sites have a significant impact on all aspects of the system, including query optimization and processing, concurrency control, and recovery. In contrast to parallel databases, the distribution of data is governed by factors such as local ownership and increased availability, in addition to performance issues.

In this chapter we look at the issues of parallelism and data distribution in a DBMS. In Section 21.1 we discuss alternative hardware configurations for a parallel DBMS. In Section 21.2 we introduce the concept of data partitioning and consider its influence on parallel query evaluation. In Section 21.3 we show how data partitioning can be used to parallelize several relational operations. In Section 21.4 we conclude our treatment of parallel query processing with a discussion of parallel query optimization. The rest of the chapter is devoted to distributed databases. We present an overview of distributed databases in Section 21.5. We discuss some alternative architectures for a distributed DBMS in Section 21.6 and describe options for distributing data in Section 21.7. In Section 21.9 we discuss query optimization and evaluation for distributed databases, in Section 21.10 we discuss updating distributed data, and finally, in Sections 21.12 and 21.13 we discuss distributed transaction management.

## 21.1 ARCHITECTURES FOR PARALLEL DATABASES

The basic idea behind parallel databases is to carry out evaluation steps in parallel whenever possible, in order to improve performance. There are many opportunities for parallelism in a DBMS; databases represent one of the most successful instances of parallel computing.

Three main architectures have been proposed for building parallel DBMSs. In a **shared-memory** system, multiple CPUs are attached to an interconnection network and can access a common region of main memory. In a **shared-disk** system, each CPU has a private memory and direct access to all disks through an interconnection network. In a **shared-nothing** system, each CPU has local main memory and disk space, but no two CPUs can access the same storage area; all communication between CPUs is through a network connection. The three architectures are illustrated in Figure 21.1.

The shared memory architecture is closer to a conventional machine, and many commercial database systems have been ported to shared memory platforms with relative ease. Communication overheads are low, because main memory can be used for this purpose, and operating system services can be leveraged to utilize the additional CPUs. Although this approach is attractive for achieving moderate parallelism—a few tens of CPUs can be exploited in this fashion—memory contention becomes a bottleneck as the number of CPUs increases. The shared-disk architecture faces a similar problem because large amounts of data are shipped through the interconnection network.

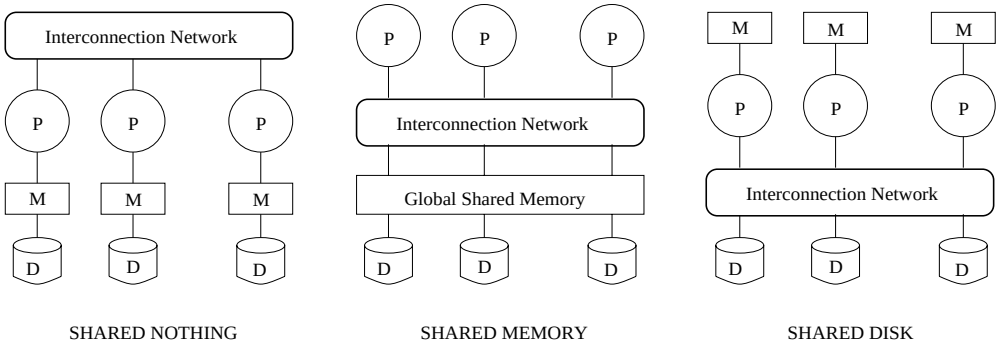


**Figure 21.1** Physical Architectures for Parallel Database Systems

The basic problem with the shared-memory and shared-disk architectures is **interference**: As more CPUs are added, existing CPUs are slowed down because of the increased contention for memory accesses and network bandwidth. It has been noted that even an average 1 percent slowdown per additional CPU means that the maximum speedup is a factor of 37, and adding additional CPUs actually slows down the system; a system with 1,000 CPUs is only 4 percent as effective as a *single CPU* system! This observation has motivated the development of the shared-nothing architecture, which is now widely considered to be the best architecture for large parallel database systems.

The shared-nothing architecture requires more extensive reorganization of the DBMS code, but it has been shown to provide linear **speed-up**, in that the time taken for operations decreases in proportion to the increase in the number of CPUs and disks, and linear **scale-up**, in that performance is sustained if the number of CPUs and disks are increased in proportion to the amount of data. Consequently, ever-more powerful parallel database systems can be built by taking advantage of rapidly improving performance for single CPU systems and connecting as many CPUs as desired.

Speed-up and scale-up are illustrated in Figure 21.2. The speed-up curves show how, for a fixed database size, more transactions can be executed per second by adding CPUs. The scale-up curves show how adding more resources (in the form of CPUs) enables us to process larger problems. The first scale-up graph measures the number of transactions executed per second as the database size is increased and the number of CPUs is correspondingly increased. An alternative way to measure scale-up is to consider the time taken per transaction as more CPUs are added to process an increasing number of transactions per second; the goal here is to sustain the response time per transaction.

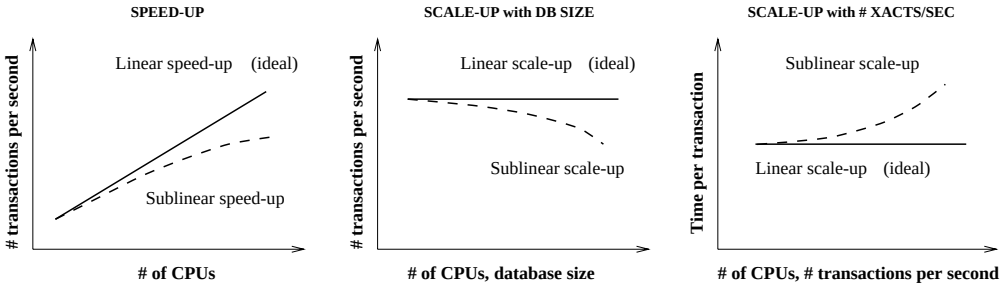


**Figure 21.2** Speed-up and Scale-up

## 21.2 PARALLEL QUERY EVALUATION

In this section we discuss parallel evaluation of a relational query in a DBMS with a shared-nothing architecture. While it is possible to consider parallel execution of multiple queries, it is hard to identify in advance which queries will run concurrently. So the emphasis has been on parallel execution of a single query.

A relational query execution plan is a graph of relational algebra operators and the operators in a graph can be executed in parallel. If an operator consumes the output of a second operator, we have **pipelined parallelism** (the output of the second operator is worked on by the first operator as soon as it is generated); if not, the two operators can proceed essentially independently. An operator is said to **block** if it produces no output until it has consumed all its inputs. Pipelined parallelism is limited by the presence of operators (e.g., sorting or aggregation) that block.

In addition to evaluating different operators in parallel, we can evaluate each individual operator in a query plan in a parallel fashion. The key to evaluating an operator in parallel is to *partition* the input data; we can then work on each partition in parallel and combine the results. This approach is called **data-partitioned parallel evaluation**. By exercising some care, existing code for sequentially evaluating relational operators can be ported easily for data-partitioned parallel evaluation.

An important observation, which explains why shared-nothing parallel database systems have been very successful, is that database query evaluation is very amenable to data-partitioned parallel evaluation. The goal is to minimize data shipping by partitioning the data and by structuring the algorithms to do most of the processing at individual processors. (We use *processor* to refer to a CPU together with its local disk.)

We now consider data partitioning and parallelization of existing operator evaluation code in more detail.

### 21.2.1 Data Partitioning

Partitioning a large dataset horizontally across several disks enables us to exploit the I/O bandwidth of the disks by reading and writing them in parallel. There are several ways to horizontally partition a relation. We can assign tuples to processors in a round-robin fashion, we can use hashing, or we can assign tuples to processors by ranges of field values. If there are $n$ processors, the $i$th tuple is assigned to processor $i \ mod \ n$ in **round-robin partitioning**. Recall that round-robin partitioning is used in RAID storage systems (see Section 7.2). In **hash partitioning**, a hash function is applied to (selected fields of) a tuple to determine its processor. In **range partitioning**, tuples are sorted (conceptually), and $n$ ranges are chosen for the sort key values so that each range contains roughly the same number of tuples; tuples in range $i$ are assigned to processor $i$.

Round-robin partitioning is suitable for efficiently evaluating queries that access the entire relation. If only a subset of the tuples (e.g., those that satisfy the selection condition $age = 20$) is required, hash partitioning and range partitioning are better than round-robin partitioning because they enable us to access only those disks that contain matching tuples. (Of course, this statement assumes that the tuples are partitioned on the attributes in the selection condition; if $age = 20$ is specified, the tuples must be partitioned on $age$.) If range selections such as $15 < age < 25$ are specified, range partitioning is superior to hash partitioning because qualifying tuples are likely to be clustered together on a few processors. On the other hand, range partitioning can lead to **data skew**; that is, partitions with widely varying numbers of tuples across partitions or disks. Skew causes processors dealing with large partitions to become performance bottlenecks. Hash partitioning has the additional virtue that it keeps data evenly distributed even if the data grows and shrinks over time.

To reduce skew in range partitioning, the main question is how to choose the ranges by which tuples are distributed. One effective approach is to take samples from each processor, collect and sort all samples, and divide the sorted set of samples into equally sized subsets. If tuples are to be partitioned on $age$, the $age$ ranges of the sampled subsets of tuples can be used as the basis for redistributing the entire relation.

### 21.2.2 Parallelizing Sequential Operator Evaluation Code

An elegant software architecture for parallel DBMSs enables us to readily parallelize existing code for sequentially evaluating a relational operator. The basic idea is to use parallel data streams. Streams (from different disks or the output of other operators) are **merged** as needed to provide the inputs for a relational operator, and the output of an operator is **split** as needed to parallelize subsequent processing.

A parallel evaluation plan consists of a dataflow network of relational, merge, and split operators. The merge and split operators should be able to buffer some data and should be able to halt the operators producing their input data. They can then regulate the speed of the execution according to the execution speed of the operator that consumes their output.

As we will see, obtaining good parallel versions of algorithms for sequential operator evaluation requires careful consideration; there is no magic formula for taking sequential code and producing a parallel version. Good use of split and merge in a dataflow software architecture, however, can greatly reduce the effort of implementing parallel query evaluation algorithms, as we illustrate in Section 21.3.3.

## 21.3 PARALLELIZING INDIVIDUAL OPERATIONS

This section shows how various operations can be implemented in parallel in a shared-nothing architecture. We assume that each relation is horizontally partitioned across several disks, although this partitioning may or may not be appropriate for a given query. The evaluation of a query must take the initial partitioning criteria into account and repartition if necessary.

### 21.3.1 Bulk Loading and Scanning

We begin with two simple operations: *scanning* a relation and *loading* a relation. Pages can be read in parallel while scanning a relation, and the retrieved tuples can then be merged, if the relation is partitioned across several disks. More generally, the idea also applies when retrieving all tuples that meet a selection condition. If hashing or range partitioning is used, selection queries can be answered by going to just those processors that contain relevant tuples.

A similar observation holds for bulk loading. Further, if a relation has associated indexes, any sorting of data entries required for building the indexes during bulk loading can also be done in parallel (see below).

### 21.3.2 Sorting

A simple idea is to let each CPU sort the part of the relation that is on its local disk and to then merge these sorted sets of tuples. The degree of parallelism is likely to be limited by the merging phase.

A better idea is to first redistribute all tuples in the relation using range partitioning. For example, if we want to sort a collection of employee tuples by salary, salary values range from 10 to 210, and we have 20 processors, we could send all tuples with salary

values in the range 10 to 20 to the first processor, all in the range 21 to 30 to the second processor, and so on. (Prior to the redistribution, while tuples are distributed across the processors, we cannot assume that they are distributed according to salary ranges.)

Each processor then sorts the tuples assigned to it, using some sequential sorting algorithm. For example, a processor can collect tuples until its memory is full, then sort these tuples and write out a run, until all incoming tuples have been written to such sorted runs on the local disk. These runs can then be merged to create the sorted version of the set of tuples assigned to this processor. The entire sorted relation can be retrieved by visiting the processors in an order corresponding to the ranges assigned to them and simply scanning the tuples.

The basic challenge in parallel sorting is to do the range partitioning so that each processor receives roughly the same number of tuples; otherwise, a processor that receives a disproportionately large number of tuples to sort becomes a bottleneck and limits the scalability of the parallel sort. One good approach to range partitioning is to obtain a sample of the entire relation by taking samples at each processor that initially contains part of the relation. The (relatively small) sample is sorted and used to identify ranges with equal numbers of tuples. This set of range values, called a **splitting vector**, is then distributed to all processors and used to range partition the entire relation.

A particularly important application of parallel sorting is sorting the data entries in tree-structured indexes. Sorting data entries can significantly speed up the process of bulk-loading an index.

### 21.3.3 Joins

In this section we consider how the join operation can be parallelized. We present the basic idea behind the parallelization and also illustrate the use of the merge and split operators described in Section 21.2.2. We focus on parallel hash join, which is widely used, and briefly outline how sort-merge join can be similarly parallelized. Other join algorithms can be parallelized as well, although not as effectively as these two algorithms.

Suppose that we want to join two relations, say, $A$ and $B$, on the *age* attribute. We assume that they are initially distributed across several disks in some way that is not useful for the join operation, that is, the initial partitioning is not based on the join attribute. The basic idea for joining $A$ and $B$ in parallel is to decompose the join into a collection of $k$ smaller joins. We can decompose the join by partitioning both $A$ and $B$ into a collection of $k$ logical buckets or partitions. By using the same partitioning function for both $A$ and $B$, we ensure that the union of the $k$ smaller joins computes

the join of $A$ and $B$; this idea is similar to intuition behind the partitioning phase of a sequential hash join, described in Section 12.5.3. Because $A$ and $B$ are initially distributed across several processors, the partitioning step can itself be done in parallel at these processors. At each processor, all local tuples are retrieved and hashed into one of $k$ partitions, with the same hash function used at all sites, of course.

Alternatively, we can partition $A$ and $B$ by dividing the range of the join attribute *age* into $k$ disjoint subranges and placing $A$ and $B$ tuples into partitions according to the subrange to which their *age* values belong. For example, suppose that we have 10 processors, the join attribute is *age*, with values from 0 to 100. Assuming uniform distribution, $A$ and $B$ tuples with $0 \leq age < 10$ go to processor 1, $10 \leq age < 20$ go to processor 2, and so on. This approach is likely to be more susceptible than hash partitioning to data skew (i.e., the number of tuples to be joined can vary widely across partitions), unless the subranges are carefully determined; we will not discuss how good subrange boundaries can be identified.

Having decided on a partitioning strategy, we can assign each partition to a processor and carry out a local join, using any join algorithm we want, at each processor. In this case the number of partitions $k$ is chosen to be equal to the number of processors $n$ that are available for carrying out the join, and during partitioning, each processor sends tuples in the $i$th partition to processor $i$. After partitioning, each processor joins the $A$ and $B$ tuples assigned to it. Each join process executes sequential join code, and receives input $A$ and $B$ tuples from several processors; a merge operator merges all incoming $A$ tuples, and another merge operator merges all incoming $B$ tuples. Depending on how we want to distribute the result of the join of $A$ and $B$, the output of the join process may be split into several data streams. The network of operators for parallel join is shown in Figure 21.3. To simplify the figure, we assume that the processors doing the join are distinct from the processors that initially contain tuples of $A$ and $B$ and show only four processors.

If range partitioning is used, the algorithm outlined above leads to a parallel version of a sort-merge join, with the advantage that the output is available in sorted order. If hash partitioning is used, we obtain a parallel version of a hash join.

### Improved Parallel Hash Join

A hash-based refinement of the approach offers improved performance. The main observation is that if $A$ and $B$ are very large, and the number of partitions $k$ is chosen to be equal to the number of processors $n$, the size of each partition may still be large, leading to a high cost for each local join at the $n$ processors.

An alternative is to execute the smaller joins $A_i \bowtie B_i$, for $i = 1 \ldots k$, one after the other, but with each join executed in parallel using all processors. This approach allows

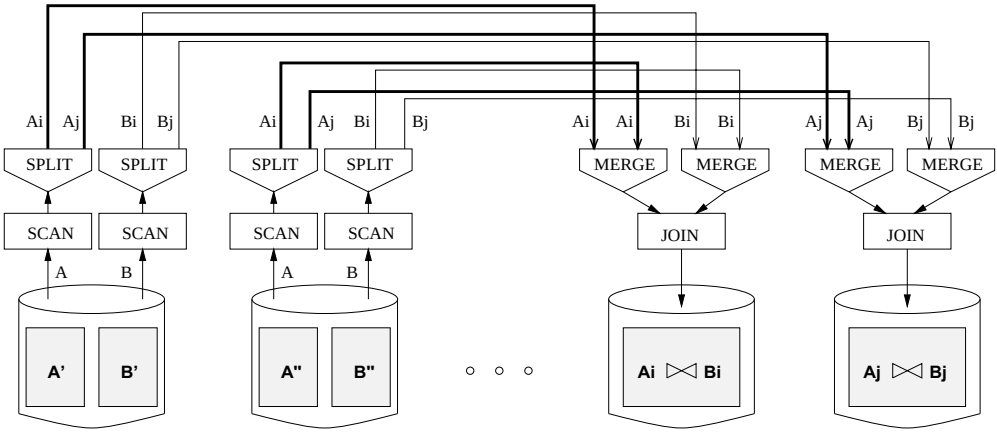


**Figure 21.3** Dataflow Network of Operators for Parallel Join

us to utilize the total available main memory at all $n$ processors in each join $A_i \bowtie B_i$ and is described in more detail as follows:

1. At each site, apply a hash function $h1$ to partition the $A$ and $B$ tuples at this site into partitions $i = 1 \ldots k$. Let $A$ be the smaller relation. The number of partitions $k$ is chosen such that each partition of $A$ fits into the *aggregate* or combined memory of all $n$ processors.

2. For $i = 1 \ldots k$, process the join of the $i$th partitions of $A$ and $B$. To compute $A_i \bowtie B_i$, do the following at every site:
   (a) Apply a second hash function $h2$ to all $A_i$ tuples to determine where they should be joined and send tuple $t$ to site $h2(t)$.
   (b) As $A_i$ tuples arrive to be joined, add them to an in-memory hash table.
   (c) After all $A_i$ tuples have been distributed, apply $h2$ to $B_i$ tuples to determine where they should be joined and send tuple $t$ to site $h2(t)$.
   (d) As $B_i$ tuples arrive to be joined, probe the in-memory table of $A_i$ tuples and output result tuples.

The use of the second hash function $h2$ ensures that tuples are (more or less) uniformly distributed across all $n$ processors participating in the join. This approach greatly reduces the cost for each of the smaller joins and therefore reduces the overall join cost. Observe that all available processors are fully utilized, even though the smaller joins are carried out one after the other.

The reader is invited to adapt the network of operators shown in Figure 21.3 to reflect the improved parallel join algorithm.

## 21.4 PARALLEL QUERY OPTIMIZATION

In addition to parallelizing individual operations, we can obviously execute different operations in a query in parallel and execute multiple queries in parallel. Optimizing a single query for parallel execution has received more attention; systems typically optimize queries without regard to other queries that might be executing at the same time.

Two kinds of interoperation parallelism can be exploited within a query:

- The result of one operator can be pipelined into another. For example, consider a left-deep plan in which all the joins use index nested loops. The result of the first (i.e., the bottom-most) join is the outer relation tuples for the next join node. As tuples are produced by the first join, they can be used to probe the inner relation in the second join. The result of the second join can similarly be pipelined into the next join, and so on.
- Multiple independent operations can be executed concurrently. For example, consider a (nonleft-deep) plan in which relations $A$ and $B$ are joined, relations $C$ and $D$ are joined, and the results of these two joins are finally joined. Clearly, the join of $A$ and $B$ can be executed concurrently with the join of $C$ and $D$.

An optimizer that seeks to parallelize query evaluation has to consider several issues, and we will only outline the main points. The cost of executing individual operations in parallel (e.g., parallel sorting) obviously differs from executing them sequentially, and the optimizer should estimate operation costs accordingly.

Next, the plan that returns answers quickest may not be the plan with the least cost. For example, the cost of $A \bowtie B$ plus the cost of $C \bowtie D$ plus the cost of joining their results may be more than the cost of the cheapest left-deep plan. However, the time taken is the time for the more expensive of $A \bowtie B$ and $C \bowtie D$, plus the time to join their results. This time may be less than the time taken by the cheapest left-deep plan. This observation suggests that a parallelizing optimizer should not restrict itself to only left-deep trees and should also consider *bushy* trees, which significantly enlarge the space of plans to be considered.

Finally, there are a number of parameters such as available buffer space and the number of free processors that will be known only at run-time. This comment holds in a multiuser environment even if only sequential plans are considered; a multiuser environment is a simple instance of interquery parallelism.