DB2 Parallel Edition

by C. K. Baru
G. Fecteau
A. Goyal
H. Hsiao
A. Jhingran
S. Padmanabhan
G. P. Copeland
W. G. Wilson

The rate of increase in database size and response-time requirements has outpaced advancements in processor and mass storage technology. One way to satisfy the increasing demand for processing power and input/output bandwidth in database applications is to have a number of processors, loosely or tightly coupled, serving database requests concurrently. Technologies developed during the last decade have made commercial parallel database systems a reality, and these systems have made an inroad into the stronghold of traditionally mainframe-based large database applications. This paper describes the DB2® Parallel Edition product that evolved from a prototype developed at IBM Research in Hawthorne, New York, and now is being jointly developed with the IBM Toronto laboratory.

Large-scale parallel processing technology has made giant strides in the past decade, and there is no doubt that it has established a place for itself. At this time, however, almost all of the applications harnessing this technology are scientific or engineering applications. The lack of commercial applications for these parallel processors may in part be due to the perceived robustness and usability of these systems. Compared to mainframe systems, large-scale parallel processing systems have not emphasized availability and reliability and have not been supported with adequate software for system management and application development. However, the current generation of massively parallel processor systems, such as the IBM Scalable POWERparallel Systems* (the SP1* and SP2* class of systems), are much more robust and easier to use.

Database management systems (DBMSs) provide important support for commercial applications. Currently there is a rapidly growing trend among businesses to analyze their increasing volumes of transaction data for various types of trends, including sales and purchasing, inventory, and budget. This class of applications, called decision support applications, poses complex queries on the large volumes of data that have been collected from various sources. Single-system (or serial) DBMSs cannot handle the capacity and the complexity requirements of these applications. In addition to decision support applications, there are other new application classes such as data mining, digital libraries, and multimedia that require either large capacity or the ability to handle complexity. The emergence of these applications has fueled the need for parallel DBMS software from commercial vendors.

In the past, a number of research prototypes, including Gamma,[1] Bubba,[2] and XPRS,[3] have tried to understand the issues in parallel databases. These and other projects addressed important issues such as parallel algorithms for execution of important database operations,[4–7] query optimization techniques,[8,9] data placement,[10–13] and database performance.[14–16] The results of these stud-

0018-8670/95/$3.00 © 1995 IBM

ies form a basis for our knowledge of parallel database issues today. However, two major limitations with many of these projects were: (1) Many of the problems were considered in isolation, so the implementation tended to be very simple, and (2) in several cases, people resorted to simulation and analysis because the implementation required enormous effort. Recognizing the importance of an "industrial strength" parallel database system, we initiated a project at IBM Research that has now led to the emergence of the product DB2* Parallel Edition.

DB2 Parallel Edition (DB2 PE) is a parallel database software program that currently operates on AIX*-based parallel processing systems, such as the IBM SP2 system, and will be available in the future on other architecture and operating system platforms. Its shared-nothing (SN) architecture and function shipping execution model (discussed later) provide two important assets: *scalability* and *capacity*. DB2 PE can easily accommodate databases with hundreds of gigabytes of data. Likewise, the system model enables databases to be easily scaled with the addition of system processor and disk resources. The architecture and implementation of DB2 PE provide the opportunity for the best query processing performance.

- The query optimization technology considers a variety of parallel execution strategies for different operations and queries and uses cost in order to choose the best possible execution strategy.
- The execution time environment is optimized to reduce process overhead, synchronization overhead, and data transfer overhead.
- The ACID (a term meaning atomicity, consistency, isolation, and durability—all properties of a transaction) transaction properties[17] are enforced in a very efficient manner in the system to provide full transaction capabilities.
- Utilities such as load, import, reorganize data, and create index have been efficiently structured to run in parallel.
- A parallel reorganization utility (redistribute) is used to effectively correct data and processing load imbalance across different nodes of the system.

It must be noted that companies such as Tandem Computers Incorporated and Teradata Corporation have built and sold parallel database products for a few years.[16,18,19] Teradata's DBC/1012 system is targeted for decision support applications while most of the Tandem systems target high-performance on-line transaction processing (OLTP) applications. Proprietary hardware increases the cost of such systems and also inhibits the development of a full set of application enablers on them. Besides the fact that DB2 PE does not impose such a limitation, we believe that there are several novel aspects that are addressed by DB2 PE that have not been addressed elsewhere. Several of these aspects are highlighted later in this paper.

The next section in this paper describes the general architecture of the DB2 PE system and discusses the merits of the shared-nothing architecture and the function shipping execution model. Following sections discuss the three layers of the system in detail: the user-controlled data layout for optimal performance, the salient features of the DB2 PE query optimization, and the run-time internals of the system. Next we discuss database utilities such as load and unload that are very useful for decision support applications. Finally we present initial performance numbers of some controlled experiments. The paper ends with a discussion of our experience, our conclusions, and directions for future work.

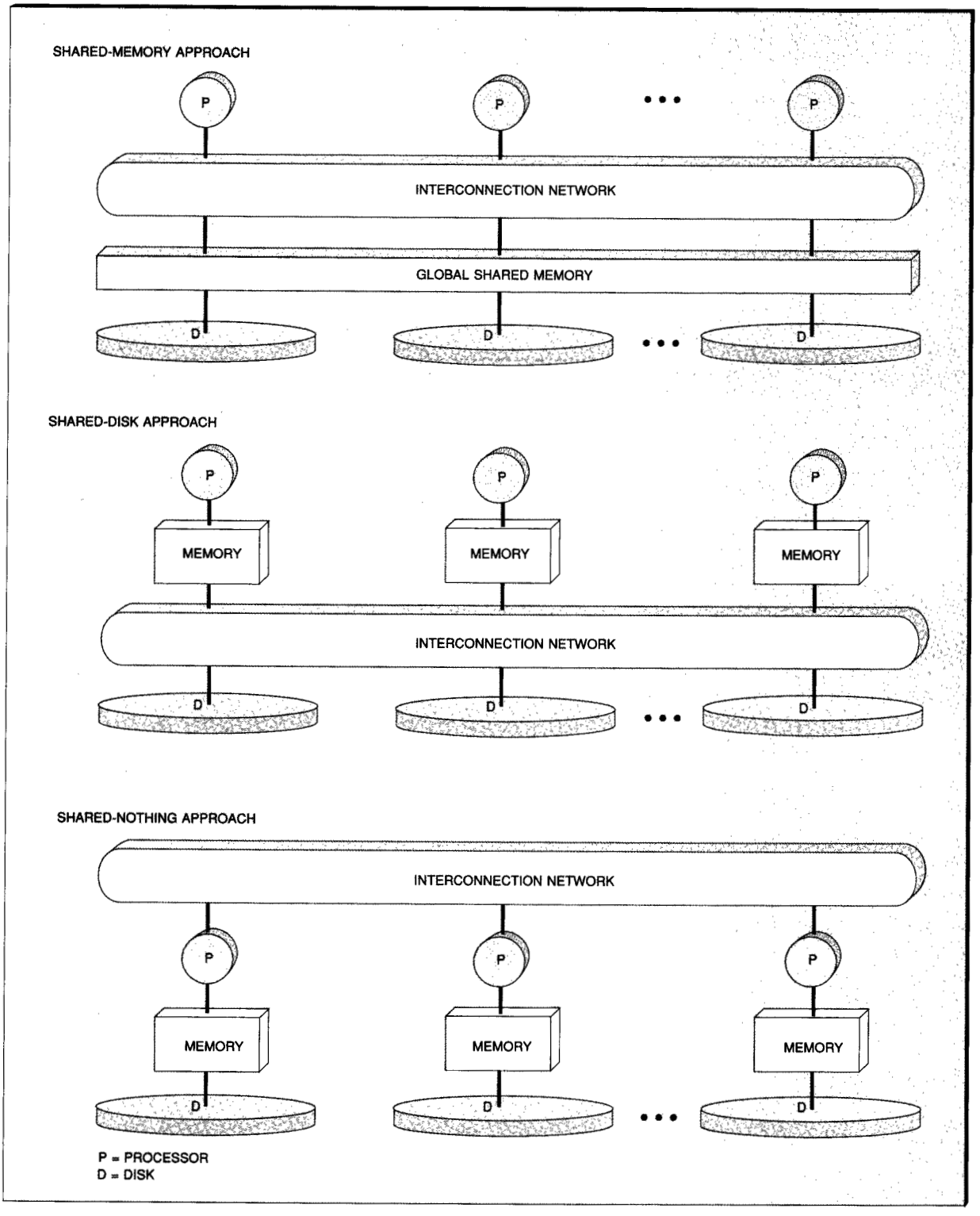
Architecture overview

Parallel database systems can be built on top of different hardware architectures, and given a specific hardware architecture, they can be implemented using one or both of the function shipping and I/O shipping paradigms. This section describes the specifics of each of these architecture and execution models and the choices we made for DB2 PE.

Hardware architecture. Three different approaches can be used in building high-performance parallel database systems,[20] namely, shared-memory (shared everything, tightly coupled), shared-disk (data sharing, loosely coupled), and shared-nothing (partitioned data) approaches. Figure 1 illustrates the three different parallel database system architectures.

In shared-memory systems, multiple processors share a common central memory. With this approach, communication among processors is through shared memory, thus there is little message overhead. In addition, the software required to provide parallel database processing is consid-

Figure 1 Shared-memory, shared-disk, and shared-nothing approaches



erably less complex. Consequently, many commercial parallel database systems available today are based on the shared-memory architecture.

Although shared-memory systems are easier to develop and support, one major limitation is that this approach cannot be scaled to a large number of processors. Research has shown that beyond a certain number of processors, access to memory becomes a bottleneck[21] and the processing speed of the system will be limited by memory access and will not be determined by the speed of the processors. State-of-the-art technology can build memory to support about 500 million instructions per second (MIPS) of CPU power. This implies that a shared-memory system can support less than 10 RISC System/6000* processors of the current generation accessing the shared memory at the same time.

In shared-disk systems,[22] multiple processors, each with its local memory, share a pool of disks. Shared-disk systems avoid the central memory access bottleneck, but introduce the difficult problem of connecting all processors to all disks. This can be especially difficult in the case of a large number of processors and disks. In addition, shared disks present the most challenging task of transaction management because of the need to coordinate global locking activities (without the help of shared memory) and to synchronize log writing among all processors.

With the shared-nothing approach, each processor has its own memory as well as local disks. Except for the communication network, no other resources are shared among processors. Shared nothing does not have the memory access bottleneck problem, nor does it have the problem of interconnecting a large number of processors and disks. The major complexity in supporting the shared-nothing approach is the requirement of breaking a Structured Query Language (SQL) request into multiple subrequests sent to different nodes in the system and merging the results generated by multiple nodes. In addition, shared nothing requires distributed deadlock detection and multiphase commit protocol to be implemented. Researchers and developers have argued that the shared-nothing approach is the most cost-effective alternative and the most promising approach for high-performance parallel database systems.[20,23,24] Many research projects, including Gamma[1] and Bubba,[2] have studied various aspects of parallel database system design based on this approach. Because a shared-nothing system can easily be scaled to hundreds of processors, while shared-memory and shared-disk systems are limited either by memory bus bandwidth or by input/output channel bandwidth, and because a shared-nothing system can grow gracefully, i.e., adding more disk capacity or processing power as needed, DB2 PE adopts the shared-nothing approach.

Function shipping. Because resources are not shared in a shared-nothing system, typical implementations use function shipping, wherein database operations are performed where the data reside. This minimizes network traffic by filtering out unimportant data and achieving good parallelism. A major task in a shared-nothing implementation is to split the incoming SQL request into many subtasks; these subtasks are then executed on different processors (if required, interprocess and interprocessor communication is used for data exchanges). Typically, a coordinator serves as the application interface, receiving the SQL request and associated host variables and returning the answers to the application.

Figure 2 shows some of the task structure for a very simple query. The table T1 is shown horizontally partitioned[13] across all the nodes; thus, based on the function shipping paradigm, the coordinator requests a slave task—one on each node—to fetch its partition of T1 and send the result to it. The results are then returned to the application. In more complicated SQL statements, the task structure is inherently more complex; it is the job of the query compiler to derive the best (i.e., optimal) task structure for the execution of a query. The query compiler determines the function to be performed by each task at run time; the coordinator task is typically instantiated (or specifically determined) on the node to which the application connects, and each slave task is instantiated on the nodes where the data it accesses reside. Thus in Figure 2, there is one coordinator, and there are five instances of slave tasks. In this paper, we use the terms *slave task*, *subordinate task*, *subsection*, and *subplan* interchangeably.

As an example of more complex function shipping, consider the following join query:

```
select T.A, S.A from T, S where T.B = S.B
```

Figure 3A shows a serial execution plan for this query. When tables *T* and *S* are horizontally par-

Figure 2 Task structure for a simple query

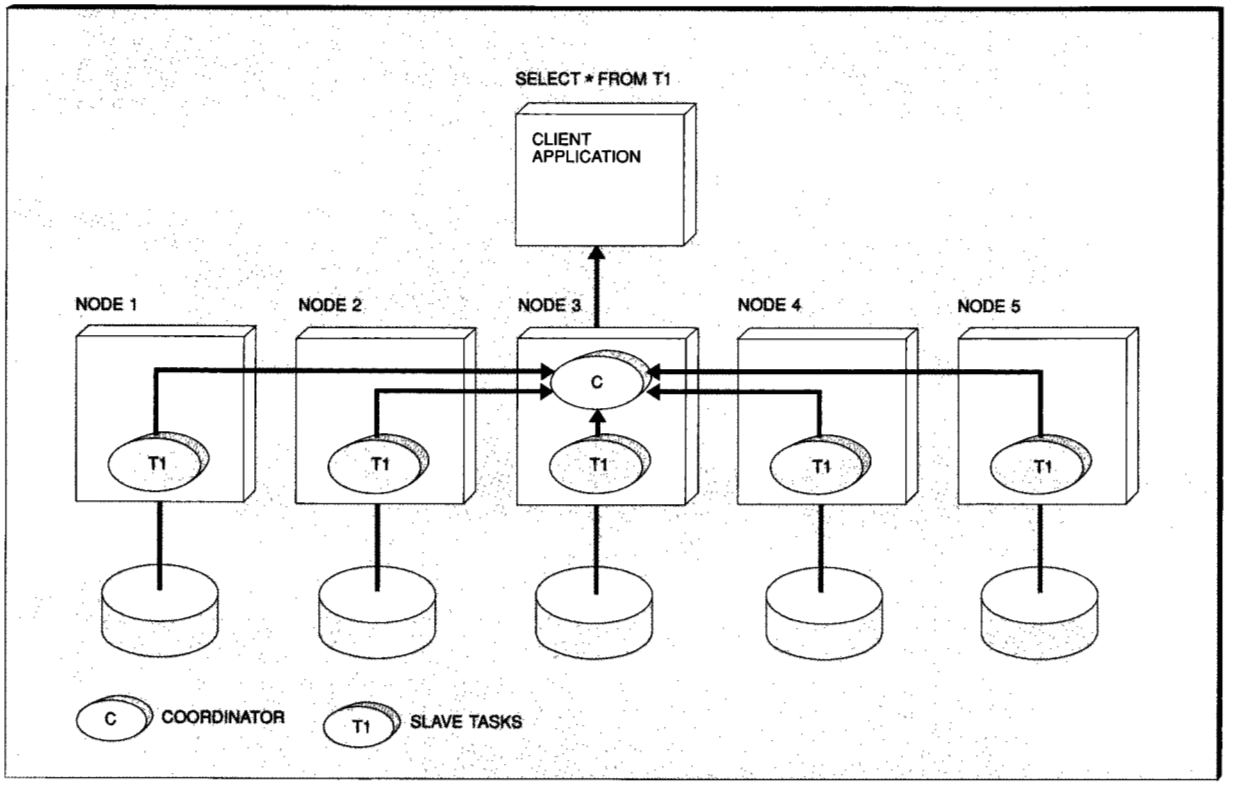
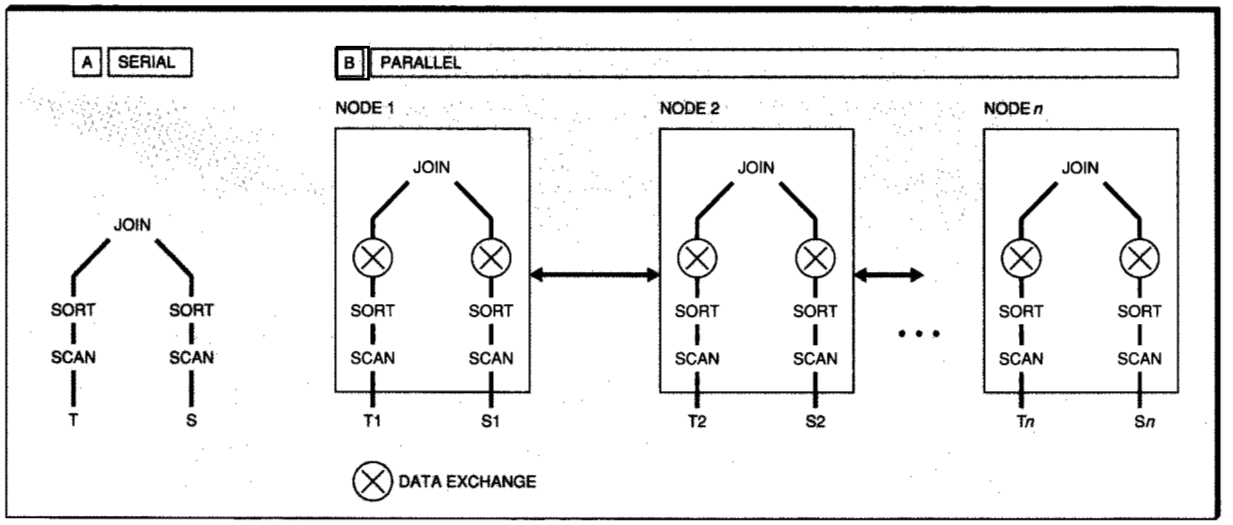


Figure 3 Examples of a serial and a parallel execution strategy



titioned, a possible parallel execution strategy could be the one that maintains the serial structure, but executes each operator in parallel. In Figure 3B, circled Xs indicate data exchanges. (We show later how in a large number of cases even such exchanges can be avoided.) The parallel execution of this query plan may require a coordinator (not shown) and three slave tasks (one slave task scans, sorts, and ships its partition of T to a second slave task, a third slave task does the same against S, and the second slave task performs the actual SQL join function).

One of the advantages that we realized from using function shipping was that we could use much of the existing code; the scans, sorts, joins, etc., shown in Figure 3B are identical to the operators in Figure 3A. A fundamental technology in Figure 3B is the mechanism that glues the nodes together to provide a single-system view to the user. In addition to function shipping, other technologies required to support an environment shown in Figure 3 are (1) generation of parallel plans, (2) streaming of data and control flow, (3) process management, (4) parallel transaction and lock management, and (5) parallel utilities.

Figure 4 describes the system architecture of one node of a DB2 PE system at a conceptual level. Operations on a node are either on behalf of external applications, or internal requests from other nodes in the system. External requests include SQL calls, utilities (load, unload, rebalance, etc.), or other calls (commit, start using database, etc.). SQL calls can be broken into data definition language (DDL) and data manipulation language (DML). DDL is used to define and manipulate the structure of the data (meta-data), such as creating databases, tables, and indices. DML is used for populating, querying, or modifying the data in the database.

Execution of the external and internal requests is primarily driven through the run-time layer. An example function of this layer is to traverse the "operator" graph of an optimized DML statement and to call lower-level functions for executing each operator. The run-time system is also responsible for allocating and deallocating processes for processing local and remote requests.

Below this layer are two distinct components: data management services (DMS), which deal with operations on local data, and communication services, which deal with operations on remote data. DB2 PE built upon and modified the DMS layer of the existing (nonparallel) IBM product DB2/6000*, but the changes are relatively modest. However, the communication services is an entirely new component.

The communication services component provides two types of interfaces, one for control messages and the other for data. The control messages can be either synchronous or asynchronous. All messaging is through a communication manager, which is responsible for multiplexing, demultiplexing, and reliable delivery to other DB2 PE processes.

In addition, the data protection services (DPS) layer of DB2/6000, responsible for locking, logging, and recovery, had to be extended to account for the fact that a transaction can activate more than one process and can involve more than one node. The extensions to DPS use the control message interface of the communication services for global deadlock detection, two-phase commit protocol, and recovery from system failures.

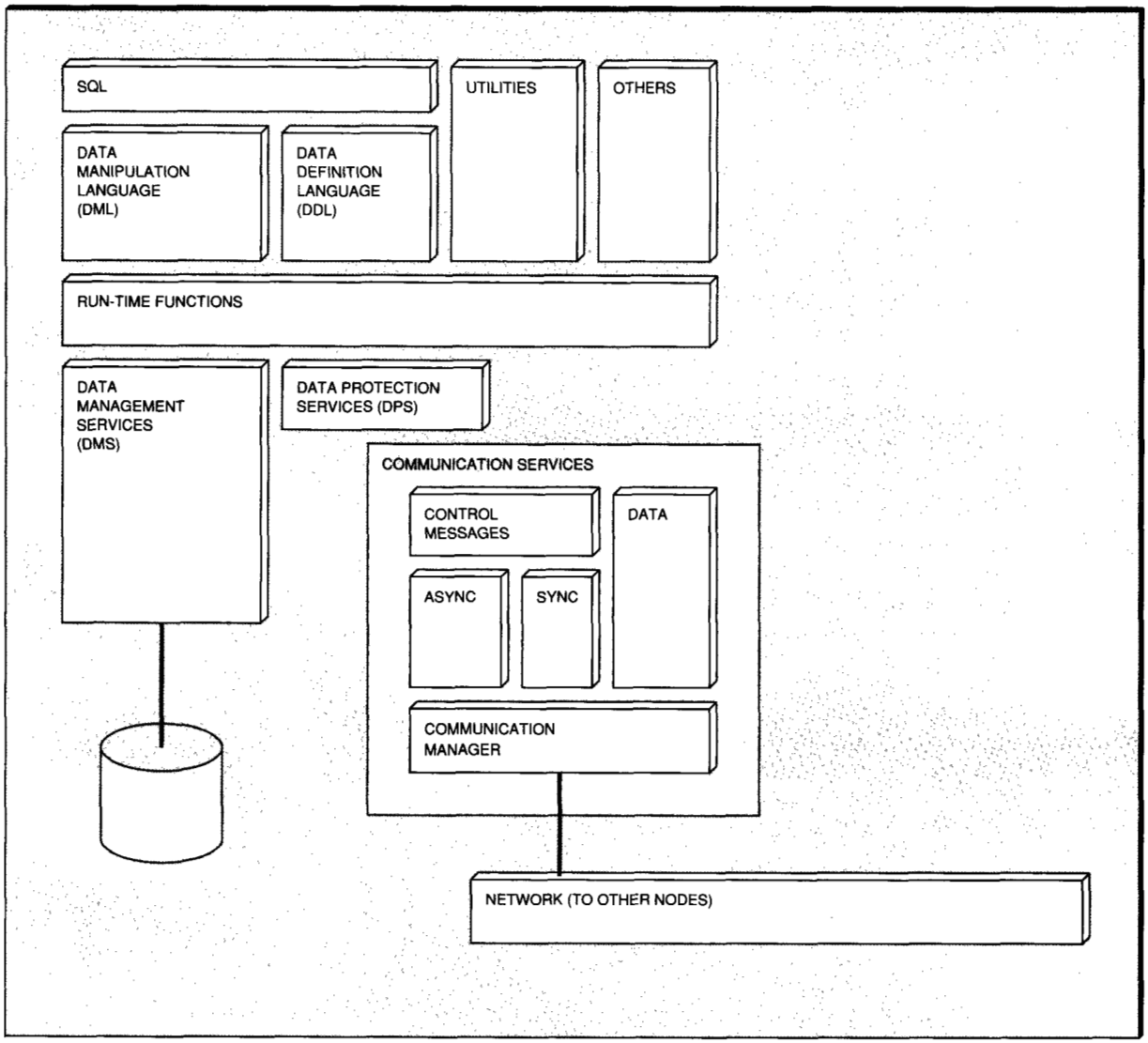
These building blocks of the DB2 PE system are discussed later in more detail. We discuss changes to the DDL and its processing, DML statements and their optimization (including the new operators required to execute them in a function shipping paradigm), changes in the run-time system and the DPS layer, the new communication component and, finally, some of the new parallel database utilities.

Data definition language

DB2 PE provides extensions to SQL in the form of new data definition language (DDL) statements that allow users to control the placement of database tables across the nodes of a parallel system. Before describing the DDL extensions, we provide a general discussion of data placement issues in shared-nothing parallel database systems.

Data placement. The data placement problem is the problem of determining the best storage strategy for the tables in a given database. Data placement in parallel database systems is known to be a difficult problem[13] and several approaches have been taken to solve it.[10,13,25,26] The three key aspects of the data placement problem are *declustering*, *assignment*, and *partitioning*.[13] Declustering refers to the technique of distributing the rows of a single table across multiple nodes. If the rows are stored across all the nodes of the parallel database

Figure 4 Major system components at one node



system, then the table is said to be fully declustered. If the rows are distributed across a subset of the nodes, then the table is said to be partially declustered. Partial declustering subsumes full declustering, and provides more flexibility for assignment of tables. The number of nodes across which a table is declustered is referred to as the degree of declustering of the table. The term *table partition* refers to the set of rows of a given table that are all stored at one node of the shared-nothing system (therefore, the number of table partitions equals degree of declustering).

After choosing the degree of declustering, it is necessary to solve the assignment problem, which is the problem of determining the particular set of nodes on which the table partitions are to be stored. The following issues arise during assignment. Given any two database tables, their assignment may be nonoverlapped, i.e., the two tables do not share any common nodes. Conversely, their assignment may be overlapped, in which case the two tables share at least one node. If both tables share exactly the same set of nodes, then the tables are said to be fully overlapped. Full declustering re-

stricts assignment to be fully overlapped, whereas partial declustering allows for full freedom in assignment of table partitions. Finally, the problem of partitioning refers to the problem of choosing a technique to assign each row of a table to a table partition. Common techniques are round-robin, hash, and range partitioning. In the last two, a set of columns (attributes) of the table are defined as the partitioning keys and their values in each row are used for hash or range partitioning.

Nodegroup DDL. DB2 PE supports partial declustering, overlapped assignment, and hash partitioning of database tables using the notion of *nodegroups*. A nodegroup is a named subset of nodes in the parallel database system. The following example illustrates the use of the nodegroup DDL statement:

```
CREATE NODEGROUP GROUP _1
                ON NODES (1 TO 32, 40, 45, 48)
CREATE NODEGROUP GROUP _2
                ON NODES (1, 3, 33)
CREATE NODEGROUP GROUP _3
                ON NODES (1 TO 32, 40, 45, 48)
```

In the above example, GROUP _1 and GROUP _3 are two different nodegroups, even though they contain the same set of nodes (nodes 1 to 32, 40, 45, and 48). Nodegroup GROUP _2 is partially overlapped with GROUP _1 and GROUP _3 (on nodes 1 and 3).

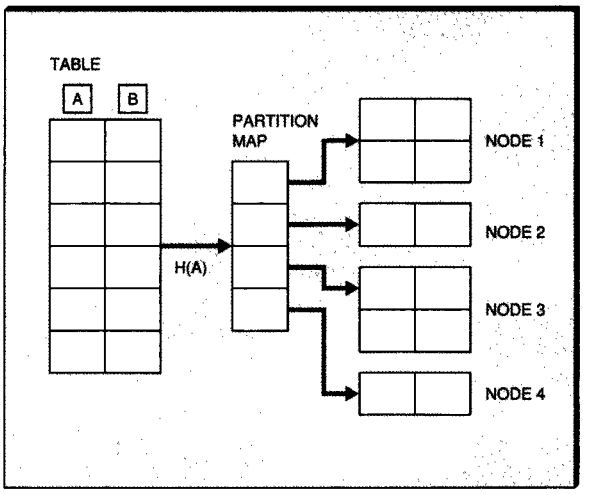
To support scalability, a data redistribution utility is provided to add and drop nodes to or from a nodegroup (see the subsection on data redistribution later in this paper).

Extensions to CREATE TABLE DDL. When creating a table, it is possible to specify the nodegroup on which the table will be declustered. The cardinality of the nodegroup is the degree of declustering of the table. In addition, it is possible to specify the columns to be used for the partitioning key. The following example illustrates the use of DDL extensions to the CREATE TABLE statement:

```
CREATE TABLE PARTS (Partkey integer,
                Partno integer) IN GROUP _1
PARTITIONING KEY (Partkey) USING HASHING

CREATE TABLE PARTSUPP (Partkey integer,
                Suppkey integer, PS_Descp char[50])
IN GROUP _1 PARTITIONING KEY (Partkey)
                USING HASHING
```

Figure 5 The concept of partitioning keys and maps




```
CREATE TABLE CUSTOMERS (Custkey integer,
                C_Nation char[20]) IN GROUP _1
PARTITIONING KEY (Custkey) USING HASHING

CREATE TABLE SUPPLIERS (Suppkey integer,
                S_Nation char[20]) IN GROUP _1
PARTITIONING KEY (Suppkey) USING HASHING

CREATE TABLE ORDERS (Orderkey integer,
        Custkey integer, Orderdate date) IN GROUP _1
PARTITIONING KEY (Orderkey) USING HASHING
```

The partitioning key of tables PARTS and PARTSUPP is Partkey. All tables are partitioned across the set of nodes identified by the nodegroup, GROUP _1.

For each row of a table, the hash partitioning strategy applies an internal hash function to the partitioning key value to obtain a partition (or bucket) number. This partition number is used as an index into an internal data structure associated with each nodegroup (the *partitioning map*), which is an array of node numbers. Each nodegroup is associated with a distinct partitioning map. If a partitioning key value hashes to partition i in the map, then the corresponding row will be stored at the node whose node number appears in the ith location in the map. Figure 5 shows a table with partitioning key A. The hash function $H(A)$ is applied on a tuple's A value and that is used as an index into the partition map to determine the actual node number.

If there are p partitions in the partitioning map and if d is the degree of declustering of a table, then it is necessary that $d \leq p$. In DB2 PE, the value of p is chosen to be 4096. Typically, $d \ll 4096$, thus several partitions are mapped to the same node. Initially, the 4096 hash partitions are assigned to nodes using a round-robin scheme. Thus, each node has at most $4096/d$ partitions of a given table.

In the above example, all tables use the same partitioning map since they are defined in the same nodegroup. In addition, if the data types of the partitioning keys are compatible, then the tables are said to be *collocated*. Since the data types of the partitioning keys of PARTS and PARTSUPP are the same, they are compatible by definition. DB2 PE provides a simple set of rules that define compatibility of unequal data types. The partitioning strategy ensures that rows from collocated tables are mapped to the same partition (and therefore the same node) if their partitioning key values are the same. This is the primary property of collocated tables. Conversely, if rows from collocated tables map to different nodes, then their partitioning key values must be different. Collocation is an important concept since the equi-join of collocated tables on the respective partitioning key attributes can be computed efficiently in parallel by executing joins locally at each node without requiring internode data transfers. Such joins are called *collocated joins* and have the property of being highly scalable (perfectly scalable in the ideal case). Thus, in the above example, the following is a collocated join:

```
select * from PARTS, PARTSUPP
where PARTS.PARTKEY = PARTSUPP.PARTKEY
```

Query optimization

The compiler component of DB2 Parallel Edition is responsible for generating the parallel query execution strategies for the different types of SQL queries. The DB2 PE compiler is implemented on the basis of a number of unique principles:

- Full-fledged cost-based optimization—The optimization phase of the compiler generates different parallel execution plans and chooses the execution plan with the least cost. The optimizer accounts for the inherent parallelism of different operations and the additional costs introduced by messages while comparing different strategies. This approach is similar to the cost-based query optimization performed by System R* for a distributed database environment.[27]
- Comprehensive usage of data distribution information—The optimizer makes full use of the data distribution and partitioning information of the base and intermediate tables involved in each query while trying to choose parallel execution strategies.
- Transparent parallelism—The user applications issuing data manipulation SQL statements do not have to change in order to execute on DB2 PE. Hence, the investment that users and customers have made already in generating applications is fully protected and the migration task for the DML applications is trivial. Application programs written for the DB2/6000 product do not even need to be recompiled fully when they are migrated to DB2 PE; the application only requires a rebind to the parallel database, which generates the least cost parallel plan for the different SQL statements, and, if appropriate, stores them.

The following subsections describe the key features of the query compilation technology in DB2 PE. We describe the important operator extensions that are required for parallel processing, the different types of operator execution strategies, and finally, the generation of the overall parallel execution plan. We use several examples to illustrate these concepts.

Operator extensions. For the most part, parallel processing of database operations implies replicating the basic relational operators at different nodes. Thus, the basic set of operators (such as table access, join, etc.) are used without much change. However, the function shipping execution paradigm introduces two new concepts that are not present in a serial engine:

- Query execution may require multiple logical tasks and each task may be executed across multiple nodes. Consequently, we need operators that the coordinator task can use to control the run-time execution of slave tasks. This operator, called distribute subsection, is described in more detail in a later section.
- As a consequence of multiple processes, interprocess communication operators (notably send and receive) are required in DB2 PE. These operators can have attributes (e.g., send can be broadcast or directed; receive can be deterministic or random).

Partitioning knowledge. In DB2 PE, we are conscious about partitioning in the DDL, data manipulation SQL, and at run time. The partitioning methodology of DB2 PE can be viewed simply as a load balancing tool (by changing the key and partition map, we can adjust the number of tuples on any node); however, by making the compiler and the run-time systems understand it, we have succeeded in improving SQL performance beyond simply load balancing. As mentioned before, an example of this is collocated joins. The compiler, being fully cognizant of table partition keys, nodegroups, etc., can evaluate the costs of different operations (collocated vs broadcast joins, for example, as described later) and thus choose the optimal execution strategy for a given SQL statement. In the case of certain directed joins, the run-time system uses the knowledge of partitioning to correctly direct tuples to the appropriate nodes.

Query optimization and execution plan generation. In this section, we describe the query execution plans as trees of operators separated into tasks. The query execution can be viewed as a data flow on this tree, with sends and receives being used for intertask communication.

A query optimizer typically chooses: (1) the optimal join order and (2) the best method to access base tables and to compute each join. This task is inherently exponential[28,29] and many optimizers use heuristics such as postponing of cross products, left-deep trees, etc., in order to prune the search space. In the case of a parallel database, query optimization is further complicated by: (3) determining the nodes on which operations need to be done (this is called the *repartitioning* strategy and is required because the inner and the outer tables may not be on the same set of nodes) and (4) choosing between system resources and response time as the appropriate metric for determining the cost of a plan.

In DB2 PE, we have made a few simplifying assumptions in order to keep the query optimization problem tractable:

- We keep track, on a per-node basis, of the total system resource accumulated during the bottom-up generation of a query plan. The maximum resources used across all the nodes and the network is a measure of the response time of a query.
- Of all the possible subsets of nodes that can be used to execute a join, we typically consider only a few subsets: all the nodes, the nodes on which the inner table is partitioned, the nodes on which the outer table is partitioned, and a few others.
- In keeping with the DB2/6000 query optimization strategy, we use a *greedy heuristic* while choosing between different parallel join execution strategies. The join execution strategies are described later. The best locally-optimized strategy is the one that survives.

In some queries, the optimal strategy is obvious. For example, consider the following query:

```
select S_NAME, S_ADDRESS from SUPPLIERS
where S_REGION= 'ASIA'
```

If a secondary index exists on SUPPLIERS.S_REGION, then the query plan will use it to restrict the tuples on each node; otherwise each node will have to fetch all its SUPPLIERS tuples and eliminate those that are not from 'ASIA'. The run-time execution strategy is very similar to Figure 2.

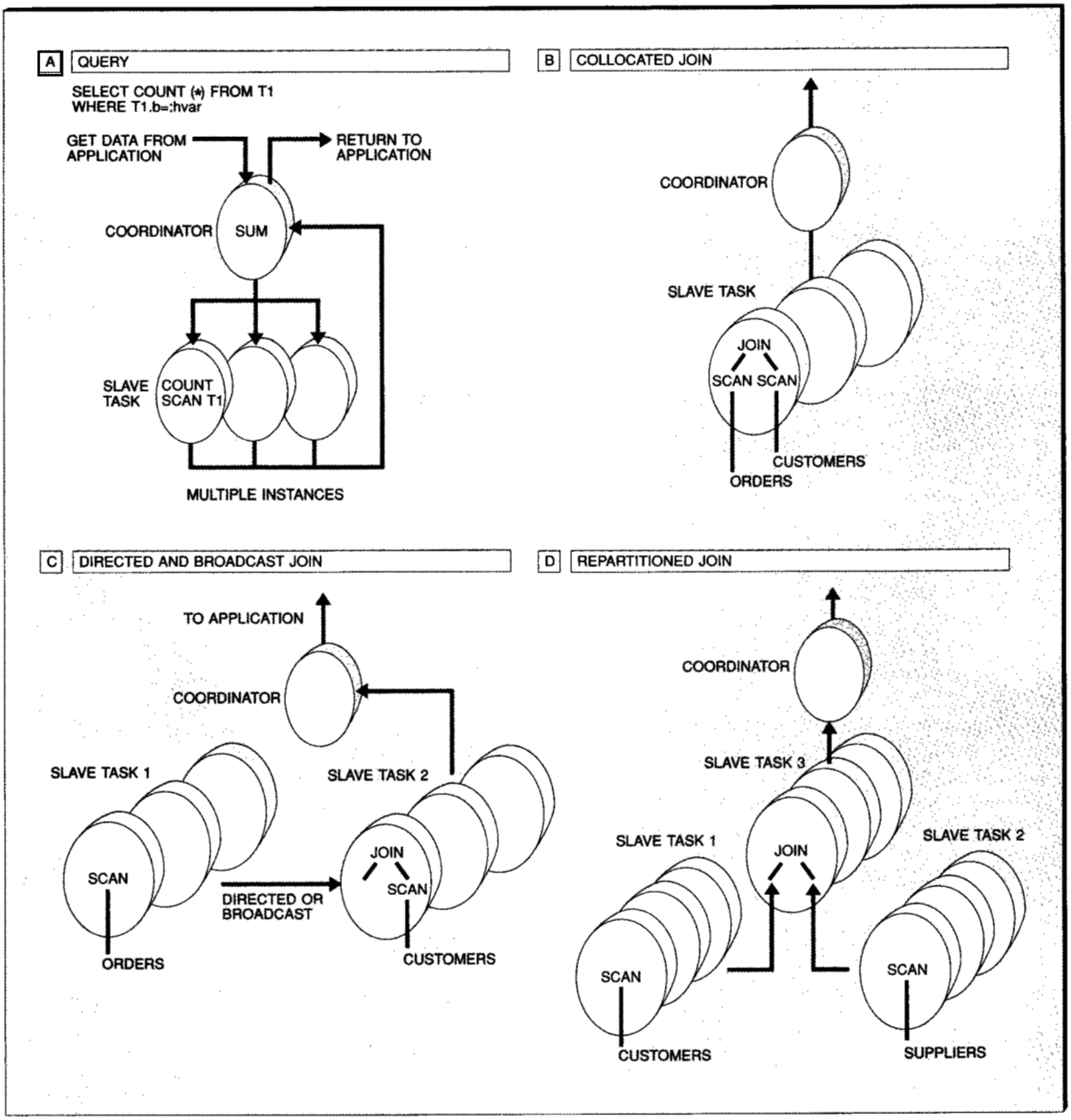
In a more complicated query, such as one shown in Figure 6A, the coordinator not only returns the answer to the application, but also binds in any information required to compute the answer (passing this information to the slave task if required). In this case, an additional feature that DB2 PE supports is to perform aggregation such as count(*) in two steps—the slave tasks compute their local counts and then the coordinator sums the counts and returns the result to the application. The arrows from the coordinator to the slave task represent the passing of all the information required for the slave task to correctly execute (i.e., the query subplan, including the input host variable), and the arrows from the slave task to the coordinator indicate return of their local counts.

In these two examples, the query optimizer had to do little; we now turn to some examples of joins where the optimizer has to actually make decisions.

```
SELECT CUSTNAME from CUSTOMERS, ORDERS
where O_CUSTKEY = C_CUSTKEY
and O_ORDERDATE > '02/02/94'
```

The query selects the names of all customers who placed orders after a certain date. It requires that the ORDERS and CUSTOMERS tables be joined on their CUSTKEY attribute. This join operation can

Figure 6 Task structure for a query and join



be performed by a variety of different strategies in a parallel database environment.

Collocated join. Let the partition keys of the ORDERS and CUSTOMERS tables be CUSTKEY and let them be in the same nodegroup. Then, the records of both tables having a particular CUSTKEY value will reside on the same node. For example, CUSTKEY value of 10 000 may be mapped to node 100 but is the same for both tables. Thus, the join op-

eration can be performed on the local partitions of the two tables. The execution strategy for this is shown in Figure 3A except that the circled cross operators are null operators—no data exchange is required, and the entire operation can be done in one slave process that scans the two tables and joins them and then ships the result to the coordinator. Figure 6B shows the task structure for this join.

Directed join. Let the partition key for CUSTOMERS be CUSTKEY and ORDERS be ORDERKEY. Here, we cannot perform a collocated join operation since records of the ORDERS table with a particular CUSTKEY value could reside on all nodes. The compiler recognizes this from the partitioning information of the CUSTOMERS and ORDERS tables. It then considers a few execution strategies, the foremost of which is the directed join.

The optimizer recognizes that the CUSTOMERS table is partitioned on CUSTKEY. So, one efficient way to match the CUSTKEYs of ORDERS and CUSTOMERS is to hash the selected ORDERS rows using its CUSTKEY attribute and *direct* the rows to the appropriate CUSTOMERS nodes. This strategy localizes the cost of the join to partitions at each node and at the same time tries to minimize the data transfer. Figure 6C shows the compiled plan for this strategy.

Broadcast join. Consider the following query between the CUSTOMERS and SUPPLIERS table.

```
SELECT CUSTNAME, SUPPNAME, C_NATION
from CUSTOMERS, SUPPLIERS
where C_NATION = S_NATION
```

The query tries to find customers and suppliers in the same region. Let the partitioning key for CUSTOMERS be CUSTKEY and that of SUPPLIERS be SUPPKEY. Note that C_NATION and S_NATION could have been the respective partition keys of the two tables; however, CUSTKEY and SUPPKEY are used more often in queries and are more likely candidates. Given this, the optimizer cannot try to localize the join operation on the C_NATION and S_NATION attributes. Hence, a strategy of broadcasting the selected rows of either table to all the nodes of the other tables is considered. The broadcast essentially causes one table to be materialized fully at each node containing a partition of the other table. Now, a join at all nodes will produce the complete result of the query. Figure 6C also shows a broadcast join (with the ORDERS table being replaced by SUPPLIERS) and the arrow connecting slave task 1 to slave task 2 being of type broadcast as opposed to directed.

The broadcast join operation is relatively expensive both in terms of network cost and join processor cost. However, there are instances where this strategy is still very useful. These instances include situations where one of the joining tables is much smaller than the other, when there is an index on a joining attribute, or in nonequijoin situations.

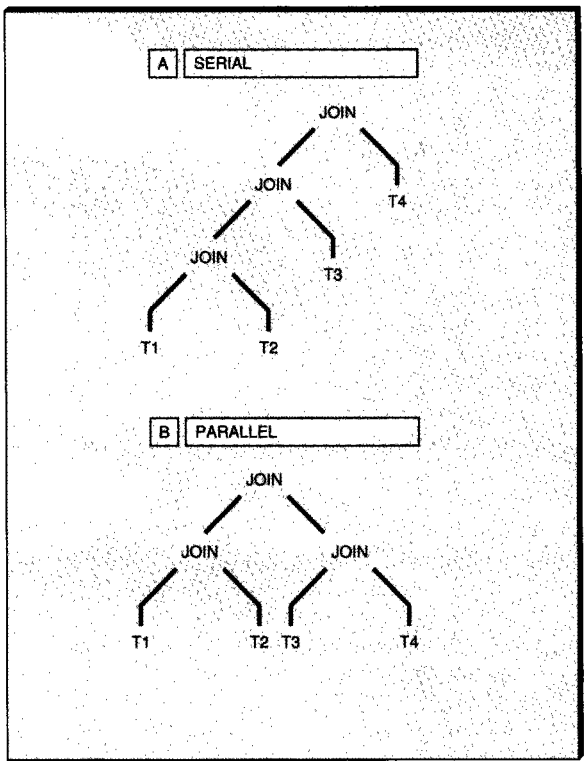
Repartitioned joins. We also consider a repartitioned strategy of join execution in cases such as the query described previously. In this strategy, the optimizer decides to explicitly repartition both tables on their joining attributes in order to localize and minimize the join effort. In the example query previously described, the optimizer will repartition the CUSTOMERS table on C_NATION and the SUPPLIERS table on S_NATION on some common set of nodes. The repartitioned tables can then be joined in a collocated join fashion at each node. Figure 6D shows the repartitioned join strategy.

The repartitioned join requires message traffic to redistribute rows of both tables involved in the join. Once redistributed, the join processor cost is similar to the collocated join case.

Cost-based optimization. One of the most important features of the optimizer is that it uses cost estimates when deciding among different execution choices. This is to be contrasted with an optimization technique that heuristically decides to go with a particular strategy. For example, given a join operation, the optimizer estimates the cost of each of the join strategies previously described and chooses the one with the least cost estimate for a given join step. The cost basis makes the optimizer decisions more robust when choosing between strategies such as broadcast or repartitioned joins.

Cost estimation also enables the optimizer to choose the best query execution plan in a parallel environment. It accounts for the messaging costs incurred by operations. Most importantly, estimation tries to influence parallel processing of different parts of the query whenever possible. Figure 7 shows two different types of query execution

Figure 7 Different execution strategies in serial and parallel environments



plans for a four-way join query. Assume that tables T1 and T2 are collocated on a nodegroup, while T3 and T4 are collocated on an entirely different nodegroup in a parallel environment. An optimizer for a serial DBMS could choose the strategy in Figure 7A, because all the operations are performed in the same node and that is the best serial strategy (possibly influenced by indexes, sort orders, etc.). However, the DB2 PE optimizer may try to favor the parallel plan represented by Figure 7B since more work can be performed in parallel and the partitioning of tables for the two lowermost joins are optimal. Thus proper "parallel cost measures" are critical for parallel query optimization.

Parallelism for all operations. A guiding principle in the compiler design has been to enable parallel processing for all types of SQL constructs and operations. For the sake of brevity, only a list of other operations and constructs where we apply parallelism while generating the execution strategies is provided.

- Aggregation—The ability to perform aggregation is required at individual slave tasks and may be required at a global level.
- Set operations—We consider collocated, directed, repartitioned, or a global strategy akin to the join strategies previously described.
- Updates—The ability to perform inserts with subselect, updates, and deletes in parallel is required.
- Subqueries—We consider collocated, directed, and broadcast methods of returning subquery results to the sites where the subquery predicate is evaluated.

DB2 Parallel Edition run time

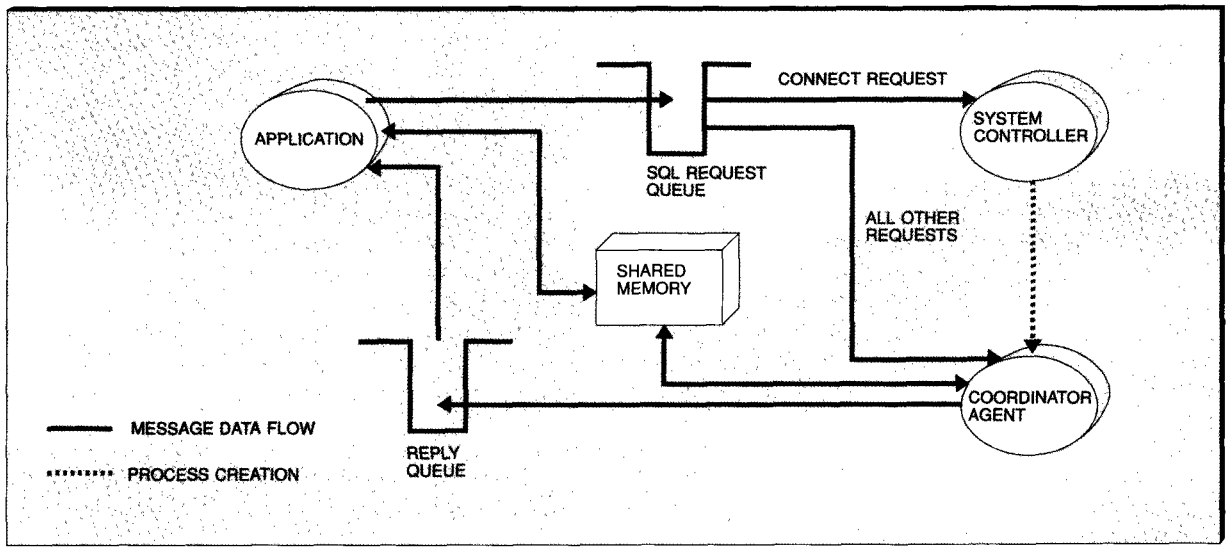
In order to execute a query plan or a DDL statement in parallel, DB2/6000 run-time mechanisms had to be augmented considerably. The following new components were added to provide interprocess (processes may be on different nodes) communication:

- Control services—These services deal with interprocess control message flow (start and stop of processes, error reporting, interrupts, parallel remote procedure call, etc.).
- Table queue services—These services deal with exchange of rows between DB2 PE agents across or within a node and are responsible for the correct execution of the data flow operators connecting different slave tasks.
- Communication manager—This service performs the actual routing of messages using the underlying communications protocol.

In addition, several existing components had to be modified for DB2 PE. They include the interpreter (the component that drives the execution of a query plan), deadlock detector, lock manager, transaction manager, etc. In this section, we describe the new components, as well as the modifications to the existing ones.

Control services. When an application connects to a database (see Figure 8), a special process (or an agent) called the *coordinator* is created. This process is responsible for executing database requests on behalf of the application, returning any results through the reply queue and shared-memory area. In the serial case this is all there is; but in the parallel case multiple processes need to be created to execute requests. These processes (*agents*) are organized into a network of producer-consumer pro-

Figure 8 Application and coordinator agent



cesses. The data flow through table queues, which are described in the following section.

The query subtask (subsection) executed by the coordinator distributes the other subsections to the appropriate nodes to be executed. Along with every request, it sends connection information for the table queues and any host variable information that may be required. There are separate *distribute* operators in the coordinator for each subsection. Typically, the compiler can make static decisions about where a particular subsection needs to be instantiated (generally based on the nodegroups of the tables that the particular slave task accesses). However, DB2 PE is capable of choosing nodes at run time, based either on the query structure (e.g., a query select * from T1, where T1.a = host-variable with T1.a being the partitioning key of T1, allows the table access to happen only on the node that contains the partition for T1.a = host-variable), or on the current I/O and processing activity of the nodes (for those subsections that are not tied to specific nodes, e.g., those that execute repartitioned joins).

Creating a process can be an expensive operation. For long-running queries, this process is amortized over many millions of instructions, but for shorter queries this can be considerable overhead. Therefore several optimizations have been done to decrease this overhead.

The purist view of process management is that the "abstract database process" is created to execute its subsection, then terminates when the subsection is finished. In DB2 PE a pool of processes is initiated when the database manager is started at a node, and any of the processes can be reused by different subtasks of the same application or different applications. The process pool can significantly reduce process creation overheads for short queries.

Certain sequences of SQL operations have a portion that is inherently state-based. For example, cursor-based updates depend on a previous statement to position the cursor. Therefore DB2 PE provides *persistent* agents that remain assigned until the application disconnects. After such an agent starts working on behalf of a request, it remains attached to the request's state until the request completes. An alternative we explored was disconnecting a process from a subsection during idle times, such as when waiting for a message to be received or sent. The extra overhead of saving and restoring state seemed to overwhelm the system savings. The parameters determining this trade-off may change as system speed increases disproportionately to the disk swap time.

In addition to the requests to start or stop processes, the control component also handles requests to stop or interrupt processes, returns con-

trol replies to applications, and provides parallel remote procedure call support for low-level database manager functions (such as "retrieve long field" or "load access plan from catalog").

Table queue services. The interprocess data flow constructs are called table queues, and are similar to Gamma's split tables.[1] However, they are richer in functionality. Intuitively, they can be thought of as a temporary table. The most important difference is that they do not need to be completely built before rows can be fetched from them. They are in effect streams of rows for interprocess communication, controlled by the receiver's ability to handle the traffic (back pressure). They have a send operator (table queue build) and a receive operator (table queue access).

Table queues are designed to provide maximum flexibility to the SQL compiler and optimizer in generating the most efficient parallel plan. The plan specifies that a certain table queue is to connect two subsections of a plan. However, each subsection can be instantiated on more than one node. Therefore a single table queue can have more than one sender and more than one receiver; thus a communication path exists between multiple producer and multiple consumer processes (see Figure 9A). Although it should be thought of as one entity, it is implemented by multiple connections, between each sender/receiver pair. Each sending process can send each row to every receiver process, or to just one process depending on the partitioning information associated with the table queue.

There are many attributes associated with table queues. An important one of them is Broadcast vs directing—does one row at the sending end go to all the receivers, or only to one? See Figure 9B for an example of a directed table queue. In this figure, all "A" values are directed to the first receive node from all sending nodes, all "B" values to the second, and so on.

Communication subsystem. The parallel communications component is layered in a similar fashion to the rest of the run time. It accepts messages (either control messages or buffers of rows) and guarantees ordered delivery between nodes (or between processes on the same node). It also performs multiplexing and demultiplexing of messages between nodes of a DB2 PE system. Underneath this layer, it uses the delivery layer, which can be the User Datagram Protocol/Internet Protocol (UDP/IP), Transmission Control Protocol/Internet Protocol (TCP/IP), or proprietary high-speed switch interface.

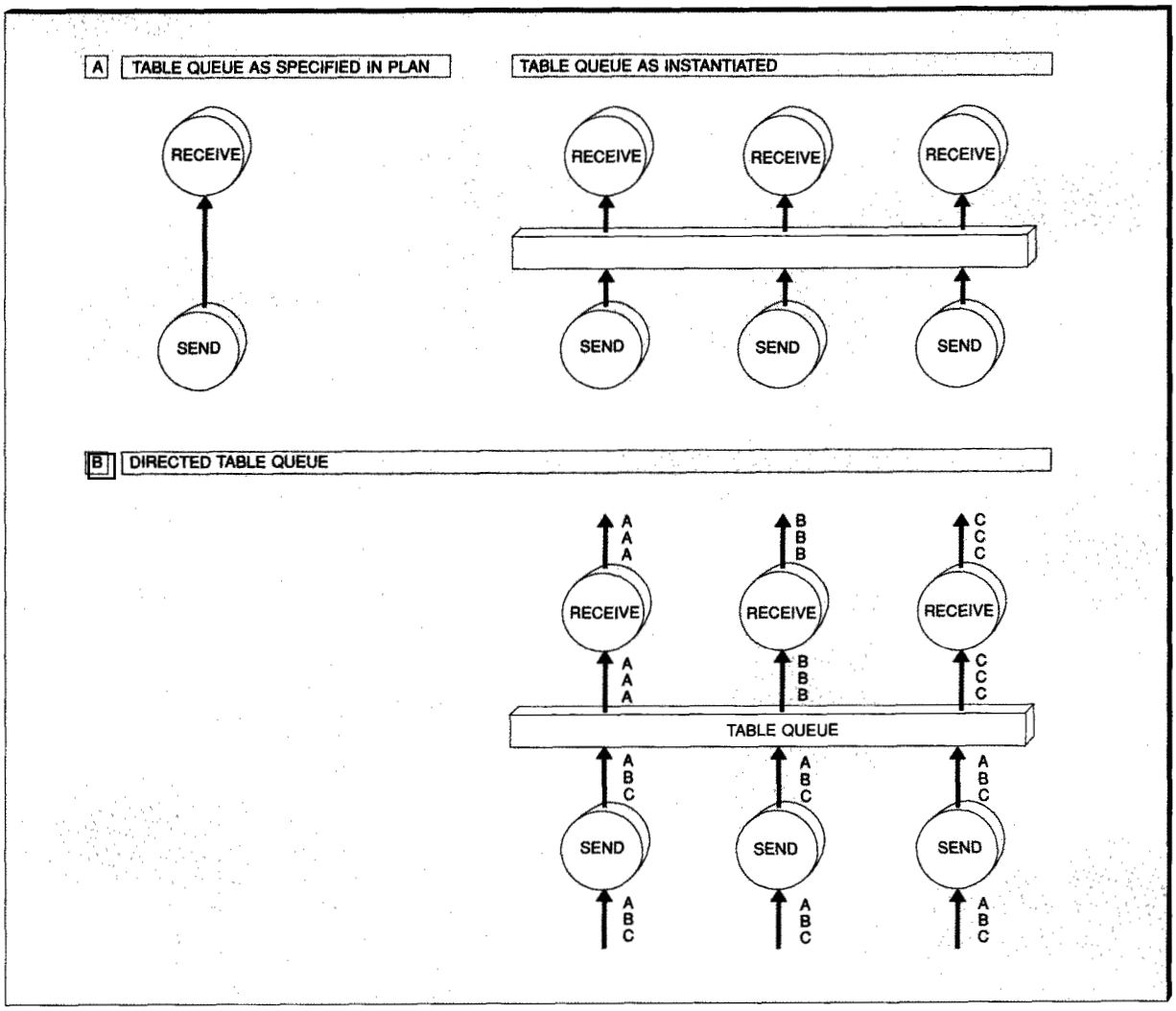
Because a message can be sent before the process waiting for it is ready to receive it, the communications layer must hold messages until the receiving process is ready. Some of the issues that had to be solved here were determining if the process to which a message was directed had already terminated, in which case the arriving (or "in-flight") message should be dropped; or whether the process had not yet been created and so the message should be kept. The solution to this question relied on the communication manager to guarantee *order of arrival* of messages. That is, if message A is sent from sender S on node 1 to receiver R on node 2, then it must be received by R before R can receive any other message sent later by S to R. (Exceptions are made for the class of "interrupt" messages.)

Interrupt and error handling. The assumption inherent in the serial database manager is that either the application is busy or the database system is busy, but not both at the same time. Further, the database system is busy handling only one request per application. In DB2 PE, not only can the database system be active concurrently with the application, it can be processing more than one query on behalf of the same application. Multiple cursors may be open at any given time. Each fetch of a cursor returns a single row, but there could be processes on many nodes working to retrieve rows for that cursor. A process may be initiated at a node when a cursor is opened (execution of an open cursor statement by the application) and it remains until it has completed processing of all its data or until the cursor is closed by the application.

So although a row may be ready to be fetched, another node may have had an error. The semantics had to be defined for when the error indication is returned to the application, whether it should be returned as soon as possible, as late as possible, or in its "natural" order. DB2 PE implements the "as soon as possible" policy, but it is by no means clear this is always the best. There are many other examples of similar problems, where serial semantics just cannot be maintained, e.g., interrupt handling, forward recovery to current time, etc.

Concurrency control and recovery. A parallel database must ensure that individual transactions can be serialized and that recovery from software and

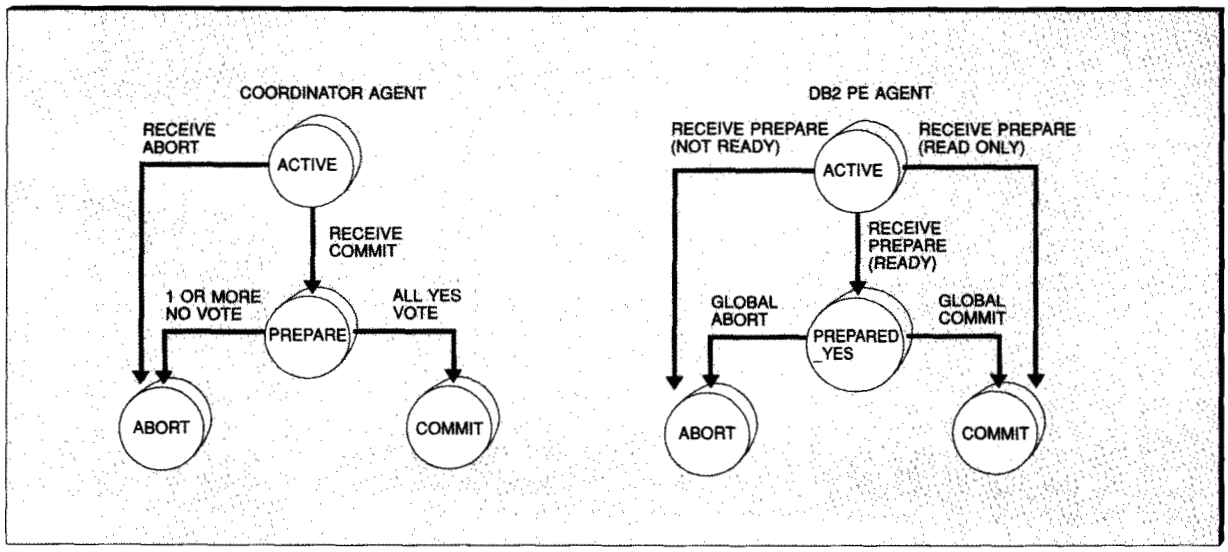
Figure 9 Table queues



hardware failures are possible. Serializability (serialization of transactions) is obtained by using a standard two-phase *locking* protocol, and recovery from failures (durability) is obtained by using a two-phase *commit* protocol. This section describes the specifics of each of these subjects.

Two-phase commit protocol. One important property of database systems is to guarantee that either all actions of a user transaction take effect or none take effect. Since a transaction can be executed on multiple processors concurrently in a parallel database system, it is much harder to provide the all or nothing property. To guarantee this property, most parallel systems adapt a two-phase commit protocol that includes a prepare phase and a commit or abort phase. The two-phase commit protocol may result in *blocking* if the coordinator fails after it has received votes but before it sends out an outcome. When blocking occurs, participants will hold or lock resources for a long time, resulting in significant degradation in system performance. Three-phase commit protocol[30] has been proposed to remedy the blocking problem. But because a three-phase commit protocol imposes much higher overhead than a two-phase protocol,

Figure 10 Transaction state transition in DB2 PE



and the blocking problem can be "resolved" by system administrators, none of the existing commercial systems currently supports the three-phase protocol.

There are three variations of the two-phase commit protocol: presumed nothing, presumed commit, and presumed abort.[31] DB2 PE adopts the presumed abort protocol that assumes a transaction has aborted when the state of the transaction was inquired by any subordinate nodes and the state cannot be found in the in-memory transaction table. Figure 10 shows the transition of transaction state in DB2 PE. When a transaction starts, a coordinator agent is activated to coordinate the execution of the transaction. Subordinate agents (DB2 PE agents), if needed, are activated by the requests sent from the coordinator agent. Before processing a commit or rollback request, the transaction at both coordinator and subordinate nodes is in the active state.

DB2 PE maintains a transaction node list for every active transaction that contains the node numbers of all nodes participating in executing the transaction. When a coordinator agent receives a commit request from an application, it sends out prepare-to-commit requests to the PDB request queues of all nodes recorded in the transaction node list, including the coordinator node itself. At this point, the coordinator agent enters the prepare state.

Upon receiving the prepare-to-commit request, the DB2 PE agent controller at the coordinator node is responsible for stopping active agents associated with the committing transaction. However, the coordinator itself is responsible for processing the prepare-to-commit locally. Notice that there is no prepare log written at the coordinator node before starting the prepare phase in DB2 PE. At a subordinate node, a prepare-to-commit request is processed by an active agent associated with the transaction if one exists. Otherwise, a new agent is selected by the DB2 PE agent controller to process the request. The commit agent first checks the transaction state stored in the local transaction table. If the transaction encountered any error and thus cannot be committed, it will vote "no" to the coordinator and enters the abort state. Otherwise, it will reply "yes" and enters the prepared state if it has modified its local database. If a participant does not update its local database and is ready to commit, it will reply "read-only" and enters the commit state.

If everyone votes "yes" or read-only, the coordinator commits the transaction and informs all participants who have voted "yes." At this point, the transaction state changes to commit at the coordinator node. Otherwise, the coordinator aborts the transaction and forwards the decision to all subordinate nodes that voted "yes." All actions performed for the transaction at all nodes are rolled

back (undone) and the transaction state is changed to abort.

Concurrency control. Parallel database systems need to maintain consistent global locking across all nodes because a database object (record, table, index, etc.) may be accessed by multiple nodes concurrently and deadlocks may occur among nodes. This requirement posts a significant challenge to parallel database system designers. In shared-memory and shared-disk systems, data can be accessed by multiple nodes concurrently. In order to maintain a consistent global locking, a node needs to obtain read and write permission either explicitly or implicitly from all other nodes before reading or writing a data object for which it does not already own an appropriate access permission—hence the requirement for some form of global lock management. In DB2 PE, each processor accesses only the portion of the database that it owns locally. Consequently, a processor does not have to request access permission from remote processors before accessing its local data; thus a global lock table is not required. However, DB2 PE does require a distributed deadlock detector to check for global deadlocks.

In DB2 PE, a lock table and local deadlock detector are created for each database and node to maintain locking information and to resolve conflicts among lock requests for a given database. A transaction may have multiple processes active on its behalf and each process requesting a lock will be assigned a separate lock request block. When two processes of the transaction make a lock request to the same object, the one lock request block per process design uses more memory space. However, it simplifies the design in processing lock conversion requests and lock release requests before the end of a transaction. The local deadlock detector is implemented as a separate process and awakens periodically (becomes active) to scan the local lock table and to build the local wait-for graph. It then sends the local wait-for graph to the global deadlock detector (GDD) for processing.

Global deadlock detection is also implemented as a separate process. There is one global deadlock detector per database opened per DB2 PE system. Currently, a transaction is not allowed to access multiple databases at the same time and thus one GDD per database is the most efficient method. The GDD process resides on a preconfigured node. On a user configurable time interval, local deadlock detectors send their local wait-for graphs to the GDD. The GDD merges the graphs received and builds a global wait-for graph based on the transaction identifier that is unique across all nodes in a DB2 PE system. After the completion of building the global wait-for graph, the GDD goes through it to find deadlock cycles in the graph. When a cycle is detected, one or more transactions are selected and rolled back to break the cycle. When a transaction has been selected as a deadlock "victim," its coordinator agent (process) is informed by the agent requesting the lock and the coordinator agent will send a rollback request to its subordinate agents (processes) to undo the action of the transaction.

Database utilities

DB2 PE provides a variety of utilities to manage the parallel database system. Some of the important utilities are described in the following subsections.

Data loading. The load utility allows bulk loading of database tables from flat files. To support applications requiring very large database sizes (hundreds of gigabytes and higher), DB2 PE provides efficient ways of loading large volumes of data into the database. Data can be loaded in parallel into a single table by invoking the load utility at each of the nodes that contains a table partition for the given table. Typically, the input data are stored in a single flat file. A data partitioning utility as well as application programming interfaces provided with the database system can be used to partition an input file into multiple files, one per table partition. The partitioned files can then be loaded in parallel. In addition, at each node, the load utility reads the input data file and creates data pages in the internal format used by the database engine. These pages are directly appended to the existing database file, thereby greatly increasing the speed of the load utility.

Adding nodes to the system. DB2 PE supports scalability by allowing incremental addition of nodes to the shared-nothing parallel system. Thus, a user can start with a system configuration that is sufficient to handle current storage and performance requirements and add new nodes as the size of the database grows. New nodes can be added to increase storage capacity as well as performance. The command, Add Node, allows users to add nodes to the parallel database system configuration and to initialize the node for use by any da-

tabase. Once added, a node can be used by a database by including it in one of the nodegroups in the database (either the CREATE or REDISTRIBUTE NODEGROUP statements can be used for this purpose). Since DB2 PE supports partial declustering of tables, the set of all tables for a given database may reside only on a subset of the nodes in the system. However, an application can connect to any database from any node in the system, regardless of whether that node contains data pertaining to that database.

The Drop Node command can be used to verify whether a node can be dropped from the database configuration. If the node to be dropped is currently in use by a database, then the node should not be dropped. The Redistribute Nodegroup command (described in the later subsection on data redistribution) should be used to remove any data from this node before dropping it.

Creating a database. Normally, issuing the Create Database command ensures that the database is defined across all the nodes that are currently in the system. Similarly, the Drop Database command drops the database definition from all nodes. However, there are situations in which one may wish to create and drop the database only at a single node. For example, the Add Node command described above implicitly performs a create-database-at-node operation for each existing database. Also, in case the database at a particular node happens to be damaged, the Drop Database At Node command allows the user to drop only the database at that node rather than dropping the entire database across all the nodes of the system. Since DB2 PE supports node-level backup and restore (see the later section on backup and restore), after dropping a database at a node, the database backup image can be used to restore the database at that node (and roll forward the logs, if necessary).

Data reorganization. As a result of insert, delete, and update activity, the physical layout of database tables may change. Insertions may result in the creation of overflow data blocks and, as a result, the disk pages containing data belonging to the table may no longer be stored contiguously. On the other hand, deletions may create gaps in disk pages, thereby resulting in an inefficient utilization of disk space. If a table is partitioned across a set of nodes, insert and delete activity may also result in table partitions at some nodes having more data than those at other nodes, thus creating a skew in data distribution. Also, in many decision support applications, the database size increases with time. Thus, it may be necessary to increase the degree of declustering of a table in order to accommodate the additional data. Finally, even if the size of the database remains the same, the workload may change, thereby requiring a change in data placement.

In all of the above situations, data reorganization utilities can be used to manage the physical storage of the table. The following subsections describe the data reorganization utilities available in DB2 PE.

Table reorganization. The Reorg utility can be used for compaction and reclustering of database files at each node. The Reorg operation executes in parallel across all the nodes that contain a table partition for a given table. The file in which the database table is stored is reorganized by creating a new file without any page gaps and overflow blocks. If the operation completes successfully on some nodes but not on others, then the table partitions remain successfully reorganized at the nodes where Reorg succeeded.

This is an example of an operation where the atomic commit semantics of the database operation have been relaxed. If the operation were to be atomic, then upon failure, the Reorg would have to be undone at all the nodes where it completed successfully. However, the Reorg operation may be time-consuming and undoing it may be even more expensive. In addition, consider the case when Reorg succeeds on, say, 60 nodes but fails on 1. It is more beneficial not to undo the operation. In this case, the operation returns an error message but is not undone since there is no penalty if some partitions are reorganized while others are not. On the other hand, the nodes at which the partitions were reorganized would benefit from the resulting file compaction.

Data redistribution. The partitioning strategy used to partition tables may, in some situations, cause a skew in the distribution of data across nodes. This can be due to a variety of factors, including the distribution of attribute values in a relation and the nature of the partitioning strategy itself. At initial placement time, it is possible to analyze the distribution of input attribute values and obtain a data placement that minimizes skew. However, data skew may be reintroduced over the database life-

time due to insertion and deletion of data. DB2 PE provides a data redistribution utility to redistribute the data in a table in order to minimize skew.

For a given nodegroup, the data redistribution operation determines the set of partitions of the partitioning map that should be moved in order to obtain an even distribution of data across the nodes of the nodegroup. The default assumption is that the data are evenly distributed across the 4K partitions; thus, if the partitions are evenly distributed among the set of nodes, then the data are also assumed to be evenly distributed across the nodes. The user may override this default assumption by providing a *distribution file* that assigns a weight to each of the 4K partitions. In this case, the redistribution operation will attempt to redistribute partitions among nodes such that the sum of the weights at each node is approximately the same.

If a nodegroup contains several tables, then redistributing only one table and not the others will result in a loss of collocation among the tables. In order to preserve table collocation at all times, the redistribution operation is applied to all the tables in the nodegroup and each table is redistributed in turn. If a redistribute operation does not complete successfully, it is likely that some tables in the nodegroup have been redistributed while others have not. In this case, the operation can be completed by issuing the Redistribute command with the *restart* option. It is also possible to issue the Redistribute command with a *rollback* option, in order to undo the effects of the failed redistribution. The Redistribute Nodegroup command is an on-line operation that locks only the table that is currently being redistributed. All other tables in the nodegroup are normally accessible.

The data redistribution utility also permits users to redistribute data by specifying a *target partitioning map* for a given nodegroup. Data redistribution of all tables in the nodegroup using the target positioning map is initiated through the application programming interface. This interface can be used to achieve "custom" redistribution of tables, e.g., send all rows with a particular partitioning key value to a particular node, create skewed distributions, etc. The current data distribution across partitions and nodes can be determined using two new SQL scalar functions, namely, PARTITION and NODE. These functions return the partition number and node number to which a given row in a table is mapped.

The following example illustrates how the new SQL functions can be used to obtain the distribution of the rows of a PARTS table:

Query 1:

```
CREATE VIEW Partition_Nums(Pnum) AS
  SELECT PARTITION(PARTS)
  FROM PARTS;

SELECT Pnum, COUNT(*)
FROM Partition_Nums
GROUP BY Pnum
ORDER BY Pnum;
```

Query 2:

```
CREATE VIEW Node_Nums(Nnum) AS
  SELECT NODENUMBER(PARTS)
  FROM PARTS;

SELECT Nnum, COUNT(*)
FROM Node_Nums
GROUP BY Nnum
ORDER BY Nnum;
```

The output of Query 1 is a set of rows where each row contains the partition number (0 to 4095) and the number of rows of the table that map to that partition. The output of Query 2 is a set of rows where each row contains the node number and the number of rows of the table that map to that node.

Backup and restore. The degree of parallelism achieved during backup and restore of a database is determined by the number of backup devices available. The DB2 PE backup and restore design allows each node in the system to be backed up independently. Thus, data from several nodes can be backed up simultaneously, if multiple backup devices are available. The backup utility creates a backup image of the entire database partition resident at a given node.

At restore time, it is necessary to ensure that the database partition that is being restored is in a consistent state with respect to the rest of the nodes in the system. This can be achieved by either restoring all nodes in the system using backup images that are known to be consistent, or by restoring the single node and rolling forward logs to a point in time where the database state is consistent across all nodes. DB2 PE supports the ability to roll forward logs across nodes to a specific point in time.

High availability. High availability is supported by the use of HACMP (highly available cluster multiprocessor)[32] software. The HACMP software provides transparent takeover of the disk and communications resources of the failed node. System nodes are paired together and each pair has access to twin-tailed disks (disks connected to two nodes). If one of the processors in a pair fails, the other processor can take over and the system can continue to operate. To enable use of HACMP software, the database engine has been designed to allow the situation where a single processor executes multiple copies of the database engine. In other words, multiple *database nodes* or *logical nodes* are mapped to the same physical node. While this method provides quick takeover of a failed node, there may be an impact on performance due to increased load on the takeover processor. In many decision support applications, it is not essential to provide instant takeover capability, whereas it is important not to degrade overall system performance. Thus, it may be acceptable to have a particular node become inaccessible, for example for ten minutes, in order to be able to recover from a failure of that node without any subsequent performance penalty. This can be achieved by configuring one or more spare nodes in the system that can take over on behalf of any failed node. When a node fails, its database files are copied to the spare node (access to the disks on the failed node is available due to twin tailing) and the spare is now restarted as the original, failed node. In this scenario, only the node that failed is inaccessible for a brief period of time while the remaining nodes in the system are still operational.

Performance monitoring and configuration management. Database monitoring tools allow users to identify performance bottlenecks and take appropriate action to relieve the bottlenecks. DB2 PE provides a database monitoring facility that allows users to gather data on resource consumption at the database manager, database, application, and individual process levels. These data are collected at each node and can be used to identify bottlenecks at individual nodes. To obtain a global picture of the performance of the entire system, it is necessary to combine performance monitoring data across all nodes. A performance monitoring tool is being developed as a separate product for this purpose.

The database manager provides several configuration parameters at the database manager and individual database levels, that can be used to tune the performance of each database and the database manager as a whole. For example, users can control the size of buffers, maximum number of processes, size of log files, etc. These parameters can be set independently at each node, thereby allowing users to tune the performance of each individual node. Thus, the configuration parameters can be adjusted to account for differences in hardware capacities, database partition sizes, and workloads at each node.

Results

We have performed a number of internal IBM and customer benchmarks on DB2 PE and a brief synopsis of these results is presented here. The results are divided into three categories:

- Stand-alone numbers—This basic metric includes capacity and load times.
- Speedup—This metric measures the performance of queries and utilities as we increase the number of nodes in the system while maintaining the same database size.
- Scaleup—This metric measures the performance of the system as the database size, the number of concurrent users, and the number of nodes are scaled proportionately.

The system configuration for many of the benchmarks has been the IBM SP2 or SP1 systems. The systems have ranged from 8 nodes to 64 nodes depending upon the database requirements of the individual benchmarks. Typically, each node has 128 or 256 megabytes of memory and 2 to 48 gigabytes of disk capacity. The nodes are interconnected using a high-speed switch. In some of the benchmarks, we have only used the slower Ethernet as the interconnect path.

The results that are described in this section were measured using early untuned drivers of the DB2 PE software, and specific hardware configurations (often having a single disk per node). As such, the results obtained in a different hardware or software environment may vary significantly. Users of these results should verify if they are applicable for their environments. The absolute values of a number of different metrics are likely to be different in the final product.

Stand-alone metrics. Table 1 describes the important stand-alone metrics based on results of bench-

marks performed to-date. One of the foremost metrics we would like to present is *system capacity*. When stating capacity, we must differentiate between user data size (the size of flat files containing the data in normalized form), database size (the space occupied once the data have been loaded in DB2 PE, relevant indexes created, and any required denormalized tables created), and disk capacity (the total disk space used to support the database workload, including internal work areas, interim load files, etc.). We have benchmarked applications with over 100 gigabytes (GB) of user data, databases of over 250 GB, and systems with more than 600 GB of disk space. One of the tables in the database has been as large as 84 GB and contained over 2 billion rows. We expect to support configurations in the terabyte size. To put this into perspective, even mainframe relational databases are rarely over 200 GB in size. Some of the measurements were done using tables larger than the 64 GB limit of many relational database management systems.

Another very important metric is *data load* times. Our fastload utility is able to load data at rates of up to 2 GB/per node/per hour. The dataload utility runs in parallel at all nodes, hence it demonstrates a completely linear speedup of load rates. In a 32-node system, one could load at the rate of 64 GB per hour.

Before loading, the data must be declustered and sent to the appropriate node. The utility used to decluster data (data splitter) is flexible and can be modified by the user in situations where fine tuning is required. The data splitter can be executed on a variety of IBM operating system platforms including Advanced Interactive Executive* (AIX), Multiple Virtual Storage (MVS), and VM. In most cases we also divided the input data so as to run the splitter in parallel. The output of the splitter must then be sent to the appropriate node for loading. In certain benchmarks this was done by sending the data in file format using FTP (File Transfer Protocol), while in other benchmarks the output from the splitter was piped directly into the load program. In most cases, the connectivity between the system containing the source data and the target database system was the limiting factor on the entire database load process. For a 100 GB database on a 46-node SP2 system, the elapsed time for partitioning all the data, loading the data, and creating indexes on the tables was just 12 hours.

Table 1 Stand-alone metrics

Data size	100 GB and larger
Database size	250 GB and larger
Table size	84 GB (2 000 000 000 rows)
Total disk space	over 600 GB
Data splitting	2 GB per node per hour
Data load rate	2 GB per node per hour

Table 2 Database description for speedup experiments

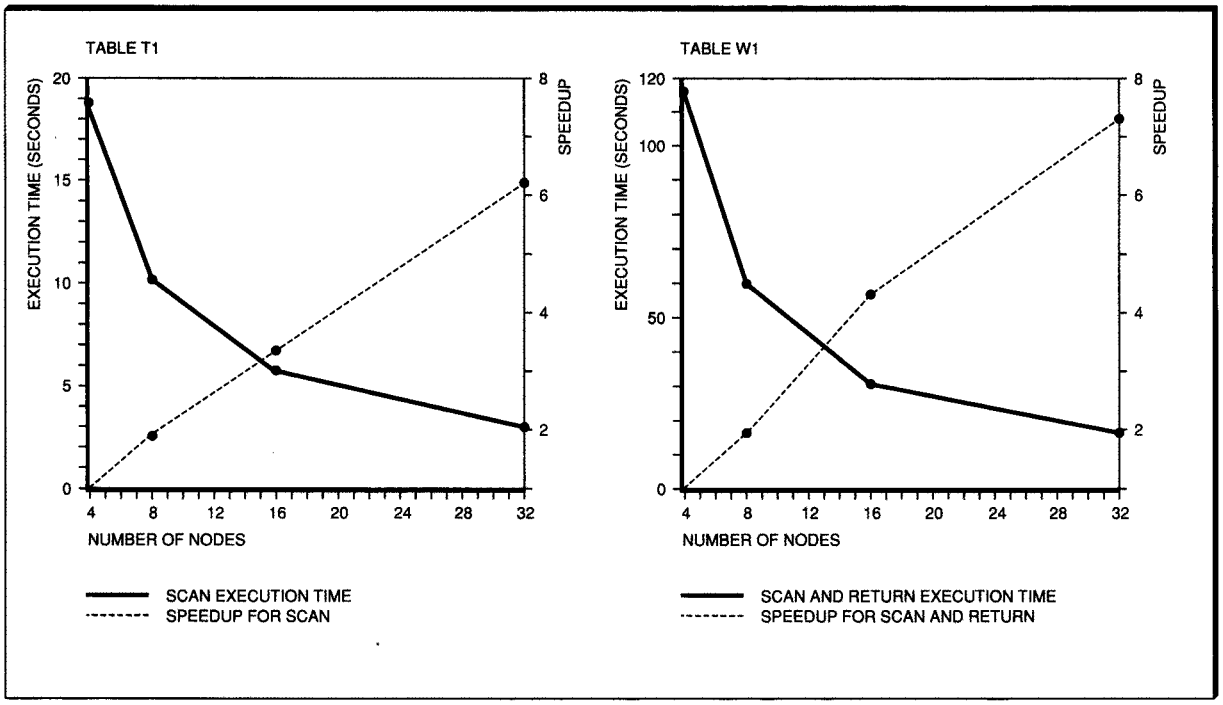
Table	No. of Rows	Row Size	Total Size
S1	100 000	100	10 MB
T1	1 000 000	100	100 MB
T2	1 000 000	100	100 MB
W1	1 000 000	1000	1 GB

The final stand-alone metric is *availability*. In these benchmarks, we have tried to maintain several spare nodes for replacement in the event of node failures. Due to the decision support nature of the benchmarks, only a few nodes (those containing database catalogs) in the system may need the use of special availability mechanisms such as twin-tailing of disks. For all other nodes, in the event of failure, the data residing on the failed node can be reloaded onto the spare node and the spare node is then used as a replacement node. We have been able to accomplish this task in a time that is only dependent on the data load rate for the node. For example, on the 100 GB database on 46 nodes, this task was accomplished in less than two hours.

Speedup results. For speedup, we present results from an internal benchmark performed on 4, 8, 16, and 32 node SP2 systems.[33] Table 2 describes the database configuration used. The database consists of four main tables (S1, T1, T2, W1) and each table contains a primary key along with other non-essential attributes. The S1 table contains 100 000 rows, while the T1, T2, and W1 tables contain a million rows, respectively. S1, T1, and T2 tables contain rows with a size of 100 bytes, while the W1 table has a maximum row size of 1000 bytes and an average row size of 560 bytes.

Scan performance. Figure 11 shows the execution times and the speedup of parallel scan operations on tables T1 and W1 returning 1 row to the application. The y axis on the left shows the execution times, while the y axis on the right measures speedup that can be a maximum of 8. The scan of

Figure 11 Execution times and speedup of parallel scan returning 1 row of T1 and W1 tables



the T1 table exhibits linear speedup, i.e., the ratio of the execution times is *exactly* the inverse of the ratio of the number of nodes, when the number of nodes is increased from 4 to 8. Beyond this point, the speedup becomes sublinear, due to the smaller size of the table. In contrast, the scan of the W1 table exhibits linear speedup up to 16 nodes and only then becomes slightly sublinear. If the table scans had been performed using more nodes, the execution times would eventually flatten out when the table partitions at each node become small enough so that the overhead of initiating the scans at the different nodes offsets the performance gain from the parallel scan. This figure illustrates that the parallelism benefits are bounded by the sizes of the tables for any operation.

Figure 12 illustrates the performance of a parallel scan operation on the W1 table that returns 10 percent of the rows to the application. The execution times improve as the system size is increased but the speedup is quite sublinear. The reason for this has to do with the processing performed at the coordinator node in fetching 100 000 rows (10 percent) of the data and returning it to the application. Amdahl's Law effectively limits the maximum performance improvement from such queries due to the *serial bottleneck*. To overcome this bottleneck, the application must be parallelized. One simple way of doing this on DB2 PE is to divide the application into multiple tasks, each running on a separate coordinator. The division can either be based on range of data or be such that each task operates on a subset of the database nodes. The issue of parallel applications is further discussed in a later section on experiences and observations.

Insert, update performance. Figure 13 shows the execution times for performing insert into a temporary table of 1 row, 1 percent of the rows, and 10 percent of the rows of the T1 table. The figure is plotted using a logarithmic scale (base 2) on both the x and y axes. In such a graph, linear execution time curves (with appropriate slope) indicate linear speedup. All three curves show near linear speedup gains with increasing system size. We are able to parallelize both the insert as well as the subselect operations of this statement resulting in linear speedup of the statement across different nodes.

Figure 14 shows the execution times of update column operations on 1 percent, 10 percent, and 100 percent of the rows of the S1 table. Once again, the execution times decrease linearly with increasing system size. The 1 percent and 10 percent update curves show somewhat anomalous behavior at 32 nodes. We conjecture that the relatively small number of updates at each node of the 32-node system (approximately 300 for 1 percent) makes the execution times really dependent on the parallel scan times for 32 nodes. Both these results illustrate the extremely parallel query execution strategies that DB2 PE is able to generate for insert, update, and delete SQL statements. The parallel insert was particularly useful in benchmarks when interim results were saved in tables for later analysis or when denormalized tables were created from normalized ones.

Index create performance. Figure 15 shows the execution times of a secondary index creation on the 100 000-row S1 table and the 1 000 000-row T1 table. Both curves illustrate close to linear performance improvement, indicating that the create index operations are very efficiently parallelized in DB2 PE. The reader should note that there is no difference between primary indexes and secondary indexes in our system due to the function shipping model of execution. However, the same is not true for other parallel database processing systems, where secondary indexes are global and cannot be efficiently parallelized.

Scalability results. We present three different types of scalability results:

1. Results from performance experiments as the database size scales from 10 GB to 100 GB on the same number of nodes
2. Results from performance experiments as the database size and the system size are scaled proportionately
3. Results from performance experiments as the number of concurrent users on the system is increased

The results for all three cases have been obtained from customer benchmarks.

Table 3 shows the performance of DB2 PE on a 46-node SP2 system for 10 GB and 100 GB versions of a scalable database. The results are shown for a variety of complex queries on the database. The scaling ratios for the different queries varies be-

Figure 12 Execution times and speedup of parallel scan returning 10 percent of the rows on W1 table

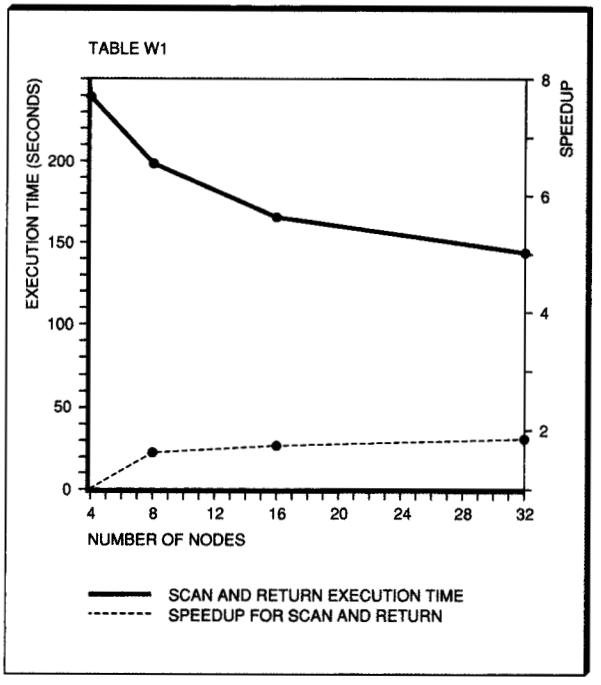


Figure 13 Execution times of 1 row, 1 percent, and 10 percent insert with subselect statements

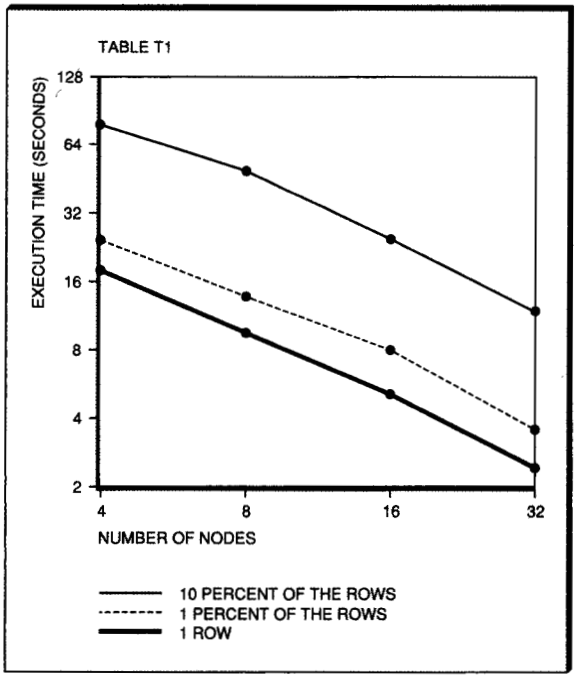


Figure 14 Execution times of 1 percent, 10 percent, and 100 percent update with subselect statements

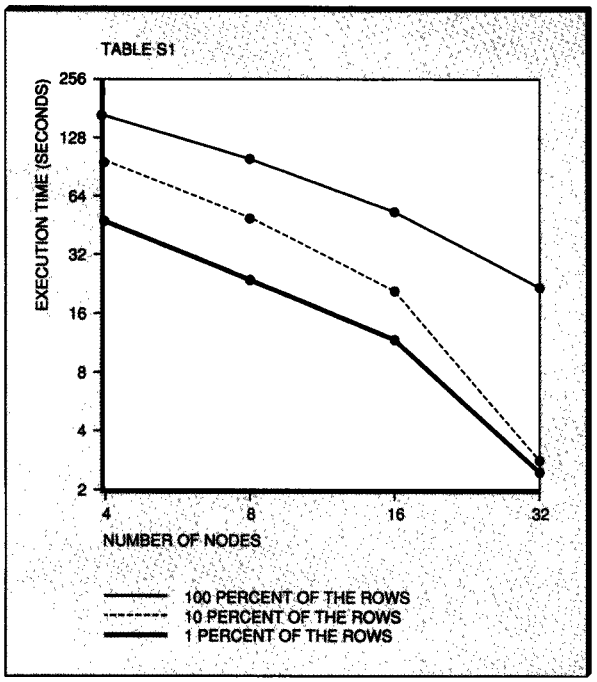


Figure 15 Execution times of create secondary index statements on the S1 and T1 tables

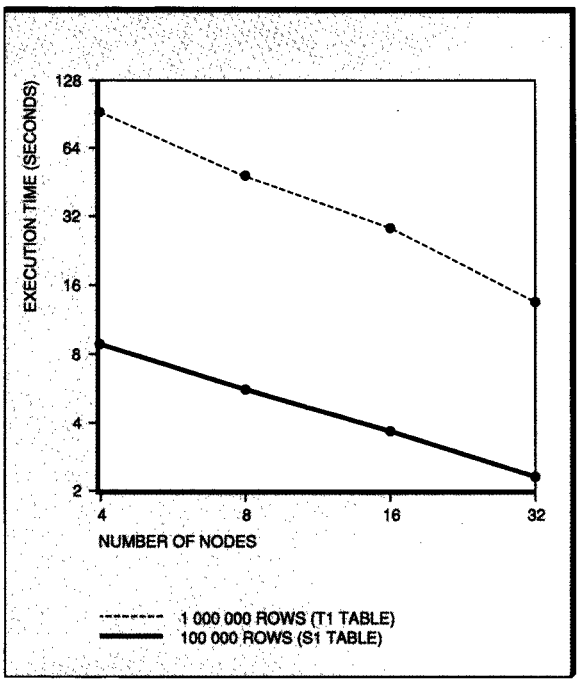


Table 3 Performance of complex queries on a 46-node SP2 for database sizes of 10 GB and 100 GB

Query Description	Response Times		Ratio
	10 GB	100 GB	
Six-way join, 14 columns, three tables	22	132	5.0
Insert/select of two-way join, select temp	26	186	7.15
Simple select, SUM, group by, order by	163	1 177	7.22
Two-way join, not equal predicate, in list	174	1 521	8.74
Create temp, insert/select (four-way join), select temp	694	7 234	10.42
Union of two two-way joins	253	2 647	10.46
Three-way join, three tables, avg., group by, order by	240	3 340	13.91
Two-way join, between predicate, group by, order by	164	3 682	22.45
Insert/select, select, three tables, distinct	157	4 147	26.4

tween 5 and 26.4. Since the 100 GB table is ten times (10×) as large as the 10 GB table, strictly linear scalability would result in response-time ratios of 10. Response-time ratios of less than 10 show superlinear speedup; thus for most of the queries, DB2 PE is able to generate an equal or larger amount of parallelism on the 100 GB database when compared to the 10 GB database. The scaling factor of the last three queries in Table 3 is sublinear (ratio greater than 10). These three queries include order by or distinct clauses that require sorting to be performed on the intermediate results at the coordinator. This causes a serial bottleneck and translates into a reduction in the scaleup ratio. Consider next the following complex business query:

```
SELECT Count(*) FROM Customers
WHERE Class IN ('1', '2', '4', '6')
  AND Cust_No NOT IN
    (SELECT O_CUST_NO FROM Offers
      WHERE O_DATE IN (list of dates))
  AND Cust_No IN
    (SELECT O_CUST_NO FROM Offers
      WHERE O_DATE IN (list of 2 dates)
      AND Response = 'Y')
```

Figure 16 shows the scalability results for this query, as the system size is increased from 4 to 64 nodes and the database size (which includes index files) is proportionately increased from 2.5 GB to 40 GB. This is an important result, since it indicates that the overhead introduced by the function shipping model of execution is relatively small and does not affect the execution times of complex queries.

Table 4 shows the results of a concurrent execution scalability test on the system. The test was performed using a 23 GB database on an eight-node SP2 system. In this test, we compare the response time of queries submitted by a single user to that of 20 and 30 concurrent users. First, we measured the execution times of the query suite consisting of 15 complex queries when they were submitted in a single stream by the single user. These execution times are shown in the second column of Table 4. Next, the queries were concurrently submitted by 20 and 30 users, respectively. Each user submitted one of the 15 queries. In order to distribute the coordinator activity over all nodes in the system, the users were connected to the 8 nodes in the SP2 in a round-robin fashion. Columns three and four show the scaling ratios of the execution times of the different queries for the 20 and 30 users, respectively. The results show that DB2 PE is able to scale superlinearly with respect to the concurrent users. There are several reasons for the superlinear performance scaleup. Similar to serial databases, DB2 PE is able to make better reuse of the database buffers at each node due to the common concurrent requests. This reuse occurs at all nodes, thereby providing a significant benefit. Another very big contributing factor is that the concurrent users can connect to all nodes in the system and distribute the application and coordinator load evenly across the nodes. In this experiment, the last row of Table 4 shows the total elapsed time for completing the entire query suite. The results show that 30 queries (two executions of the query suite) were completed concurrently in a time that was only 1.5 times worse than the single user, single stream test. Not all parallel database systems have this feature. Many of the parallel database systems are backend machines that have specific interfacing systems for application entry, and their multiprogramming levels are limited by the capabilities of this front-end system. DB2 PE does not have this restriction and provides significant concurrency benefits to users.

Figure 16 Performance of a complex query as both system size and database size are proportionately increased

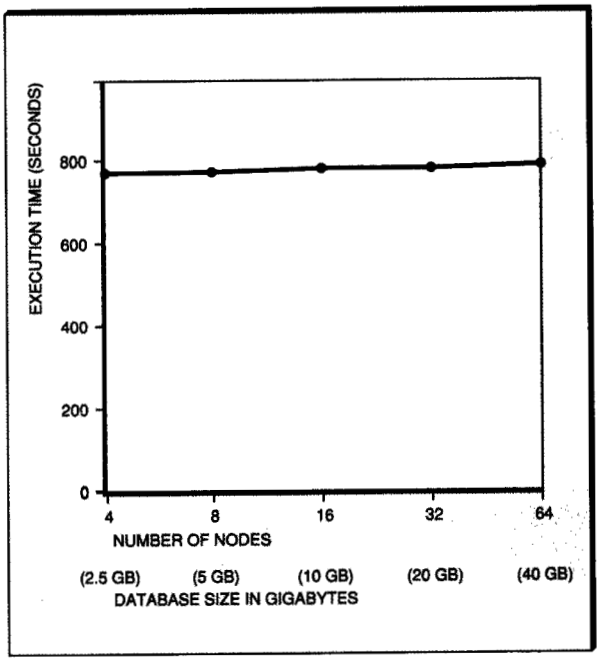


Table 4 Performance of queries with scaling of the number of concurrent users

Query	Single User Exec. Times (secs)	Scaling Ratios 20 Users	Scaling Ratios 30 Users
Q01	2	5	8
Q02	15	3.4	3.4
Q03	45	7.1	12.4
Q04	68	12.4	23.24
Q05	93	1.34	2.08
Q06	331	10.6	18.9
Q07	447	3.53	6.55
Q08	493	4.84	8.62
Q09	541	4.77	8.57
Q10	722	4.9	8.77
Q11	755	4.69	8.57
Q12	1140	4.75	8.74
Q13	1159	3.4	6.17
Q14	1557	4.24	8.23
Q15	1592	4.23	7.95
Total Elapsed Time	8491	8435	12828

Discussion. We have presented a flavor of the results obtained from several internal and external benchmarks performed so far using DB2 PE. The results that are presented here have all been extremely positive on the performance of the system. The speedup and scalability results have been consistently excellent for most types of queries and utilities. The results vindicate most of our design decisions in the generation and execution of optimal parallel strategies.

There are a few types of queries that do not result in linear speedup or scaleup. These queries are typically those that require significant serial work with respect to the total work in the queries. The performance of an example query type, which returns a significant number of rows to the application, was described in the section on scan performance. Another example is a query requiring coordinator sorts or distinct operations. When the coordinator activity is high in proportion to the total activity, the performance improvement of the system can decrease. We are working on improving the execution strategies for such types of queries to improve performance.

Also, the performance of queries that are executed in extremely short times on a serial database cannot be improved much further by the use of parallelism. This is because the serial database execution strategy is quite efficient, and parallelism is not going to provide any improvement on the execution of such a strategy. An example is index-only selection of a single row of values. Here, the result is a single row and only requires access to the appropriate index entry. Parallelism can benefit such a query only if the index happens to be extremely large.

Overall, the capacity, speedup, and scaleup improvements of DB2 PE for a very high majority of the queries far outweigh the very small class of queries described above with smaller performance gains.

Experiences and observations

We have learned much, both technically and organizationally, during the four years we have worked on this project. The Research Division produced an initial prototype, with function incrementally added. This allowed us to show "proof of concept," and maintain project momentum during the organizational changes that occurred. When the development laboratories became involved with the project, we continued in a closely allied joint development mode, with the product development meeting ISO 9000 standards. The joint development worked much better than the alternatives—specifying the product in intimate detail, or just handing over the prototype. It has certainly been a remarkable experience that two teams—research and development—with such disparate backgrounds have been able to work so closely together.

As the work progressed, it became increasingly clear that while the initial prototyping efforts in the project had given us a good understanding of the fundamental issues (section management, run time, initial query optimization, etc.), the work required to produce an industrial strength parallel database manager was still sizeable.

In the rest of this section, we highlight some of our technical observations about the product.

Function shipping. At the onset, we made a technical decision to use function shipping with shared-nothing hardware. This decision provided multiple benefits. Because we were working on a shared-nothing platform, we had to parallelize every database operation—query operator, DDL statement, and utility. While this resulted in a larger development effort, its positive impact was that it disciplined us to think in parallel concepts. The result is a scalable product where parallelism is the cornerstone—not just in simple scans and joins, but also in updates, subqueries, utilities, etc. Contrast this with some of the alternatives, which build limited parallelism on top of a *shared-something* environment and leave the more difficult operators (updates, subqueries, etc.) single-threaded.

Another benefit of our initial decision has been that most of our system limits scale linearly. Thus our tables (base and intermediate) can be N times larger, a query can typically acquire N times the number of locks, etc. This is a straightforward consequence of doing all operations in parallel.

Query optimization. Our initial effort for generating parallel plans was to transform the optimized serial plan into a parallel plan. This decision was a matter of programming convenience, encouraged by initial studies that indicated that this often produced the best parallel plan.[3] However, as we explored more and more complex SQL, it became increasingly clear that our approach, far from min-

imizing coding, was doing quite the opposite. For example, we were making the same kinds of discovery about ordering requirements in the post-optimization phase as the optimizer had made. Furthermore, it was also clear that without knowledge of partitioning, the optimizer was likely to come up with some very inefficient plans. For example, in

```
select * from T1, T2, T3 where T1.A = T2.B
                           and T2.C = T3.D
```

the optimizer might decide to join T2 and T3 first (because of its internal decisions on size, etc.), whereas in the parallel environment, T1 and T2 might be compatibly partitioned and hence should be joined first.

We therefore rejected the post-optimization approach and developed an integrated cost-based optimizer that understands parallelism, partitioning, messages, etc. This has been a sound decision resulting in a quality product.

Parallel utilities. Database literature and research in parallel databases have generally concentrated on what is considered the hardest problem—join processing. However, as we dug more and more into the making of a product, we realized that in a true decision support environment, we cannot operate on data until the data are correctly in place. We thus put in a lot of effort to make sure that the data can be initially loaded, balanced, and indexes created and generally prepared for subsequent queries, all at speeds viable for hundreds or thousands of gigabytes that are typical in these environments. People have criticized the shared-nothing approach because its static data partitionings can be skewed. We have provided rebalancing tools that do not bring database operations to a grinding halt and work one table at a time. Overall, our parallel utilities are as important as our parallel query processing, and we will strive to continuously improve on them.

Serial semantics. We have tried to provide transparent parallelism by maintaining serial semantics. Since we try and achieve the maximum parallelism by invoking read-ahead operations wherever possible, the users can see behavior different from their serial applications. An example of such behavior can happen when cursor-controlled operations interact with read-ahead operations. Other examples where the semantics are different from a serial database engine include error reporting and recovery triggered during blocked inserts. In serial machines the behavior is fairly predictable, since all operations are done by one process. The same cannot be said of the parallel product. In such cases, the user will have to choose between performance and deterministic semantics.

Parallel applications. Sometimes the amount of data crossing the database/application interface can overwhelm the improvements due to parallelism (explained by Amdahl's Law). We are studying ways to improve the performance of such applications by:

- Providing for parallel application support (programming model, SQL language extensions, and DBMS extensions required to transfer data in parallel)
- Examining means for pushing down some of the application logic into the database. This could involve object-oriented extensions such as user-defined functions and extending SQL to understand statistical issues such as correlation and sampling.

Summary and future. In summary, our parallel database implementation on an open platform has been a successful one, especially for complex query environments. We have shown that a scalable shared-nothing database system can be built by extending a serial database system with software that glues nodes together in order to provide a single system image to the user. The key components of this software—query optimization and processing, run-time system and database utilities—have been described in this paper. We have also shown that such a system can handle decision support applications on hundreds or thousands of gigabytes (thus extending them beyond the reach of shared-memory architectures).

In the future we will be enhancing it to incorporate better technology, newer applications and paradigms (e.g., parallel applications); exploit fully other hardware platforms (e.g., SMPs); and ensure our technology is best-of-breed in business-critical environments.

Acknowledgments

The seeds for DB2 PE were sown at the IBM T. J. Watson Research Center, starting in 1989. We started prototyping work with IBM Austin, and our initial technology achievements were showcased

in the Fall Comdex 1990 industry computer show. For this demonstration, a small number of local area network- (LAN) connected PS/2* processors were used. Although much of the underlying runtime infrastructure had been prototyped by then, the parallel query plans were hand-generated. The initial architecture design was led by Ambuj Goyal and Francis Parr from Research, and George Copeland from IBM Austin, with help from the IBM Almaden Research Center. Joint work started with IBM's Software Solutions Division (then Programming Systems) in Toronto when the latter got the DB2 client/server mission in late 1992. A full development team under Harry Chow was put in place starting mid-1993, when it became clear that the market was ripe for an open massively parallel processing- (MPP) based database system.

This product would not be possible without the hard work and contributions of the Toronto and Hawthorne development teams. Several of the architectural issues discussed in this paper were developed by people on these teams. In addition to the core development team, a successful product involves testing, benchmarking, marketing and sales. The authors acknowledge the contributions of the performance benchmarking teams from IBM Toronto and IBM Kingston (POWER Parallel Division)—several of their results are presented here. In addition, initial versions of DB2 Parallel Edition have had success in the field, mainly because of the tremendous job done by a number of DB2 PE specialists, some from marketing and sales, and others from consultancy organizations like the IBM ISSC. We have freely borrowed from their benchmark results and thank all the people involved.

Cited references and note

1. D. DeWitt et al., "The Gamma Database Machine Project," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 1, 44–62 (March 1990).
2. H. Boral et al., "Prototyping Bubba: A Highly Parallel Database System," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 1, 4–24 (March 1990).
3. W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS," *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)* (1991), pp. 218–225.
4. C. Baru and S. Padmanabhan, "Join and Data Redistribution Algorithms for Hypercubes," *IEEE Transactions on Knowledge and Data Engineering* **5**, No. 1, 161–168 (February 1993).
5. M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash-Join Method for Data Skew in the Super Database Computer (SDC)," *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman, Palo Alto, CA (August 1990), pp. 210–221.
6. M. Lakshmi and P. Yu, "Effectiveness of Parallel Joins," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 4, 410–424 (December 1990).
7. D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proceedings of the ACM SIGMOD Conference*, Portland, Oregon (May 1989), pp. 110–121.
8. S. Khoshafian and P. Valduriez, "Parallel Execution Strategies for Declustered Databases," *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, Editors, Kluwer Acad. Publishers, Boston, MA (1988), pp. 458–471.
9. D. Schneider and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman (August 1990), pp. 469–481.
10. G. Copeland et al., "Data Placement in Bubba," *Proceedings of the 1988 ACM SIGMOD Conference*, Chicago, IL (June 1988), pp. 99–108.
11. S. Ghandeharizadeh and D. Dewitt, "Performance Analysis of Alternative Declustering Strategies," *Proceedings of 6th International Conference on Data Engineering*, Los Angeles, CA (Feb. 1990), pp. 466–475.
12. K. A. Hua and C. Lee, "An Adaptive Data Placement Scheme for Parallel Database Computer Systems," *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman (August 1990), pp. 493–506.
13. S. Padmanabhan, *Data Placement in Shared-Nothing Parallel Database Systems*, Ph.D. thesis, EECS Department, University of Michigan, Ann Arbor, MI 48109-2122 (1992).
14. D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Analysis of the Gamma Database Machine" *Proceedings of the 1988 ACM SIGMOD Conference*, Chicago, IL (June 1988), pp. 350–360.
15. D. DeWitt, M. Smith, and H. Boral, "A Single-User Performance Evaluation of the Teradata Database Machine," *Proceedings of the 2nd International Workshop on High Performance Transaction Systems (Lecture Notes in Computer Science, No. 359)*, Pacific Grove, CA, Springer-Verlag (September 1987), pp. 244–269.
16. The Tandem Performance Group, "A Benchmark of NonStop SQL on the Debit Credit Transaction," *Proceedings of the ACM SIGMOD Conference*, Chicago, IL (June 1988), pp. 337–341.
17. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys* **15**, No. 4, 287–317 (1983).
18. The Tandem Database Group, "Nonstop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL," *Proceedings of the 2nd International Workshop on High Performance Transaction Systems (Lecture Notes in Computer Science, No. 359)*, Pacific Grove, CA, Springer-Verlag (September 1987), pp. 60–104.
19. *DBC/1012 Data Base Computer Concepts and Facilities*, c02-0001-05 Edition, Teradata Corp., CA (1988).
20. M. Stonebraker, "The Case for Shared Nothing," *Database Engineering* **9**, No. 1 (March 1986).

21. A. Bhide and M. Stonebraker, "A Performance Comparison of Two Architectures for Fast Transaction Processing," *Proceedings of the 1988 International Conference on Data Engineering*, Los Angeles, CA (February 1988), pp. 536–545.
22. C. Mohan, H. Pirahesh, W. Tang, and Y. Wang, "Parallelism in Relational Database Management Systems," *IBM Systems Journal* **33**, No. 2, 349–371 (1994).
23. D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM* **35**, No. 6, 85–98 (June 1992).
24. D. Davis, "ORACLE's Parallel Punch for OLTP," *Datamation* **38,** No. 16, 67–71 (August 1992).
25. E. Ozkarahan and M. Ouksel, "Dynamic and Order Preserving Data Partitioning for Database Machines," *Proceedings of the 1985 Very Large Data Bases International Conference* (1985), pp. 358–368.
26. S. Ghandeharizadeh and D. J. DeWitt, "Magic: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines," *IEEE Transactions on Parallel and Distributed Systems* **5,** No. 5, 509–524 (May 1994).
27. C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*, ACM (August 1993), pp. 76–88.
28. P. Selinger et al., "Access Path Selection in a Relational Database Management System," *Proceedings of the 1979 ACM SIGMOD Conference* (1979), pp. 23–34.
29. S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution," *Proceedings of the 1992 ACM SIGMOD Conference* (May 1992), pp. 9–18.
30. D. Skeen, "Non-Blocking Commit Protocol," *Proceedings of the 1981 ACM SIGMOD Conference*, Orlando, FL (1981), pp. 133–142.
31. G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in R*," *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Editors, Springer-Verlag, Inc., New York (1985), pp. 31–48.
32. *AIX High Availability Cluster Multi-Processing/6000, Concepts and Facilities*, SC23-2699, IBM Corporation; available through IBM branch offices.
33. SP2 performance measurement data are available from the IBM POWER Parallel Division, P.O. Box 100, Somers, NY 10589.

Accepted for publication January 11, 1995.

Chaitanya K. Baru *IBM Toronto Laboratory, 1150 Eglinton Ave. East, North York, Ontario, M3C 1H7 Canada.* Dr. Baru is currently an advisory development analyst in the Database Technology division at the IBM Toronto laboratory, and one of the team leaders in the DB2 Parallel Edition (DB2 PE) project. He joined IBM Canada in 1992. He has been involved in the design and development of the data manager extensions and some of the database utilities for DB2 PE. Prior to joining IBM, he was Assistant Professor of Computer Science and Engineering in the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor, Michigan. He has published several papers related to parallel database systems in technical conferences and journals. Dr. Baru is a senior member of the IEEE and a member of the Computer and Information Sciences Grants Selection Committee of the Natural Sciences and Engineering Research Council (NSERC) of Canada. He received his B. Tech. degree from the Indian Institute of Technology, Madras, India in 1979, and M.E. and Ph.D. degrees in electrical engineering from the University of Florida, Gainesville, in 1983 and 1985, respectively.

Gilles Fecteau *IBM Toronto Laboratory, 1150 Eglinton Ave. East, North York, Ontario, M3C 1H7 Canada.* Mr. Fecteau has been in information technology for 25 years, as an IBM Canada employee. His experience includes application programming, systems programming, capacity planning, database design, operating systems, and database management systems design and implementation. His primary technical focus for the past nine years has been IBM's relational database systems design, where he has focused on performance and manageability enhancements. He holds the degree of Bachelor of Engineering Science (applied physic) from Laval University in Quebec city, Canada. Since 1990 Mr. Fecteau has been involved in several advanced technology efforts in IBM to bring to market a parallel database. In 1991 he joined the workstation database group in Toronto as a lead designer on the implementation of a parallel version of the DB2 workstation database products (DB2/2 and DB2/6000).

Ambuj Goyal *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: ambuj@watson.ibm.com).* Dr. Goyal joined IBM as a research staff member at the Thomas J. Watson Research Center in Yorktown Heights, New York, in May 1982 and assumed a first-line management position in July 1988. He became the senior manager of the Parallel and Fault-Tolerant Computing group in October 1990, and then was promoted to his present position of Director, Servers and Parallel Systems. All of Dr. Goyal's assignments have been with the IBM Research Division at the Yorktown Heights, New York, location. He received the B. Tech. degree from the Indian Institute of Technology, Kanpur, in 1978, and the M.S. and Ph.D. degrees from The University of Texas at Austin, in 1979 and 1982, respectively. As director, he is responsible for setting Research Division strategy in servers and parallel systems, including establishing a technical program in this area that is vital to IBM's success and is known for its technical innovations.

Hui-I Hsiao *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: hhsiao@watson.ibm.com).* Dr. Hsiao has been a research staff member at the IBM Thomas J. Watson Reseach Center since 1990. Currently, he is a member of the parallel database project, working on the research and development of IBM's first highly parallel database system. His research interests include parallel database architecture, parallel query and transaction processing strategies, and database performance analysis. Prior to joining IBM, he worked as a software engineer at Nicolet Instrument Corporation, where he received a Nicolet Associate Fellow Award. He was a member of the Gamma database machine project at the University of Wisconsin. Dr. Hsiao received a B.S. degree from National Taiwan University, and M.S. and Ph.D. degrees in computer science from the University of Wisconsin at Madison.

Anant Jhingran *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: anant@watson.ibm.com).* Dr. Jhingran

is currently the manager of information systems at the IBM Thomas J. Watson Research Center. His group is engaged in several areas of database research, including parallelism, availability, transaction processing, data mining, and warehousing. Over the last several years, he has been working with IBM Toronto on all aspects of DB2 Parallel Edition. Dr. Jhingran has contributed significantly to various components of DB2 Parallel Edition, including query optimization and processing, runtime subsystem, data partitioning, and reliability and availablity. He has filed several invention disclosures and patents in several areas of parallel database systems, and has actively published in leading conferences like ACM SIGMOD, VLDB, and Data Engineering. He has also served on the program committee of SIGMOD. He received his Ph.D. and M.S. degrees in computer science from the University of California, Berkeley, in 1990 and 1987 respectively, and his bachelor's degree in electrical engineering from the Indian Institute of Technology, Delhi, India, in 1985.

Sriram Padmanabhan *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: srp@watson.ibm.com).* Dr. Padmanabhan is a research staff member at the IBM Thomas J. Watson Research Center, Hawthorne, New York, which he joined in 1992. He is an active member of the DB2 Parallel Edition research and development effort and has contributed to several architectural aspects of the system and significantly to the query optimization and parallel query execution areas. Besides parallel databases, he is interested in object-oriented and distributed databases, information retrieval, massive data-storage applications, and parallel architectures and algorithms. Dr. Padamanabhan received his Ph.D. and M.S. degrees in computer science from the University of Michigan, Ann Arbor, in 1992 and 1990, respectively, and his B. Tech. degree from the Indian Institute of Technology, Madras, India in 1986.

George P. Copeland *IBM Corporation, Personal Software Products Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: copeland@austin.ibm.com).* Dr. Copeland is currently a Senior Technical Staff Member at IBM Austin. He received his B.S. degree from Christian Brothers College, Memphis, in 1969, and his M.E. and Ph.D. degrees from the University of Florida, Gainesville, in 1970 and 1974, respectively. He has worked for NASA, the Bank of America, and Tektronix, and was chief architect and a founder of Servio Corporation, an object-oriented database (OODB) vendor. At MCC, he served as chief architect of the Bubba parallel OODB project. He joined IBM in 1989 as chief architect of the OS/2 Database Manager, helping initiate and contribute to projects to make it portable and parallel. He has also contributed to the IBM microkernel and file systems. His major area of interest is applying to objects the same kinds of systems services that database systems apply to data, such as persistence, concurrency, recovery, transactions, distribution, security, etc. He is a member of IEEE and ACM, and has served on numerous conference program committees and panels.

Walter G. Wilson *Red Brick Systems, 485 Alberto Way, Los Gatos, California 95032 (electronic mail: walter@redbrick.com).* Dr. Wilson is currently Director, Parallel Systems, at Red Brick Systems. He received his B.S. in mathematics from Stanford University, and his M.S. and Ph.D. in computer science from Syracuse University. Previous experience includes that of project leader for the parallel database project at IBM's Thomas J. Watson Research Center, manager of systems advanced technology at IBM's Enterprise Systems division, and manager of the Symbolic Programming department at the Thomas J. Watson Research Center.

Reprint Order No. G321-5570.