# Exploiting database parallelism in a message-passing multiprocessor

by R. A. Lorie
J.-J. Daudenarde
J. W. Stamos
H. C. Young

**Parallel processing may well be the only means of satisfying the long-term performance requirements for database systems: an increase in throughput for transactions and a drastic decrease in response time for complex queries. In this paper, we review various alternatives, and then focus entirely on exploiting parallel-processing configurations in which general-purpose processors communicate only via message passing. In our configuration, the database is partitioned among the processors. This approach looks promising but offers challenging problems. The paper reports on our solutions to some of them: how to express strategies for efficiently executing complex queries, how to minimize overhead in operations such as parallel joins and sorts, and how to deal with transaction management in a highly distributed system. The paper ends with a discussion of the lessons we learned from exercising a prototype developed in IBM Research.**

## Introduction

Parallel processing may be harnessed to increase throughput (using $n$ times as many computers to do $n$ times as much work in the same amount of time) or to reduce response time (using $n$ times as many computers to solve a problem in $1/n$th the time). Parallel processing is applicable to database systems, but several factors make the problem challenging. First, the nature of the workload is unknown when the database system is deployed, since interactive users can construct arbitrary queries and execute them at any time. Second, the characteristics of the subset of data manipulated by a query are in general not known until run time, because it depends on both the predicates in the query and the state of the database. Third, shared-database systems permit concurrent access, which means that a parallel database system must support parallelism and multiprogramming simultaneously. These three challenges emphasize the need for run-time flexibility, load balancing, and effective scheduling.

Introducing parallelism and distribution (spreading data among several locations) in any system introduces new possibilities for overhead, and database systems are no

exception. Supporting a parallel, distributed database requires communication and additional synchronization over what is needed in a single-processor configuration. Moreover, using multiple threads for each transaction increases the overhead per transaction. In addition, the penalty for distributing locks (for concurrency control) is that global deadlock may occur, while the penalty for distributing logs (for recovery) is that a commit protocol must be employed [1]. This collective overhead can decrease transaction throughput and increase transaction response time, partially negating the reasons for introducing parallelism.

The ARBRE project (for Almaden Research Backend Relational Engine) was established in IBM Research to study database parallelism. The technology we adopted and the algorithms we invented attack some, but not all, of the above challenges and sources of overhead.

This paper reports on what we learned in the ARBRE project. First it discusses the requirements of large database applications and briefly reviews architectures that have been proposed to meet these requirements. The architecture in ARBRE uses a collection of independent processors that communicate by means of messages. After illustrating how the parallel database system works, the paper describes those parts of the system that are most interesting and new.

## The need for parallelism in database applications

We illustrate the need for powerful database systems by using a simple banking application as a running example. In a relational model, the banking application uses two tables. The table ACCOUNT has one row for each account, while the table ACTIVITY contains a row for each modification of an account:

```
ACCOUNT (ACCOUNT_NUMBER, CUSTOMER_NAME, BALANCE)
ACTIVITY (ACCOUNT_NUMBER, DATE, AMOUNT,
 OTHER_ACCOUNT_NUMBER)
```

- *Simple transaction*

A typical transaction $T$ transfers a certain amount of money from one account to another. For example, a transfer of \$100 from account $x$ to account $y$ updates the balances in both ACCOUNT rows and inserts the following two rows into ACTIVITY:

```
ACTIVITY (x, 9/9/91, −100, y)
ACTIVITY (y, 9/9/91,  100, x)
```

Transaction $T$ has a path length of the order of 200 000 instructions. With reasonable CPU utilization, we can run four transactions per second on a one-MIPS (million instructions per second) processor. Thus, the CPU time is a fraction of a second on a multiple-MIPS processor. The I/O time is also a fraction of a second, since the transaction requires only six I/O operations. If the transaction were running alone, the total response time would be less than a second, which is quite reasonable. The real problem is how to run hundreds or thousands of such transactions per second and maintain a low response time. For 1000 transactions per second, one needs 250 MIPS of power and a capability for 6000 disk accesses each second. If one disk can support a maximum of 30 accesses per second, one needs about 200 disks.

- *Complex queries*

The banking data may be queried for the purpose of decision-making. A query $Q$ may, for example, be used to find the accounts for which the average amount of individual deposits for the last month is larger than \$1000. We assume that the database contains ten million accounts, with an average of 20 activity records per account. A scan of table ACTIVITY is necessary. Scanning 200 million tuples at a cost of 1000 instructions per tuple takes 200 billion instructions. On a 10-MIPS machine, the response time is almost six hours, even if we ignore the time for I/O. This example shows that the sequential execution of a complex query can take a long time and may prevent a query from being run in an enterprise, even if its answer could be very valuable for decision making.

To improve the response time for complex queries, one needs more computing capability. One also needs a system organization that can use all of the computing power on one query and can overlap the query I/O with its processing. The question is, Can we provide such a system at a reasonable cost? Although uniprocessor systems continue to improve with respect to price and performance, the need to execute thousands of transactions per second and to rapidly answer complex queries on large databases strongly suggests that alternatives be investigated.

## Alternatives

From a hardware viewpoint, the alternatives fall into two categories: those that use special-purpose hardware customized for database operations (see [2], for example) and those that construct a parallel database system in software running on general-purpose processors [3–6]. We chose general-purpose processors for four reasons. First, relatively inexpensive, high-performance microprocessors are available. Second, we expect this hardware base to give us substantial improvements in price and performance. Third, special-purpose hardware for relational databases typically supports read-only databases and frequently does not directly support the relational join operator. We want to provide efficient support for all the relational operators, including updates. Fourth, using

general-purpose processors does not prevent us from incorporating special-purpose hardware should it become generally available and desirable.

Our first decision was whether the multiple processors should share memory, disks, both, or neither (message-passing). Because approaches that share global memory do not scale (i.e., the improvement in performance per additional processor decreases as the number of processors increases) for large numbers of processors, we considered only shared-disk and message-passing architectures. A shared-disk architecture has two advantages. First, it permits any transaction to be executed on only one processor, since every processor can access all disks in the system. This avoids the overhead of distributed transactions. Second, a shared-disk approach simplifies CPU load balancing, because any transaction can execute on any processor. A message-passing architecture has different advantages. Instead of sending entire disk pages on the communication network, it sends only selected records. This reduces bandwidth consumption and may require a less expensive network. In addition, algorithms and techniques developed for distributed databases are applicable to message-passing architectures and may be adopted without change. Finally, there is no cache-consistency problem with disk pages as there is in a shared-disk architecture. For simplicity, we did not consider a hybrid architecture that uses a shared-disk configuration as a node in a message-passing architecture. Ultimately, we decided upon a message-passing architecture.

We chose a message-passing architecture in which nodes are interconnected by a network that lets each pair of nodes communicate. Each node has its own processor(s), memory, channels, disks, operating system, and database software. Each table in the relational model is partitioned into subsets of rows, and each subset is stored and managed at one node, although data may be replicated at one node or at several nodes. This horizontal partitioning of data can be controlled by various mechanisms, such as hashing or key ranges.

## The ARBRE prototype

We were interested in exploring the use of parallelism within complex relational queries and among large numbers of independent transactions. The ARBRE project involved the design, construction, and evaluation of a prototype of a multiprocessor database machine that exploits parallelism in three different ways [7]. First, the prototype simultaneously executes independent transactions and thus supports *inter-transaction* parallelism. Second, the prototype simultaneously executes different operations from the same transaction and thus supports *inter-operation* parallelism. Third, the prototype uses several nodes to execute a single operation in parallel and thus supports *intra-operation* parallelism. Note that the latter two forms of parallelism are *intra-transaction* parallelism.

The ARBRE prototype runs on equipment that was available for use in our database laboratory: three dyadic IBM 4381 systems, which provide a total of six processors. Communication is supported by an IBM 3088 channel-to-channel interconnection unit and a prototype high-performance interprocessor communication subsystem implemented by our colleague Kent Treiber. The operating system is MVS. We measured the performance of our prototype and compared it with the results of a simulation model. Having validated the model, we were able to use it to study larger and more powerful hardware configurations.

The database system is organized as follows: On each of the three 4381 computers, four virtual nodes are emulated by running four database systems in four address spaces, providing 12 virtual nodes in total. We chose this number of virtual nodes because 12 concurrent scans provide enough parallelism to keep all CPUs busy. Even nodes on the same 4381 computer communicate with messages, since we did not exploit shared memory. Each data table is split horizontally into partitions; each partition is stored in a different virtual node. The access method is borrowed from SQL/DS (DBSS) [8]. Since we were interested primarily in studying the performance of such a system as it executes database operations, we did not implement a query optimizer; instead we hand-coded specific strategies that would normally be chosen by a query optimizer.

A strategy indicates how a transaction or query is executed and is expressed as a series of code fragments. The code fragments rely on three interfaces that handle lower-level functions: the database functions themselves, the communication primitives, and the operating-system primitives that are used to create and manage threads of execution. A code fragment may be duplicated at several nodes to provide intra-operation parallelism. We compiled code fragments into object code; an alternative is to represent the code fragments with data structures and interpret them at run time. An *executor* controls the execution of the fragments.

## The executor

Every virtual node executes the same software, including the database system, which manages the data kept at that node. Since data are not shared, a node executing a request that involves data stored only at another node must send a message to the other node instructing it to manipulate the data. As a result of this function shipping, the logic must be decomposed into fragments, with each fragment being executed by a thread at a node. A fragment may be as simple as a single procedure call, or it may be an arbitrary procedure that uses sequencing, loops,

conditional execution, and expression evaluation. A node may also use multiple threads for one transaction, to overlap I/O and processing or to exploit any multiprocessing capability available at the node.

The function-shipping mechanism used in the ARBRE prototype is called REV (Remote Evaluation) [9], which is an improvement over the classical RPC (Remote Procedure Call). The improvement permits the incorporation of several RPCs along with other program logic into a single message and thus does not require a separate request message for each RPC.

The node that receives a database request from an application coordinates the execution of that request: It ensures that each fragment of the request is loaded at the correct node(s), each fragment is executed at the right time(s), and each thread used by the request is released at the appropriate time. Several techniques have been developed to coordinate fragments [10]; the particular techniques we used are not significant for the following discussion.

Let us show how fragments interact, with transaction $T$ defined above as an example. Suppose the application is executing at node 1. When the application wants to execute $T$, the application invokes the first fragment for $T$, called $F_a$, which reads the balance in account $x$. If account $x$ is local, $F_a$ will invoke the database interface; if not, $F_a$ will use REV to invoke $F_b$ (to read the balance) at another node, say node 2. This implies that the identifier of the receiving node must be known to $F_a$ or must be determined dynamically by $F_a$. Suppose account $x$ is not local. When received by node 2, the REV message from $F_a$ initiates execution of $F_b$, which accesses account $x$ and sends the data to $F_a$. The data are read by $F_a$, which checks the balance and sends messages for updating account $x$ (at node 2) and account $y$ (probably at another node, 3), in parallel. $F_a$ inserts two rows into the local ACTIVITY table and then waits until completion messages from nodes 2 and 3 are received. As we can see from the example, the data returned by each fragment are small, and the data are always returned to the fragment caller. Performing the updates of the balances of $x$ and $y$ in parallel corresponds to inter-operation parallelism.

Now, we use a portion of query $Q$ to illustrate how the executor deals with data-intensive operations. Consider the case in which the ACTIVITY rows for a single account are scattered among all processors. Let us assume that the strategy for evaluating the aggregate function (average amount of deposits) is to send all ACTIVITY rows to the nodes that own the corresponding ACCOUNT rows. This can be done in parallel by having two fragments at each node. One fragment ($F_c$) sends each local ACTIVITY row to the node that owns the account in the row. The other fragment ($F_d$) receives ACTIVITY rows, merges them, and computes the average of the deposits for each local account.

In the case of query $Q$, several instances of $F_c$ run in parallel and exploit intra-operation parallelism. ACTIVITY rows flow from one node to another once a channel has been opened between the two nodes (actually, between the two threads).

Although we have mentioned the execution of fragments at one or more nodes, we have not discussed how they arrived there in the first place. The simplest way is for all fragments for a particular program (a set of database calls) to be stored initially at one node. Then, each fragment can be sent to another node, together with the request for its execution. This and other alternatives are discussed in [11].

Now that we have described the functions of the executor, we discuss how the details of strategies can be specified.

## Distributed Access Specification Language (DASL)

The strategy determined by the query optimizer is encoded as a graph in which vertices represent operations, and arcs represent communication. Since a vertex often represents several operations "blocked" together, we often refer to a vertex as a block. Each arc connects the output of one operation block to the input of another one. The way vertices and arcs are combined can be viewed as a language. The Access Specification Language (ASL) in [12] supported only single-node strategies. Here, we extend it to describe strategies that exploit parallelism in one or more nodes. The extended language is called DASL (Distributed ASL). In this section, we briefly describe the functionality of the language, first for single-node operations and then for distributed operations that execute in parallel.

- *Single-node operations*

Tables, tuples, indexes, and temporary relations are some of the objects in DASL; they have their normal meaning for relational systems. There are two additional objects:

- The *stream*: It is a named sequence of tuples. Although a stream may be specified as a constant, it is generally the result of an operation.
- The *G-stream*: It implements the GROUP BY operation of SQL [8] and is a stream of groups. Each group is a stream of tuples that are related in some way; a group may be empty (but its existence as an empty group is not lost). For example, in one implementation of a G-stream, flags may be used to identify the boundaries of the groups.

The DASL language supports the following operations:

- A *scan* operation is applied to a table (or to an input stream) and produces a stream. The scan evaluates the

scan predicates for each row in the table; for each row that qualifies, a tuple is computed and fed out. The scan may have an auxiliary input stream. If it consists of a single tuple, the values in that tuple can be used in the predicate evaluation and output computation, but the output is still a stream. If the auxiliary input stream comprises more than one tuple, the operation is repeated for each tuple in it, and the output is a G-stream (with one group for each tuple in the auxiliary input).

- The *sort* operation accepts a stream as input and reorders it according to a given sort key; the output is also a stream.
- The *group* operation receives a sorted stream and a grouping condition and produces a G-stream.
- The *aggregate* operation computes any aggregate (such as sum or maximum) on a stream or on each group of a G-stream.
- The *join* operation accepts two streams or G-streams and combines their tuples in a one-to-one, group-to-group, or one-to-group way, to produce a stream or a G-stream.

To illustrate how operations are combined to build complex strategies, we consider the query $Q$. First, assume that all data are stored at one node. **Figure 1** is a sketch of the strategy discussed in the previous section. Block 1 represents a scan of the ACTIVITY table. Since $Q$ concerns deposits only, a predicate selects the tuples with amount >0. Tuples produced by block 1 enter block 2, which sorts them by ACCOUNT_NUMBER. This sorted stream goes to block 3, which produces a G-stream, in which each group corresponds to a different account. Block 4 computes the aggregate (the average amount of individual deposits); its output is a stream in which each tuple corresponds to a different account and contains the corresponding aggregate value. Scan block 5 evaluates the predicate "value > 1000" and produces a stream $q$; each tuple of $q$ corresponds to an account with average deposits >1000. A relational join is needed to combine that information with the information in table ACCOUNT. One of the most efficient methods is to use a merge-join algorithm. We have a first thread execute the operations described above, producing stream $q$, sorted by ACCOUNT_NUMBERs, and a second thread produce a stream *temp*, composed of ACCOUNT tuples, also sorted by ACCOUNT_NUMBERs. Then the join simply consists of merging the two streams and finding the matching tuples.

The technique described relies on multiprogramming only; it is a form of software parallelism. We now look at parallelism in its general form.

- *Expressing parallelism*

We turn our attention to a strategy for a multiprocessor configuration. We still assume that all of the ACTIVITY

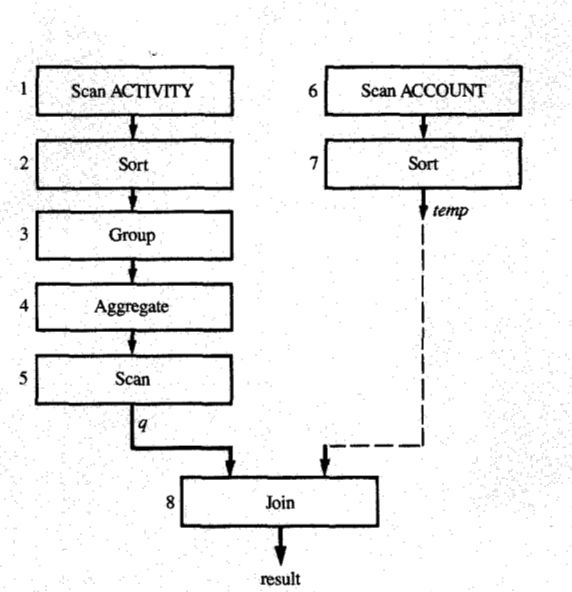


**Figure 1**

DASL representation of a strategy for query $Q$ with multiprogramming.

rows are stored at one node, as in the above example, but this time we assume that the entire ACCOUNT table is stored at another node and the application runs at a third. We exploit *inter-operation* parallelism by scanning the ACTIVITY table and the ACCOUNT table in parallel. **Figure 2** shows five threads of execution distributed among the three nodes involved. The threads are linked in a dataflow fashion. The first thread (at block 0) runs at the node that executes the application (the A-node). Block 0 generates a stream $p$ with a single dummy row (a token). Two other threads, at the nodes that own data, wait for the stream $p$. As soon as they receive the token, they execute the indicated operations until they are blocked. Blocks 1 to 5 describe an activity identical to the one described above for the single-node case. The same is true for blocks 6 and 7. The difference is that these fragments of the query evaluation are started in parallel. They produce two streams, $q$ and *temp*. These streams are input to another thread, which executes block 8. (This thread may be at any node; however, the A-node seems to be the best choice.) The output $s$ is sent to the A-node, where a new thread receives it (block 9). Note that we used the maximum number of threads; in practice, the number can be reduced. For example, blocks 0 and 9 can share the same thread at the A-node.

The only functions we added to the language to support inter-operation parallelism are the send and receive operations:

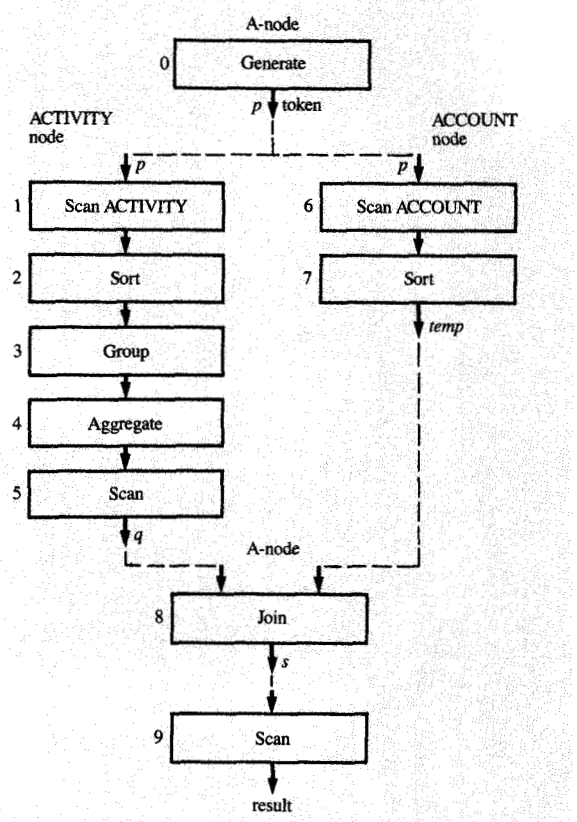


**Figure 2**

Representation of a strategy for query $Q$ exploiting inter-operation parallelism.

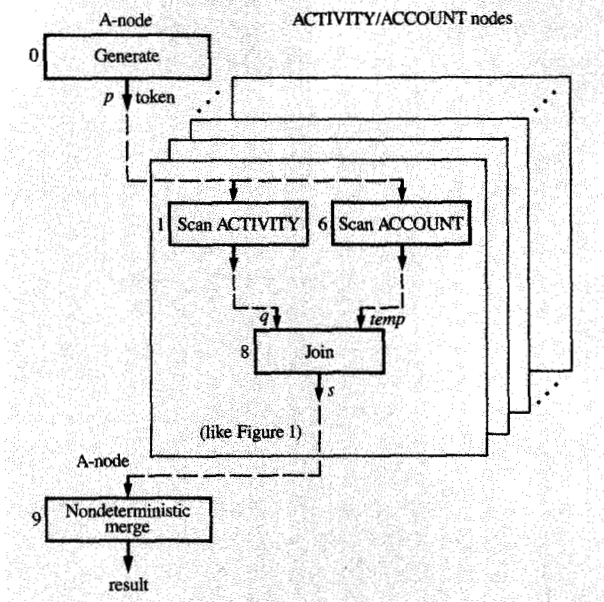


**Figure 3**

Representation of a strategy exploiting intra-operation parallelism.

- A *send* operation specifies a destination (a ⟨node, stream⟩ pair).
- The *receive* operation specifies only an input stream.

Note that the fragments executed in parallel scan and sort large amounts of data. This improves response time (actually, the time to execute blocks 1 to 5 is overlapped with the execution of blocks 6 and 7). Nevertheless, the degree of parallelism is very low, leaving most of the nodes uninvolved.

To remedy the situation, we exploit *intra-operation* parallelism by executing a single operation in parallel. In other words, several nodes execute the same fragment on different portions of the data. Consider the query $Q$, assuming this time that the ACCOUNT rows are stored at $n$ nodes and that all ACTIVITY rows for an account are stored at the node where the ACCOUNT row is stored. The execution strategy is depicted in **Figure 3**. At the A-node, a thread (block 0) receives the request and initiates the parallel execution by sending a token to all nodes that have ACTIVITY (and therefore, ACCOUNT) rows. Each of these nodes executes blocks 1 to 8 on local data. Each of the $n$ instances of block 8 sends a stream $s$ to the A-node, which is received by block 9. Since block 9 receives its input from various nodes, a merge must be performed. In this case, the order is irrelevant, so the merge may be nondeterministic. If the $n$ streams coming from the ACTIVITY/ACCOUNT nodes are ordered, the substreams may be merged by value in order to produce a fully ordered stream.

In summary, to support intra-operation parallelism we introduced

- The notion of *multicasting* a stream to several nodes (send to ALL or a selected set), which activates multiple instances of execution fragments that execute in parallel.
- The notion of *merge*, which applies to a node block receiving from various nodes substreams that must be merged into a single stream.

- *Scheduling*

Note that the structures shown in the examples represent the maximum parallelism consistent with dataflow execution. In practice, scheduling can be used to reduce the degree of parallelism, when appropriate. For example,

in Figure 3, you may be quite satisfied with the amount of parallelism provided by the simultaneous execution of many instances of blocks 1–2, and decide to execute blocks 6 and 7 at a node only when block 2 at that node has completed. Such a limitation on parallelism may require storing intermediate results, but may be preferable when too much parallelism causes unacceptable contention for critical resources. In DASL, tokens can be used to control parallelism, since they trigger execution.

- *Extensibility*

The DASL constructs are sufficient for expressing useful strategies. DASL can be extended easily by defining other operators that accept and produce streams. Routines to perform these operations must then be incorporated in the system and made accessible to the executor.

- *Intra-transaction deadlock detection*

In DASL, a database query is represented by an acyclic dataflow graph that consists of operator vertices and directed communication arcs. A transaction running in isolation can become deadlocked by itself if its DASL graph, when viewed as an undirected graph, contains a cycle. These deadlocks, a surprise to us, happened frequently for certain distributed algorithms we developed. The intra-transaction deadlocks we encountered were caused because buffers between operators in the dataflow graph are finite.

An example of this buffering-dependent deadlock is illustrated in **Figure 4**, where F1, F2, J1, and J2 are operator vertices. The dataflow graph in Figure 4(a) could represent the redistribution phase of a distributed join in which F1 and F2 receive sorted substreams and send the "odd" values to J1 and the "even" values to J2. F1 and F2 are fork-type operators (each has multiple outputs), whereas J1 and J2 are join-type operators (each has multiple inputs and a single output stream) that deterministically merge the sorted tuples they receive from F1 and F2 to produce sorted output streams. A deadlock, illustrated in Figure 4(b), can occur as follows. Assume that J1 already has a tuple from F1 and is waiting for a tuple from F2. F2 might be waiting to send a tuple to J2 but cannot, because of communication pacing (i.e., the buffer between F2 and J2 is full). J2 could be waiting for a tuple from F1 because it already has a tuple from F2. If F1 is waiting to send a tuple to J1 but cannot because of pacing, a deadlock exists, since J1 is indirectly waiting for itself.

A buffering-dependent deadlock like this can occur only if a *blocking* fork-type operator is an ancestor of a blocking join-type operator. A dataflow operator is said to be blocking if either of the following conditions causes the dataflow operator to stop processing inputs: Some but not all of its input arcs are empty; some but not all of its output arcs are full. One way to avoid buffering-dependent deadlock is to let buffers overflow onto disk, essentially providing "infinite" buffers. An alternative approach is to replace some blocking join and fork operators in a DASL graph with nonblocking operators when deadlocks may occur. We chose the latter approach and implemented a nonblocking version for each blocking operator in DASL. This was done by providing one or more internal buffers or by exploiting the cursor facilities provided by the underlying access method. The nonblocking operators let us avoid intra-transaction deadlock when we experimented with different algorithms. If the query optimizer checks whether a blocking fork-type operator is an ancestor of a blocking join-type operator, it can substitute nonblocking operators for blocking operators to avoid the possibility of deadlock.

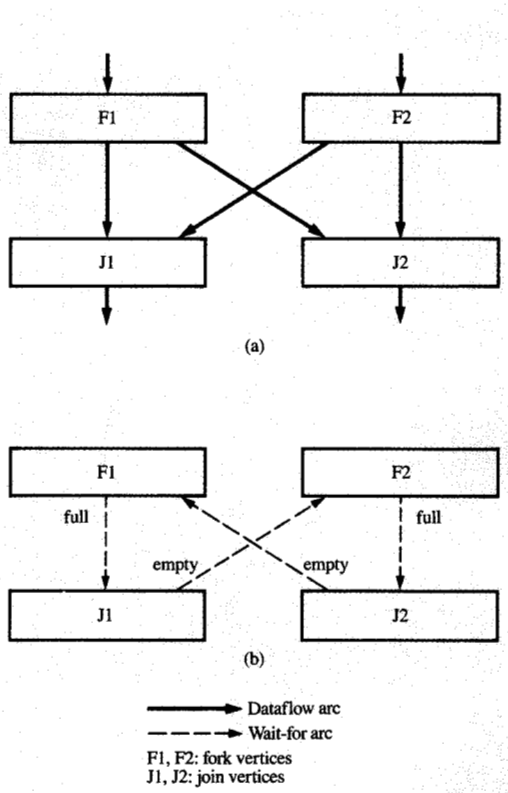


**Figure 4**

Scenario of a self-deadlock: (a) dataflow graph; (b) dependency graph.

## Algorithms

We developed several parallel algorithms for sorts and joins in a distributed system that require less

communication than existing ones. The next three subsections describe our algorithms.

- *Fully parallel sort algorithm*

In this section, we consider the problem of sorting a file in a distributed system, such as ARBRE. We assume that the result of the sort is needed at a single node called the A-node, while the file is originally distributed over several nodes. The algorithm exploits parallelism during the sort, merge, and return-to-A-node phases. In addition, this algorithm requires less inter-node communication than existing parallel sort algorithms. Our algorithm labels nodes as being data nodes, merge nodes, the coordinator node, or the A-node. A single node may have more than one role in the algorithm.

We exploit the size difference between the sort key and the entire record by manipulating only sort keys instead of records whenever possible. The data nodes eventually send sorted records directly to the A-node, which efficiently performs the final merge step without doing sort-key comparisons. As explained below, the A-node merge algorithm uses information computed during the processing of the sort keys by the merge nodes. The merge nodes send only this information to the A-node; the sort keys are discarded as the records are merged.

We present a high-level description of the algorithm and then give an example. Each data node sorts locally the portion of the file that it owns; in doing so, it computes the distribution of the sort keys that it encounters and sends the distribution to the coordinator. On the basis of such information from all data nodes, the coordinator assigns each merge node a range of sort keys for which it is responsible and multicasts this partition function to all the data nodes and the A-node. (Multicasting is more efficient than sending individual messages.) Each data node sends each of its sort keys (in sorted order) to the appropriate (assigned) merge node. Each merge node receives these sort keys from all data nodes and merges them. At the same time, it builds a list of node numbers. This list remembers, for each record in the result of the merge, the node from which it came. During the return-to-A-node phase, each merge node sends this stream of node identifiers to the A-node. The A-node reads the node-identifier streams from the merge nodes in the order specified by the global coordinator. (That is, the node-identifier stream from the merge node responsible for the smallest sort-key range is read first, and so on.) Then each data node sends its stream of sorted records to the A-node. The A-node uses the node identifier as an index to select the next data node and then reads the next record in the stream from that node.

We illustrate this sort algorithm by an example that involves two data nodes and two merge nodes. (The sort key is the first character, and the data fields are represented as *-dddddd*.) After the local sort phase, the contents of the sorted file are as follows:

| *Data node 0* | *Data node 1* |
|---|---|
| A-dddddd | B-dddddd |
| D-dddddd | C-dddddd |

The global coordinator determines that sort keys of value *B* or less should be sent to merge node 2 for merging, and all other keys should be sent to node 3.

The merge process at node 2 receives the following streams:

A (*from node 0*) B (*from node 1*)

(Note that data fields are not sent.) It generates *0* and *1* in its output stream to the A-node, meaning that the smallest record is from node 0 and the next smallest record is from node 1. Similarly, the merge process at node 3 receives *C* (from node 1) and *D* (from node 0) and produces *1* and *0* in its output stream to the A-node. In the meantime, the sort process at node 0 sends the A-node a stream consisting of *A-dddddd* followed by *D-dddddd*, and the sort process at node 1 sends the A-node *B-dddddd* followed by *C-dddddd*. Thus, the A-node receives two node identifier streams (from the merge processes) and two sorted record streams (from the sort processes). The contents of these streams are as follows:

| *Node identifier streams* | | *Sorted record streams* | |
|---|---|---|---|
| *From node 2* | *From node 3* | *From node 0* | *From node 1* |
| 0 | 1 | A-dddddd | B-dddddd |
| 1 | 0 | D-dddddd | C-dddddd |

The A-node first reads the node identifier stream from node 2. The first element in that node identifier stream has a *0*, indicating that the smallest record is from node 0 (*A-dddddd*). The next element in the node identifier stream from node 2 has a *1*, indicating that the next smallest record is from node 1 (*B-dddddd*). Similarly, the node identifier stream from node 3 has *1* and *0*, indicating that the next two records should be read from node 1 (*C-dddddd*) and node 0 (*D-dddddd*), respectively.

An alternative method to the fully parallel algorithm is to send both the sort key and the data (i.e., the entire record) to the merge nodes, which then send the merged records to the A-node. For the same example shown above, the contents of the merge nodes after the merge are

| *Node 2* | *Node 3* |
|---|---|
| A-dddddd | C-dddddd |
| B-dddddd | D-dddddd |

Node 2 sends all of its merged records to the A-node first; then node 3 sends all of its merged records. Only one merge process sends merged records to the A-node at any given time. This sequential portion can limit the degree of parallelism of the algorithm, especially when the merged records do not fit in the main memory of a merge node. On the contrary, in our fully parallel sort algorithm, the A-node reads records from all data nodes in an interleaved fashion. Even though the interleaved reading is slightly more expensive than a sequential reading, an A-node employing current microprocessor technology can accept records from disk drives on five to ten data nodes without becoming the bottleneck.

Let us look at the amount of inter-node communication of our fully parallel algorithm and the alternative method just mentioned. We use $K$, $D$, and $I$ to represent the lengths of the sort key, data, and node identifier. (A record consists of the sort key and the data; thus, the record size is $K + D$.) For our fully parallel method, the communication per tuple is $K + I + (K + D)$. For the alternative method, the communication per tuple is $2(K + D)$. The difference is $D - I$ per tuple.

The semi-sort idea (i.e., sorting keys) is unrelated to the parallel return-to-A-node phase idea. However, it is advantageous to use a semi-sort in our algorithm, because the data are not needed by the nodes that merge.

The efficiency of the algorithm is affected by the amount of skew in the data (the uneven distribution of data among the nodes). Since skew affects only the early phases of the sort (before tuples are redistributed), the adverse effect may be diluted.

We implemented this sorting algorithm, developed an analytical model, and validated the model. The details are in [13].

Survey articles on parallel sorting show that the problem has been studied extensively [14, 15]. Here we compare our fully parallel sort algorithm with recently developed sort algorithms that are appropriate for message-passing database systems. One important difference is that all the other algorithms send entire records between nodes two or more times, whereas our algorithm sends records only once. In the FastSort algorithm [16], all sorted streams are merged at the A-node. This sequential merge takes longer than the interleaved reading used in our fully parallel sort algorithm, because merging requires sort-key comparisons. The sort algorithms presented in [17, 18] use a multiphase merge to reduce the number of sorted streams an A-node merges, but the A-node still does merging. In [19], records are partitioned by range according to sort keys, and the A-node simply reads the sorted streams sequentially from one node at a time. Whether this algorithm or our algorithm minimizes execution time at the A-node depends on main-memory sizes and disk configurations. If each sorted stream fits into the main memory of the corresponding sort node, or if each sorted stream is stored in an interleaved fashion across sufficiently many disk drives, sequential reading is preferable, because the A-node is the bottleneck. Otherwise, our form of interleaved reading is preferable.

In summary, our sorting algorithm has the following advantages: 1) Each phase (except for very short intervals) is performed in parallel, yielding high utilization of resources and a low sorting time. 2) The algorithm reduces the amount of inter-node communication by using a semi-sort algorithm that sends only sort keys on the inter-node communication network.

- *Parallel join algorithm using semi-join*

In this section, we describe a parallel join algorithm that is useful when the join results are needed at a single node, called the A-node.

Consider tables $R$ and $S$. Each row has an identifier, a join column (a single letter in this example), and data. The result of the join is obtained by combining all rows of $R$ with all rows of $S$ for which the join columns match. For example, $R$, $S$, and the result of the join might be the following:

| *R* | *S* | *join(R, S)* |
|---|---|---|
| 1 a r11111 | 1 b s11111 | b r22222 s11111 |
| 2 b r22222 | 2 c s22222 | b r33333 s11111 |
| 3 b r33333 | 3 d s33333 | c r44444 s22222 |
| 4 c r44444 | 4 e s44444 | c r55555 s22222 |
| 5 c r55555 | | d r66666 s33333 |
| 6 d r66666 | | |

In a distributed database system organization, the rows of $R$ and $S$ are generally scattered over many nodes. For example, $R$ and $S$ may be stored on two nodes, as follows:

| *R* | | *S* | |
|---|---|---|---|
| *Node 1* | *Node 2* | *Node 1* | *Node 2* |
| 1 a r11111 | 3 b r33333 | 1 b s11111 | 2 c s22222 |
| 2 b r22222 | 4 c r44444 | 3 d s33333 | 4 e s44444 |
| 5 c r55555 | 6 d r66666 | | |

Computing the join in distributed databases in an efficient way is challenging. The following method is based on the idea of semi-join [20]:

- A first phase computes a *semi-join* by manipulating (and sending on the network) only the join column values and not the (much longer) data. (In general, the semi-join algorithm will use several nodes and leave the result spread over these nodes.) For example, we may obtain

| *Node 1* | *Node 2* |
|---|---|
| (1) 2 b (1) 1 | (1) 5 c (2) 2 |
| (2) 3 b (1) 1 | (2) 4 c (2) 2 |
| | (2) 6 d (1) 3 |

Each number in parentheses represents the home node of the row.

- The second phase, called *join materialization*, replaces the row identifiers with the actual data. This operation is itself a join and requires, if we are not careful, the exchange of a potentially large amount of data. In ARBRE, we developed two techniques that decrease the communication. Depending upon the circumstances, one or the other may be chosen.

*Technique 1*

The novelty of Technique 1 is that the A-node participates in the join materialization. The data are sent to the A-node directly without redistribution. Assume that we arbitrarily establish the order 1, 2, with the intention of first sending the data from node 1, then from node 2. Thus, if the following sequential streams ($R$ stream and $S$ stream) were available to the A-node, together with the result of the semi-join, the A-node could read them in parallel and perform a one-to-one join (a straightforward merge) between the semi-join and these streams:

| *R stream* | *S stream* |
|---|---|
| (1) 2 r22222 | (1) 1 s11111 |
| (2) 3 r33333 | (1) 1 s11111 |
| (1) 5 r55555 | (2) 2 s22222 |
| (2) 4 r44444 | (2) 2 s22222 |
| (2) 6 r66666 | (1) 3 s33333 |

These streams can be constructed by extracting from the semi-join the row identifiers and ordinal positions in which the data are needed and sending this information to the nodes storing the particular rows. (The node identifier comes in handy here.) E.g., consider $R$. The corresponding sequential stream can be obtained by sending

| *Ordinal number* | *Row-id* | *To node* |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 5 | 1 |
| 4 | 4 | 2 |
| 5 | 6 | 2 |

The nodes receive the following information:

| *Node 1* | *Node 2* |
|---|---|
| 2 | 3 |
| 5 | 4 |
| | 6 |

The individual nodes replace these row identifiers with the data, obtaining

| *Node 1* | *Node 2* |
|---|---|
| r22222 | r33333 |
| r55555 | r44444 |
| | r66666 |

Similar streams are prepared for the data in $S$.

The join materialization and the sending of the data to the A-node are done simultaneously. The A-node reads the node identifiers in the semi-join result; e.g., the first tuple is $\langle 1, 1 \rangle$. Thus, the A-node picks up the data for $R$ and for $S$ from node 1. In general, the source nodes are different.

*Technique 2*

A larger reduction in communication is possible when the rows needed from $R$ (without duplication) fit in the memory of the A-node. The result of the semi-join indicates which rows of $R$ are used in the result. The A-node stores these rows in a structure in memory with a direct-access capability based on the row identifier (clearly, without duplication). This makes the building of the $R$ stream unnecessary. The same technique can be used for $S$ and generalizes to $n$-way joins as well.

The proposed algorithm reduces the amount of communication because it eliminates duplication. The ideal case occurs when all the rows needed by the join fit in the A-node memory. Then, the amount of communication for the materialization is proportional to the sum of the amounts of data from the various tables to be joined, rather than a product. The rows are sent to the A-node directly from where they reside in the database, without duplication. If the rows do not fit in memory, Technique 1 ensures that data are also sent directly from the nodes where they reside, although this time, duplication cannot be avoided.

- *SFR: A new distributed join algorithm*

Multiprocessor hash-based join algorithms [21, 22] are desirable for the equijoin of two tables with little or no data skew. However, if the join is not an equijoin, if there are more than two tables to be joined, or if there is significant data skew, a multiprocessor hash-based join may not be desirable or even applicable. The fragment-and-replicate (FR) join algorithm [23] is a useful alternative in these cases, since it handles any kind of join and can perform load balancing. One drawback of the FR algorithm is its high communication cost. We have generalized the FR algorithm and, in so doing, reduced its communication requirements.

The FR algorithm fragments one table by row across several processing nodes and completely replicates the second table at each of these processing nodes. Each node performs a local join and sends its results to the A-node or redistributes the results for any additional processing. The FR algorithm is asymmetric because it treats the two

tables differently: It fragments one table and completely replicates the second one across all nodes. It is possible to reduce communication by using a symmetric fragment-and-replicate (SFR) join algorithm.

The SFR algorithm fragments both tables and replicates each fragment across some, but usually not all, nodes. The replication pattern minimizes the amount of data communicated, yet it does not affect the result of the join operation. Under certain circumstances, the SFR algorithm degenerates into the FR algorithm. Under all other circumstances, the SFR algorithm requires less communication than the FR algorithm.

The differences between the FR and SFR algorithms are best illustrated by example. Assume that 100 nodes are available to join two 5-megabyte (MB) tables. The FR algorithm broadcasts one table to all 100 join nodes and breaks the other table into 100 disjoint fragments, sending each fragment to a different join node. The SFR algorithm, in contrast, breaks each table into ten disjoint fragments and, at least conceptually, arranges the 100 nodes into a $10 \times 10$ square. The $i$th fragment of one table is multicast to all nodes in the $i$th row of the square, while the $j$th fragment of the other table is multicast to the $j$th column of the square. The $(i, j)$th node then joins these two fragments and sends its results to the destination node(s). Each join node in the FR algorithm receives 5 MB + 0.05 MB = 5.05 MB of data, whereas each join node in the SFR algorithm receives 0.5 MB + 0.5 MB = 1 MB of data. In this example, using the SFR algorithm instead of the FR algorithm reduces communication by more than a factor of 5.

In general, the SFR algorithm uses a rectangular arrangement of join nodes rather than a square arrangement. The rectangular dimensions that minimize communication depend on the number of nodes performing the join and on the size of each table. In addition, we have devised a round-robin method that automatically balances the load at the join nodes. The SFR algorithm, its analysis, and the load-balancing enhancement have all been extended to accommodate $N$-way joins. The details of the method may be found in [24].

We implemented the SFR algorithm for a two-way join and measured its performance on ARBRE. In one set of experiments on the join of two tables of equal size, using the SFR algorithm instead of the FR algorithm reduced communication by 40% and improved the response time by 10%. These gains were realized even though only nine join nodes were used; our mathematical analysis predicts even larger reductions in communication as the number of join nodes is increased.

## Transaction management

Using large numbers of nodes in a software database machine can substantially increase the overhead associated with executing a distributed transaction. In this section we present algorithms for reducing the communication and I/O activity associated with atomic "commitment" and the communication associated with global deadlock detection in a database machine. The algorithms exploit the selective centralization of information: Certain information about a transaction is stored at a single node, regardless of what nodes generated the information. Storing information about different transactions at different nodes avoids bottlenecks and helps balance the load.

- *Atomic commitment protocol*

"*Committing*" a transaction makes permanent the updates called for by the transaction, while *aborting* a transaction removes its updates from the system. All nodes at which a transaction updates data need to agree on whether to commit or abort the transaction, in order to maintain the consistency of the data in the system. The sequence of steps the nodes execute in order to decide whether to commit or abort a distributed transaction is called an atomic commitment protocol [1]. An atomic commitment protocol requires I/O activity when a log is forced, since *forcing* the log moves the log tail from volatile storage to nonvolatile, stable storage. The log tail consists of information, recently appended to the log, that has not yet been forced to nonvolatile, stable storage.

An atomic commitment protocol can require many messages and log forces. For example, the classic two-phase commit protocol [25, 26] requires four rounds of messages. For a distributed transaction involving $N$ nodes, each round in this protocol requires $N - 1$ messages; therefore, committing this transaction with the two-phase commit protocol requires $4(N - 1)$ messages. Executing the transaction may require as few as $2(N - 1)$ messages, which means that the communication overhead for atomic commitment may be as large as 200%. The node that ultimately decides whether to commit or abort, called the coordinator, must force its log once, and each of the other $N - 1$ nodes must force its log twice, when a transaction is committed using the two-phase commit protocol. Hence, $2N - 1$ log forces are needed to commit a transaction. Optimized atomic commit protocols exist [27], but their communication and I/O costs are still high.

We developed the coordinator log protocol to reduce the communication and I/O activity associated with atomic commitment. Because of space limitations, we present only an overview of the protocol here. A complete description of the coordinator log protocol and its effect on database crash recovery is in [28].

When several nodes participating in a transaction share a common log, some of the log forces of existing protocols may be converted into append operations without affecting the correctness of the protocol [27]. The coordinator log protocol extends this optimization to its ultimate limit by

forcing each transaction to use only one log, the log of the node coordinating the transaction. All log records that each noncoordinator node generates while executing a transaction accompany its response to the coordinator. The coordinator appends to its log the log records it receives from other nodes as well as the log records it generates while executing its own portion of the transaction. Once the work associated with the transaction has been completed, the coordinator unilaterally commits the transaction by appending a commit record to its log and by forcing its log to nonvolatile, stable storage. Then the coordinator notifies the other nodes that the transaction has committed. Committing a transaction that involves $N$ nodes requires only one log force and $N - 1$ messages. Compared to the two-phase commit protocol, the coordinator log protocol saves $3(N - 1)$ messages and $2(N - 1)$ log forces when a transaction commits.

• *Deadlock-detection algorithm*

Whenever deadlocks are unavoidable, they must be detected and broken. A global deadlock is a deadlock that spans several nodes, and many distributed algorithms exist to detect and break global deadlocks [29]. Unfortunately, the communication required by existing distributed deadlock-detection algorithms is frequently proportional to the number of nodes at which a transaction may be requesting locks. For example, if one transaction is waiting for a second transaction that happens to be setting locks at 100 nodes, the system may need to check whether the second transaction is (indirectly) waiting for the first transaction at each of the 100 nodes. We circumvent this problem by redistributing deadlock information and running a distributed deadlock-detection algorithm on the redistributed information. A detailed description of our approach to deadlock detection is contained in [30]. Below we give a simple introduction.

A deadlock corresponds to a cycle in the transaction wait-for graph (TWFG), in which each vertex represents a transaction and each directed edge represents a transaction waiting for another transaction. The TWFG changes as transactions request and release locks. Before we look for cycles in the TWFG, we centralize the wait-for information about each transaction in the following manner. All edges emanating from a given vertex are sent to a deadlock-detection node that depends only on the transaction associated with the vertex. For example, if there are two deadlock-detection nodes, we could send all edges emanating from "odd" vertices to one deadlock-detection node (call that node ODD) and all edges emanating from "even" vertices to the other deadlock-detection node (call that node EVEN). A distributed deadlock-detection algorithm would use the TWFG edges maintained by the nodes ODD and EVEN and ignore the TWFG edges at the data nodes. The set of nodes at which a transaction may be requesting locks is thus irrelevant to the deadlock-detection algorithm. Edges that disappear from the TWFG when a transaction terminates or releases a lock are also removed from the TWFG maintained by the deadlock-detection nodes.

Any hashing function that maps transaction identifiers to nodes may be used to redistribute the TWFG. The range of the hashing function determines which nodes serve as deadlock-detection nodes. Note that if there is only one deadlock-detection node, our approach degenerates to a centralized approach for global deadlock detection.

Although our approach to deadlock detection can reduce communication, redistributing the TWFG has some cost. Transactions with a small degree of average parallelism suffer from this approach, while transactions with a larger degree of average parallelism benefit. Using a simple mathematical analysis, we examined several distributed deadlock-detection algorithms and learned that benefit was frequently gained when average parallelism is greater than six nodes. Since a hybrid approach is viable, we recommend redistributing wait-for information only for transactions that may have a moderate to large degree of parallelism. The deadlock-detection algorithm must then distinguish between the two classes of transactions.

## Discussion and future work

A message-passing multiprocessor is an appealing configuration for a parallel relational-database system. ARBRE is a prototype that exploits such a configuration. Building and evaluating the prototype taught us several important lessons about parallel processing for distributed databases. In this section we discuss these lessons, review our accomplishments, and describe important areas that need additional exploration.

The most important lesson is that certain database interfaces designed and tuned for serial execution need adjustment for parallel and distributed execution. Adjustment is needed because the introduction of parallelism and distribution affects the frequency with which database facilities are used and the circumstances under which they are used. Consider, for example, the overhead for starting and committing a transaction. In a single-node database, all database operations for each transaction are executed at one node, and the transaction overhead is typically small in comparison to the database work. In a parallel distributed database, each operation in the transaction may execute at a different node, and the transaction overhead at each node may be comparable to the actual database work at that node. This increase in overhead reduces transaction throughput and makes evident the need for a transaction facility with much less overhead.

Another illustration of this lesson involves temporary tables. In a single-node database, temporary tables are

used to hold intermediate results. In a parallel distributed database, temporary tables are also used to hold data either before or after they are redistributed, and a good strategy for evaluating the answer to a complex query may frequently redistribute data in order to exploit parallelism fully. For example, redistribution of data pages can minimize the effect of a poor horizontal partitioning of data, whereas redistribution of records can eliminate the skew introduced by scan predicates or join predicates. Redistribution may also be used to introduce skew to counter an unbalanced load on the system. The resulting increased dependence on temporary tables makes the performance for manipulating such tables an important factor in overall system performance.

In some cases, the adjustment for parallel or distributed operation requires adding functionality rather than improving performance. For example, the access method we used for ARBRE had no intra-transaction parallelism: The access method could not exploit multiple processors in a shared-memory multiprocessor, and it could not overlap processing and I/O for a single transaction. Running several database systems at each node is a quick and easy way to circumvent these two limitations. The only requirement is that the database systems cooperate as independent databases in a distributed database system. The recommended way to enhance a database system is to make it support multiple threads for a single transaction and make it exploit multiple processors in a shared-memory multiprocessor. We chose the former, easier alternative because it required much less work. In hindsight, we believe we made the right choice for a research project.

The second lesson that we learned from the ARBRE prototype is that there is typically no single horizontal partitioning of data that provides good performance for all transactions and queries. The performance of a given transaction or query depends on the way in which data are partitioned. Centralizing small transactions improves their throughput, because each transaction executes on fewer nodes, while decentralizing complex queries improves their response times, because each complex query is executed on more nodes. One way to address this conflict between throughput and response time is to have the horizontal partitioning depend on how many records are related to one another. Consider the ACTIVITY table in our banking application. A personal savings account may have a small number of records in the ACTIVITY table, and these records could be assigned to the node that contains the corresponding ACCOUNT record. Joining the two tables would not require any communication for that account, and sequentially scanning or aggregating the activity records for the account would not take long. The activity for a large corporate account may be thousands of times greater than that for a personal account, so it makes sense to spread the ACTIVITY records for a large corporate account over several nodes. Although this placement of data requires communication when a join occurs, it lets the system scan or aggregate the ACTIVITY records for the corporate account in parallel. The differences between the personal account and the corporate account illustrate how different groups of records must be treated differently. Given the number of related records in a group, a threshold function can determine the number of nodes that should manage the group. Other database researchers have advocated similar approaches [31, 32].

One important goal in designing a parallel distributed database is load balancing. For simplicity, we considered individual queries rather than the entire system load. Because query predicates make the number and distribution of intermediate results unpredictable, our parallel algorithms gather statistics and use this information to redistribute data appropriately.

Another important goal is overhead minimization. Unfortunately, no single technique works for all sources of overhead introduced by parallelism and distribution. We reduce communication whenever possible by exchanging record components rather than entire records, by introducing symmetry into an algorithm, by introducing hashing into an algorithm, and by exploiting existing communication messages. In addition, we reduce logging by centralizing logging on a per-transaction basis.

Query optimization, load balancing, and scheduling are important topics that need further study. One approach to query optimization is to parallelize the best sequential implementation of a query [31, 33], while another approach is to consider parallel strategies during the optimization process. Scheduling must be considered at run time, at compile time, or at both times. In the absence of scheduling, unrestricted dataflow execution can unleash too much parallelism and consume too many resources at one time. Without a priority mechanism, work on the critical path cannot receive favorable treatment.

We believe that these problems are not insurmountable and that using general-purpose processors to implement a parallel database system is a promising approach. There is still much to be learned about parallelism, databases, and their interrelationships.

## Acknowledgments

Gary Hallmark and Georg Zoerntlein helped us develop and exercise the ARBRE prototype. The anonymous reviewers provided numerous suggestions that improved the presentation.

## References

1. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Reading, MA, 1987.

2. Esen Ozkarahan, *Database Machines and Database Management*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.
3. Philip M. Neches, "The Anatomy of a Data Base Computer System," *Proceedings of the 2nd International Conference on Supercomputing: Volume I*, ACM Press, New York, 1987, pp. 102–104.
4. Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah, "A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases," *Technical Report 89.4*, Tandem Computer Inc., Cupertino, CA, May 1989.
5. Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez, "Prototyping Bubba, a Highly Parallel Database System," *IEEE Trans. Knowledge & Data Eng.* **2**, 4–24 (March 1990).
6. David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen, "The Gamma Database Machine Project," *IEEE Trans. Knowledge & Data Eng.* **2**, 44–62 (March 1990).
7. Raymond Lorie, Jean-Jacques Daudenarde, Gary Hallmark, James Stamos, and Honesty Young, "Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience," *IEEE Data Eng. Bull.* **12**, 2–8 (March 1989).
8. *SQL/Data System General Information*, Order No. GH24-5012, available through IBM branch offices.
9. James W. Stamos and David K. Gifford, "Remote Evaluation," *ACM Trans. Programming Lang. & Syst.* **12**, 537–565 (October 1990).
10. William Alexander and George Copeland, "Process and Dataflow Control in Distributed Data-Intensive Systems," *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ACM Press, New York, June 1988, pp. 90–98.
11. Gary Hallmark, "Function Request Shipping in a Database Machine Environment," *Proceedings of the 6th International Workshop on Database Machines*, Morgan Kaufmann Publishers, San Mateo, CA, June 1989, pp. 171–186.
12. Raymond A. Lorie and Jorgen F. Nilsson, "An Access Specification Language for a Relational Data Base System," *IBM J. Res. Develop.* **23**, 286–298 (May 1979).
13. Raymond A. Lorie and Honesty C. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine," *Proceedings of the 15th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Mateo, CA, August 1989, pp. 125–134.
14. Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surv.* **16**, 287–318 (September 1984).
15. S. Lakshmivarahan, Sudarshan K. Dhall, and Leslie L. Miller, "Parallel Sorting Algorithms," *Adv. in Computers* **23**, 295–354 (1984).
16. Betty Salzberg, Alex Tsukerman, Jim Gray, Michael Stewart, Susan Uren, and Bonnie Vaughan, "FastSort: A Distributed Single-Input Single-Output External Sort," *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ACM Press, New York, August 1990, pp. 94–101.
17. Micah Beck, Dina Bitton, and W. Kevin Wilkinson, "Sorting Large Files on a Backend Multiprocessor," *IEEE Trans. Computers* **37**, 769–778 (July 1988).
18. Balakrishna R. Iyer and Daniel M. Dias, "System Issues in Parallel Sorting for Database Systems," *Proceedings of the 6th International Conference on Data Engineering*, IEEE Computer Society Press, Los Alamitos, CA, February 1990, pp. 246–255.
19. Bjorn Arild W. Baugsto and Jarle Fredrik Greipsland, "Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer," *Proceedings of the 6th International Workshop on Database Machines*, Morgan Kaufmann Publishers, San Mateo, CA, June 1989, pp. 127–141.
20. Philip A. Bernstein and Nathan Goodman, "Power of Natural Semijoins," *SIAM J. Computing* **10**, 751–771 (November 1981).
21. James Richard Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Data Base Management," Ph.D. thesis, University of California at Berkeley, May 1981. Available as *Technical Report UCB/ERL M81/33.*
22. David J. DeWitt and Robert Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the 11th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Mateo, CA, August 1985, pp. 151–164.
23. R. Epstein, M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System," *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, ACM Press, New York, May 1978, pp. 169–180.
24. James W. Stamos and Honesty C. Young, "A Symmetric Fragment and Replicate Algorithm for Distributed Joins," *Research Report RJ-7188*, IBM Almaden Research Center, San Jose, CA, December 1989.
25. J. N. Gray, *Notes on Database Operating Systems, Lecture Notes in Computer Science*, Vol. 60, Springer-Verlag, New York, 1978, pp. 393–481.
26. Butler W. Lampson, *Distributed Systems—Architecture and Implementation: An Advanced Course*, Ch. 11, Atomic Transaction, Springer-Verlag, Berlin, 1981, pp. 246–265.
27. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Trans. Database Syst.* **11**, 378–396 (December 1986).
28. James W. Stamos and Flaviu Cristian, "A Low-Cost Atomic Commit Protocol," *Proceedings of the 9th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, October 1990, pp. 66–75.
29. Ahmed K. Elmagarmid, "A Survey of Distributed Deadlock Detection Algorithms," *Sigmod Record* **15**, 37–45 (September 1986).
30. James W. Stamos, "Using Hash Partitioning to Reduce Communication in Distributed Deadlock Detection Algorithms," *Research Report RJ-7098*, IBM Almaden Research Center, San Jose, CA, October 1989.
31. Michael Stonebraker, Randy Katz, David Patterson, and John Ousterhout, "The Design of XPRS," *Proceedings of the 14th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Mateo, CA, August–September 1988, pp. 318–330.
32. Shahram Ghandeharizadeh and David J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Partitioning Strategy for Multiprocessor Database Machines," *Proceedings of the 16th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Mateo, CA, August 1990, pp. 481–492.
33. Michael Stonebraker, Paul Aoki, and Margo Seltzer, "Parallelism in XPRS," *Technical Report M89/16*, University of California at Berkeley, ERL, February 1989.

*Received October 10, 1990; accepted for publication July 8, 1991*

**Raymond A. Lorie** *IBM Research Division, Almaden Research Center, San Jose, California 95120.* Mr. Lorie is a research staff member at the IBM Almaden Research Center, San Jose, California. He has been working on relational database systems since the early 1970s. Between 1973 and 1978, he contributed to many aspects of System R, a full relational database system prototype, and was the main architect of the SQL compiler. He later led the XSQL project for extending relational technology to support nonbusiness applications and the ARBRE project for the exploitation of parallelism in database systems. Mr. Lorie graduated as Ingénieur Electricien-Mécanicien from the University of Brussels, Belgium, in 1959, joining IBM in 1960.

**Jean-Jacques Daudenarde,** *IBM Storage Systems Product Division, 5600 Cottle Road, San Jose, California 95193.* Mr. Daudenarde is currently working on advanced applications for large manufacturing databases at the IBM Storage Systems Product Division, San Jose, California. He joined the IBM France Scientific Center in 1973 and transferred to the IBM Almaden Research Center to work on extensions of the SQL language and on the use of parallelism in database systems. His interests include the application of relational technology to new areas such as engineering, software development, office automation, and image databases. Mr. Daudenarde received an M.S. in mathematics from the University of Rouen, France.

**James W. Stamos** *IBM Research Division, Almaden Research Center, San Jose, California 95120.* Dr. Stamos received the S.B. and S.M. degrees in computer science in 1982 and the Ph.D. degree in computer science in 1986, all from the Massachusetts Institute of Technology. He is currently a research staff member at the IBM Almaden Research Center. Dr. Stamos's research interests include distributed systems, database systems, and programming languages.

**Honesty C. Young** *IBM Research Division, Almaden Research Center, San Jose, California 95120.* Dr. Young received his B.S. degree in electrical engineering from the National Taiwan University in 1978 and his M.S. degree in electrical engineering and Ph.D. degree in computer sciences from the University of Wisconsin at Madison in 1984 and 1985, respectively. He is currently a research staff member at the IBM Almaden Research Center. Dr. Young's technical interests include database systems, computer architecture, parallel processing, and compilers.