Chapter 13

PARALLEL DATABASE SYSTEMS

A parallel computer, or multiprocessor, is itself a distributed system made of a number of nodes (processors and memories) connected by a fast network within a cabinet. Distributed database technology can be naturally revised and extended to implement *parallel database systems*, i.e., database systems on parallel computers [DeWitt and Gray, 1992], [Valduriez, 1992]. Parallel database systems exploit the parallelism in data management [Boral, 1988a] in order to deliver high-performance and high-availability database servers at a much lower price than equivalent mainframe computers.

Research on parallel database systems has been initiated in the context of the relational model, which explains why all commercial products are SQL-based. In this chapter, we present the parallel database system approach as a solution to high-performance and high-availability distributed database management. The objectives are to exhibit the advantages and disadvantages of the various parallel system architectures and to present the generic implementation techniques.

This chapter marks the beginning of the relaxation of the assumptions we made in Chapter 1. Here we relax the assumptions about the nature of computers and interconnection between them. In subsequent chapters, we relax others.

This chapter is organized as follows. In Section 13.1, we present the objectives of the database server approach and of the integration of database servers and application servers in a distributed database. In Section 13.5, we make precise the objectives, the functional and architectural aspects of parallel database systems. In particular, we discuss the respective advantages and limitations of the parallel system architectures along several important dimensions including the perspective of both end-users, database administrators and system developers. In Section 13.3, we present the implementation techniques for data placement, data processing, query optimization and load balancing. In Section 13.4, we focus on the problems of parallel query execution for which we provide a complete solution in Section 14.3 in the context of hierarchical architectures.

13.1 DATABASE SERVERS

In this section we introduce the database server approach, which enables distributed applications to access a remote database server. We also discuss their use in distributed databases.

13.1.1 Database Server Approach

Typically, a DBMS runs as a system program on a computer, which is shared by other system and application programs. This traditional approach has several shortcomings. As a result of the recent advances in database theory and technology, the size of the databases and the variety of applications have significantly increased. Nowadays, some databases have several hundred gigabytes or even several terabytes of data. Their management by a general-purpose computer may result in poor utilization of computer resources shared between applications and other complex software programs with different requirements. For example, the DBMS can easily congest the main memory with useless data and saturate the central processor when selecting the relevant data. This approach is inefficient because the general-purpose operating system does not satisfy the particular requirements of database management. This situation stems from the excessive centralization of data and application management functions in the same computer.

A solution to that problem appeared in the early 1970s [Canaday et al., 1974]. The idea is to offload the central processor by isolating the database management functions from the main computer and grouping them in another computer dedicated to their execution. The main computer executing the application programs was termed the *host computer*; the dedicated computer was called the *database machine*, *database computer*, or *backend computer*. We will use, instead, the more recent terms *application server* for the host computer and *database server* for the dedicated computer. Today, application servers can be anything from a personal computer or workstation to a more general-purpose computer connected to client servers such as personal computers or network computers. Figure 13.1 illustrates a simple view of the database server approach, with application servers connected to one database server via a communication network. This follows the architectural model of Figure 4.7, with the user processor and data processor functions performed by the application server and database server, respectively. The application server manages the application which includes the user interface and the parsing of user queries to be submitted to the database server, and manages the interface and communication with the database server for sending commands and receiving results. The application servers may also run other system and application programs. The database server manages the interface and communication with the application server and performs the database functions.

The database server approach has several potential advantages. First, the single focus on data makes possible the development of specific techniques for increasing data reliability and availability. Second, the overall performance of database management can be significantly enhanced by the tight integration of the

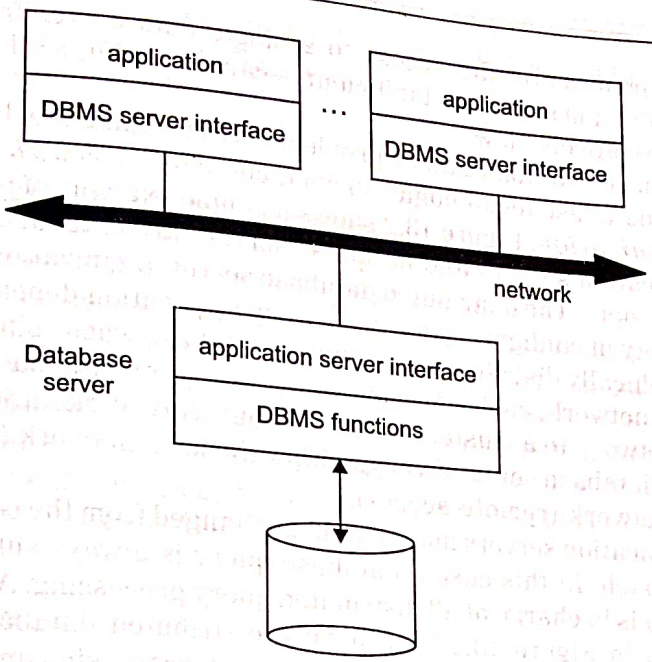

Figure 13.1. Database Server Approach

database system and a dedicated database operating system. Third, a database server fits naturally in a client-server or distributed environment. Finally, a database server can also exploit recent hardware architectures, such as multiprocessor computers to enhance both performance and data availability.

Although these advantages are significant, they can be offset by the overhead introduced by the additional communication between the application and the data servers. For example, accessing the database server one record at a time may incur a prohibitive communication cost since at least two messages must be exchanged for each record that is useful to the application program. The communication cost can be amortized only if the server interface is sufficiently high level to allow the expression of complex queries involving intensive data processing. The relational model, which favors set-oriented manipulation of data, has therefore been the natural data model supported by the database server approach. As a result, most commercial database servers today are relational.

13.1.2 Database Servers and Distributed Databases

The centralized server approach enables distributed applications to access a single database server efficiently. It is often a cost-effective alternative to distributed databases, whereby all the difficult problems of distributed database management disappear at the local database server level. However, this centralized approach is likely to suffer from the traditional limitations of centralized databases. The addition of new application servers in a local network is technically easy but may require the expansion of the database server's processing power and storage

capacity. Furthermore, the access to a single data server from geographically distant application servers is inefficient because communication over a wide area network is relatively slow.

The natural solution to these problems is to combine the database server and distributed database technologies in what could be termed *distributed database server organization*. Figure 13.2 shows a simple example of this organization, in which application servers and database servers are extended with a distributed DBMS component. The distributed database server organization can accommodate a large variety of configurations, each being application dependent. For example, in a geographically distributed distributed database whose sites are connected by a wide area network, each site can consist of a single database server connected by a local network to a cluster of workstations. Any workstation could access the data at any database server through either the local network (local access) or the wide area network (remote access).

The application servers may remain unchanged from the centralized database server approach. In this case a database query is always submitted to the local server, which is in charge of all distributed query processing. An alternative which is illustrated in Figure 13.2 is to have a distributed database interface at the application servers to provide distributed query processing and distributed transaction control. This avoids the systematic access to a single database server in order to retrieve from a remote database server. Although more complex, this solution more efficiently supports configurations where several database servers are connected by a local network.

In the distributed server organization, each database server is fully dedicated to distributed and centralized database management. Therefore, a first solution

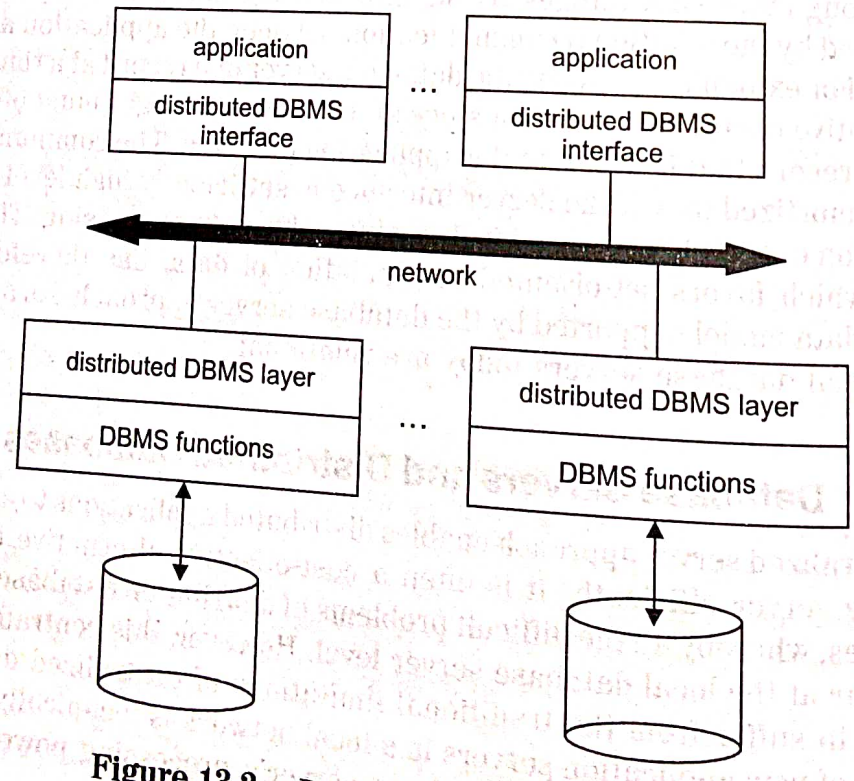

Figure 13.2. Distributed Database Servers

to improve performance is to implement the DBMS and distributed DBMS modules on top of a distributed database operating system running on a traditional (uniprocessor) computer. Another solution goes one step further and uses a parallel database system.

13.2 PARALLEL ARCHITECTURES

In this section we demonstrate the value of parallel systems for efficient database management. We motivate the needs for parallel database systems by reviewing the requirements of very large information systems using current hardware technology. We present the functional and architectural aspects of parallel database systems. In particular, we present and compare the conventional architectures: shared-memory, shared-disk and shared-nothing architectures [Bergsten et al., 1992] and the hybrid exemplified by hierarchical and non-uniform memory access (NUMA) architectures [Graefe, 1993].

13.2.1 Objectives

Parallel processing exploits multiprocessor computers to run application programs by using several processors cooperatively, in order to improve performance. Its prominent use has long been in scientific computing by improving the response time of numerical applications [Kowalik, 1985], [Sharp, 1987]. The recent developments in both general-purpose MIMD parallel computers using standard microprocessors and parallel programming techniques [Osterhaug, 1989] have enabled parallel processing to break into the data processing field.

Parallel database systems combine database management and parallel processing to increase performance and availability. Note that performance was also the objective of the *database machines* (DBMs) in the 70s and 80s [Hsiao, 1983]. The problem faced by conventional database management has long been known as "I/O bottleneck" [Boral and DeWitt, 1983], induced by high disk access time with respect to main memory access time (typically hundreds of thousands times faster). Initially, DBM designers tackled this problem through special-purpose hardware (e.g., by introducing data filtering devices within the disk). However, they failed because of a poor price/performance when compared to the software solution which can easily benefit from hardware progress in silicon technology. A notable exception to these failures is the CAFS-ISP filtering device [Babb, 1979] which is bundled within ICL disk controllers for fast associative search and can be used by the INGRES system (when the optimizer decides to do so).

An important result of DBM research, however, is in the general solution to the I/O bottleneck. We can summarize this solution as *increasing the I/O bandwidth through parallelism*. For instance, if we store a database of size D on a single disk with throughput T, the system throughput is bounded by T. On the contrary, if we partition the database across n disks, each with capacity D/n and throughput T' (hopefully equivalent to T), we get an ideal throughput of $n * T'$ which can be better consumed by multiple processors (ideally n). Note that the main

memory database system solution [Eich, 1989] which tries to maintain the database in main memory is complementary rather than alternative. In particular, the "memory access bottleneck" can also be tackled using parallelism in a similar way.

Therefore, parallel database system designers strived to develop software-oriented solutions in order to exploit multiprocessor hardware. The objectives of parallel database systems can be achieved by extending distributed database technology, for example, by partitioning the database across multiple (small) disks so that much inter- and intra-query parallelism can be obtained. This can lead to significant improvements in both response time and throughput (number of transactions per second). Motivated by set-oriented processing and application portability, most of the work in this area has focused on supporting SQL. Some relational database products implement this approach, e.g., Teradata's DBC and Tandem's NonStopSQL and the number of such products will increase as the market for general-purpose parallel computers expands. In fact, there are now excellent implementations of existing DBMSs such as INFORMIX and ORACLE on parallel computers.

A parallel database system can be loosely defined as a DBMS implemented on a tightly-coupled multiprocessor. This definition includes many alternatives ranging from the straightforward porting of an existing DBMS, which may require only rewriting the operating system interface routines, to a sophisticated combination of parallel processing and database system functions into a new hardware/software architecture. As always, we have the traditional trade-off between portability (to several platforms) and efficiency. The sophisticated approach is better able to fully exploit the opportunities offered by a multiprocessor at the expense of portability. Interestingly, this gives different advantages to computer manufacturers and software vendors. It is therefore important to characterize the main points in the space of alternative parallel system architectures. In order to do so, we will make precise the parallel database system solution and the necessary functions. This will be useful in comparing the parallel database system architectures.

13.2.2 Functional Aspects

A parallel database system acts as a database server for multiple application servers in the now common client-server organization in computer networks. The parallel database system supports the database functions and the client-server interface, and possibly general-purpose functions. The latter capability distinguishes a parallel database system from a database machine which is fully dedicated to database management and cannot, for instance, run a C program written by a user. To limit the potential communication overhead between client and server, a high-level powerful interface (set-at-a-time rather than record-at-a-time) that encourages data-intensive processing by the server is necessary.

Ideally, a parallel database system should provide the following advantages with a much better price/performance than its mainframe counterparts. To some extent, these advantages are also those of distributed database systems.

1. **High-performance.** This can be obtained through several complementary solutions: database-oriented operating system support, parallelism, optimization, and load balancing. Having the operating system constrained and "aware" of the specific database requirements (e.g., buffer management) simplifies the implementation of low-level database functions and therefore decreases their cost. For instance, the cost of a message can be significantly reduced to a few hundred of instructions by specializing the communication protocol. Parallelism can increase throughput, using inter-query parallelism, and decrease transaction response times, using intra-query parallelism. However, decreasing the response time of a complex query through large-scale parallelism may well increase its total time (by additional communication) and hurt throughput as a side-effect. Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query. Load balancing is the ability of the system to divide a given workload equally among all processors. Depending on the multiprocessor architecture, it can be achieved by static physical database design or dynamically at run-time.

2. **High-availability.** Because a parallel database system consists of many similar components, it can exploit data replication to increase database availability. In a highly-parallel system with many small disks, the probability of a disk failure at any time can be higher (than in an equivalent mainframe). Therefore, it is essential that a disk failure does not imbalance the load, e.g., by doubling the load on the available copy. Solutions to this problem require partitioning copies in such a way that they can also be accessed in parallel [Hsiao and DeWitt, 1991].

3. **Extensibility.** In a parallel environment, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should be easier. Extensibility is the ability of smooth expansion of the system by adding processing and storage power to the system. Ideally, the parallel database system should demonstrate two advantages [DeWitt and Gray, 1992]: *linear scaleup* and *linear speedup*. Linear scaleup refers to a sustained performance for a linear increase in both database size and processing and storage power. Linear speedup refers to a linear increase in performance for a constant database size and linear increase in processing and storage power. Furthermore, extending the system should require minimal reorganization of the existing database. Linear scaleup and linear speedup are not appropriate objectives for distributed DBMS.

Assuming a client-server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical RDBMS. The differences, though, have to do with implementation of these functions which must now deal with parallelism, data partitioning and replication, and distributed transactions. Depending on the architecture, a processor can support all (or a subset) of these subsystems. Figure 13.3 shows the architecture using these subsystems named after [Bergsten et al., 1991].

1. **Session Manager.** It plays the role of a transaction monitor (like TUXEDO [Andrade, 1989]), providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems. Therefore, it initiates and closes user sessions (which may contain multiple transactions). In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.

2. **Request Manager.** It receives client requests related to query compilation and execution. It can access the database directory which holds all meta-information about data and programs. The directory itself should be managed as a database in the server. Depending on the request, it activates the various compilation phases, triggers query execution and returns the results as well as error codes to the client application. Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.

3. **Data Manager.** It provides all the low-level functions needed to run compiled queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc. If the request manager is

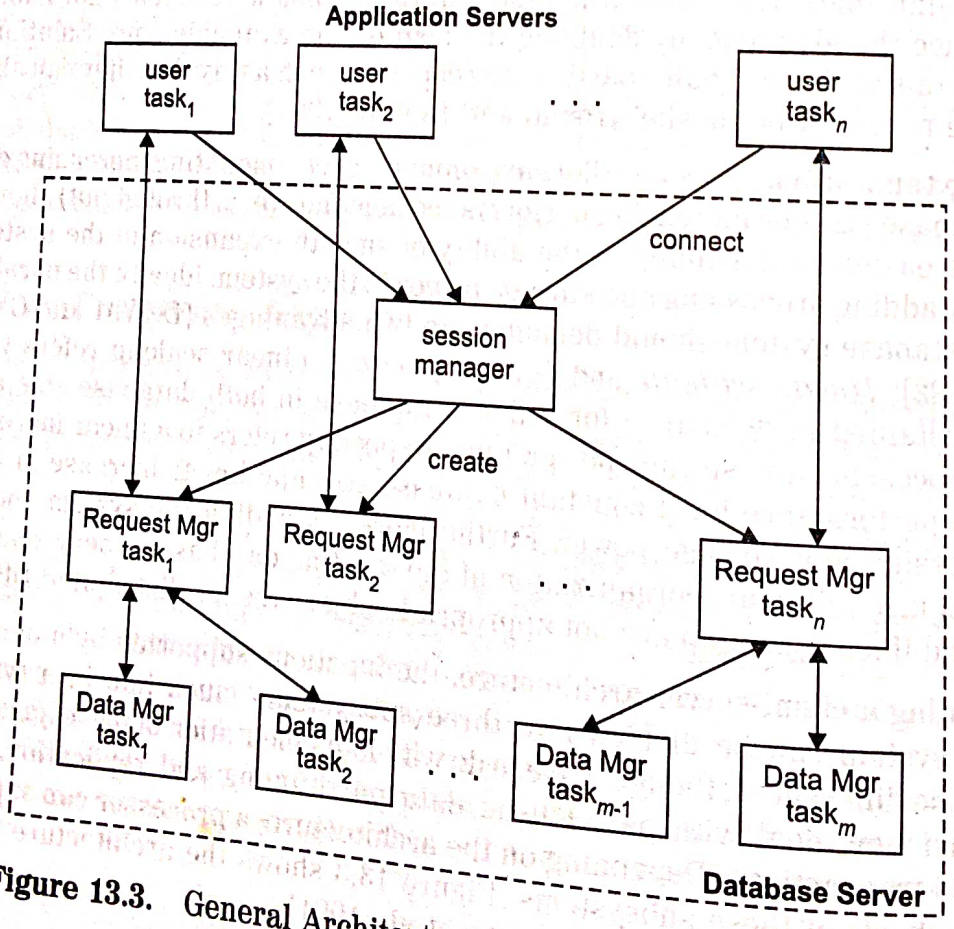

Figure 13.3. General Architecture of a Parallel Database System

able to compile dataflow control, then synchronization and communication among data manager modules is possible. Otherwise, transaction control and synchronization must be done by a request manager module.

13.2.3 Parallel System Architectures

A parallel system represents a compromise in design choices in order to provide the aforementioned advantages with a better cost/performance. One guiding design decision is the way hardware components, i.e., processors, memories, and disks, are interconnected through some fast communication medium. Parallel system architectures range between two extremes, the *shared-memory* and the *shared-nothing* architectures, and a useful intermediate point is the *shared-disk* architecture [Pirahesh et al., 1990]. More recently, hybrid architectures such as *hierarchical* or *NUMA* architectures try to combine the benefits of shared-memory and shared-nothing.

Shared-Memory

Tightly Coupled

In the shared-memory approach (see Figure 13.4), any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch). Several new mainframe designs such as the IBM3090, and symmetric multiprocessors such as Sequent and Bull's Escala follow this approach.

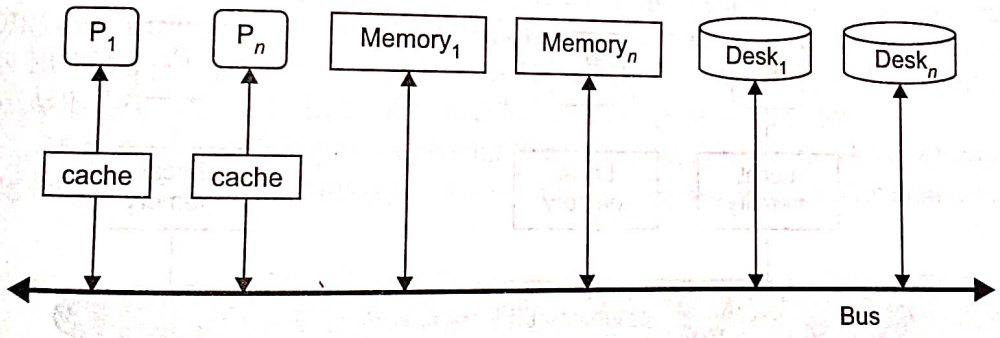

Figure 13.4. Shared Memory Architecture

Examples of shared-memory parallel database systems include XPRS [Hong, 1992], DBS3 [Bergsten et al., 1991], and Volcano [Graefe, 1990], as well as portings of major commercial DBMSs on shared-memory multiprocessors. In a sense, the implementation of DB2 on an IBM3090 with 6 processors [Cheng et al., 1984] was the first example. Most shared-memory commercial products today can exploit inter-query parallelism to provide high transaction throughput and intra-query parallelism to reduce response time of decision-support queries.

Shared-memory has two strong advantages: simplicity and load balancing. Since meta-information (directory) and control information (e.g., lock table) can

be shared by all processors, writing database software is not very different than for single-processor computers. In particular, inter-query parallelism comes for free. Intra-query parallelism requires some parallelization but remains rather simple. Load balancing is excellent since it can be achieved at run-time using the shared-memory.

Shared-memory has three problems: cost, limited extensibility and low availability. High cost is incurred by the interconnect which is fairly complex because of the need to link each processor to each memory module or disk. With faster processors (even with larger caches), conflicting accesses to the shared-memory increase rapidly and degrade performance [Thakkar and Sweiger, 1990]. Therefore, extensibility is limited to tens of processors (20 on a Sequent or Encore). Finally, since the memory space is shared by all processors, a memory fault may affect most processors thereby hurting database availability. A solution is to use duplex memory as in Sequoia systems.

Shared-Disk

In the shared-disk approach (see Figure 13.5), any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory. Then, each processor can access database pages on the shared disk and copy them into its own cache. To avoid conflicting accesses to the same pages, global locking and protocols for the maintenance of cache coherency are needed [Mohan, 1991].

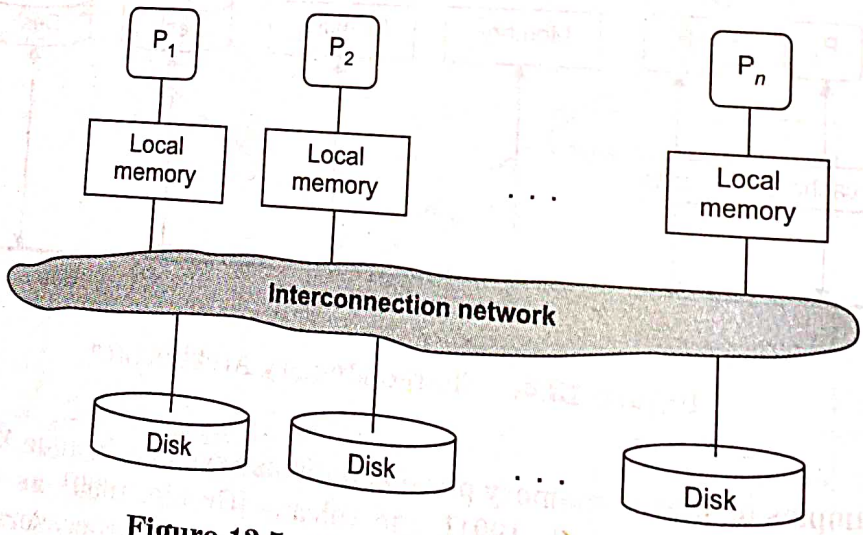

Figure 13.5. Shared-Disk Architecture

Examples of shared-disk parallel database systems include IBM's IMS/VS Data Sharing product and DEC's VAX DBMS and Rdb products. The implementation of ORACLE on DEC's VAXcluster and NCUBE computers is also using the shared-disk approach since it requires minimal extensions of the RDBMS kernel.

Shared-disk has a number of advantages: cost, extensibility, load balancing, availability, and easy migration from uniprocessor systems. The cost of the interconnect is significantly less than with shared-memory since standard bus technology may be used. Given that each processor has enough cache memory, interference on the shared disk can be minimized. Thus, extensibility can be better (in the hundreds of processors). Since memory faults can be isolated from other processor-memory nodes, availability can be higher. Finally, migrating from a centralized system to shared-disk is relatively straightforward since the data on disk need not be reorganized.

Shared-disk suffers from higher complexity and potential performance problems. It requires distributed database system protocols, such as distributed locking and two-phase commit. As we have discussed in previous chapters, these can be complex. Furthermore, maintaining the coherency of the copies can incur high communication overhead among the nodes. Finally, access to the shared-disk is a potential bottleneck.

Shared-Nothing

In the shared-nothing approach (see Figure 13.6), each processor has exclusive access to its main memory and disk unit(s). Then, each node can be viewed as a local site (with its own database and software) in a distributed database system. Therefore, most solutions designed for distributed databases such as database fragmentation, distributed transaction management and distributed query processing may be reused.

Examples of shared-nothing parallel database systems include the Teradata's DBC and Tandem's NonStopSQL products as well as a number of prototypes such as BUBBA [Boral et al., 1990], EDS [EDS, 1990], GAMMA [DeWitt et al., 1986], GRACE [Fushimi et al., 1986], and PRISMA [Apers et al., 1992].

As demonstrated by the existing products, e.g., [Tandem, 1988], shared-nothing has three main virtues: cost, extensibility, and availability. The cost advantage

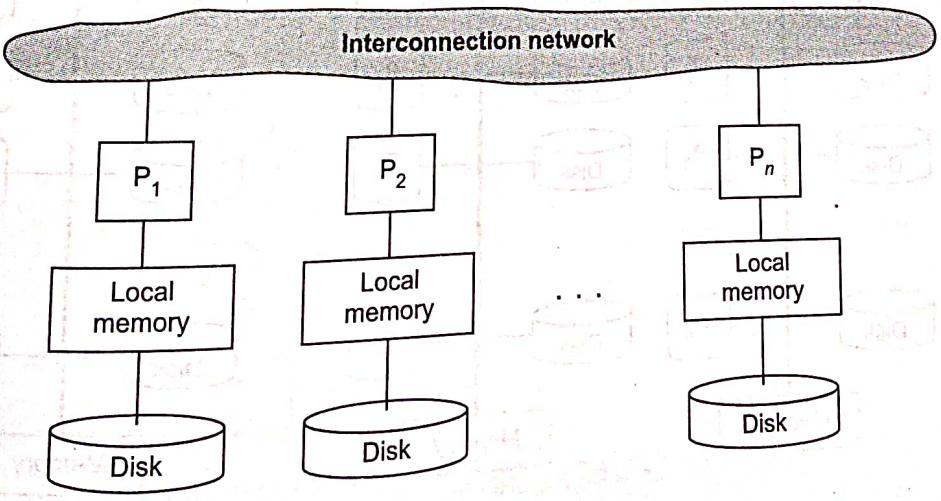

Figure 13.6. Shared-Nothing Architecture

is the same as for shared-disk. By implementing a distributed database design which favors the smooth incremental growth of the system by the addition of new nodes, extensibility can be better (in the thousands of nodes). For instance, Teradata's DBC can accommodate 1024 processors. With careful partitioning of the data on multiple disks, linear speedup and linear scaleup could be achieved for simple workloads. By replicating data on multiple nodes, high availability can be also achieved.

Shared-nothing is also more complex than shared-memory. Higher complexity is due to the necessary implementation of distributed database functions assuming large numbers of nodes. In addition, load balancing is more difficult to achieve because it relies on the effectiveness of database partitioning for the query workloads. Unlike shared-memory and shared-disk, load balancing is decided based on data location and not the actual load of the system. Furthermore, the addition of new nodes in the system presumably requires reorganizing the database to deal with the load balancing issues.

Hierarchical Architectures

Hierarchical architecture (also called cluster architecture), is a combination of shared-nothing and shared-memory. The idea is to build a shared-nothing machine whose nodes are shared-memory. This architecture was first proposed by Bhide [Bhide, 1988], then by Pirahesh [Pirahesh et al., 1990] and Boral [Boral et al., 1990]. A detailed description is proposed by Graefe [Graefe, 1993] and shown in Figure 13.7.

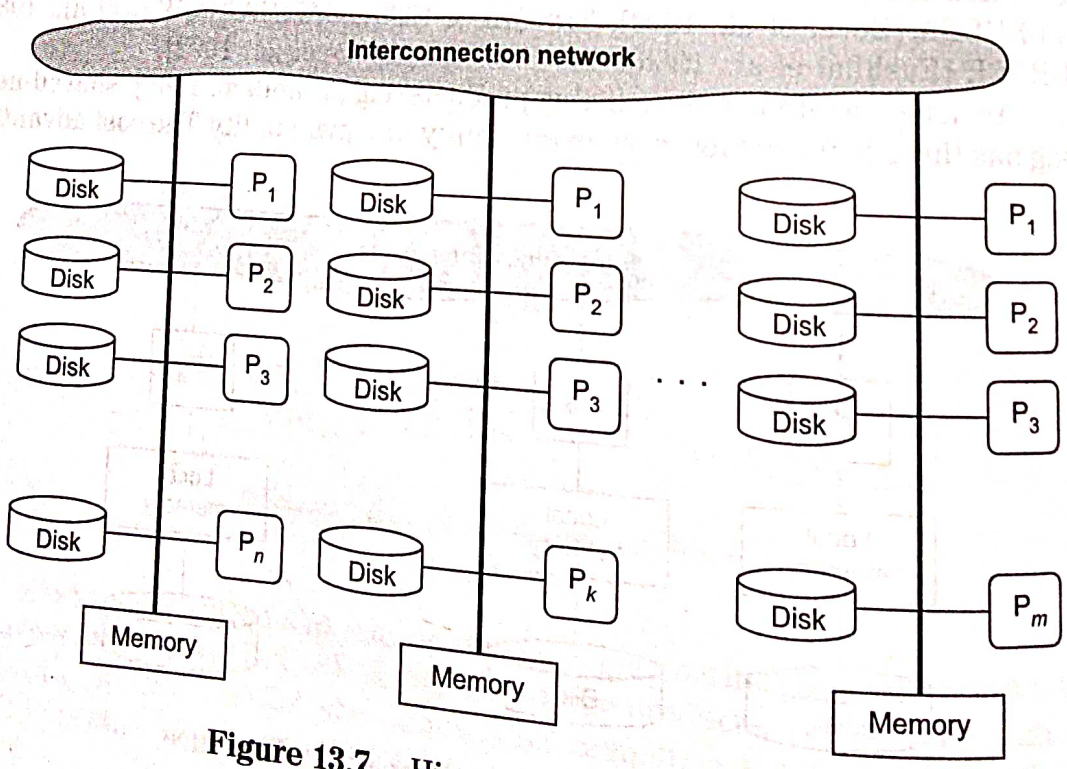

Figure 13.7. Hierarchical Architecture

The advantages of such an architecture are evident. It combines flexibility and performance of shared-memory with high extensibility of shared-nothing. In each shared-memory node (SM-node), communication is done efficiently using the shared-memory, thus increasing performance. Finally, load balancing is eased by the shared-memory component of this architecture.

As an evidence, symmetric multiprocessors (SMP), e.g., Sequent, are moving to scalable cluster architectures, while massively parallel processors (MPP), e.g., NCR's Teradata, are evolving to use shared-memory nodes. Another example is Bull's PowerCluster which is a cluster of PowerPC-based SMP nodes.

NUMA Architectures

With the same goal of combining extensibility and flexibility, shared-memory multiprocessors are evolving towards NUMA architectures. The objective is to provide a shared-memory programming model and all its benefits, in a scalable parallel architecture.

Two classes of NUMA architecture have emerged: Cache Coherent NUMA machines (CC-NUMA) [Goodman and Woest, 1988], [Lenoski et al., 1992], which statically divide the main memory among the nodes of the system and Cache Only Memory Architectures (COMA) [Hagersten et al., 1992], [Frank et al., 1993], which convert the per-node memory into a large cache of the shared address space. Thus, the location of a data item is fully decoupled from its physical address and the data item is automatically migrated or replicated in main memory.

Because shared-memory and cache coherency are supported by hardware, remote memory access is very efficient, only several times (typically 4 times) the cost of local access (See Figure 13.8).

NUMA is now based on international standards and off-the-shelf components. For instance, the Data General nuSMP machine and the Sequent NUMA-Q 2000 are using the ANSI/IEEE Standard Scalable Coherent Interface (SCI) [IEEE, 1992] to interconnect multiple Intel Standard High Volume (SHV) server nodes. Each SHV node consists of 4 Pentium Pro processors, up to 4 gigabytes of memory and dual

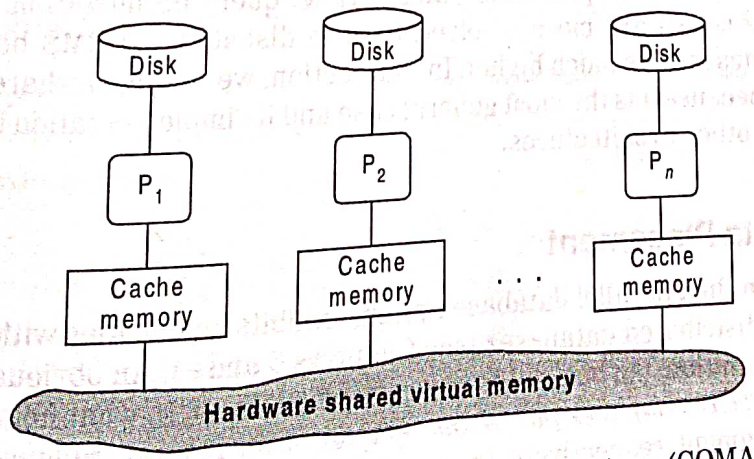

Figure 13.8. Cache Only Memory Architecture (COMA)

peer PCI/IO subsystems [Intel, 1997], [Data General, 1997c]. Other examples of NUMA computers are Kendal Square Research's KSR1 and Convex's SPP1200 which can scale up to hundreds of processors.

The "strong" argument for NUMA is that it does not require any rewriting of application software. However some rewriting is necessary in the operating system and in the database engine [Bouganim et al., 1999]. In response to the nuSMP announcement from Data General, SCO has provided a NUMA version of Unix called Gemini [Data General, 1997b], Oracle has modified its kernel [Data General, 1997a] in order to optimize the use of 64 Gbytes of main memory allowed by NUMA multiprocessors.

Comparisons

Let us briefly compare these alternative design approaches based on their potential advantages (high-performance, high-availability, and extensibility). It is fair to say that, for a small configuration (e.g., less than 20 processors), shared-memory can provide the highest performance because of better load balancing [Bhide, 1988].

Distributed architectures outperform shared-memory in terms of extensibility. NUMA has good performance until a high number of processors. Some years ago, shared-nothing was the only choice for high-end systems (e.g., requiring more than thousands of transactions-per-second of the TPC-B benchmark [Gray, 1993]). Today, NUMA seems the best choice for medium systems. The advantage of such architecture is the simple (shared-memory) programmation model which eases database tuning. High-end systems may use hierarchical architectures for better extensibility and availability.

13.3 PARALLEL DBMS TECHNIQUES

Implementation of parallel database systems naturally relies on distributed database techniques. Essentially, the transaction management solutions can be reused. However, the critical issues for such architectures are data placement, query parallelism, parallel data processing and parallel query optimization. The solutions to these issues are more involved than in distributed DBMS because the number of nodes may be much higher. In this section, we assume a shared-nothing architecture because it is the most general case and its implementation techniques also apply to other architectures.

13.3.1 Data Placement

Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases (see Chapters 5 and 8). An obvious similarity is that fragmentation can be used to increase parallelism. In what follows, we use the terms *partitioning* and *partition* instead of horizontal fragmentation and horizontal fragment respectively, in contrast to the alternative strategy, which consists of *clustering* a relation at a single node. In some papers, e.g., [Livny

et al., 1987], the term *declustering* is also used to mean partitioning. Vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases. Another similarity is that since data is much larger than programs, programs should be executed as much as possible where the data reside [Khoshafian and Valduriez, 1987]. However, there are two important differences with the distributed database approach. First, there is no need to maximize local processing (at each node) since users are not associated with particular nodes. Second, load balancing is much more difficult to achieve in the presence of a large number of nodes. The main problem is to avoid resource contention, which may result in thrashing the entire system (e.g., one node ends up doing all the work while the others remain idle). Since programs are executed where the data resides, data placement is a critical performance issue.

Data placement must be done to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries. In Chapter 9 we have seen that maximizing response time (through intra-query parallelism) results in increased total work due to communication overhead. For the same reason, inter-query parallelism results in increased total work. On the other hand, clustering all the data necessary to a program minimizes communication and thus the total work done by the system in executing that program. In terms of data placement, we have the following trade-off: maximizing response time or inter-query parallelism leads to partitioning, whereas minimizing the total amount of work leads to clustering. As we have seen in Chapter 5, this problem is addressed in distributed databases in a rather static manner. The database administrator is in charge of periodically examining fragment reference frequencies, and when .necessary, must move and reorganize fragments.

An alternative solution to data placement is *full partitioning,* whereby each relation is horizontally fragmented across *all* the nodes in the system. Full partitioning is used in the DBC/1012, GAMMA, and NonStop SQL. There are three basic strategies for data partitioning: round-robin, hash, and range partitioning (Figure 13.9).

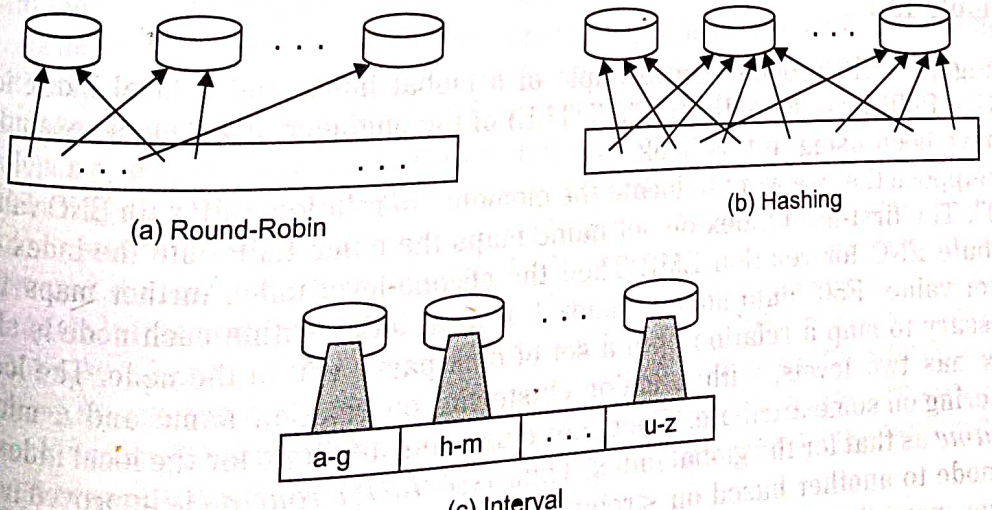

Figure 13.9. Different Partitioning Schemes

1. *Round-robin partitioning* is the simplest strategy, it ensures uniform data distribution. With n partitions, the ith tuple in insertion order is assigned to partition ($imodn$). This strategy enables the sequential access to a relation to be done in parallel. However, the direct access to individual tuples, based on a predicate, requires accessing the entire relation.

2. *Hash partitioning* applies a hash function to some attribute which yields the partition number. This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.

3. *Range partitioning* distributes tuples based on the value intervals (ranges) of some attribute. In addition to supporting exact-match queries as with hashing, it is well-suited for range queries. For instance, a query with a predicate "A between A_1 and A_2" may be processed by the only node(s) containing tuples whose A value is in $[A_1, A_2]$. However, range partitioning can result in high variation in partition size.

In [Livny et al., 1987], the performance of full partitioning is compared to that of clustering the relations on a single disk. The results indicate that for a wide variety of multiuser workloads, partitioning is consistently better. However, clustering may dominate in processing complex queries (e.g., joins). In [Tandem, 1987], the throughput of a system running debit-credit transaction workload [Anon, 1985] in the presence of full partitioning is shown to increase linearly with the number of nodes for up to 32 nodes.

Although full partitioning has obvious performance advantages, high parallel execution might cause a serious performance overhead for complex queries involving joins. For example, in a 1024-node architecture, the worst-case number of messages for a binary join (without select) would be 10,242. Furthermore, full partitioning is not appropriate for small relations that span a few disk blocks. These drawbacks suggest that a compromise between clustering and full partitioning (i.e., *variable partitioning*), needs to be found.

Example 13.1

Figure 13.10 provides an example of a global index and a local index for relation EMP(ENO, ENAME, DEPT, TITLE) of the engineering database example we have been using in this book.

Suppose that we want to locate the elements in relation EMP with ENO value "E50". The first-level index on set name maps the name EMP onto the index on attribute ENO for relation EMP. Then the second-level index further maps the cluster value "E50" onto node number j. A local index within each node is also necessary to map a relation onto a set of disk pages within the node. The local index has two levels, with a major clustering on relation name and a minor clustering on some attribute. The minor clustering attribute for the local index is the *same* as that for the global index. Thus *associative routing* is improved from one node to another based on <relation name, cluster value>. This local index further maps the cluster value "E5" onto page number 91.

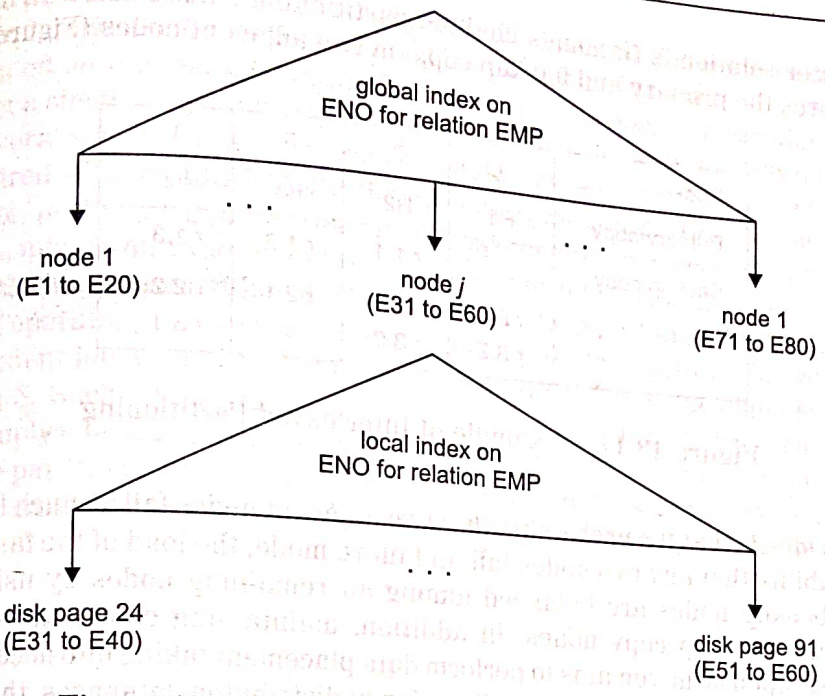

Figure 13.10. Example of Global and Local Indexes

[Copeland et al., 1988] provides experimental results for variable partitioning of a workload consisting of a mix of short transactions (debit–credit like) and complex ones. The results indicate that as partitioning is increased, throughput continues to increase for short transactions. However, for complex transactions involving several large joins, further partitioning reduces throughput because of communications overhead.

A serious problem in data placement is dealing with skewed data distributions which may lead to non-uniform partitioning and hurt load balancing. Range partitioning is more sensitive to skew than either round-robin or hash partitioning. A solution is to treat non-uniform partitions appropriately, e.g., by further fragmenting large partitions. The separation between logical and physical nodes is also useful since a logical node may correspond to several physical nodes.

A final complicating factor is data replication for high availability. The simple solution is to maintain two copies of the same data, a primary and a backup copy, on two separate nodes. This is the *mirrored disks* architecture as promoted by Tandem's NonStop SQL system. However, in case of a node failure, the load of the node having the copy may double, thereby hurting load balancing. To avoid this problem, several high-availability data replication strategies have been proposed for parallel database systems [Hsiao and DeWitt, 1991]. An interesting solution is Teradata's interleaved partitioning which partitions the backup copy on a number of nodes (Figure 13.11). In failure mode, the load of the primary copy gets balanced among the backup copy nodes. But if two nodes fail, then the relation cannot be accessed thereby hurting availability. Reconstructing the primary copy from its separate backup copies may be costly. In normal mode, maintaining copy consistency may also be costly.

A better solution is Gamma's chained partitioning [Hsiao and DeWitt, 1991] which stores the primary and backup copy on two adjacent nodes (Figure 13.12).

Node	1	2	3	4
Primary copy	R1	R2	R3	R4
Backup copy	r 2.3 r 3.2	r 1.1 r 3.2	r 1.2 r 2.1	r 1.3 r 2.2 r 3.1

Figure 13.11. Example of Interleaved Partitioning

The main idea is that the probability that two adjacent nodes fail is much less than the probability that any two nodes fail. In failure mode, the load of the failed node and the backup nodes are balanced among all remaining nodes by using both primary and backup copy nodes. In addition, maintaining copy consistency is cheaper. An open issue remains to perform data placement taking into account data replication. Similar to the fragment allocation in distributed databases, this should be considered an optimization problem.

Node	1	2	3	4
Primary copy	R1	R2	R3	R4
Backup copy	r4	r1	r2	r3

Figure 13.12. Example of Chained Partitioning

13.3.2 Query Parallelism

Inter-query parallelism enables the parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput. Within a query (*intra-query parallelism*), *inter-operator* and *Intra-operator parallelism* are used to decrease response time. Inter-operator parallelism is obtained by executing in parallel several operators of the query tree on several processors while with intra-operator parallelism, the same operator is executed by many processors, each one working on a subset of the data.

Intra-operator Parallelism

Intra-operator parallelism is based on the decomposition of one operator in a set of independent sub-operators, called *operator instances*. This decomposition is done using static and/or dynamic partitioning of relations. Each operator instance

will then process one relation partition also called *bucket*. The operator decomposition frequently benefits from the initial partitioning of the data (e.g., the data is partitioned on the join attribute). To illustrate intra-operator parallelism, let us consider a simple select-join query. The select operator can be directly decomposed into several select operators, each on a different partition and no redistribution is required (Figure 13.13). Note that if the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances. For example, in an exact-match select, only one select instance will be executed if the relation was partitioned by hashing (or range) on the select attribute. For the join operator, it is more complex to decompose the operator. In order to have independent joins, each bucket of the first relation R_i may be joined to the entire relation S. Such a join will be very inefficient (unless if S is very small) because it will imply a broadcast of S on each participating processor. A more efficient way is to use partitioning properties. For example, if R and S are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join

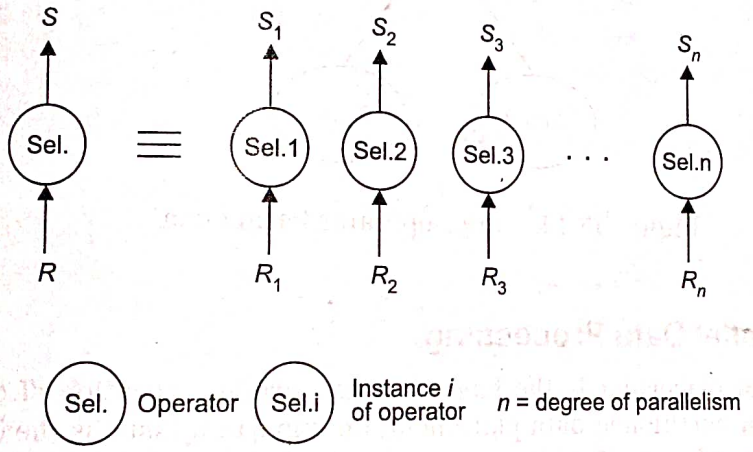

Figure 13.13. Intra-operator Parallelism

into independent joins (see Algorithm 13.3.3 in Section 13.3.3). This is the ideal case which cannot be always used because it depends on the initial partitioning of R and S. In the other cases, one or two operands may be repartitioned [Valduriez and Gardarin, 1984]. Finally, we may notice that the partitioning function (hash, range, round robin) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator (i.e., on each processor). For instance, a hash join using a hash partitioning needs two hash functions. The first one, h_1, is used to partition the two base relations on the join attribute. The second one, h_2, which can be different for each processor, is used to process the join on each processor.

Inter-operator Parallelism

Two forms of inter-operator parallelism can be exploited. With pipeline parallelism, several operators with a producer–consumer link are executed in parallel. For instance, the select operator in Figure 13.14 will be executed in parallel with the subsequent join operator. The advantage of such execution is that the intermediate result is not materialized, thus saving memory and disk accesses. In the example of Figure 13.14, only S may fit in memory. Independent parallelism is achieved when there is no dependency between the operators executed in parallel. For instance, the two select operators of Figure 13.14 can be executed in parallel. This form of parallelism is very attractive because there is no interference between the processors. However, it is only possible for bushy execution (see Section 13.3.4) and may consume more resources [Shekita et al., 1993].

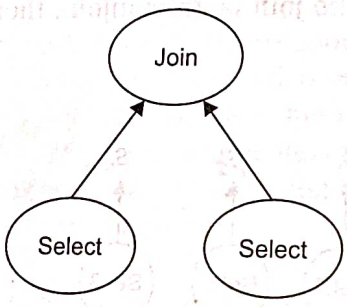

Figure 13.14. Inter-operator Parallelism

13.3.3 Parallel Data Processing

Partitioned data placement is the basis for the parallel execution of database queries. Given a partitioned data placement, an important issue is the design of parallel algorithms for an efficient processing of database operators (i.e., relational algebra operators) and database queries which combine multiple operators. This issue is difficult because a good trade-off between parallelism and communication cost must be reached. Parallel algorithms for relational algebra operators are the building blocks necessary for parallel query processing.

Parallel data processing should exploit intra-operator parallelism. As in Chapter 9, we concentrate our presentation of parallel algorithms for database operators on the select and join operators, since all other binary operators (such as union) can be handled very much like join [Bratbersengen, 1984]. The processing of the select operator in a partitioned data placement context is identical to that in a fragmented distributed database. Depending on the select predicate, the operator may be executed at a single node (in the case of an exact match predicate) or in the case of arbitrary complex predicates at all the nodes over which the relation is partitioned. If the global index is organized as a B-tree-like structure (see Figure 13.10), a select operator with a range predicate may be executed only by the nodes storing relevant data.

The parallel processing of join is significantly more involved than that of select. The distributed join algorithms designed for high-speed networks (see Chapter 9) can be applied successfully in a partitioned database context. However, the availability of a global index at run time provides more opportunities for efficient parallel execution. In the following, we introduce three basic parallel join algorithms for partitioned databases: the parallel nested loop (PNL) algorithm, the parallel associative join (PAJ) algorithm, and the parallel hash join (PHJ) algorithm. We describe each using a pseudo-concurrent programming language with three main constructs: **do-in-parallel**, **send**, and **receive**. **Do-in-parallel** specifies that the following block of actions is executed in parallel. For example,

```
for i from 1 to n do in parallel action A
```

indicates that the action A is to be executed by n nodes in parallel. **Send** and **receive** are the basic communication primitives to transfer data between nodes. **Send** enables data to be sent from one node to one or more nodes. The destination nodes are typically obtained from the global index. **Receive** gets the content.of the data sent to a particular node. In what follows we consider the join of two relations R and S that are partitioned over m and n nodes, respectively. For the sake of simplicity, we assume that the m nodes are distinct from the n nodes. A node at which a fragment of R (respectively, S) resides is called an R-node (respectively, S-node).

The parallel nested loop algorithm [Bitton et al., 1983] is the simplest one and the most general. It basically composes the Cartesian product of the relations R and S in parallel. Therefore, arbitrarily complex join predicates may be supported. This algorithm has been introduced in Chapter 9 in the context of Distributed INGRES. It is more precisely described in Algorithm 13.1, where the join result is produced at the S-nodes. The algorithm proceeds in two phases.

Algorithm 13.1 *PNL*

input: R_1, R_2, . . . , R_m: fragments of relation R;
S_1, S_2, . . . , S_n: fragments of relation S;
JP: join predicate
output: T_1, T_2, . . . ,T_n: result fragments
begin
  for i **from** 1 **to** m **do in parallel** {send R entirely to each S-node}
    send R_i to each node containing a fragment of S
  end-for
  for j **from** 1 **to** n **do in parallel** {perform the join at each S-node}
  begin
    $R \leftarrow \bigcup_{i=1}^{m} R_i$ {receive R_i from R-nodes; R is replicated on $S-nodes$}
    $T_j \leftarrow$ JOIN(R, S_j, JP) {JOIN is a generic funption}
  end-for
end. {PNL}

In the first phase, each fragment of R is sent and replicated at each node containing a fragment of S (there are n such nodes). This phase is done in parallel by m nodes and is efficient if the communication network has a broadcast capability. In this case each fragment of R can be broadcast to n nodes in a single transfer, thereby incurring a total communication cost of m messages. Otherwise, $(m * n)$ messages are necessary.

In the second phase, each S-node j receives relation R entirely, and locally joins R with the fragment S_j. This phase is done in parallel by n nodes. The local join can be done as in a centralized DBMS. Depending on the local join algorithm, join processing may or may not start as soon as data are received. In the first case (e.g., with the nested loop join algorithm), join processing can be done in a pipelined fashion as soon as a tuple of R arrives. In the latter case (e.g., with the sort merge join algorithm), all the data must have been received before the join of the sorted relations begins.

To summarize, the parallel nested loop algorithm can be viewed as replacing the operator $R \bowtie S$ by

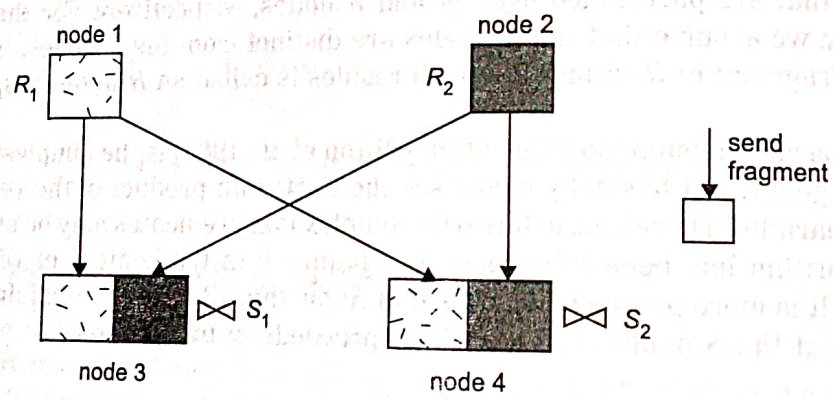

Figure 13.15. Example of Parallel Nested Loop

$$\bigcup_{i=1}^{n} (R \bowtie S_i)$$

Example 13.2

Figure 13.15 shows the application of the parallel nested loop algorithm with $m = n = 2$.

The parallel associative join algorithm, shown in Algorithm 13.2, applies only in the case of equijoin with one of the operand relations partitioned according to the join attribute. To simplify the description of the algorithm, we assume that the equijoin predicate is on attribute A from R, and B from S. Furthermore, relation S is partitioned according to the has h function h applied to join attribute B,

meaning that all the tuples of S that have same value for $h(B)$ are placed at the same node. No knowledge of how R is partitioned is assumed. The application of the parallel associative join algorithm will produce the join result at the nodes where S_i exists (i.e., the S-nodes).

Algorithm 13.2 *PAJ*

```
input: R_1, R_2, ..., R_m: fragments of relation R;
       S_1, S_2, ..., S_n: fragments of relation S;
       JP: join predicate
output: T_1, T_2, ..., T_n: result fragments
begin      {we assume that JP is R.A = S.B and relation S is fragmented
                                          according to the function h(B)}
  for i from 1 to m do in parallel {send R associatively to each S-node}
  begin
    Rij ← apply h(A) to R_i (j = 1, ..., n)
    for j from 1 to n do
      send R_ij to the node storing S_j
    end-for
  end-for
  for j from 1 to n do in parallel          {perform the join at each S-node}
  begin
    R_j ← ⋃_{i=1}^{m} R_ij                  {receive only the useful subset of R}
    T_j ← JOIN(R_j, S_j, JP)
  end-for
end. {PAJ}
```

The algorithm proceeds in two phases. In the first phase, relation R is sent associatively to the S-nodes based on the function h applied to attribute A. This guarantees that a tuple of R with hash value v is sent only to the S-node that contains tuples with hash value v. The first phase is done in parallel by m nodes where R_i's exist. Unlike the parallel nested loop algorithm, the tuples of R get distributed but not replicated across the S-nodes. In the second phase, each S-node j receives in parallel the relevant subset of R, (i.e., R_j), and joins it locally with the fragments S_j. Local join processing can be done as in the parallel nested loop join algorithm.

To summarize, the parallel associative join algorithm replaces the operator $R \bowtie S$ by

$$\bigcup_{i=1}^{n} (R_i \bowtie S_i)$$

Example 13.3

Figure 13.16 shows the application of the parallel associative join algorithm with $m = n = 2$. The squares that are hatched with the same pattern indicate fragments whose tuples match the same hash function.

The parallel hash join algorithm, shown in Algorithm 13.3, can be viewed as a generalization of the parallel associative join algorithm. It also applies in the case of equijoin but does not require any particular partitioning of the operand relations. The basic idea is to partition relations R and S into the same number p of mutually exclusive sets (fragments) $R_1, R_2, \ldots, R_p$, and $S_1, S_2, \ldots, S_p$, such that

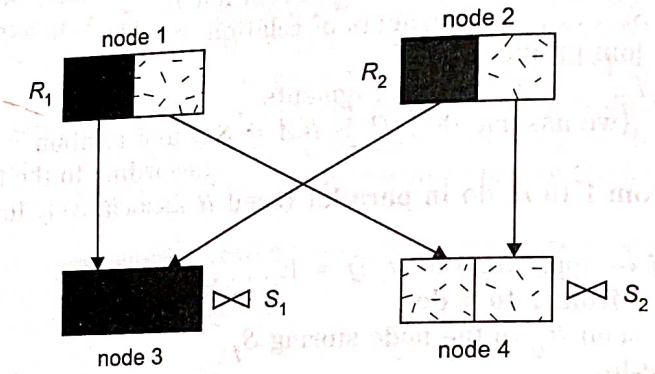

Figure 13.16. Example of Parallel Associative Join

$$R \bowtie S = \bigcup_{i=1}^{p} (R_i \bowtie S_i)$$

Algorithm 13.3 *PHJ*

input: $R_1, R_2, \ldots, R_m$: fragments of relation R;
$S_1, S_2, \ldots, S_n$: fragments of relation S;
JP: join predicate
output: $T_1, T_2, \ldots, T_n$: result fragments
begin {we assume that JP is $R.A = S.B$ and h is a hash function that returns an element of $[1, p]$}

for i **from** 1 **to** m **do in parallel** {hash R on the join attribute}
begin
  $R_{ij} \leftarrow$ apply $h(A)$ to R_i $(j = 1, \ldots, p)$
  for j **from** 1 **to** p **do**
    send R_{ij} to node j
  end-for
end-for

for i **from** 1 **to** n **do in parallel** {hash S on the join attribute}
begin
  $S_{ij} \leftarrow$ apply $h(B)$ to S_i $(j = 1, \ldots, p)$
  for j **from** 1 **to** p **do**

send S_{ij} to node j
end-for
end-for

for j **from** 1 **to** p **do in parallel**
begin
$R_j \leftarrow \cup_{i=1}^{p} R_{ij}$ {perform the join at each S-node}
$S_j \leftarrow \cup_{i=1}^{p} S_{ij}$ {receive from R-nodes}
$T_j \leftarrow \text{JOINC}(R_j, S_j, JP)$ {receive from S-nodes}
end-for
end. {PAJ}

As in the parallel associative join algorithm, the partitioning of R and S can be based on the same hash function applied to the join attribute. Each individual join $(R_i \bowtie S_i)$ is done in parallel, and the join result is produced at p nodes. These p nodes may actually be selected at run time based on the load of the system. The main difference with the parallel associative join algorithm is that partitioning of S is necessary and the result is produced at p nodes rather than at n S-nodes.

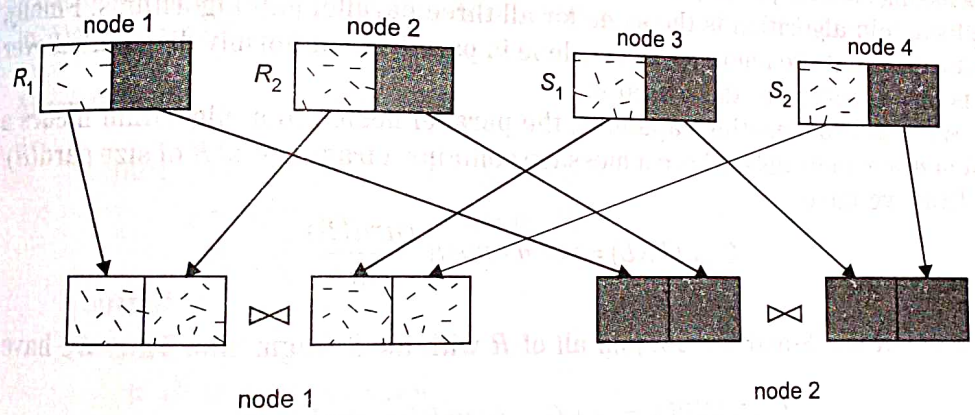

Figure 13.17. Example of Parallel Hash Join

Example 13.4

Figure 13.17 shows the application of the parallel hash join algorithm with $m = n = 2$. We assumed that the result is produced at nodes 1 and 2. Therefore, an arrow from node 1 to node 1 or node 2 to node 2 indicates a local transfer.

Variations of this algorithm for specific multiprocessor architectures are given in [Valduriez and Gardarin, 1984]. An interesting variation is to divide the algorithm in two phases, a *build* phase and a *probe* phase, in order to pipeline the join result to the subsequent operator [DeWitt and Gerber, 1985]. The build phase

hashes R on the join attribute, sends it to the target p nodes which build a hash table for the incoming tuples. The probe phase sends S associatively to the target p nodes which probe the hash table for each incoming tuple. Thus, as soon as the hash tables have been built for R, the S tuples can be sent and processed in pipeline by probing the hash tables.

These parallel join algorithms apply and dominate under different conditions. Join processing is achieved with a degree of parallelism of either n or p. Since each algorithm requires moving at least one of the operand relations, a good indicator of their performance is total cost. To compare these algorithms, we now give a simple analysis of cost, defined in terms of total communication cost, denoted by C_{COM} and processing cost, denoted by C_{PRQ}. The total cost of each algorithm is therefore

$$Cost(Alg.) = C_{COM}(Alg.) + C_{PRO}(Alg.)$$

For simplicity, C_{COM} does not include control messages, which are necessary to initiate and terminate local tasks. We denote by $msg(\#tup)$ the cost of transferring a message of $\#tup$ tuples from one node to another. Processing costs (total I/O and CPU cost) will be based on the function $C_{LOC}(m, n)$ which computes the local processing cost for joining two relations of cardinalities m and n. We assume that the local join algorithm is the same for all three parallel join algorithms. Finally, we assume that the amount of work done in parallel is uniformly distributed over all nodes allocated to the operator.

Without broadcasting capability, the parallel nested loop algorithm incurs a cost of $m * n$ messages, where a message contains a fragment of R of size card(R)/m. Thus we have

$$C_{COM}(PNL) = m * n * msg\left(\frac{card(R)}{m}\right)$$

Each of the S-nodes must join all of R with its S fragments. Thus we have

$$C_{PRO}(NPL) = n * C_{LOC}(card(R), card(S)/n)$$

The parallel associative join algorithm requires that each R-node partitions a fragment of R into n subsets of size $card(R)/(m * n)$ and sends them to n S-nodes. Thus we have

$$C_{COM}(PAJ) = m * n * msg\left(\frac{card(R)}{m}\right)$$

and

$$C_{PRO}(PAJ) = n * C_{LOC}(card(R)/n, card(S)fn)$$

The parallel hash join algorithm requires that both relations R and S be partitioned across p nodes in a way similar to the parallel associative join algorithm. Thus we have

$$C_{COM}(PHJ) = m * p * msg\left(\frac{card(R)}{m * p}\right) + n * p * msg\left(\frac{card(S)}{n * p}\right)$$

and

$$C_{PRO}(PHJ) = n * C_{LOC}(card(R)/n, card(S)/n)$$

Let us first assume that $p = n$. In this case the join processing cost for the PAJ and PHJ algorithms is identical. However, it is higher for the PNL algorithm because each S-node must perform the join with R entirely. Prom the equations above, it is clear that the PAJ algorithm incurs the least communication cost. However, the least communication cost between the PNL and PHJ algorithms depends on the values of relation cardinality and degree of partitioning. If we now choose p so that it is smaller than n, the PHJ algorithm may well incur the least communication cost but at the expense of increased join processing cost. For example, if $p = 1$, the join is processed in a purely centralized way.

In conclusion, the PAJ algorithm is most likely to dominate and should be used when applicable. Otherwise, the choice between the PNL and PHJ algorithms requires estimation of their total cost with the optimal value for p. The choice of a parallel join algorithm can be summarized by the procedure CHOOSE_JA shown in Algorithm 13.4.

Algorithm 13.4 *CHOOSEJA*

```
input: prof(R): profile of relation R;
       prof(S): profile of relation S;
       JP: join predicate
output: JA: join algorithm
begin
  if JP is equijoin then
    if one relation is partitioned according to the join attribute then
      JA ← PAJ
    else if Cost(PNL) < Cost(PHJ) then
      JA← PNL
      else
        JA ← PH J
      end-if
    end-if
  else
    JA ← PNL
  end-if
end. {CHOOSE.JA}
```

13.3.4 Parallel Query Optimization

Parallel query optimization exhibits similarities with distributed query processing. It should take advantage of both ultra-operator parallelism (using the algorithms described above) and inter-operator parallelism. This second objective can be achieved using some of the techniques devised for distributed DBMSs.

Parallel query optimization refers to the process of producing an execution plan for a given query that minimizes an objective cost function. The selected plan is the best one within a set of candidate plans examined by the optimizer, but not necessarily the optimal one among all possible plans. A query optimizer is usually seen as three components: a search space, a cost model, and a search strategy. The *search space* is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operators and the way these operators are implemented. The *cost model* predicts the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the parallel execution environment. The *search strategy* explores the search space and selects the best plan. It defines which plans are examined and in which order.

Search Space

Execution plans are abstracted, as usual, by means of operator trees, which define the order in which the operators are executed. Operator trees are enriched with *annotations*, which indicate additional execution aspects, such as the algorithm of each operator. An important execution aspect to be reflected by annotations is the fact that two subsequent operators can be executed in pipeline. In this case, the second operator can start before the first one is completed. In other words, the second operator starts *consuming* tuples as soon as the first one produces them. Pipelined executions do not require temporary relations to be materialized, i.e., a tree node corresponding to an operator executed in pipeline is not stored.

Pipeline and store annotations constrain the scheduling of execution plans. They split an operator tree into non-overlapping sub-trees, called *phases*. Pipelined operators are executed in the same phase, whereas a storing indication establishes the boundary between one phase and a subsequent phase. Some operators and some algorithms require that one operand be stored. For example, one variation of the parallel hash join algorithm consists of two consecutive phases: build and probe. In the build phase, a hash table is constructed in parallel on the join attribute of the smallest relation. In the probe phase, the largest relation is sequentially scanned and the hash table is consulted for each of its tuples.

Example 13.5

In the left-hand part of Figure 13.18, the temporary relations *Temp1* must be completely produced and the hash table in *Build2* must be finished before *Probe2* can start consuming R_3. The same is true for *Temp2*, *Build3* and *Prpbe3*. Thus, this tree is executed in four consecutive phases: build R_1's

hash table, then probe it with R_2 and build *Temp1*'s hash table, then probe it with R_3 and build *Temp2*'s hash table, then probe it with R_3 and produce the result. In the right-hand part of Figure 13.18, the pipeline annotations are indicated by arrows. This tree can be executed in two phases if enough memory is available to build the hash tables: build the tables for R_1 R_3 and R_4, then execute *Probe1*, *Probe2* and *Probe3* in pipeline.

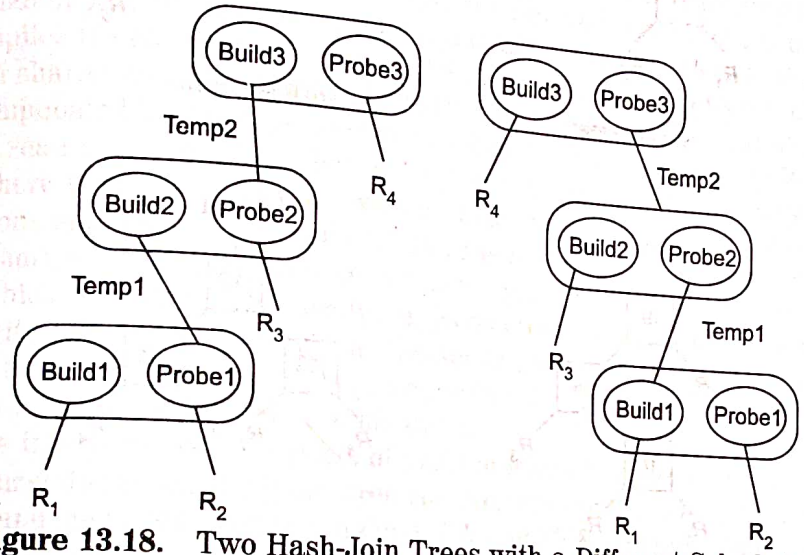

Figure 13.18. Two Hash-Join Trees with a Different Scheduling.

The set of nodes where a relation is stored is called its *home*. The *home* of *an operator* is the set of nodes where it is executed and it must be the home of its operands in order for the operator to access its operand. For binary operators such as join, this might imply repartitioning one of the operands. The optimizer might even sometimes find that repartitioning both the operands is of interest. Operator trees bear execution annotations to indicate repartitioning.

Figure 13.19 shows four operator trees, that represent execution plans for a three-way join. An operator tree is a labelled binary tree where the leaf nodes are relations of the input query and each non-leaf node is an operator node (e.g., join, union) whose result is an *intermediate* relation. A join node captures the join between its operands. Execution annotations (e.g., join algorithm) are not shown for simplicity. Directed (respectively undirected) arcs denote that the intermediate relation generated by a tree node is consumed in pipeline (respectively stored) by the subsequent node. Operator trees may be *linear*, i.e. at least one operand of each join node is a base relation or *bushy*.

It is convenient to represent pipelined relations on as right-hand side input of an operator. Thus, right-deep trees express full pipelining while left-deep trees express full materialization of intermediate results. Thus, long right-deep trees are more efficient then corresponding left-deep trees but tend to consume more memory to store left-hand side relations.

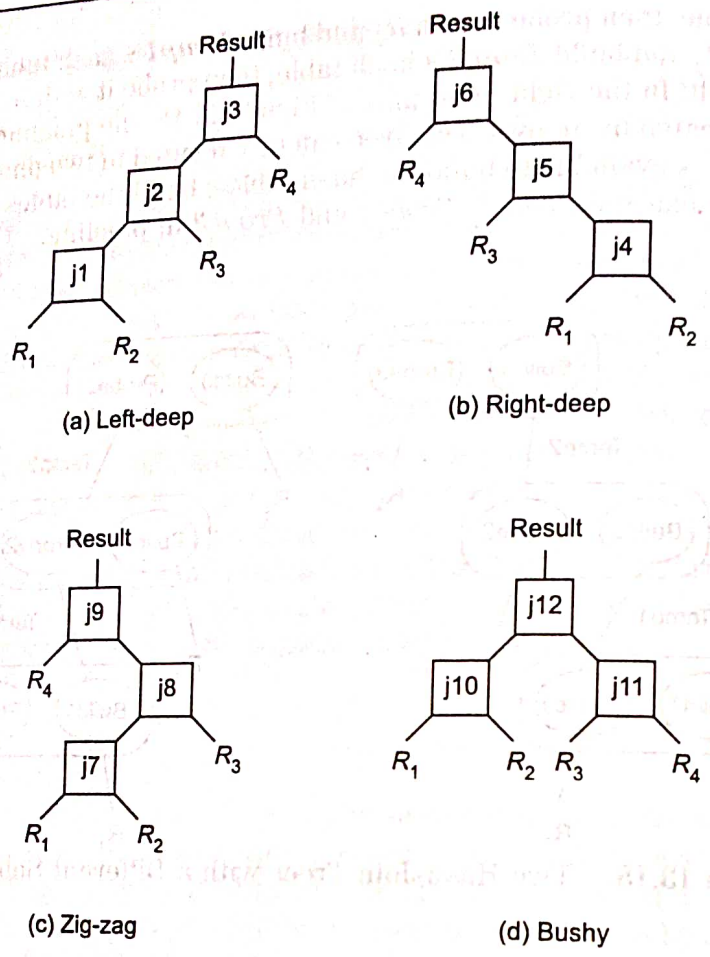

Figure 13.19. Execution Plans as Operator Trees

Parallel tree formats other than left or right-deep are also interesting. For example, bushy trees (Figure 13.19.(d)) are the only ones to allow independent parallelism. Independent parallelism is useful when the relations are partitioned on disjoint homes. Suppose that the relations in Figure 13.19 are partitioned two (R_1 and R_2) by two (R_3 and R_4) on disjoint homes (resp. h_1 and h_2). Then, joins j_{10} and j_{11} could be independently executed in parallel by the set of nodes that constitutes h_2 and h_2.

When pipeline parallelism is beneficial, *zigzag trees*, which are intermediate formats between left-deep and right-deep trees, can sometimes outperform right-deep trees due to a better use of main memory [Ziane et al., 1993]. A reasonable heuristic is to favor right-deep or zigzag trees when relations are partially fragmented on disjoint homes and intermediate relations are rather large. In this case, bushy trees will usually need more phases and take longer to execute. On the contrary, when intermediate relations are small, pipelining is not very efficient because it is difficult to balance the load between the pipeline stages.

Cost Model

The optimizer cost model is responsible for estimating the cost of a given execution plan. It may be seen as two parts: architecture-dependent and architecture-independent [Lanzelotte et al., 1994]. The architecture-independent part is constituted by the cost functions for operator algorithms, e.g., nested loop for join and sequential access for select. If we ignore concurrency issues, only the cost functions for data repartitioning and memory consumption differ and constitute the architecture-dependent part. Indeed, repartitioning a relation's tuples in a shared-nothing system implies transfers of data across the interconnect, whereas it reduces to hashing in shared-memory systems. Memory consumption in the shared-nothing case is complicated by inter-operator parallelism. In shared-memory systems, all operators read and write data through a global memory, and it is easy to test whether there is enough space to execute them in parallel, i.e., the sum of the memory consumption of individual operators is less than the available memory. In shared-nothing, each processor has its own memory, and it becomes important to know which operators are executed in parallel on the same processor. Thus, for simplicity, it can be assumed that the set of processors (home) assigned to operators to execute do not overlap, i.e., either the intersection of the set of processors is empty or the sets are identical.

To take into account the aspects of parallel execution, we define the cost of a plan [Lanzelotte et al., 1993] as three components: total work (TW), response time (RT), and memory consumption (MC). TW and RT are expressed in *seconds*, and MC in *Kbytes*.

The first two components are used to express a trade-off between response time and throughput. The third component represents the size of memory needed to execute the plan. The cost function is a combination of the first two components, and plans that need more memory than available are discarded. Another approach [Ganguly et al., 1992] consists of using a parameter, specified by the system administrator, by which the maximum throughput is degraded in order to decrease response time. Given a plan p, its cost is computed by a parameterized function defined as follows:

$$cost_{(W_{RT},W_{TW})}(p) = \begin{cases} W_{RT} * RT + W_{TW} * TW & \text{if } MC \text{ of plan } p \text{ does not exceed} \\ \infty & \text{the available memory otherwise} \end{cases}$$

where W_{RT} and W_{TW} are weight factors between 0 and 1, such that $W_{RT} + W_{TW} = 1$.

A major difficulty in evaluating the cost is in assigning values to the weight of the first two components. These factors depend on the system state (e.g., load of the system and number of queried submitted to the system), and are ideally determined at run time. This is impossible with static optimization. The total work can be computed by a formula that simply adds all CPU, I/O and communication cost components as in distributed query optimization. The response time is more involved as it must take pipelining into account.

The response time of p, scheduled in phases (each denoted by ph), is computed as follows [Lanzelotte et al., 1994]:

$$RT(p) = \sum_{ph \in p} (max_{OP \in ph}(_{resp}Time(Op) + pipe\_delay(OP)) + store\_delay(ph))$$

where Op denotes an operator and respTime(Op) the response time of Op. $pipe\_delay(Op)$ is the waiting period of Op, necessary for the producer to deliver the first result tuples. It is equal to 0 if the input relations of O are stored. $store\_delay(ph)$ is the time necessary to store the output results of phase ph. It is equal to 0 if ph is the last phase, assuming that the result are delivered as soon as they are produced.

To estimate the cost of an execution plan, the cost model uses database statistics and organization information, such as relation cardinalities and partitioning, as with distributed query optimization.

Search Strategy

The search strategy does not need to be different from either centralized or distributed query optimization. However, the search space tends to be much larger because there are more alternative parallel execution plans. Thus, randomized search strategies (see Section 9.1.2) generally outperform deterministic strategies in parallel query optimization.

13.4 PARALLEL EXECUTION PROBLEMS

Parallel query response time can be hurt by several barriers [DeWitt et al., 1992]. This section shows the principal problems introduced by parallel query execution.

13.4.1 Initialization

Before the execution takes place, an initialization step is necessary. This first step is generally sequential. It includes process (or thread) creation and initialization, communication initialization, etc. The duration of this step is proportional to the degree of parallelism and can actually dominate the execution time of low complexity queries. Thus, the degree of parallelism should be fixed according to the query complexity.

In [Wilshut et al., 1992], a formula is given to estimate the maximal speedup reachable during the execution of an operator and to deduce the optimal number of processors. Let us consider the execution of an operator which processes N tuples with n processors. Let c be the average processing time of each tuple and a the initialization time per processor. In the ideal case, the response time of the operator execution is

$$Response\ Time = an + \frac{cN}{n}$$

By derivation, we can obtain the optimal number of processor no to allocate and the maximal speedup S_0 reachable.

$$n_0 = \sqrt{\frac{cN}{a}} \qquad S_0 = \frac{n_0}{2}$$

In a main memory database system (i.e., no disk accesses during execution) which is the framework of [Wilshut et al., 1992], the initialization time on is important thus limiting the optimal number of processors. In disk-based database systems, relations are read from the disks. Initialization time also limits the optimal number of processors. These equations show that we may not use only intra-operator parallelism (i.e., maximal parallelism degree for each operator) but combine intra and inter-parallelism in order to achieve a better speed-up on each operator.

13.4.2 Interferences and Convoy Effect

A highly parallel execution can be slowed down by interference. Interference appears when several processors simultaneously access the same resource, hardware or software.

A typical example of hardware interference is the contention created on the bus of a shared-memory system. When the number of processors is increased, the number of conflicts on the bus increases, thus limiting the extensibility of shared-memory systems. A solution to these interferences is to duplicate shared resources. For instance, disk access interference can be eliminated by adding several disks and partitioning the relations.

Software interference occurs when several processors want to access shared data. To prevent incoherency, mutual exclusion variables are used to protect shared data, thus blocking all but one processor which accesses the shared data. However, shared variables may well become the bottleneck of query execution, creating hot spots and convoy effects [Blasgen et al, 1979]. A typical example of software interference is the access of database internal structures such as indexes. For simplicity, the earlier versions of database systems were protected by a unique mutual exclusion variable. Studies have shown the overhead of such strategy: 45% of the query execution time was consumed by interference between 16 processors.

A general solution to software interference is to partition the data structure. Each global structure may be decomposed in several independent structures protected by a mutual exclusion variable.

13.4.3 Load Balancing

The response time of a set of parallel operators is that of the longest one. Load balancing problems can appear with intra-operator parallelism (variation in partition size), namely *data skew*, and inter-operator parallelism (variation in the complexity of operators). Load balancing is a crucial issue for parallel execution and has been extensively studied. In the rest of this section, we present techniques for intra and inter-operator load balancing.

Intra-Operator Load Balancing

Walton et al. [Walton et al., 1991] classifies the effects of skewed data distribution on a parallel execution. *Attribute value skew* (*AVS*) is skew inherent in the dataset (e.g., there are more citizens in Paris than in Rocquencourt) while *tuple placement skew* (*TPS*) is the skew induced when the data is initially partitioned (on disk) (e.g., with range partitioning). *Selectivity skew* (*SS*) is induced when there is variation in the selectivity of select predicates on each node. *Redistribution skew* (*RS*) occurs in the redistribution step between two operators. It is similar to TPS. Finally *join product skew* (*JPS*) occurs because the join selectivity may vary between nodes. Figure 13.20 illustrates this classification on a query applied to two relations R and S which are poorly partitioned. The boxes are proportional to the size of the corresponding partitions. Such poor partitioning stems from either the data (AVS) or the partitioning function (TPS). Thus, the processing times of the two instances scan1 and scan2 are not equal. The case of the join operator is worse. First, the number of tuples received is different from one instance to another because of poor redistribution of the partitions of R (RS) or variable selectivity according to the partition of R processed (SS). Finally, the uneven size of S partitions (AVS/TPS) yields different processing times for tuples sent by the scan operator and the result size is different from one partition to the other due to join selectivity (JPS).

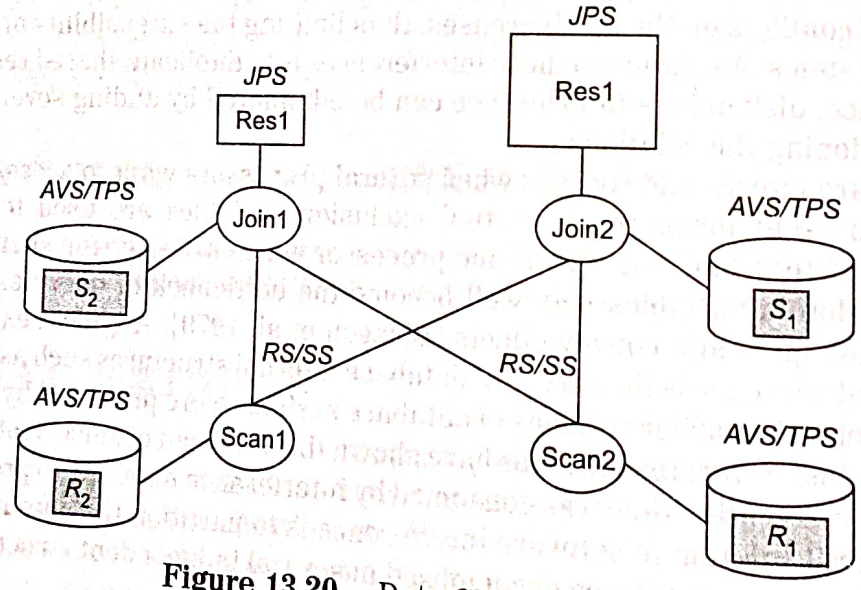

Figure 13.20. Data Skew Example

Clearly, it seems difficult to propose a solution based on estimates on the relations involved. Such strategy would require statistics like histogram on join attribute, fragmentation attribute, attributes involved in predicates—potentially on all attributes of all relations. Furthermore, a cost model would be necessary to evaluate distribution on intermediate results. A more reasonable strategy is to use a dynamic approach, i.e., redistribute the load dynamically in order to balance the execution.

Several solutions have been proposed to reduce the negative impact from skew. [Kitsuregawa and Ogawa, 1990] presents a robust hash-join algorithm for a specific

parallel architecture based on shared-nothing. The idea is to partition each hash bucket in fragments and spread them among the processors (bucket spreading). Then a sophisticated network, the Omega network, is used to redistribute buckets onto the processors. The Omega network contains logic to balance the load during redistribution. [Omiecinski, 1991] proposes a similar approach in a shared-memory parallel system, using the first fit decreasing heuristic to assign buckets to processors. DBS3 [Bergsten et al., 1991], [Dageville et al. 1994] has pioneered the use of an execution model based on relation partitioning (as in shared-nothing) for shared-memory. This model reduces processor interference and shows excellent load balancing for intra-operator parallelism [Bouganim et al., 1996a], [Bouganim et al. 1996b]. [DeWitt et al., 1992] suggests the use of multiple algorithms, each specialized for a different degree of skew, and the use of a small sample of the relations to determine which algorithm is appropriate.

Wolf et al. [Wolf et al., 1993] propose to modify hash join and sort merge join algorithms to insert a scheduling step which is in charge of redistributing the load. Each join is decomposed in a number of join superior to the number of processors and distributed among the processors using estimates on complexity and heuristics like LPT. This strategy is iterated until a "good" load balancing is achieved.

Shatdal et al. [Shatdal and Naughton, 1993] propose to use shared virtual memory (implemented by software) to redistribute dynamically the load. When a processor is idle, it steals work from a random chosen processor using shared virtual memory.

Inter-Operator Load Balancing

In order to obtain a good load balancing at the inter-operator level, it is necessary to choose, for each operator, how many and what processors to assign for its execution. Suppose we have a perfect cost model which allows evaluating the sequential execution time of each operator. We then need to find a way to assign processors to operators in order to obtain the best load balancing.

Metha et al. [Mehta and DeWitt 1995] propose to determine dynamically (just before the execution) the degree of parallelism and the localization of the processors for each operator. The *Rate Match* algorithm uses a cost model in order to match the rate at which tuples are produced and consumed. Six algorithms are then proposed in order to choose the set of processors which will be used for query execution (based on available memory, CPU or disks utilization, etc...)

[Rahm and Marek, 1995] and [Garofalakis and Ioanidis, 1996] propose other algorithms for the choice of the number and localization of processors. They try to distribute efficiently the load in order to maximize the use of several resources using statistics on this usages.

The potential reasons for poor load balancing in shared-nothing are studied in [Wilshut et al., 1995]. First, the degree of parallelism and the allocation of processors to operators, decided in the parallel optimization phase, are based on a possibly inaccurate cost model. Second, the choice of the degree of parallelism is subject to discretization errors because both processors and operators are discrete entities. Finally, the processors associated with the latest operators in a

pipeline chain may remain idle a significant time. This is called the pipeline delay problem. These problems stem from the fixed association between data operators and processors, In the next section, we present an execution model which do not realize any association between processors and operators.

13.5 PARALLEL EXECUTION FOR HIERARCHICAL ARCHITECTURE

In this section we describe a complete solution for query execution on hierarchical architectures, based on [Bouganim et al., 1996c]. In the context of hierarchical systems, load balancing is exacerbated because it must be addressed at two levels, locally among the processors of each shared-memory node and globally-among all nodes. None of the approaches presented in Section 13.4.3, can be easily extended to deal with this problem. Load balancing strategies for shared-nothing would experience even more severe problems worsening (e.g., complexity and inaccuracy of the cost model). On the other hand, adapting solutions developed for shared-memory systems would incur high communication overhead.

In this section, we present an execution model, called *Dynamic Processing* (*DP*) for hierarchical systems which dynamically performs intra- and inter-operator load balancing. The fundamental idea is that the query work is decomposed in self-contained units of sequential processing, each of which can be carried out by any processor. Intuitively, a processor can migrate horizontally and vertically along the query work. The main advantage is to minimize the communication overhead of inter-node load balancing by maximizing intra and inter-operator load balancing within shared-memory nodes.

13.5.1 Problem Formulation

The result of parallel query optimization is a *parallel execution plan* that consists of an operator tree with operator scheduling and allocation of computing resources to operators Figure 13.21 shows a join tree and the corresponding operator tree.

The operator scheduling constraints are:

```
Build1 < Probe1
Build2 > Probe2
Build3 > Probe3
```

The operator scheduling heuristics are:

```
Heuristic1: Build1 < Scan2, Build2 < Scan4, Build3 < Scan3
Heuristic1: Build3 < Scan3
```

Finally, the operator homes are:

```
home (Scan1) = Node,A
home (Build1, Probe1, Scan2, Scan3) - Node B
home (Scan4) = Node C
home (Build3, Build2, Probe2, Probe3) = Nodes B and C
```

Given this information, the problem is to produce an execution on a hierarchical architecture which minimizes response time. A necessary condition to minimize response time is to avoid processor idle time.' This can be done by using a dynamic load balancing mechanism at two levels: (i) within a SM-node, load balancing is achieved via fast interprocess communication; (ii) between SM-nodes, more expensive message-passing communication is needed. Thus, the problem is to come up with an execution model so that the use of local load balancing is maximized while the use of global load balancing is minimized. Intuitively, parallelizing a query amounts to partition the total work along two dimensions. First, each operator is horizontally partitioned to yield intra-operator parallelism. Second, the query is vertically partitioned into dependent or independent operators to yield inter-operator parallelism. We call *activation* the smallest unit of sequential processing that cannot be further partitioned. The main property of the DP model is to allow any processor to process any activation of its SM-node. Thus, there is no static association between threads and operators. This should yield good load-balancing for both intra-operator and inter-operator parallelism within a SM-node, and thus, reduce to the minimum the need for global load balancing, i.e., when there is no more work to do in a SM-node. In the rest of this section, we present the basic concepts underlying the DP model and its load balancing strategy which we illustrate with an example.

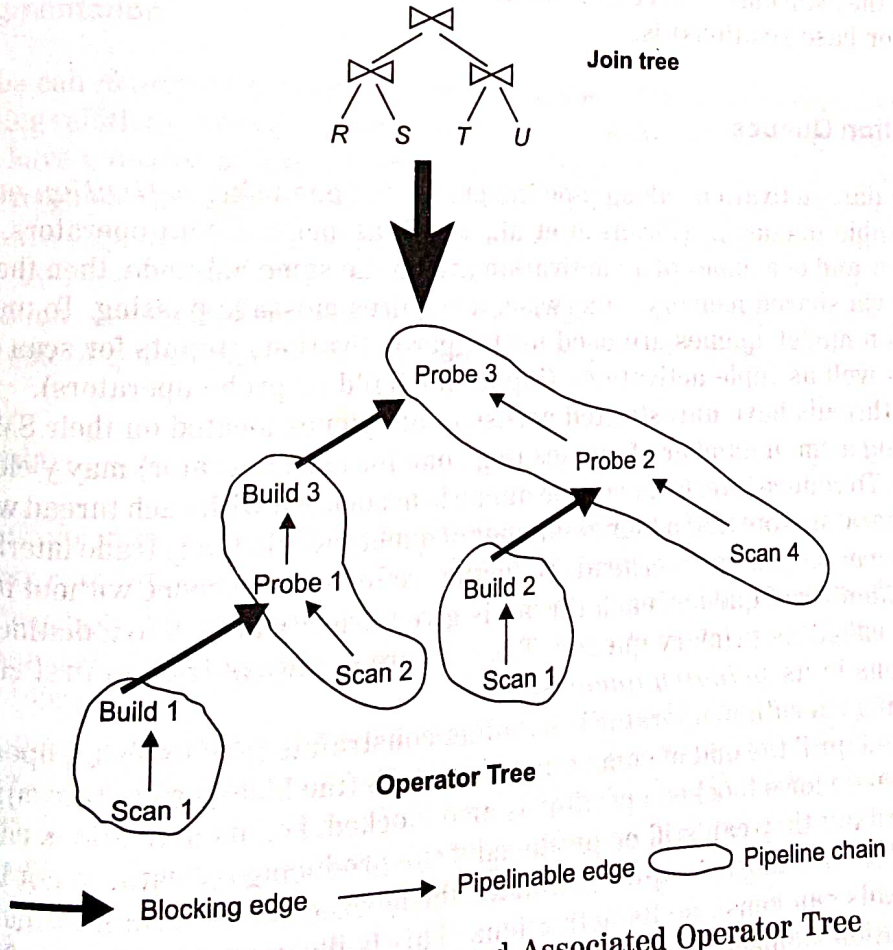

Figure 13.21. A Join Tree and Associated Operator Tree

13.5.2 Basic Concepts

The DP execution model is based on a few concepts: activations, activation queues, fragmentation, and threads. It assumes a modern multithread system. These concepts are simple and their combination provides much flexibility and generality.

Activations

An activation represents a sequential unit of work. Since any activation can be executed by any thread (by any processor), activations must be self-contained and reference all information necessary for their execution: the code to execute and the data to process. Two kinds of activations can be distinguished: trigger activations and data activations. A *trigger activation* is used to start the execution of a leaf operator, i.e., scan. It is represented by an (*Operator*, *Bucket*) pair which references the scan operator and the base relation bucket to scan. A *data activation* describes a tuple produced in pipeline mode. It is represented by an (*Operator*, *Tuple*, *Bucket*) triple which references the operator to process. For a build operator (see Section 13.3.3), the data activation specifies that the tuple must be inserted in the hash table of the bucket and for a probe operator, that the tuple must be probed with the bucket's hash table. Although activations are self-contained, they can only be executed on the SM-node where the associated data (hash tables or base relations) is.

Activation Queues

Moving data activations along pipeline chains is done using *activation queues*, called table queues in [Pirahesh et al., 1990], associated with operators. If the producer and consumer of an activation are on the same SM-node, then the move is done via shared-memory. Otherwise, it requires message-passing. To unify the execution model, queues are used for trigger activations (inputs for scan operators) as well as tuple activations (inputs for build or probe operators).

All threads have unrestricted access to all queues located on their SM-node. Managing a small number of queues (e.g., one for each operator) may yield interference. To reduce interference, one queue is associated with each thread working on an operator. Note that a higher number of queues would likely trade interference for queue management overhead. To further reduce interference without increasing the number of queues, each thread is given priority access to a distinct set of queues, called its primary queues. Thus, a thread always tries to first consume activations in its *primary queues*.

During execution, operator scheduling constraints may imply an operator to be blocked until the end of some other operators (the blocking operators). Therefore, a queue for a blocked operator is also blocked, i.e., its activations cannot be consumed but they can still be produced if the producing operator is not blocked. When all its blocking operators terminate, the blocked queue becomes consumable, i.e., threads can consume its activations. This is illustrated in Figure 13.22 with an execution snapshot for the operator tree of Figure 13.21.

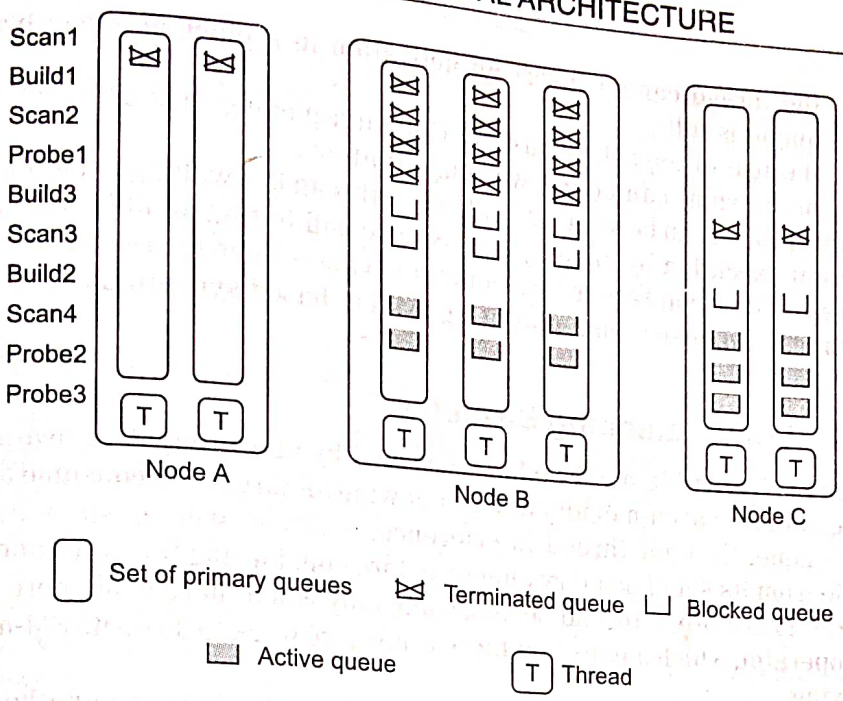

Figure 13.22. Snapshot of an execution

Fragmentation

Let us call *degree of fragmentation* the number of buckets of the building and probing relations. To reduce the negative effects of data skew, the typical solution is to have a degree of fragmentation much higher than the degree of parallelism [Kitsuregawa and Ogawa, 1990], [DeWitt et al., 1992]. If a queue is used for each bucket, a high degree of fragmentation would imply an important queue management overhead [Bouganim et al. 1996b]. Since activations are self-contained, activations of different buckets in the same queue can be mixed, thus reducing the overhead of queue management with a high degree of fragmentation.

Threads

A simple strategy for obtaining good load balancing inside a SM-node is to allocate a number of threads much higher than the number of processors and let the operating system do thread scheduling. However, this strategy incurs high numbers of system calls due to thread scheduling, interference and convoy problems [Pirahesh et al., 1990], [Hong, 1992].

Instead of relying on the operating system for load balancing, it is possible to allocate only one thread per processor per query. This is made possible by the fact that any thread can execute any operator assigned to its SM-node. The advantage of this one-thread-per-processor allocation strategy is to significantly reduce the overhead of interference and synchronization provided that a thread is never blocked. Waiting for some event would cause processor idle time. During the processing of an activation, a thread can be blocked in the following situations:

- the thread cannot insert an activation in a pipeline queue because the queue is full[1];
- the use of asynchronous I/O (for multiplexing disk accesses with data processing) can create waiting situations.

This problem can be solved as follows. A thread in a waiting situation suspends its current execution by making a procedure call to find another local activation to process. The advantage is that context saving is done by procedure call, which is much less expensive than operating system based synchronization.

13.5.3 Load Balancing Strategy

Load balancing within a SM-node is obtained by allocating all activation queues in a segment of shared-memory and by allowing all threads to consume activations in any queue. To limit thread interference, a thread will consume as much as possible from its set of primary queues before considering the other queues of the SM-node. Therefore, a thread becomes idle only when there is no more activation of any operator, which means that there is no more work to do on its SM-node which is starving.

When an SM-node starves, we can apply load sharing with another SM-node by acquiring some of its workload [Shatdal and Naughton, 1993]. However, acquiring activations (through message-passing) incurs communication overhead. Furthermore, activation acquisition is not enough since associated data, i.e., hash tables, must also be acquired. Thus, we need a mechanism that can dynamically estimate the benefit of acquiring activations and data.

Let us call "requester" the SM-node which acquires work and "provider" the SM-node which gets off-loaded by providing work to the requester. The problem is to select a queue to acquire activations and decide how much work to acquire. This is a dynamic optimization problem since there is a trade-off between the potential gain of off-loading the provider and the overhead of acquiring activations and data. This trade-off can be expressed by the following conditions: (i) the requester must be able to store in memory the activations and corresponding data; (ii) enough work must be acquired in order to amortize the overhead of acquisition; (iii) acquiring too much work should be avoided; (iv) only probe activations can be acquired since triggered activations require disk accesses and build activations require building hash tables locally; (v) there is no gain to move activations associated with blocked operators which could not be processed anyway. Finally, to respect the decisions of the optimizer, a SM-node cannot execute activations of an operator that it does not own, i.e., the SM-node is not in the operator home.

The amount of load balancing depends on the number of operators that are concurrently executed which provides opportunities for finding some work to share

[1] Without any restriction on the queue size, memory consumption may well increase. For instance, consider the concurrent execution of two pipelined operators. If the select strategy favors the producer operator, then it may well end up materializing the entire intermediate result. To avoid this situation, queues have a limited size.

in case of idle times. Increasing the number of concurrent operators can be done by allowing concurrent execution of several pipeline chains or by using non-blocking hash-join algorithms which allow to execute all the operators of the bushy tree concurrently [Wilshut et al., 1995]. On the other hand, executing more operators concurrently can increase memory consumption. Static operator scheduling as provided by the optimizer should avoid memory overflow and solve this tradeoff.

13.5.4 Performance Evaluation

The performance of the DP model is evaluated in [Bouganim et al., 1996c] using an implementation on a 72-processor KSR1 computer. KSR1's shared virtual memory architecture and high number of processors have been organized as a hierarchical parallel system. To experiment with many different queries, large relations and different relation parameters (cardinality, selectivity, skew factor, etc.), the execution of atomic operators has been simulated. Various experiments have been performed at two levels: locally within an SM-node and globally among SM-nodes.

In the shared-memory case, the DP load balancing strategy is compared with synchronous pipelining (SP) and fixed processing (FP). The results are as follows. SP is best for shared-memory but does not work in shared-nothing whereas FP is designed for shared-nothing and also works in shared-memory. FP is always worse because of discretization errors which worsen as the number of processors decreases. The performance of DP is very close to that of SP from 8 to 32 processors and remain close for higher numbers. Both SP and DP strategies show very good speedup, even with highly skewed data.

REVIEW QUESTIONS

13.1 Explain database server approach and distributed database servers through diagrams.

13.2 Explain general architecture of a parallel database system and shared memory architecture.

13.3 Explain through diagrams the following: shared-disk architecture, shared-nothing architecture, hierarchical architecture and cache-only memory architecture.

13.4 Elucidate the following with the help of suitable diagrams: data placement, different partitioning schemes and query parallelism.

13.5 Explain parallel data processing through algorithms.

13.6 What do you mean by query optimization?

13.7 What are parallel execution problems? Explain in detail.

13.8 In the case of hierarchical architecture, explain the following: problem formation, basic concepts, load balancing strategy, and performance evaluation.