DB2 Parallel Edition

Chaitanya Baru , Gilles Fecteau
IBM SWSD, Toronto
Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, Walter Wilson
IBM TJ Watson Research Center

Abstract

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 I/O 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 parallel database project initiated at IBM Research at Hawthorne and the DB2/AIX-PE product based on it.

1 Introduction

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. However, almost all of the applications harnessing this technology are scientific or engineering applications. The lack of commercial applications for these parallel processors is due largely to the questionable *robustness* and *usability* of these systems. Compared to mainframe systems, large scale parallel processing systems have a history of poor availability and reliability . They are also lack of good software for system management and application development. However, the current generation of massively parallel processor systems, in particular, IBM's *Scalable Parallel* (SP1 and SP2) class of systems, are much more robust and easy to use. The commercial market has recognized these improvements and are eager to take advantage of this exciting technology.

One of the main enablers for commercial applications is Database Management Systems (DBMS). Thus, a parallel DBMS is a natural step. Several businesses and industries are investing in *Decision Support* applications in order to understand various sales and purchase trends.

These applications pose complex questions (queries) against large sets of data in order to gain an insight into the trends. Single system (or Serial) DBMSs cannot handle the capacity and the complexity requirements of these applications. Besides decision support, there are other new application classes such as Data Mining, Electronic Libraries, and Multimedia that require either large capacity or the ability to handle complexity. All these applications require parallel DBMS software.

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, 5, 6, 7], query optimization techniques [8, 9], data placement [10, 11, 12, 13], and database performance [14, 15, 16]. The results of these studies form a basis for our knowledge of parallel database issues today. However, two major limitations with these projects are: (i) Many of the problems were considered in isolation, so the implementation tended to be very simple, and (ii) In several cases, people resorted to simulation and analysis because the implementation requires enormous effort. Recognizing the importance of a *commercial strength* parallel database system, we started a project at IBM Research that has now led to the announcement of the DB2 Parallel Edition product.

DB2 Parallel Edition (DB2 PE) is a parallel database software solution that can execute atop any UNIX-based parallel processing system. Its Shared-Nothing (SN) architecture model and Function Shipping execution model 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 more system CPU and disk resources. DB2 PE has been architect4ed and implemented to provide 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 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 parallelized. We also provide a parallel reorganization utility called *Rebalance* which will effectively correct data and processing load imbalance across different nodes of the system.

In sum, DB2 PE is a comprehensive, full-fledged, parallel database system.

It must be noted that companies such as Tandem and Teradata have built and sold parallel database products for a few years [18, 16, 19]. Teradata's DBC/1012 system is targeted for Decision Support Applications while most of the Tandem systems target high-performance OLTP applications. However, both products are based on proprietary hardware architectures which are only usable for the database processing task. The 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 this project which have not been addressed elsewhere. Several of these aspects will be highlighted in the ensuing sections.

The rest of the paper is organized as follows. Section 2 describes the general architecture of the DB2 PE system. The relative merits of the Shared Nothing architecture and the Function Shipping execution model are described. The next three sections discuss the three layers of the system in detail. Section 3 discusses the user controlled data layout for optimal performance. The next section describes some of the salient features of DB2 PE's query optimization. The run-time internals of the system are then discussed in Section 5. Since decision support applications depend a lot on database utilities like load and unload, we discuss that in section 6. Section 7 presents some of the initial performance numbers of some controlled experiments, and the paper then ends with a discussion on our experience, conclusions, and directions for future work.

2 Architecture Overview and Project History

There are three different approaches 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). Figures 1, 2, and 3 illustrate the three different parallel database system architectures.

In shared memory systems, as shown in Figure 1, multiple processors share a common central memory. With this architecture, communication among processors is through shared

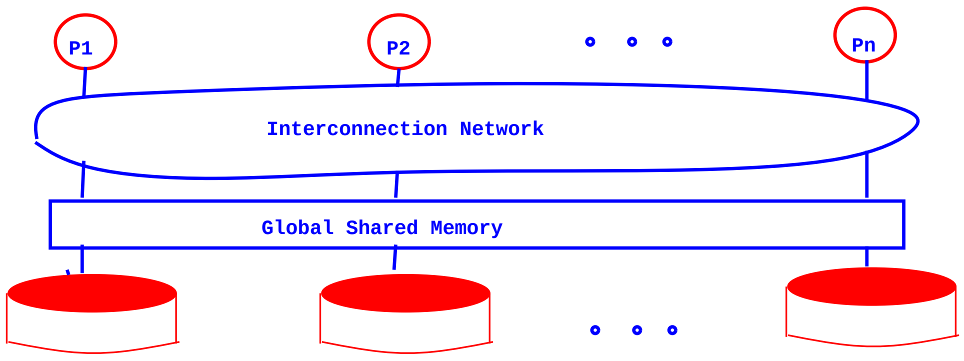

Figure 1: Shared memory architecture.

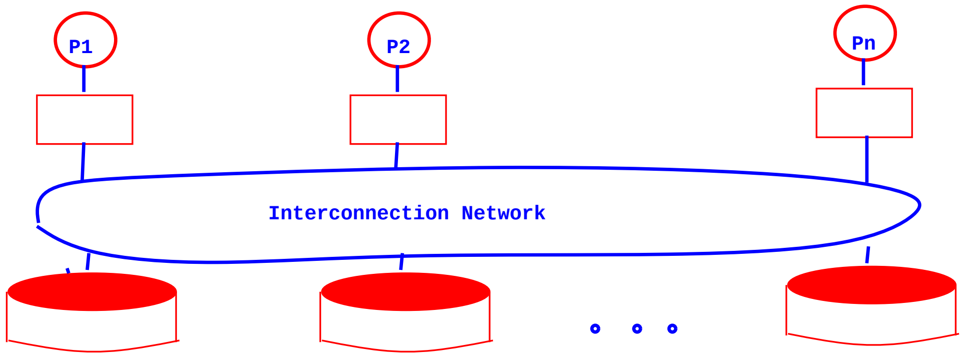

Figure 2: Shared disk architecture.

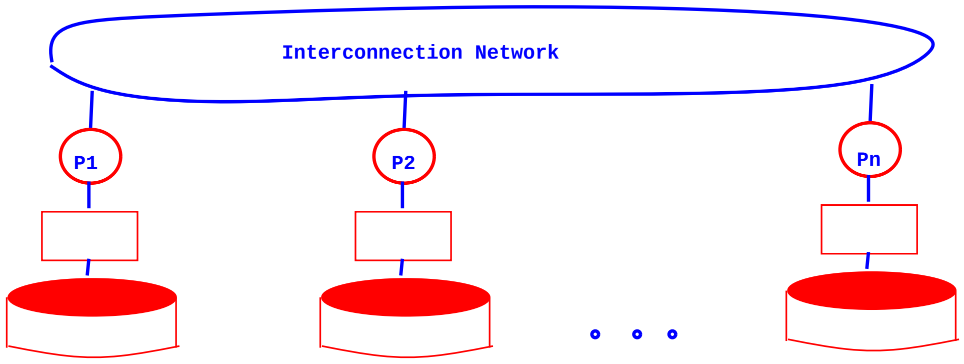

Figure 3: Shared nothing architecture.

memory, thus there is little message overhead. In addition, the software required to provide parallel database processing is considerably less complex with shared memory than with the other two architectures. 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 it can not scale to 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 not determined by how fast the processors are. State of the art technology can build memory to support about 500 MIPS of CPU power. This implies that a shared memory system can support less than 10 RISC processors of the current generation.

In shared disk systems [22], as illustrated in Figure 2, 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 large number of processors and disks. In addition, shared disk presents the most challenging task of transaction management because it needs to coordinate global locking activities – but without the help of a shared memory – and to synchronize log writes among all processors.

With the shared nothing architecture (Figure 3), each processor has its own memory as well as local disks. Except communications media, no other resources are shared among processors. shared nothing does not have the memory access bottleneck problem, nor does it have the problem of inter-connecting a large number of processors and disks. The major complexity in supporting the shared nothing architecture is the requirement of breaking a SQL request into multiple sub-requests sent to different nodes in the system and merging the results generated by multiple nodes. In addition, shared nothing requires distributed deadlock detection and multi-phase commit protocol to be implemented. Researchers and developers have argued that the shared nothing architecture 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 architecture.

Because a shared nothing system can easily be scaled to hundreds of processor while shared memory and shared disk systems are limited either by memory bus bandwidth or by I/O channel bandwidth and because a shared nothing system can grow gracefully, i.e. adding more disk

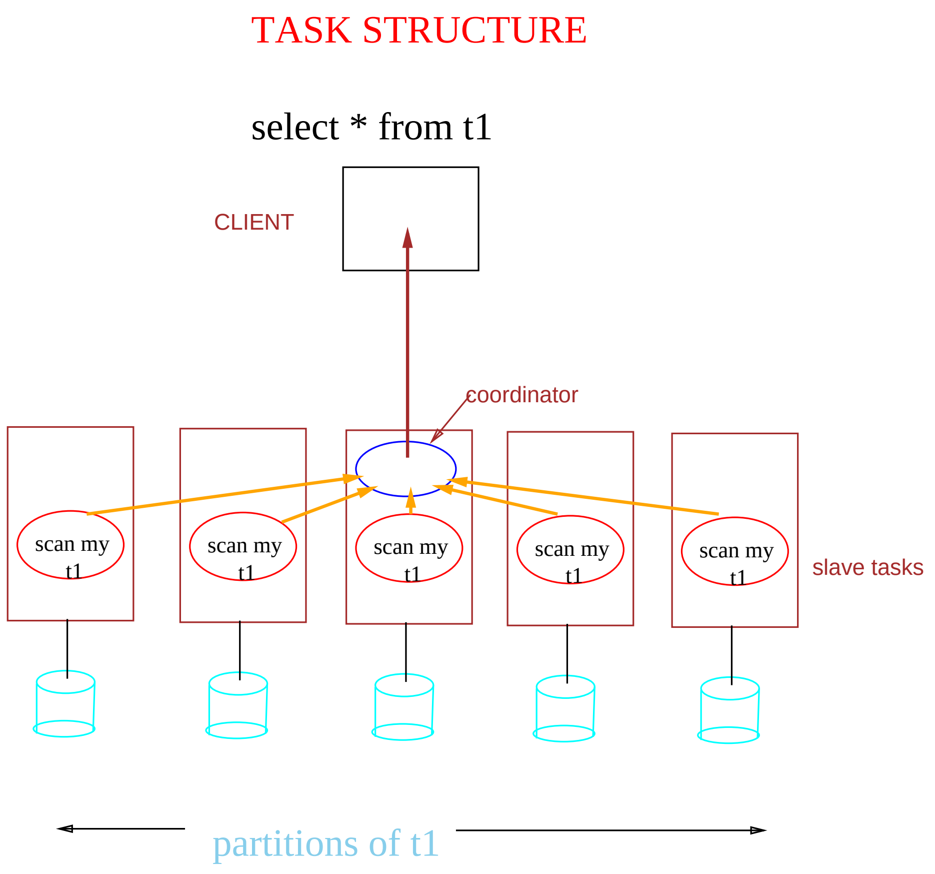

Figure 4: Run-Time Execution in DB2 PE

capacity and/or processing power as needed, DB2 PE adopts the shared nothing architecture.

2.1 Function Shipping

Because resources are not shared in an shared nothing system, typical implementations use *function shipping*. In this, database operations are performed where the data resides. This minimizes network traffic by filtering out unimportant data, as well as achieves good parallelism. So a major task in an shared nothing implementation is to split the incoming SQL 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 – it receives the SQL and associated host variables, if any, and returns the answers back to the application.

Figure 4 shows some of the task structure for a very simple query. The table **t1** is shown

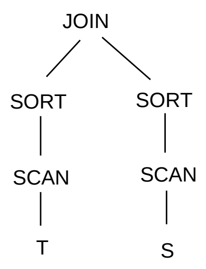

Figure 5: An example of a DB2/6000 Execution Strategy

horizontally partitioned [13] across all the nodes and thus, based on function shipping paradigm, the coordinator requests a slave task – one on each node – to fetch its partition of **t1** and stream the result to it. The results are then returned to the application as it issues "EXEC SQL FETCH" statements. In more complicated SQL statements, the task structure is inherently more complex – it is the job of the query compiler to come up with 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 on the node to which the application connects, and each slave task is instantiated on the nodes on which the data it needs to access resides. Thus in Figure 4, there is one coordinator, and five instances of slave task #1.[1]

As an example of a more complex function shipping, consider the query and its serial execution strategy shown in Figure 5 for the query:

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

When **T** and **S** are horizontally partitioned, a possible parallel execution startegy could be the one that maintains the serial structure, but executes each operator in parallel (Figure 6).

Circled crosses indicate data exchanges (we show later how in a large number of cases even such exchanges can be avoided). It is clear that this execution startegy requires a coordinator (not shown) and three slave tasks (slave task #1 scans, sorts, and ships its partition of **T** to slave task #3, slave task #2 does the same against **S**, and slave task #3 does the actual join).

One of the advantages that we realized from using function shipping was that we could leverage a lot of the existing DB2/6000 code – the scans, sorts, joins etc shown in Figure 6 is

[1]We use the term slave task, subordinate task, subsection and subplan interchangeably.

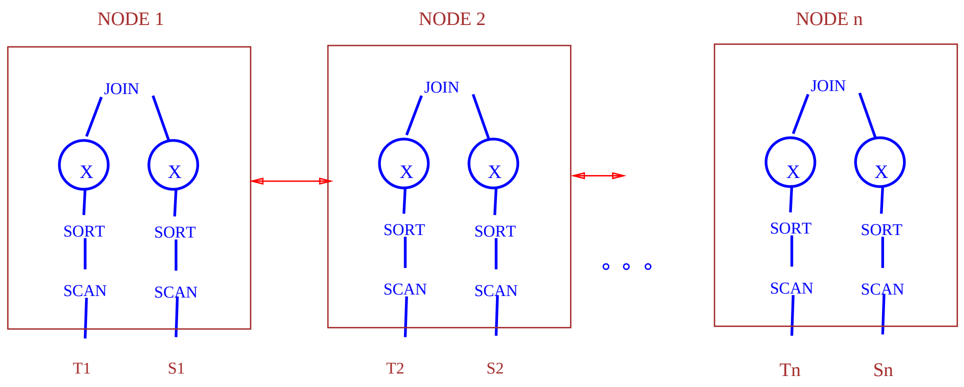

Figure 6: An example of a DB2/6000 Execution Strategy

identical to the operators in Figure 5 – they are transparently parallelized because their inputs are. The real fundamental technology in Figure 6 is in the mechanism that glues the nodes together to provide a single system view to outside world. In addition to function shipping, other technologies required are (1) generation of parallel plan, (2) streaming of data and control flow, (3) process management, (4) parallel transaction and lock management, and (5) parallel utilities.

Figure 7 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 DDL (Data Definition Language) and DML (Data Manipulation Language). DDL is used to define and manipulate 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 – DMS (Data Management Services), which deals with operations on local data, and Communication Services, which deals with operations on remote data. DB2 PE had to make modifications to the DMS layer of DB2/6000, but the changes were relatively modest. However, the communication services was an entirely new

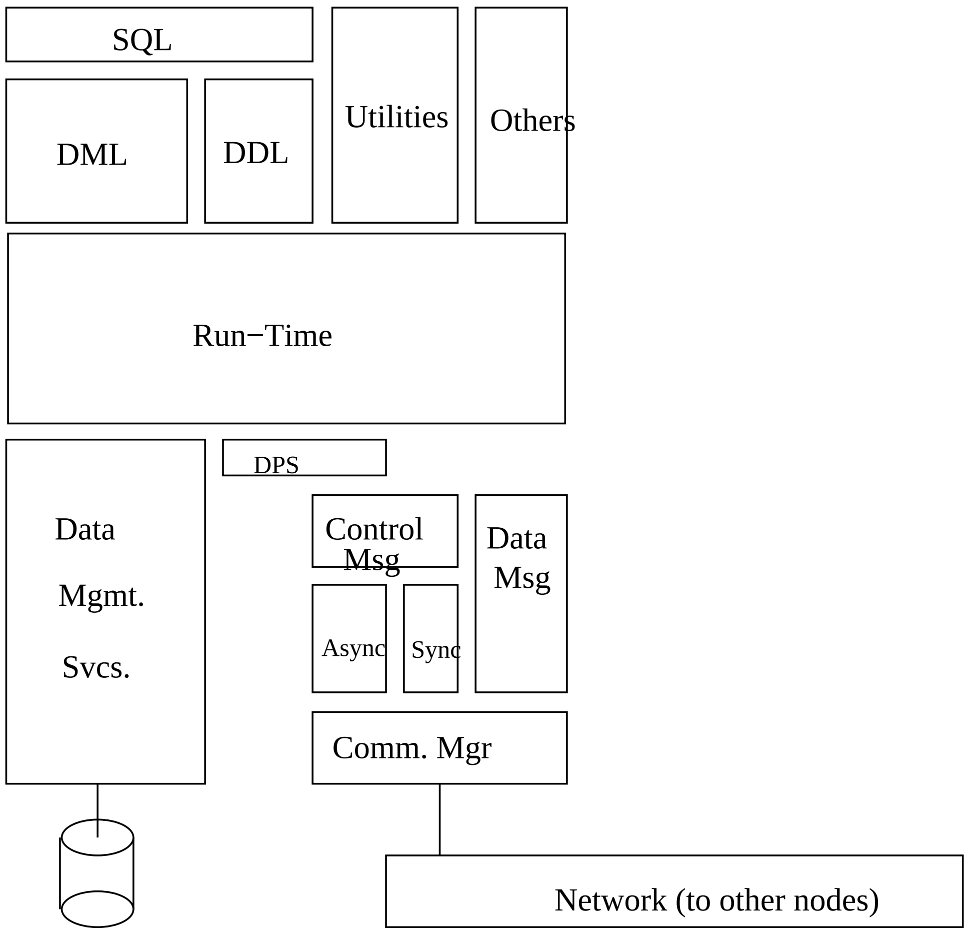

Figure 7: Major System Components

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 resposible for multiplexing, demultiplexing and reliable delivery to other DB2 PE processes.

In addition, the DPS (Data protection services) layer of DB2/6000, responsible for locking, logging and recovery, had to be extended to account for the fact that a transaction 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 will be discussed in more detail in the following sections. Changes to the DDL and its processing are described in Section 3. DML statements and their optimization, including the new operators required to execute them in a function shipping paradigm are discussed next. All changes in the run-time system and the DPS layer, as well as the new communication component are discussed in Section 5. Finally, Section 6 discusses some of the new parallel database utilities.

2.2 Project History

The seeds for DB2 PE were laid at IBM TJ Watson Research Center, starting 1989. Initial technology achievements were showcased in the Fall Comdex 1990. For this demo, a small number of LAN-connected PS/2's were used. Though a lot of the underlying run-time infrastructure had been prototyped by then, the parallel query plans were hand generated. Research kept adding to the functionality of the parallel database system, and experimented with several different approaches. We began joint work with product divisions, first with Advanced Workstations Division in Austin. 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 was put in place starting mid 1993 when it became clear that the market was ripe for an open MPP based database system in 1994.

3 Data Definition Language (DDL)

3.1 Background

DB2 PE provides extensions to SQL in the form of new data definition language (DDL) statements which 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.

The problem of determining the best storage strategy for the tables in a given database is called the *data placement problem.* Data placement in parallel database systems is known to be a difficult problem [13] and several approaches have been taken to solve this problem [25, 10, 13, 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 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.* 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 = degree of declustering).

After choosing the degree of declustering, it is neccesary 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 *non-overlapped*, 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 node, then the tables are said to be *fully overlapped.* 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. In the last two, a set of columns (attributes) of the table are defined as the *partitioning keys* and their value(s) in each row is used for hashing or range partitioning.

3.2 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, viz. 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, an on-line utility called REDISTRIBUTE NODEGROUP is provided to allow addition/removal of nodes to/from a nodegroup. Further details are provided in Section 6.4.2.

3.3 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 column(s) 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

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**

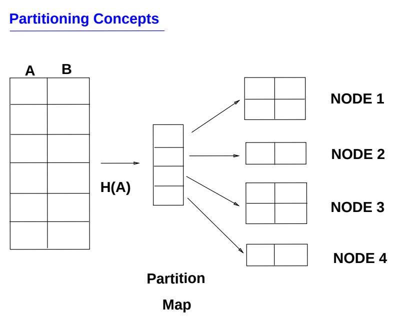

Figure 8: The concept of Partitioning Keys and Maps

PARTITIONING KEY (*Suppkey*) 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, called the *partitioning map (PM)*, 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 i^{th} location in the map, i.e. at $PM[i]$. (Figure 8 shows a table with partitioning key A. The hash function $H(.)$ 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 neccesary that $d \leq p$. In DB2 PE, the value of p is chosen to be 4096. Typically, $d << 4096$, 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 $\lceil \frac{4096}{d} \rceil$ 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 inter-node 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:

$PARTS \bowtie_{(Partkey=Partkey)} PARTSUPP.$

4 Query Optimization for DB2 Parallel Edition

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 best cost. The optimizer accounts for the inherent parallelism of different operations and the additional costs introduced by messages while comparing different strategies.

- **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 best 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.

4.1 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 database architecture introduces two new concepts that are not present in a serial engine:

- Query execution over multiple logical tasks (recall that each logical task, at run-time, can be executed on 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 sub-section, is described in more detail in a later section.

- As a consequence of multiple processes, interprocess communication operators (notably send/receive) are required in DB2 PE. These operators can have attributes (e.g., send can be broadcast or directed; receive could be deterministic or random).

4.2 Partitioning Knowledge

In DB2 PE, we are conscious about partitioning in the DDL, Data Manipulation SQL, and at run-time. DB2 PE's partitioning methodology 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 partitionings, node groups 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 approriate nodes.

4.3 Query Optimization and Execution Plan Generation

The compiler is responsible for generating the *optimal* strategy for parallel execution of a given query. In this section, we will 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 send/receives being used for inter-task communication.

A query optimizer typically chooses (a) the optimal join order and (b) the best way to access base tables and to compute each join. This task is inherently exponential ([27, 28]) and many optimizers use heuristics like postponing of cross products, left-deep trees etc. in order to prune the search space. In the case of a parallel database, this operation is further complicated by (c) 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 may not be on the smae set of nodes) and (d) choosing between system resources and response time as the appropriate metric for deteriming 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 of, on a per node basis, the total system resource accumulated during the bottom-up generation of a query plan. The maximum across all the nodes is a measure of the response time of a query. This ignores the costs associated with the coordinator, as well as the response time complications associated with multiple processes.
- Of all the possible subsets of nodes that can be used to execute a join, we typically consider only a few – all the nodes, the nodes on which the inner table is partitioned, the nodes on which the ouer is partitioned, and a few others.
- In keeping with the DB2/6000 query optimization strategy, we are greedy in choosing the repartitioning startegy as well – the best locally optimal strategy is the one that survives.

In some queries, the optimal strategy is obvious:

```
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 SUPPLIER tuples and eliminate those which are not from 'ASIA'. The run-time execution strategy is very similar to 4.

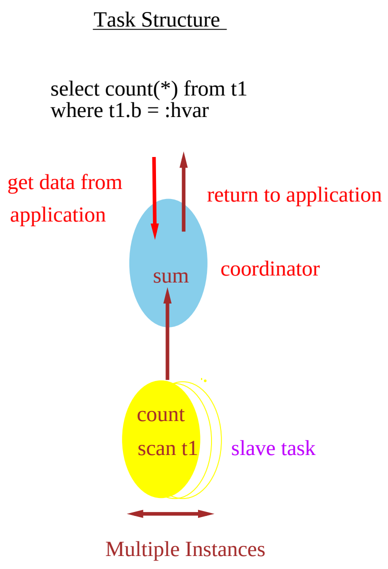

Figure 9: Task structure for a Query

In a more complicated query, such as one shown in Figure 9, 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 do aggregation such as count(*) in two steps – the slave tasks compute their local counts and then the coordinator sums the counts and returns to the application. The arrows from the slave task to the coordinator are send/receive end – the arrow from the coordinator to the slave task is the passing of all the information required for the slave task to correctly execute (i.e. the function, including the input host variable).

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

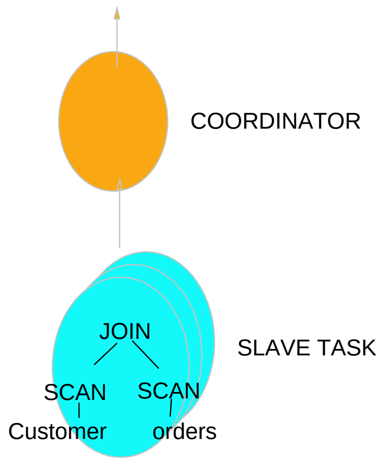

Figure 10: Collocated Join

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 10000 may be mapped to node 100 but is the same for both tables. Thus, the Join operation can be performed on the local partitions of the two tables. The execution strategy for this is shown in Figure 6 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 10 shows the task structure for this join.

Directed Join

Let the Partition Key for CUSTOMER 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 that this is the case based on the partitioning knowledge. 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 the 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 11 shows the compiled plan

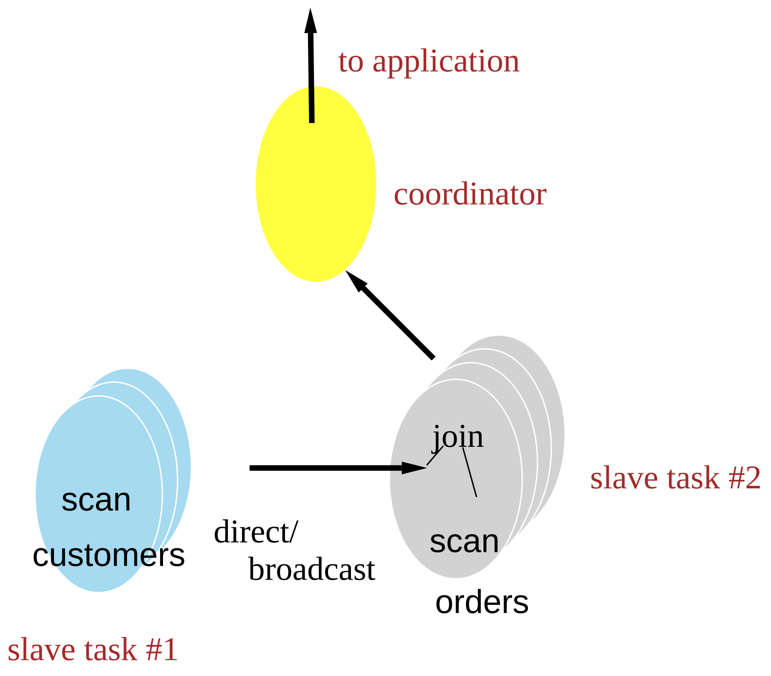

Figure 11: Directed/Broadcast Join

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 11 also shows a broadcast

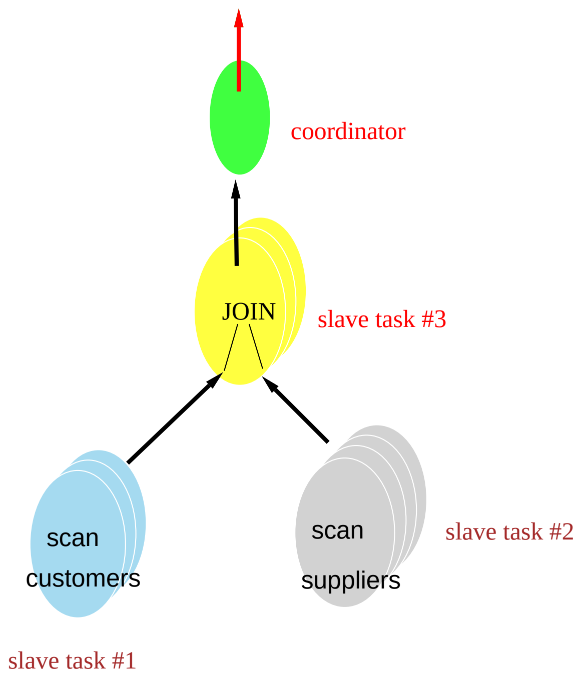

Figure 12: Repartitioned Join

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 CPU 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 or when there is an index on a joining attribute(s).

Repartitioned Joins

We also consider a repartitioned strategy of Join execution in cases such as the query described above. 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 described above, 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 12 shows the repartitioned join strategy.

The repartitioned join requires message traffic to redistributes rows of both tables involved

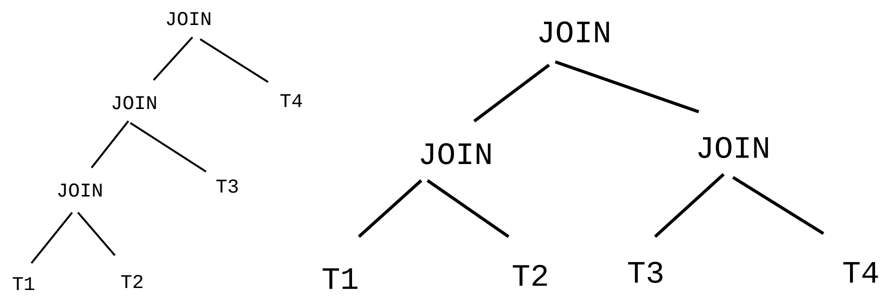

Figure 13: Different Execution Strategies in Serial and Parallel Environments

in the join. Once redistributed, the join CPU cost is similar to the colocated Join case.

Cost Based Optimization

One of the most important features of the optimizer is that it uses *Cost Estimates* whne deciding between different execution choices. This is to be contrasted with an optimization technique which 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 described above 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 13 shows two different types of query execution plans for a 4-way Join query. Let the tables be allocated to two disjoint sets of nodes. An optimizer for a serial DBMS could choose the strategy in Figure 13 (a) 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 13 (b) since more work can be performed in parallel and the partitionings for the two lowermost joins are optimal. Thus proper "parallel cost measures" are critical for parallel query optimization.

4.4 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, we only list the other operations and constructs where we apply parallelism while generating the execution strategies.

- Aggregation: Ability to perform aggregation at individual slave tasks and, if required, at a global level.
- Set operations: We consider collocated, directed, repartitioned, or a global strategy akin to the Join strategies described above.
- Inserts with subselect, updates, deletes.
- Subqueries: We consider collocated, directed, and broadcast methods of returning subquery results to the sites where the subquery predicate is evaluated.

5 DB2 Parallel Edition Runtime

In order to execute a query plan or a DDL statement in parallel, DB2/6000 run-time mechanism had to be augmented considerably. A new component, the *communication manager*, was added to provide interprocess communication between various DB2 PE processes. On top of it, the following new components now exist:

- : **Table Queue Service**: This deals with exchange of rows between DB2 PE agents across or within a node and is responsible for the correct execution of the data flow operators connecting different slave tasks.
- : **Control Service**: This deals with interprocess control message flow (start/stop of processes, error reporting, interrupts, Parallel Remote Procedure Call, etc.).

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

5.1 Control Services

To summarize the discussion in section 4, the compiler takes each SQL statement and produces a *plan* to be executed. This plan consists of operators such as *Access Table*, *Join Tables*, *Sort*, etc. Plans are organized based on a data-flow model. For example, two Sort operators may feed rows to a Join operator (as shown in Figure 5).

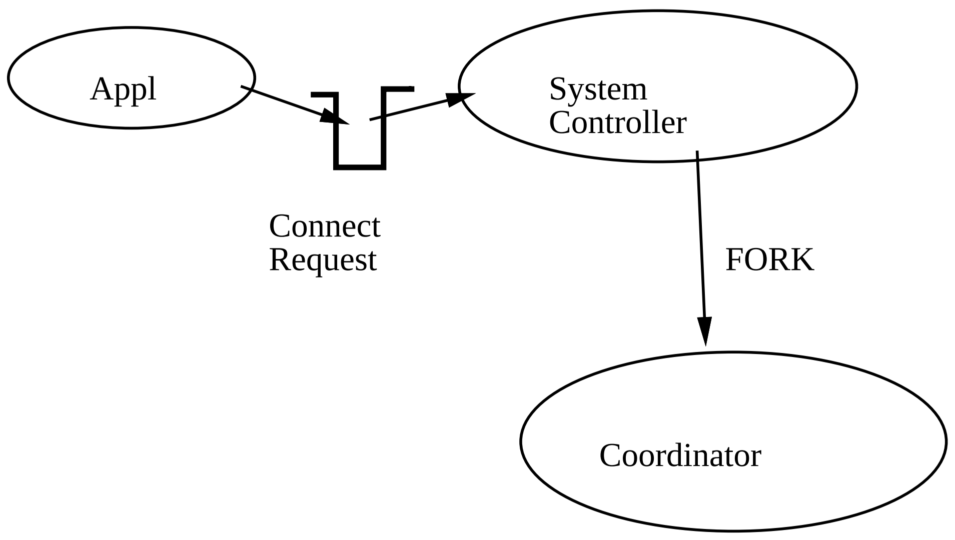

Figure 14: Application Connects

In making the query parallel, multiple processes are involved on different nodes. Unless optimization combines them, each operator is effectively executing in its own process. DB2 PE consequently adds explicit dataflow operators where data-flow between processes is necessary.

When an application connects to a database (14), a special process 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 (15). 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, called *agents*, are organized into a network of producer-consumer processes. Data flow over *Table Queues*, which are described in the section 5.2.

The 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, either based 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 needs the table

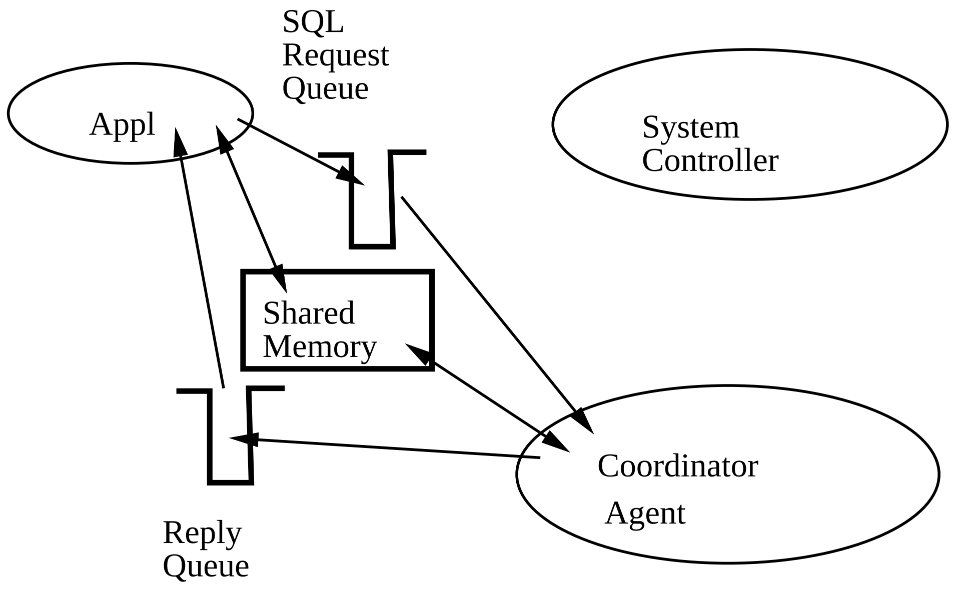

Figure 15: Application and Coordinator Agent

access to happen only on the node that contains the partition for t1.a = host-variable), or on the current load (for those subsections that are not tied to specific nodes, e.g. those which 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. Process management is done by *DB2 PE's control component*(see 16).

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. DB2 PE implements a "process pool", which allows processes to be re-used for different applications, and different subsections of the same application; and can grow or shrink as required.

Certain sequence 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 which 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 till 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

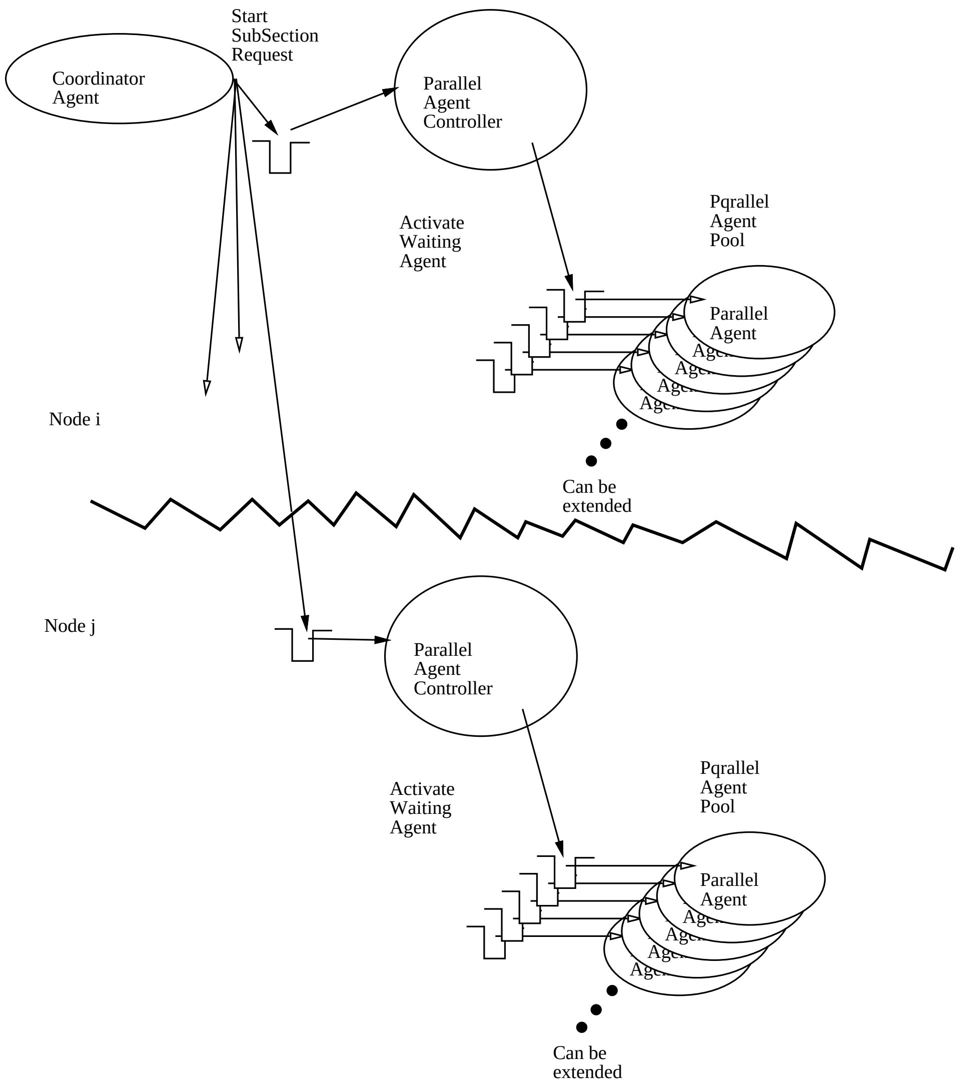

Figure 16: Subsections Distributed

parameters determining this tradeoff 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; return control replies such as the SQLCA to applications; provides Parallel Remote Procedure Call support for low-level database manager functions such as "retrieve long field" or "load access plan from catalog".

5.2 Table Queues

The inter-process 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 which is visible to more than one process. 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 inter-process communication, controlled by 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. It is a communication path between multiple producer and multiple consumer processes (see 17). 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.

Each connection on a table queue has a specified capacity, which is used to synchronize the sending and receiving processes. A receiver can consume all of the rows which have been sent, then wait for the producer to send more. Conversely, if the producer is sending rows faster than they can be consumed, it will wait for room when it has a row that needs to be sent on a full connection.

There are many attributes associated with table queues. Some of them are:

- : Broadcast vs. Directing: Does one row at the sending end go to all the receivers, or only to one? See 18 for an example of a directed table queue.

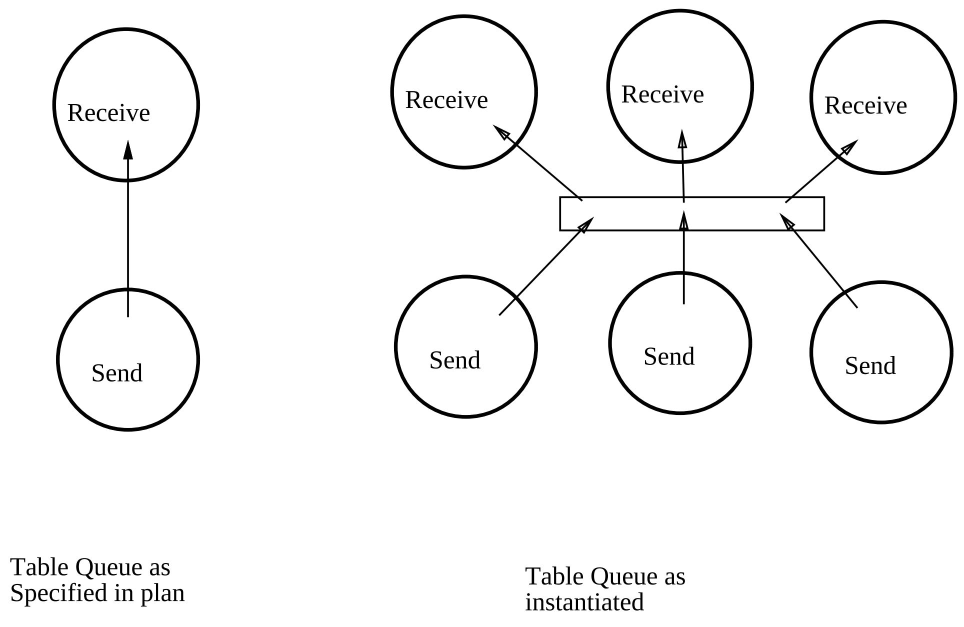

Figure 17: Table Queues

- : Read-Ahead vs. Stepping: Does the table queue builder wait to fill a block, or does it send across just one row at a time? While the latter might seem inefficient, it is required for some operations where the position of Table Access needs to be maintained for possible subsequent updates.

5.3 Communication Subsystem

The parallel communications component is layered in a similar fashion to the rest of the runtime. It accepts messages (either control messages or buffers of rows) and guarantees ordered delivery between nodes (or between processes on the same node). It performs multiplexing and de-multiplexing of messages between nodes of a DB2 PE system. Underneath, it uses the delivery layer, which can be UDP, TCP, or proprietary high-speed switch interface (see 19).

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 to receive it. Some of the complications 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 has not yet been created and so the message should

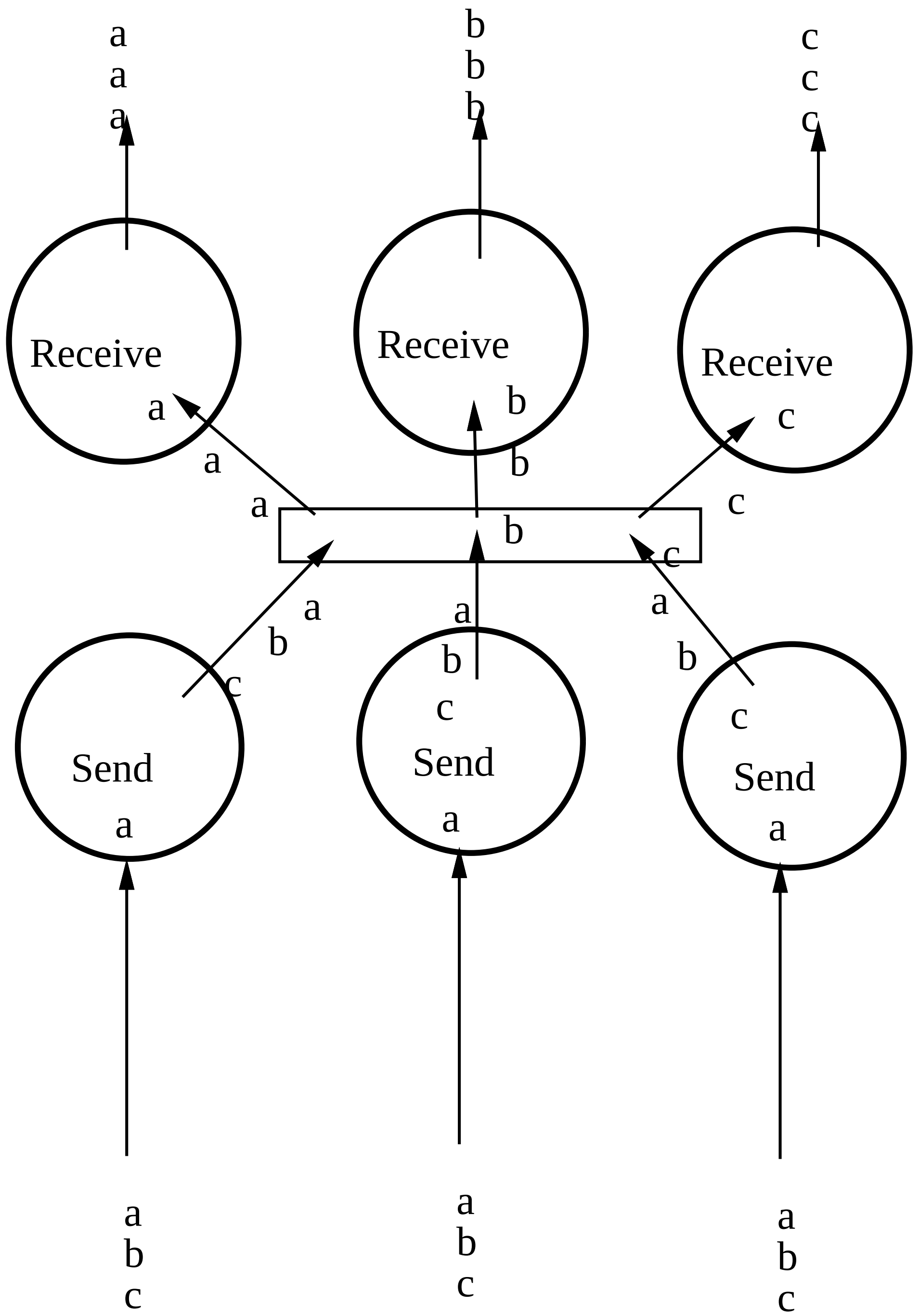

Figure 18: Directed Table Queue

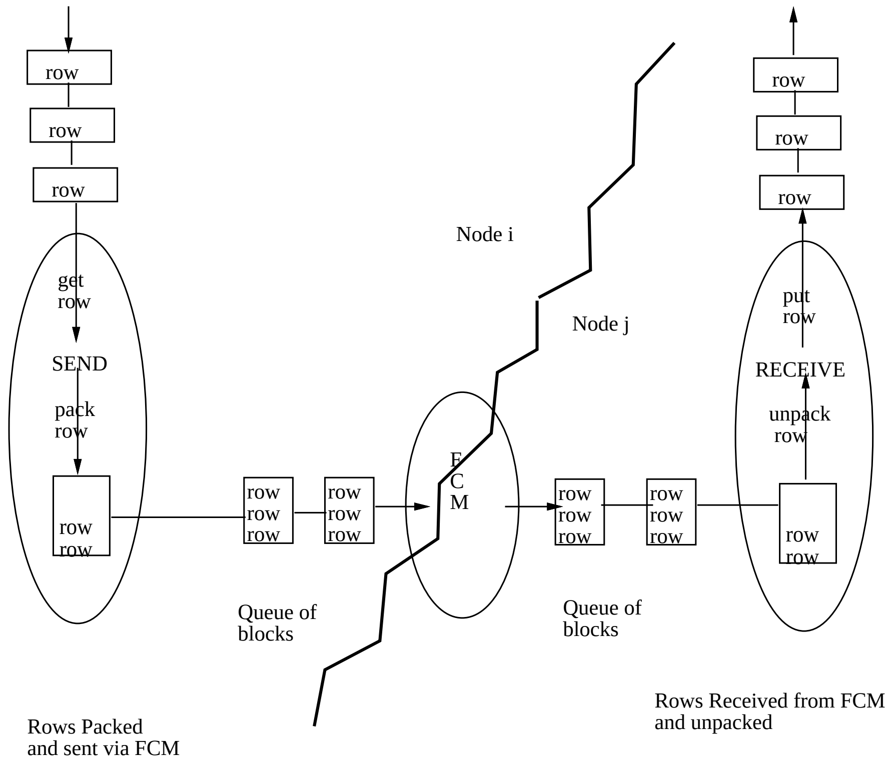

Figure 19: Details of One Table Queue Connection via Communication Manager

be kept. The solution to this question relied on the communications 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.)

5.4 Interrupt and Error Handling

The assumption inherent in the serial database manager is that either the application is busy, or the database is busy, but not both at the same time. Further, the database is busy doing only one request per application. In DB2 Parallel Edition, not only can the database 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. Each set of processes is started when the cursor is opened, and continues until the end of its partition is reached, or until the cursor is closed.

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. Should it be returned as soon as possible, as late as possible, or in its "natural" order? DB2 Parallel Edition 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 [29].

5.5 Concurrency Control and Recovery

Two-phase commit protocol. One important property of database systems is to guarantee 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/abort phase. The two-phase commit protocol may result in *blocking* if the coordinator fails after it has received votes but before sends out an outcome. When blocking occurs, participants will hold/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 does and the blocking problem can be "resolved" by system administrators, none of the existing

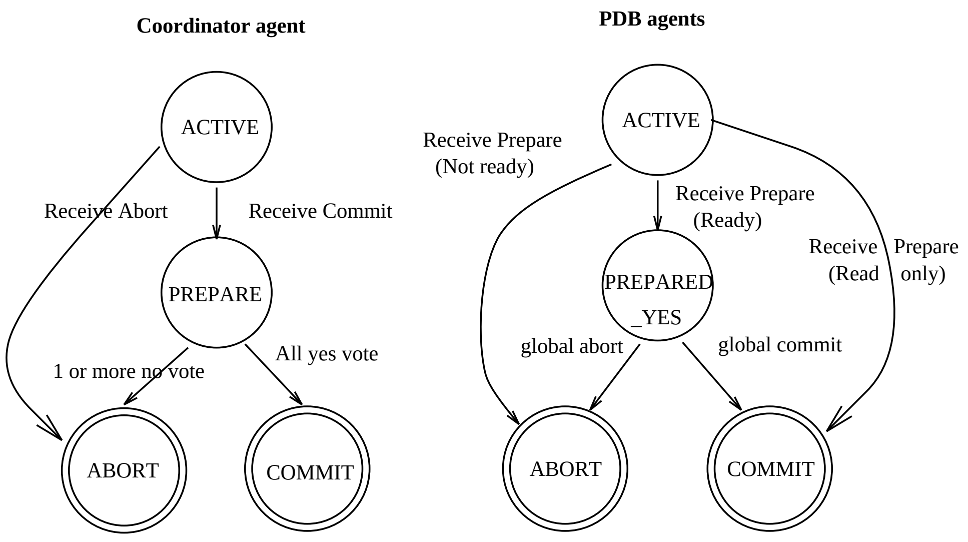

Figure 20: Transaction State Transition in DB2 PE.

commercial systems to-date support the three-phase protocol.

There are three variations of the two phase commit protocol: presumed nothing, presumed commit, and presumed abort [?]. DB2 PE adopts the presumed abort protocol which 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 20 shows the transition of transaction state in DB2 PE. When a transaction starts, a coordinator agent is activated to coordinating the execution of the transaction. Subordinate agents (PDB agents), if needed, are activated by the requests sending from the coordinator agent. Before processing a commit/rollback request, the transaction at both coordinator and subordinate agents are in **active** state.

DB2 PE keeps a transaction node list, **TNL** for short, for every active transaction. The TNL of a transaction keeps the node numbers of all nodes participated 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's TNL including the coordinator node itself. At this point, the coordinator agent enters the **prepare** state.

Upon receiving the prepare-to-commit request, the PDB 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 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 PDB 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 inform all participants who have voted "yes". At this point, the transaction state changed to **commit** at the coordinator node. Otherwise, the the coordinator aborts the transaction and forwards the decision to all subordinate nodes who voted "yes". All actions performed for the transaction at all nodes are rolled back 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 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 SM and SD systems, data can be accessed by multiple nodes concurrently. In order to maintain a consistent global locking, a node needs to get read/write permission from all other nodes before reading or writing a data object that it does not already own an appropriate access permission. In DB2 PE, each processor accesses only the portion of database that it owns locally. Consequently, a processor does not have to request access permission from remote processors before accessing its local data; thus 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 per database per node to maintain locking information and to resolve conflict in lock request 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 (LRB). When two processes of the transaction make lock request to the same object, the one LRB per process design uses more memory space. However, it simplifies the design in processing lock conversion request and lock release request before end of a transaction. The local deadlock detector is implemented as a separate process and is awakens periodically to scan the local lock table and build 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 alternative. The GDD process resides on a pre-configured 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 build a global wait-for graph based on transaction id which is unique across all nodes in a DB2 PE system. After the completion of building the global wait-for graph, the GDD goes through a depth first search to find deadlock cycles in the graph. When a cycle is detected, one or more victims 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 GDD and the coordinator agent will send a rollback request to its subordinate agents (processes) to undo the action of the transaction.

6 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.

6.1 Data Loading

The Load utility allows bulk loading of database tables from flat files. To support applications requiring very large database sizes (100's of Gbytes 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 is stored in a single flat file. Application programming interfaces (APIs) 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 directly 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.

6.2 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 “initialize” the node for use by any database. 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 ALTER 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 remove an existing node from the database configuration. However, if the node to be dropped contains any data belonging to any of the currently defined databases, then the node is not dropped. The Rebalance Nodegroup command (described in Section 6.4.2) must be used to remove data from this node before it can be dropped.

6.3 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 node is damaged for some reason, 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 section on backup/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).

6.4 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/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.

6.4.1 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 has 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.

6.4.2 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 lifetime due to insertion and deletion of data. DB2 PE provides the Rebalance Nodegroup operation to redistribute the data in a table in order to minimize skew.

For a given nodegroup, the rebalance operation considers the 4K partitions in the partitioning map and determines the set of partitions 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 is evenly distributed across the 4K partitions, thus, if the partitions are evenly distributed among the set of nodes, then the data is also assumed to be evenly distributed across the nodes. The user may override this default assumption by providing a *distribution file* which assigns a weight to each of the 4K partitions. In this case, the rebalance 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 rebalancing 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 rebalance operation is applied to all the tables in the nodegroup and each table is rebalanced in turn. If a rebalance operation does not complete successfully, it is likely that some tables in the nodegroup have been rebalanced while others have not. In this case, the operation can be completed by issuing the rebalance command with the *restart* option. It is also possible to issue the rebalance command with a *rollback* option, in order to undo the effects of the failed rebalance. The Rebalance Nodegroup command is an on-line operation which locks only the table that is currently being rebalanced. All other tables in the nodegroup are normally accessible.

In addition to the Rebalance command, an application programming interface (API) is provided that permits users to redistribute data by specifying a *target partitioning map (PM)* for a given nodegroup. The API initiates data redistribution of all tables in the nodegroup using the target PM. This API 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, viz. 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 PARTS table:

Query 1:

```
SELECT PARTITION(PARTS), COUNT(*)
FROM PARTS
GROUP BY PARTITION(PARTS)
ORDER BY PARTITION(PARTS)
```

Query 2:

```
SELECT NODE(PARTS), COUNT(*)
FROM PARTS
GROUP BY NODE(PARTS) ORDER BY NODE(PARTS)
```

The output of Query 1 is a set of 4K 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.

6.4.3 Backup/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/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 neccesary 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.

6.4.4 High Availability

High availability is supported by the use of HACMP [31] 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. 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 take over 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 take over capability, whereas it is important not to degrade overall system performance. Thus, it may be acceptable to have a particular node become inaccessible for say, tens of 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 which 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.

6.5 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. This data is 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.

7 Results

We have performed a number of internal and customer benchmarks on DB2 PE and a brief synopsis of these results are presented here. The results are divided into 3 categories:

- Stand-alone numbers - on capacity, load times, etc.
- 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 sizes.
- Scaleup - this metric measures the performance of the system as the database sizes, the number of concurrent users and/or the number of nodes are scaled proportionately.

The system configuration for many of the benchmarks has been IBM's 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 MB of memory and 2 to 48 GB 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.

The results that are described in this section were measured using available versions of the DB2 PE software and specific hardware configurations. As such, the results obtained in a different hardware and/or software environment may vary significantly. Users of these results should verify if they are applicable for their environments. We encourage the reader to look for the general trend in these results (particularly for the speedup and scaleability experiments) rather than focusing on the particular performance numbers in the figures.

7.1 Stand-alone Metrics

Table 1 describes the important stand-alone metrics based on results of benchmarks 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 date in normalized form), database size (The space occupied once the data has been loaded in DB2 PE, relevant indexes created and any denormalized tables required have been 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 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

Data size	100GB and larger
Database size	250GB and larger
Table size	84GB (2,000,000,000 rows)
Total disk space	over 600GB
Data Splitting	2 GB/Node/Hr
Data Load Rate	2 GB/Node/Hr

Table 1: Stand-alone Metrics

over 2 billion rows. We expect to support configurations in the TeraByte size. To put this in perspective, even the mainframe relational databases are rarely over 200 GB in size. Some of the measurements were done using tables larger than the 64GB limit of many RDBMS.

Another very important metric is **Data Load times**. Our FastLoad utility is able to load data at rates of up to **2 GB/node/hour**. The dataload utility runs in parallel at all nodes, hence it demonstrates completely linear speedup of load rates. In a 32 node system, one could load at the rate of 64 GB/hour.
Before loading the data, it 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 OS platforms including AIX, 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 some cases this was done by sending the data in file format (using FTP). In other cases 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**.

The final stand-alone metric is **availability**. In these benchmarks, we have tried to maintain a couple of *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 fancy availability mechanisms such as twin-tailing of disks. For all other nodes, in the event of failure, failure, the data residing on the failed node can be reloaded onto the spare node and the spare node is then wheeled in as a replacement node. We have been

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

Table 2: Database Description for Speedup Experiments

able to accomplish this task in times that are only dependent on the data load times for the node. For example, on the 100 GB database on 46 nodes, this task was accomplished in less than 2 hours.

7.2 Speedup Results

For speedup, we present results from an internal benchmark performed on 4, 8, 16, and 32 node SP2 systems. 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 besides 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 row size of 1000 bytes.

7.2.1 Scan Performance

Figure 21 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 which can be a maximum of 8.

The scan of 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 upto 16 nodes and only then becomes slightly sublinear. If the table scans had been performed using more nodes, the execution times will eventually flatten out when the table partitions at each node become small enough 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

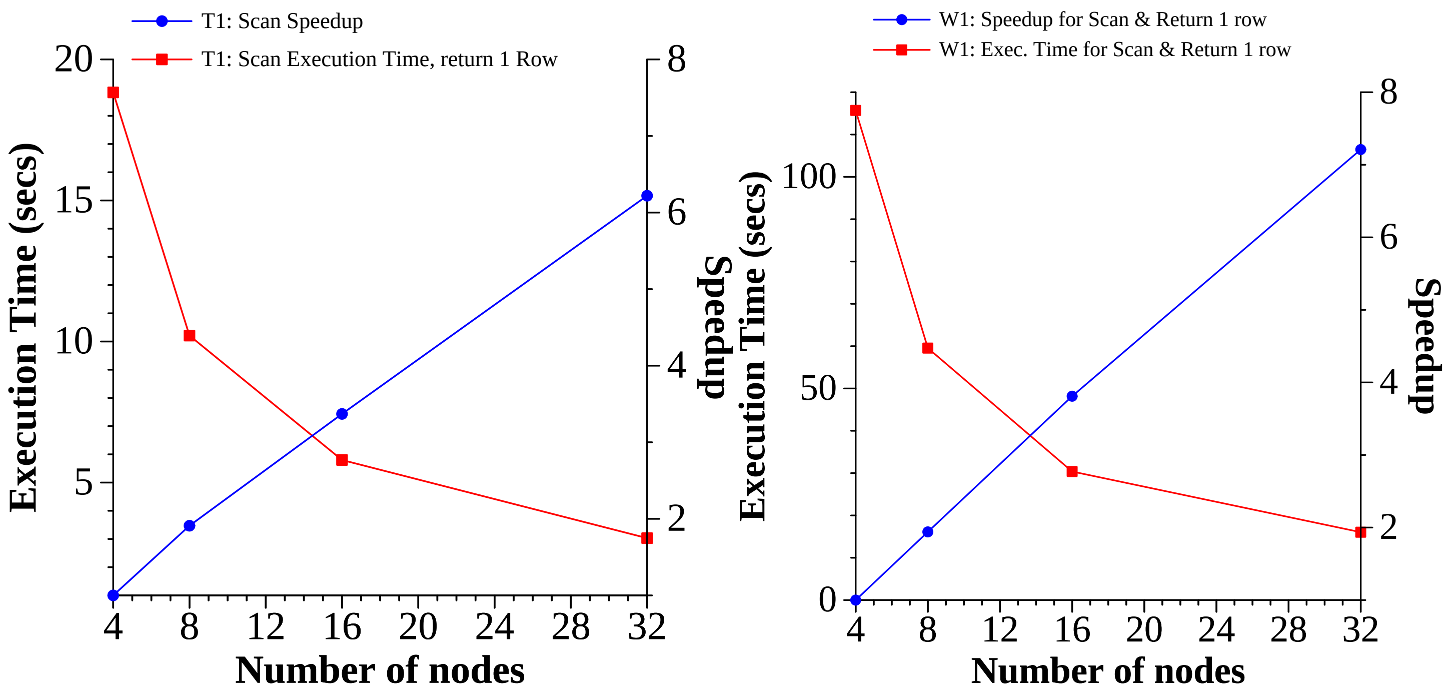

Figure 21: Execution Times and Speedup of Parallel Scan returning 1 Row on T1 and W1 tables

the sizes of the tables for any operation.

Figure 22 illustrates the performance of a Parallel Scan operation on the W1 table that returns 10% 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 done at the Coordinator node in fetching 100,000 rows (10%) 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 requires the application 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 such that each task operates on a subset of the database nodes. Section 8 discusses further the issue of parallel applications.

7.2.2 Insert, Update Performance

Figure 23 shows the execution times for performing Insert into a Temporary table of 1 Row, 1% rows, and 10% rows of the T1 table. The figure is plotted using logarithmic scale (base 2) on both the X and Y axes. In such a graph, linear execution time curves (with appropriate

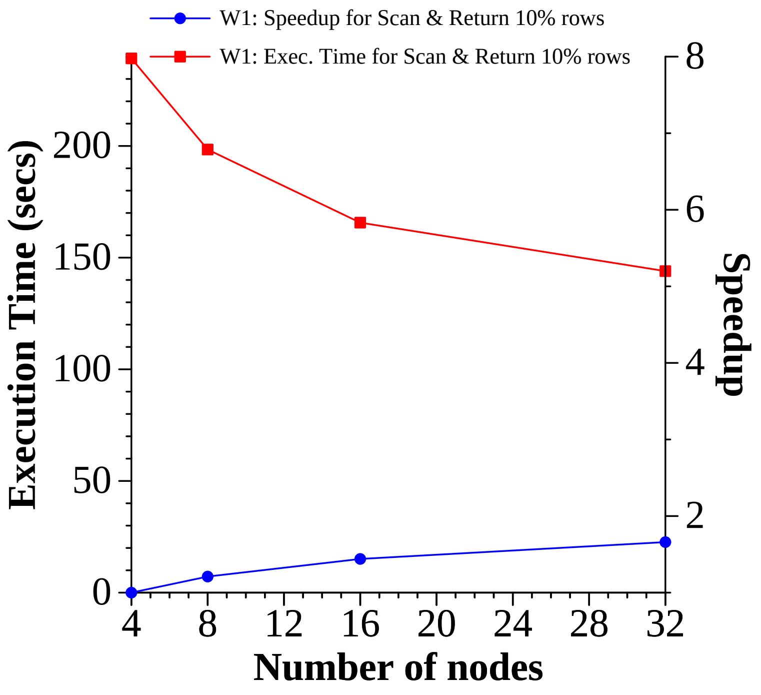

Figure 22: Execution Times and Speedup of Parallel Scan returning 10% Rows on W1 table

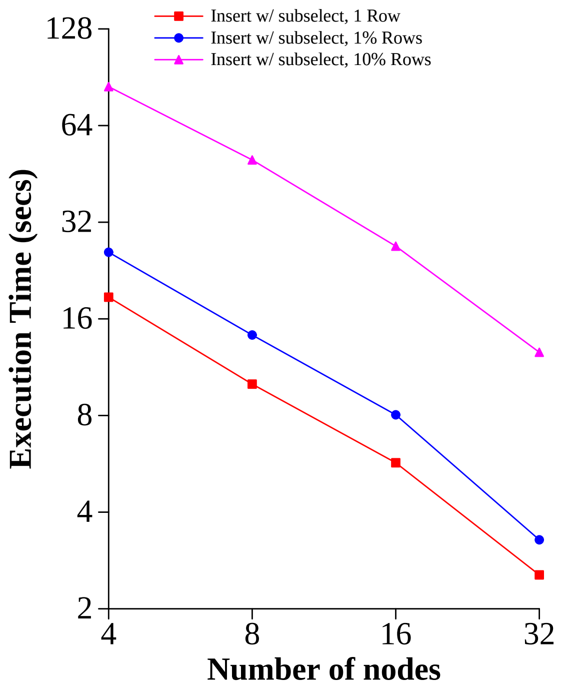

Figure 23: Execution Times of 1 Row, 1%, and 10% Insert with Subselect statements

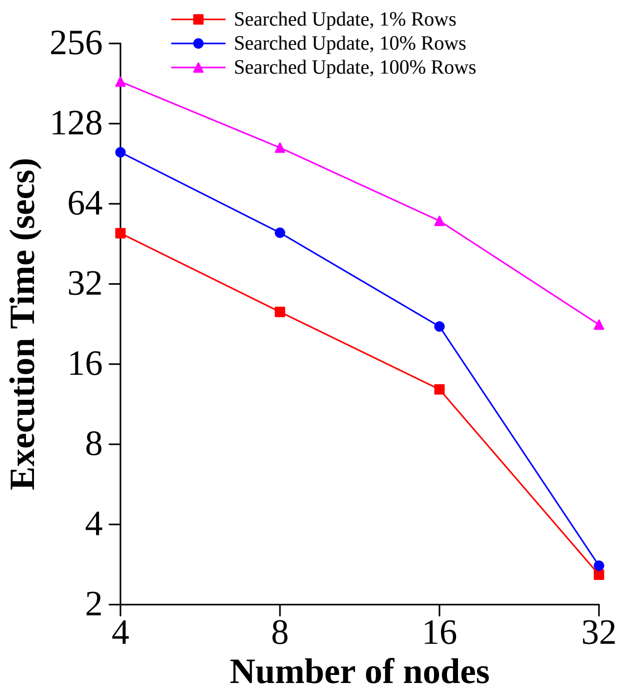

Figure 24: Execution Times of 1%, 10%, and 100% Update with Subselect statements

slope) indicate linear speedup. All three curves show near linear speedup gains with increasing system size. We are able parallelize both the insert as well as the subselect operations of this statement resulting in linear speedup of the statement across different nodes. Figure 24 shows the execution times of update column operations on 1%, 10% and 100% of the rows of the S1 table. Once again, the execution times decrease linearly with increasing system size. The 1% and 10% 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%) 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 where interim results were saved in tables for later analysis or where denormalized tables were created from normalized ones.

7.2.3 Index Create Performance

Figure 25 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

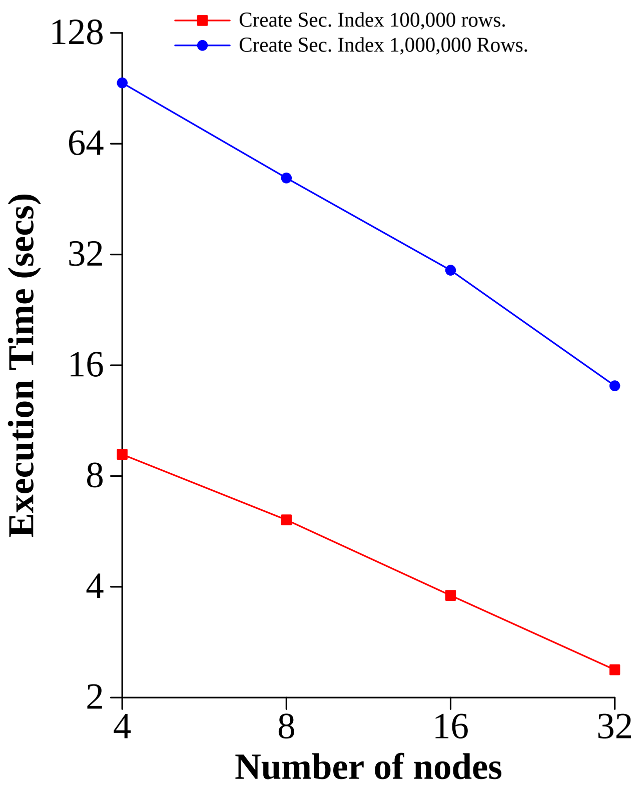

Figure 25: Execution Times of Create Secondary Index Statements on the S1 and T1 tables

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

Table 3: Performance of Complex Queries on 46 node SP2 for database sizes of 10GB and 100GB

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 [].

7.3 Scaleability Results

We present three different types of scaleability results.

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

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

Table 3 shows the performance of DB2 PE on a 46 node SP-2 system for 10 GB and 100GB versions of a scaleable database. The results are shown for a variety of complex queries on the database. The scaling ratios for the different queries varies between 5 and 26.4. Most of the queries show linear or superlinear scalability (ratios less than or equal to 10). For this set

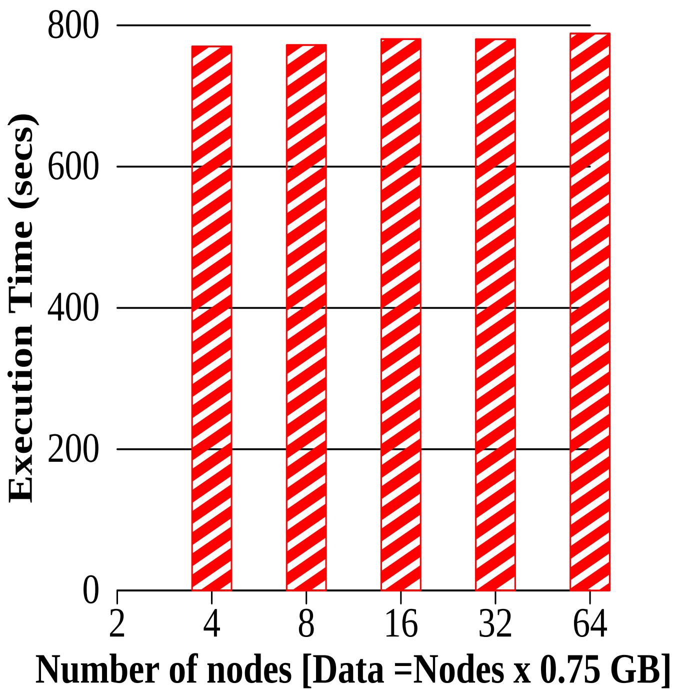

Figure 26: Performance of a Complex Query as both system size and database size are proportionately Increased.

of queries, DB2 PE is able to generate 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 are sublinear ($>$ than 10). These three queries include Order By or Distinct clauses that require sorting to be performed on the results at the coordinator. This causes a serial bottleneck and translates into a reduction in the scaleup ratio.

Figure 26 shows the scaleability results for a complex business query shown below as the system size is increased from 4 to 64 nodes and the database size (which includes index files also) is proportionately increased from 2.5 GB to 40 GB. The query performs without any significant difference in execution cost showing that no additional costs are introduced with the scaling of the system. This is an important result as it indicates that the overhead introduced by the function shpping model of execution are relatively small and do not affect the execution times of complex queries.

```
SELECT Count(*)  FROM Customers
   WHERE  Class IN ('1','2','4','6')
      AND Cust_No NOT IN
        (SELECT O_CUST_NO FROM Offers
```

Query	Single user Exec. Times (secs)	Scaling Ratios	
		20 Users	30 Users
Q1	2	5	8
Q2	15	3.4	3.4
Q3	45	7.1	12.4
Q4	68	12.4	23.24
Q5	93	1.34	2.08
Q6	331	10.6	18.9
Q7	447	3.53	6.55
Q8	493	4.84	8.62
Q9	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 (secs)	8491	8435	12828

Table 4: Performance of Queries with scaling of the number of Concurrent Users

```
     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')
```

Table 4 shows the results of a Concurrent Execution Scaleability test on the system. The test was performed using a 23 GB database on a 8-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 3 and 4 show the Scaleup ratios of the execution times 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 that 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. The multiprogramming level of the system is not constrained by the frontend or coordinator process. Not all parallel database systems have this feature. Many of the parallel database systems are backend machines which have specific interfacing systems for application entry and their multiprogramming levels are limited by the capabilities of this frontend system. DB2 PE does not have this restriction and provides significant concurrency benefits to users.

7.4 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 and all the others have all been extremely positive on the performance of the system. The speedup and scaleability 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 speedup or scaleup linearly. 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 Section 7.2.1. 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.

8 Experiences and Future

Having worked on this project for over four years, we have learnt a lot technically as well as organizationally. The Research division produced an initial prototype, with function being incrementally added. This allowed us to show "proof of concept", and maintain project momentum during the inevitable reorganizations. When the development laboratories picked up the project, we continued in a closely allied joint development mode, producing a product audited to pass ISO 9000 standards. This worked much better than the alternatives: "speccing" the product to death, or just throwing the prototype "over the wall". It has certainly been a remarkable experience how two teams – research and development – with such disparate backgrounds have been able to work so closely together.

Technically, we realized that "The devil is in the details." As the work progressed, it became increasingly clear that while the initial prototyping efforts in the project had given us a good handle on the fundamental issues (section management, run-time, initial query optimization etc.), but the work required to produce an industrial strength parallel database manager was still sizeable.

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

- **Function Shipping**: Right at the onset, we had made a technical decision to go with function shipping with shared nothing hardware. This decision has had multiple benefits. Because we were working on a shared nothing platform, we had to parallelize every database operation – query operator, DDL statement and utility. This resulted in a lot of development effort. However, the positive impact of this was that it forced a discipline on us to think parallel. The result is a scalable product where parallelism is the cornerstone not just in simple scans and joins, but also updates, subqueries, utilities etc. This is in contrast 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 take the best serial plan and to parallelize it. This decision was a matter of programming convenience, encouraged by initial studies which indicated that it often produced the best parallel plan [3]. However, as we delved into more and more complex SQL, it became increasingly clear that our approach, far from minimizing coding, was doing quite the opposite. For example, we were making the same kinds of discovery about order classes 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.

- **Serial Semantics**: We have tried to provide transparent parallelism by maintaining serial semantics. We have succeeded inasmuch as the semantics are explicitly stated in the manuals. However, there are a lot of implicit semantics that a user sometimes comes to expect, and we have realized that it is impossible or inefficient to be able to maintain all of these. For example, a user *expects* that if he issues the same query twice, with no intermediate activity from him or other users, he will see the same results in the same order. However, because we do not guarantee the ordering among nodes, this implicit assumption can be violated in our parallel product. Other examples of such behavior happen when cursor controlled operations interact with read ahead operations, In serial engines 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 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 is correctly in place. We thus put in a lot of effort to make sure that that the data can be initially loaded, balanced, 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 shared nothing because its static data partitionings can be skewed. We have provided rebalancing tools that do not bring a database to a grinding halt and work a 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.

- **Application Interface Bottleneck** (also known as *Amdahl's Law*): Amdahl's Law states that if we can parallelize a fraction f of the total pathlength, the best parallelism that we can achieve is $1/(1-f)$. Consequently, in the queries which return a large answer set, we can only parallelize the generation of the answer set – the coordinator must still return one tuple a time to the application, and hence the parallelism is limited.

  There are a number of ways to alleviate this problem – reduction in the cost of transfer from database to the application, block transfers, parallel applications etc. – we are working on all these aspects as part of our future and continuing work. In addition, it should be realized that the product can simply achieve throughput parallelism of these large answer set queries by submitting different queries from different coordinators. This is not possible in those products where all external interaction is through a dedicated node/processor.

- **Network as a bottleneck**: Because of function shipping, we are able to filter out a lot of the data before it needed to be sent over the network. This coupled with an elaborate partitioning model enables us to minimize data movement between nodes. Still, we have observed that query processing, especially large cross product computations (where there are no predicates, and there is no compatibility of partitioning) can overwhelm a local area network. We are planning on moving our communication to a more lightweight (and hence higher bandwidth) protocol to alleviate this.

- **Complex Queries vs. Transaction Processing**: We realized that to provide a timely product we would have to choose our initial focus carefully. Consequently, we decided to concentrate on parallel complex queries rather than transactions processing. The decision was partially tactical – our primary competitors were playing in the transaction domain and we saw this as an opportunity to leapfrog them. Also, it was felt that the UNIX on line transaction processing market is not yet mature.

  As a consequence of this, while we show completely scalable transaction throughput, we know of several ways to improve. This is an ongoing effort.

As a part of our future work (apart from those mentioned in the passing above), we are concentrating on the following:

- **Parallel Applications**: We believe that the current data mining applications (e.g., those which discover patterns) as well as more complex commercial applications do not execute well against parallel databases mainly because of the large volumes of data crossing the database/application boundary.. 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..
- **Other hardware platforms**: What is the best model for query parallelism on shared memory symmetric multiprocessors (SMP) and clusters of SMPs? While our shared nothing approach provides parallelism on an SMP architecture, we need to make sure that the technical decisions made for the massively parallel environment are also valid in an SMP, and whenever they are not, come up with the best technology.

In summary, our first cut at IBM's parallel database implementation on an open platform has been a successful one, especially for complex query environments. 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., SMP's); and ensure our technology is best of breed in business critical environments.

Acknowledgments

Many people have contributed to the ideas presented in this paper. It is impossible to list all of them. Some of them have worked on this project in the past and have moved on to other projects; their legacy still lives on. The authors acknowledge the contributions from other people in IBM at Austin, Toronto, Santa Teresa Laboratory, Almaden and Hawthorne. Thanks are also due to the performance and benchmarking teams from IBM Power Parallel Systems, Federal Systems Company and Marketing and Sales.

References

[1] Highly available cluster multiprocessor, 1994.

[2] C.K. Baru and S. Padmanabhan. Join and Data Redistribution algorithms for Hypercubes. *IEEE Transactions on Knowledge and Data Engineering*, Apr 1993.

[3] A. Bhide and M. Stonebraker. A performance comparison of two architectures fir fast transaction processing. In *Proceedings of the 1988 Intl. Conf. on Data Engineering*, Los Angeles, CA, Feb 1988.

[4] H. Boral et al. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[5] G. Copeland et al. Data placement in Bubba. In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 99–108, Chicago, IL, June 1988.

[6] D. Davis. ORACLE's parallel punch for OLTP. *Datamation*, Aug 1992.

[7] D.J. DeWitt et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[8] D.J. DeWitt, S. Ghandeharizadeh, and D. Schneider. A performance analysis of the Gamma database machine. In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 350–360, Chicago, IL, June 1988.

[9] D.J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[10] D.J. DeWitt, M. Smith, and H. Boral. A single-user performance evaluation of the Teradata database machine. In *Proceedings of the 2nd International Workshop on High performance Transaction systems (Lecture Notes in Computer Science, No. 359)*, pages 244–269, Pacific Grove, CA, September 1987. Springer-Verlag.

[11] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD COnference*, May 1992.

[12] S. Ghandeharizadeh and D. J. DeWitt. Magic: A multiattribute declustering mechanism for multiprocesor database machines. *IEEE Transactions on Parallel and Distributed Systems*, 5(5), May 1994.

[13] S. Ghandeharizadeh and D.J. Dewitt. Performance analysis of alternative declustering strategies. In *Proceedings of 6th Intl. Conference on Data Engineering*, Los Angeles, CA, February 1990.

[14] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), 1983.

[15] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the 1991 PDIS Conference*, 1991.

[16] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th VLDB Conference*, pages 493–506. Morgan Kaufman, August 1990.

[17] S. Khoshafian and P. Valduriez. Parallel execution strategies for declustered databases. In M. Kitsuregawa and H. Tanaka, editors, *Database Machines and Knowledge base Machines*, pages 458–471, Boston, MA, 1988. Kluwer Acad. Publishers.

[18] 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). In *Proceedings*

of the 16th International conference on Very Large Data Bases, pages 210–221, Brisbane, Australia, August 1990. Morgan Kaufman.

[19] M.S. Lakshmi and P.S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knowledge and Data Engineering*, 2(4):410–424, December 1990.

[20] C. Mohan, H. Pirahesh, W. Tang, and Y. Wang. Parallelism in relational database management systems. *IBM System Journal*, 33(2), 1994.

[21] E. Ozkarahan and M. Ouksel. Dynamic and order preserving data partitioning for database machines. In *Proceedings of the 1985 VLDB Intl. Conference*, 1985.

[22] S. Padmanabhan. *Data Placement in Shared-Nothing Parallel Database Systems.* PhD thesis, EECS Department, University of Michigan, Ann Arbor, 1992.

[23] D. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 469–481, Brisbane, Australia, 1990. Morgan Kaufman.

[24] D.A. Schneider and D.J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD Conference*, Portland, Oregon, May 1989.

[25] P. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Conference*, pages 23–34, 1979.

[26] D. Skeen. Non-blocking commit protocol. In *Proceedings of the 1981 ACM SIGMOD COnference*, Orlando, FL, 1981.

[27] S.Padmanabhan and W. Wilson. On parallel database semantics. In *Proceedings of the CASCON 93 Parallel Databases Workshop*, Toronto, Canada, Oct 1993.

[28] M. Stonebraker. The case for Shared Nothing. *Database Engineering*, 9(1), March 1986.

[29] Teradata Corp., CA. *DBC/1012 Data Base Computer Concepts and Facilities*, c02-0001-05 edition, 1988.

[30] The Tandem Database Group. NonStop SQL: A distributed, high-performance, high-availability implementation of SQL. In *Proceedings of the 2nd International Workshop on High performance Transaction systems (Lecture notes in Computer Science, No. 359)*, pages 60–104, Pacific Grove, CA, September 1987. Springer-Verlag.

[31] The Tandem Performance Group. A benchmark of Non-Stop SQL on the Debit Credit Transaction. In *Proceedings of the ACM SIGMOD Conference*, pages 337–341, Chicago, IL, June 1988.