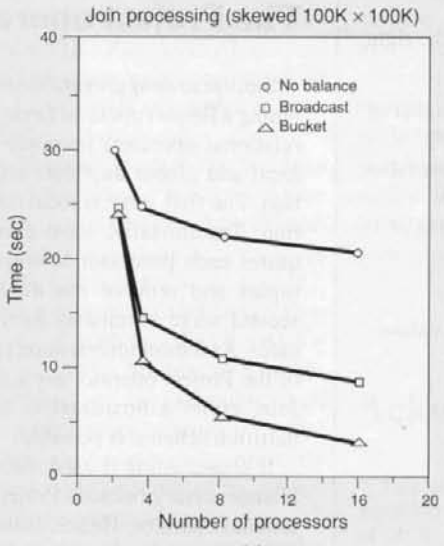
In general, the broadcast-based Join, as compared with a bucket-based Join, reduces the required communication at the expense of redundant computation. Redundant comparisons in the broadcast Join result from the unnecessary comparison of every tuple in the smaller relation, R_1, against all tuples in the larger relation, R_2. In the bucket-based Join, each tuple in R_2 is compared with only the relevant portion of R_1. In terms of communication requirements, unlike the bucket Join where both relations must be redistributed, the communication demands of the broadcast Join require that only R_1 be routed. Thus, when R_1 and R_2 are comparable in size, the bucket Join is better. However, if the difference in the communication times for the two Join approaches is significant (R_1 is much smaller than R_2), the broadcast Join is better. The precise size disparity is based on the degree of data skew and various system parameters.

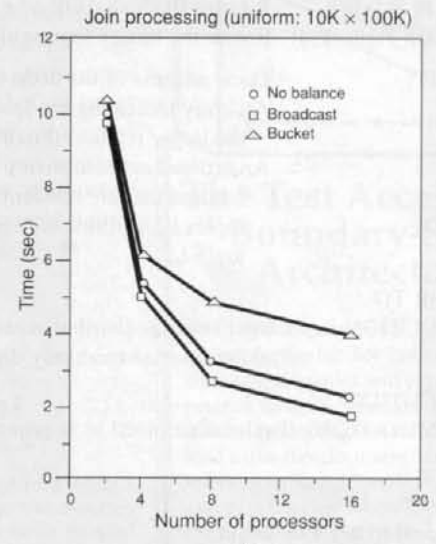
The timing results summarized in Figures 14a through 14f coincide with size disparity intuition. The scalability of the bucket Join exceeds that of the broadcast Join when the relations are comparable in size (Figures 14a-c), and the broadcast Join is superior when the relations differ in size (Figures 14c-f).

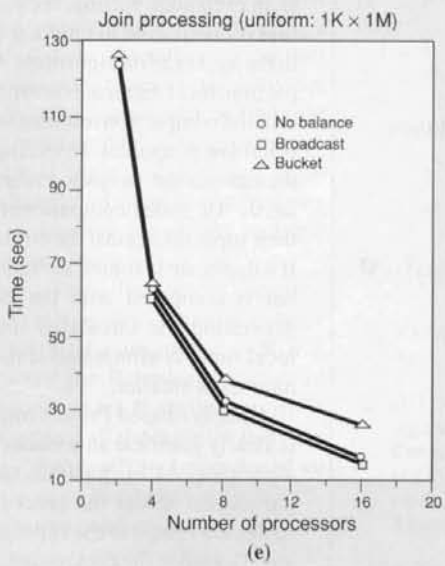
The graphs also show the performance obtained when the balancing step is not performed as part of the broadcast Join. When the tuples are uniformly distributed across the nodes, the overhead of balancing is negligible, but the reduction in processing time resulting from the balancing step is also negligible (Figures 14a, b,











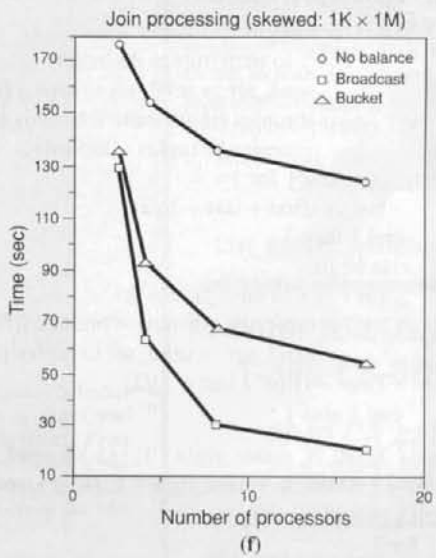


Figure 14. Experimental timing results.

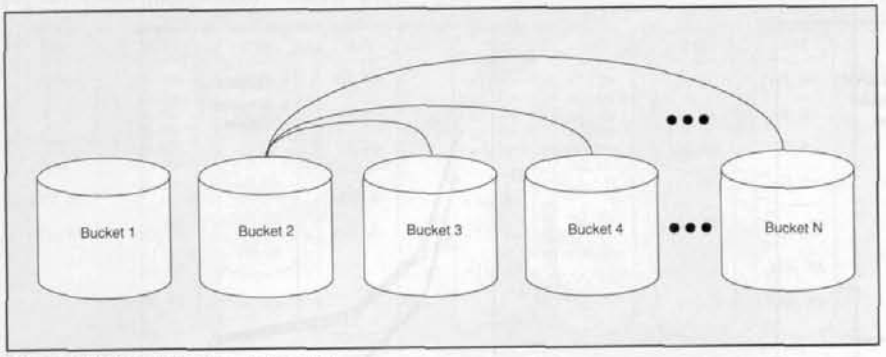


Figure 15. Theta-Join example.

```
begin   { Bucket-Based Less-Than Theta-Join }
    { Redistributes data of the smaller relation according to attribute(s) values
      for a LESS-THAN Theta-Join.
    PREDEFINED FUNCTIONS:
      XOR (k, j)          Bitwise Exclusive OR of k and j
      rotate_right (k, j) Rotate the binary representation of k, j bits to the right
    INPUT:                Local address of the node (X)
                          An array indicating the lower bound of each bucket of
                            the larger relation (bounds)
                          An ordered set comprising all the tuples of the smaller
                            relation that are resident at the node (tuples).
                            tuples [i].attribute is the joining attribute value of
                            tuple i.
    PRIOR TO
    EXECUTION:    Larger relation distributed according to attribute values
                  Smaller relation randomly distributed
    EXECUTION
    TERMINATION:  Tuples distributed as required by Theta-Join  }
    first := 0;
    last := N - 1;
    for j := 0 to n - 1 do begin
      send_set := Ø ;
      num_tuples := | tuples |;
      if X[j] = 0 then begin
            for i := 1 to num_tuples do begin
                  send_set := send_set ∪ tuples [i];
                  if tuples [i].attribute ≥ bounds [(first + last + 1) / 2] then
                    tuples := tuples - tuples[i];
                end { for }
            last := (first + last - 1)/2;
          end { then }
          else begin
            for i := 1 to num_tuples do
                if tuples [i].attribute < bounds [(first + last + 1) / 2] then
                  send_set := send_set ∪ tuples [i];
            first := (first + last + 1)/2;
          end { else }
      send ( XOR( X, rotate_right ( 1, j+1 ) ), send_set );
      receive ( XOR( X, rotate_right ( 1, j+1 ) ), receive_set );
      tuples := tuples ∪ receive_set;
    end { for }
  end { Bucket-Based Less-Than Theta-Join }
```

Figure 16. Theta-Join data-redistribution pseudocode.

d, and e). In fact, if the original tuple distribution is exactly even across the nodes, balancing will increase the total processing time. The observed minimal performance improvement results from the existence of some variance, albeit low, in the number of tuples resident at each node. However, as the variance of the tuple distribution increases (Figures 14b and f), the savings resulting from balancing become substantial. Thus, at least when faced with a skewed distribution of tuples across the nodes, the balancing step should be incorporated as part of the broadcast Join. All the results show that in a database system both Join algorithms should be implemented. The system should choose the approach to employ, depending on the input data set.

The Project operator

Similar to an aggregation operator, computing a Project operator (a multiscan, unirelational operator) involves two stages: local and global duplicate-tuple elimination. The first stage is local tuple elimination. Traditionally, local elimination requires each processor to sort its resident tuples and remove the duplicates. The second stage eliminates internode duplicates. As the communication requirements of the Project operator are a subset of the Join, either a broadcast or an attribute-partition scheme is possible.

If broadcasting is used, data replication is unnecessary because Project is a unirelational operator. Hence, instead of RCR, only relation compaction is required. That is, in each node pairing, the projected relation is compacted to only a single node, as in the aggregation algorithm. At each compaction, local duplicates are removed. Once relation compaction terminates, the cycling primitive is applied. In cycling, a copy of the compacted, projected relation is circulated. All nodes compare simultaneously their tuple set against the circulated tuples. If a duplicate is found, the local node number is compared with the node number generating the circulated tuple set. The local tuple is eliminated if the local node number is smaller.

A bucket-based Project implementation is nearly identical to a bucket-based Join algorithm. As in the Join, the domain is partitioned across the processors and the tuples are routed to the appropriate processor. To reduce the data routed, after each of the n steps, local duplicate elimination is performed. Thus, duplicates are eliminated as soon as possible.

Additional operators

Common set operators such as Union, Intersection, and Difference are typically incorporated into database systems. These multiscan operators can be implemented using the data-redistribution algorithms previously described. For example, the union of relations R[XYZ] and S[XYZ], written $T[XYZ] = R[XYZ] \cup S[XYZ]$, can be implemented by defining the new relation T[XYZ] as all the tuples of R and all the tuples of S, and then eliminating the duplicates in T, as in the Project operator.

The Intersection of R[XYZ] and S[XYZ], $T[XYZ] = R[XYZ] \cap S[XYZ]$, is the Join of the two relations in which only those tuples that match in all the attributes are kept, namely R.X = S.X, R.Y = S.X, and R.Z = S.Z. Finally, the difference of R and S, T[XYZ] = R[XYZ] – S[XYZ], is computed as a Join where only those tuples of R that *do not* join with any tuples in S are maintained.

Another popular operator that includes the Join described above as a special case is the *Theta-Join*. The Theta-Join of relations R[X] and S[Y], where $X \cap Y = \emptyset$, $A \in X$, $B \in Y$, and $\theta \in \{=, \neq, <, \leq, >, \geq\}$, written R[A θ B]S, is defined as $T[XY] = \{x \mid y \in R, z \in S$, such that $y[A] \theta z[B], y = x[X]$, and $z = x[Y]\}$. Modifying a broadcast-based Join algorithm to encompass all the Theta-comparators requires only that the local equality comparison presently made at each node be replaced by the appropriate Theta-comparator.

Because a Theta-Join may require tuples in an attribute range to be compared against tuples in a set of attribute ranges, altering a bucket-based Join algorithm to support Theta-comparators requires not only that the local equality comparator be replaced by the appropriate Theta-comparator but that the actual tuple-redistribution algorithm be changed to support comparisons between an attribute range and multiple other attribute ranges.

Consider a Theta-Join where the comparator operator is Less Than. Relations R[ABC] and S[DE] are joined so R[B < D]S. Then, all the R tuples that map to bucket 2 based on the B attribute must be compared against all the S tuples that map to buckets 2 through N inclusive based on the D attribute, as illustrated in Figure 15.

The algorithm proceeds as follows. Initially, relation S is partitioned according to attribute values using the algorithm in Figure 12. Then, using the same bucket ranges, relation R is partitioned into buckets so all tuples in bucket i ($1 \leq i \leq N$) are routed to those processors that contain buckets j ($i \leq j \leq N$) of relation S. Figure 16 shows a pseudocode description of an algorithm that achieves the necessary bucket-based Theta-Join data routing. In contrast to the algorithm in Figure 12, where no additional copy of the data is generated, here multiple copies of the data assigned to the lower numbered buckets are produced, resulting in a vast volume of internode data transfer. Algorithms for the remaining Theta-Join comparators likewise result in a vast replication of data.

Operator implementation is crucial, but fully exploiting multiprocessors in database processing involves numerous additional database concerns not discussed in this tutorial. Such topics as data placement, execution-site selection, query optimization, security, and recovery must be addressed before large-scale (thousands of processors) multiprocessor database systems become an alternative to high-performance mainframe solutions for very large databases and complex database-based applications. ■

Acknowledgment

This work was partially performed while I was at Bellcore using the computational resources of the Syracuse University Northeast Parallel Architectures Center (NPAC), which is funded by DARPA/RADC contract #F306002-88-C-0031.

I wish to thank Paul Jackson, Mark Segal, Bruce Shriver, and the anonymous referees, whose input vastly improved this article.

References

1. M. Stonebraker, "The Case for Shared Nothing," *Data Eng.*, Vol. 9, No. 1, Mar. 1986.
2. O. Frieder and C.K. Baru, "Query Scheduling and Site Selection Algorithms for a Cube-Connected Multicomputer System," *Proc. IEEE Eighth Int'l Conf. Distributed Computing Systems*, June 1988, CS Press, Los Alamitos, Calif., Order No. 865, pp. 94-101.
3. O. Frieder and G.E. Herman, "Protocol Verification Using Database Technology," *IEEE J. Selected Areas in Comm.*, Vol. 7, No. 3, Apr. 1989, pp. 324-334.

4. C. Seitz, "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

5. W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *Computer*, Vol. 21, No. 8, Aug. 1988, pp. 9-24.

6. D.J. DeWitt et al., "The Gamma Database Machine Project," *IEEE Trans. Knowledge and Data Eng.*, Vol. 2, No. 1, Mar. 1990, pp. 44-62.

7. D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proc. ACM SIGMOD Conf.*, ACM, New York, May 1989, pp. 110-121.

8. C.K. Baru, and O. Frieder, "Database Operations in a Cube-Connected Multicomputer System," *IEEE Trans. Computers*, Vol. 38, No. 6, June 1989, pp. 920-927.

9. C.K. Baru et al., "A Comparison of Join Algorithms for Hypercubes," *Proc. Fourth Hypercube Conf.*, SIAM, Philadelphia, 1989.

10. E. Omiecinski and E. Tien, "A Hash-Based Join Algorithm for a Cube-Connected Parallel Computer," *Information Processing Letters*, Vol. 30, No. 5, Mar. 1989, pp. 269-275.

11. O. Frieder, "A Database-Driven, Parallel Protocol Verification System Prototype," George Mason Univ., Dept. of Computer Science Tech. Report, TR-10-90, Oct. 1990. Submitted for publication.

12. E. Lusk et al., *Portable Programs for Parallel Processors*, Holt, Reinehart and Winston, Inc., 1987.

Further reading

The following list merely highlights some of the more recent parallel database efforts, and is not meant to be exhaustive. For additional background and earlier research, refer to the database machine textbooks listed in the last section below.

Hypercube-based algorithms

Arild, B., W. Baugsto, and J.F. Greipsland, "Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer," *Proc. Sixth Int'l Workshop on Database Machines*, 1989.

Baru, C.K., et al., "Join on a Cube: Analysis, Simulation, and Implementation," in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, eds., Kluwer Academic Publishers, Boston, 1988, pp. 61-74.

Bratbergsengen, K., "Algebra Operations on a Parallel Computer: Performance Evaluation," in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, eds., Kluwer Academic Publishers, Boston, 1988, pp. 415-428.

Frieder, O., *Database Processing on a Cube-Connected Multicomputer*, doctoral dissertation, Univ. of Michigan, Ann Arbor, Mich., 1987.

Frieder, O., and P. Jackson, "On the Design, Implementation, and Evaluation of a Portable Parallel Database System," *Proc. IEEE Int'l Conf. Databases, Parallel Architectures, and Their Applications*, CS Press, Los Alamitos, Calif., Order No. 2035, Mar. 1990, pp. 516-518.

Frieder, O., "Fault Tolerance on a Hypercube: A Database Application," to be published in *J. of Systems and Software*, Nov. 1990.

Rishe, N., D. Tal, and Q. Li, "A Sequenced Hypercube Topology for a Massively Parallel Database Computer," *Proc. Second Symp. Frontiers of Massively Parallel Computation*, CS Press, Los Alamitos, Calif., Order No. 892, Oct. 1988, pp. 521-524.

Gamma (ring-based LAN)

DeWitt, D.J., et al., "Gamma — A High Performance Dataflow Database Machine," *Proc. Conf. Very Large Data Bases*, 1986.

DeWitt, D.J., et al., "A Single-User Evaluation of the Gamma Database Machine," in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, eds., Kluwer Academic Publishers, Boston, 1988, pp. 370-386.

Bubba (hypercube interconnection used for routing only)

Boral, H., "Parallelism in Bubba," *Proc. IEEE First Int'l Symp. on Databases in Parallel and Distributed Systems*, CS Press, Los Alamitos, Calif., Order No. 893, Dec. 1988, pp. 68-71.

Copeland, G., et al., "Data Placement in Bubba," *Proc. ACM SIGMOD Conf.*, ACM, New York, 1988, pp. 99-108.

Smith, M., et al., "An Experiment on Response Time Scalability in Bubba," *Proc. Sixth Int'l Workshop on Database Machines*, 1989.

Tandem (fault-tolerant bus-based) Architecture

Tandem Database Group, "Nonstop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL," *Proc. Second Int'l Workshop High Performance Transaction Systems*, 1987.

Tandem Performance Group, "A Benchmark of Nonstop SQL on the Debit Credit Transaction," *Proc. ACM SIGMOD Conf.*, ACM, New York, 1988, pp. 337-341.

BBN Butterfly

Rosenau, T.J. and S. JaJodia, "Basic Database Operations on the Butterfly Parallel Processor: Experiment Results," Memorandum Report 6173, Naval Research Laboratory, 1988.

Oracle (database software for NCube's NCube-2 hypercube)

Oracle for the NCube 2 — Technology Overview, Doc. No. 50333-0789, Oracle, 1989.

Teradata DBC/1012 (a customized interconnection network)

DBC/1012 Data Base Computer System Manual, Doc. No. C10-0001-01, Teradata, 1985.

Database machine textbooks and tutorials

Boral, H., and P. Faudemay, *Database Machines*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1989.

Hurson, A.R., L.L. Miller, and S.H. Pakzad, *Parallel Architectures for Database Systems*, CS Press, Los Alamitos, Calif., Order No. 838, 1989.

Kitsuregawa, M., and H. Tanaka, eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, 1988.

Ozkarahan, E., *Databases Machines and Database Management*, Prentice Hall, Englewood Cliffs, N.J., 1986.

Sood, A.K., and A.H. Qureshi, *Database Machines: Modern Trends and Applications*, Springer-Verlag, Heidelberg, Germany, 1986.

Su, S., *Database Computers: Principles, Architectures, and Techniques*, McGraw-Hill, New York, 1988.

Ophir Frieder is a faculty member of the Computer Science Department at George Mason University. From 1987 to 1990, he was a member of technical staff in the Applied Research Area of Bell Communications Research. His research interests include parallel and distributed architectures, database systems, operating systems, and medical imaging architectures.

Frieder received his BSc (1984) in computer and communications science and his MSc (1985) and PhD (1987) in computer science and engineering, all from the University of Michigan. He is a member of Phi Beta Kappa and the IEEE Computer Society.

Frieder can be reached at the Department of Computer Science, George Mason University, Fairfax, VA 22030-4444.