around. Our description of parallelization of operations is based loosely on this model.

Parallel query-optimization techniques are described by Lu et al. [1991], Hong and Stonebraker [1991], Ganguly et al. [1992], Lanzelotte et al. [1993], and Hasan and Motwani [1995].

CHAPTER 18

DISTRIBUTED DATABASES

In Chapter 16, we discussed the basic structure of distributed systems. Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components. Furthermore, the database systems that run on each site may have a substantial degree of mutual independence.

Each site may participate in the execution of transactions that access data at one site, or several sites. The main difference between centralized and distributed database systems is that, in the former, the data reside in one single location, whereas in the latter, the data reside in several locations. This distribution of data is the cause of many difficulties in transaction processing and query processing. In this chapter, we address these difficulties.

We start with the question of how to store data in a distributed database, in Section 18.1. In Section 18.2, we consider issues of network transparency and of naming of data items. Several techniques have been developed for processing queries in a distributed database; they are examined in Section 18.3.

The next few sections address transaction processing. The basic model and problems caused by different kinds of failures are examined in Section 18.4. Commit protocols designed to provide atomic transaction commit, in spite of these problems, are described in Section 18.5. Recovery issues related to selecting new coordinators when a site designated as a coordinator fails are considered in Section 18.6. Concurrency control in distributed systems is examined in Section 18.7; the issue of handling deadlocks is dealt with in Section 18.8.

In recent years, the need has arisen for accessing and updating data from a variety of preexisting databases, which differ in their hardware and software envi-

ronments, and in the schemas under which data are stored. A *multidatabase system* is a software layer that enables such a heterogeneous collection of databases to be treated like a homogeneous distributed database. Section 18.9 deals with query-processing and transaction-processing issues related to multidatabase systems.

18.1 Distributed Data Storage

Consider a relation r that is to be stored in the database. There are several approaches to storing this relation in the distributed database:

- **Replication**. The system maintains several identical replicas (copies) of the relation. Each replica is stored at a different site, resulting in data replication. The alternative to replication is to store only one copy of relation r.
- **Fragmentation**. The relation is partitioned into several fragments. Each fragment is stored at a different site.
- **Replication and fragmentation**. The relation is partitioned into several fragments. The system maintains several replicas of each fragment.

In the following subsections, we elaborate on each of these techniques.

18.1.1 Data Replication

If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, we have *full replication*, in which a copy is stored in every site in the system.

There are a number of advantages and disadvantages to replication.

- **Availability**. If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.
- **Increased parallelism**. In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.
- **Increased overhead on update**. The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.

In general, replication enhances the performance of **read** operations and increases the availability of data to read-only transactions. However, update transactions incur greater overhead. Controlling concurrent updates by several transactions to replicated data is more complex than is using the centralized approach to

concurrency control that we saw in Chapter 14. We can simplify the management of replicas of relation r by choosing one of them as the *primary copy of* r. For example, in a banking system, an account can be associated with the site in which the account has been opened. Similarly, in an airline-reservation system, a flight can be associated with the site at which the flight originates. We shall examine the options for distributed concurrency control in Section 18.7.

18.1.2 Data Fragmentation

If relation r is fragmented, r is divided into a number of *fragments* $r_1, r_2, \ldots, r_n$. These fragments contain sufficient information to allow reconstruction of the original relation r. As we shall see, this reconstruction can take place through the application of either the union operation or a special type of join operation on the various fragments. There are two different schemes for fragmenting a relation: *horizontal* fragmentation and *vertical* fragmentation. Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme R of relation r in a special way that we shall discuss. These two schemes can be applied successively to the same relation, resulting in a number of different fragments. Note that some information may appear in several fragments.

We discuss the various ways for fragmenting a relation in Sections 18.1.2.1 to 18.1.2.3. We shall illustrate these approaches by fragmenting the relation *account*, with schema

Account-schema = (*branch-name*, *account-number*, *balance*)

The relation *account* (*Account-schema*) is shown in Figure 18.1.

18.1.2.1 Horizontal Fragmentation

The relation r is partitioned into a number of subsets, $r_1, r_2, \ldots, r_n$. Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

A fragment can be defined as a *selection* on the global relation r. That is, we use a predicate P_i to construct fragment r_i as follows:

$$r_i = \sigma_{P_i}(r)$$

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Valleyview	A-177	205
Valleyview	A-402	10000
Hillside	A-155	62
Valleyview	A-408	1123
Valleyview	A-639	750

Figure 18.1 Sample *account* relation.

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2$

Figure 18.2 Horizontal fragmentation of relation *account*.

We obtain the reconstruction of the relation r by taking the union of all fragments; that is,

$$r = r_1 \cup r_2 \cup \ldots \cup r_n$$

As an illustration, suppose that the relation r is the *account* relation of Figure 18.1. This relation can be divided into n different fragments, each of which consists of tuples of accounts belonging to a particular branch. If the banking system has only two branches—Hillside and Valleyview—then there are two different fragments:

$$account_1 = \sigma_{branch\text{-}name\ =\ \text{"Hillside"}}\ (account)$$
$$account_2 = \sigma_{branch\text{-}name\ =\ \text{"Valleyview"}}\ (account)$$

These two fragments are shown in Figure 18.2. Fragment $account_1$ is stored in the Hillside site. Fragment $account_2$ is stored in the Valleyview site.

In our example, the fragments are disjoint. By changing the selection predicates used to construct the fragments, we can have a particular tuple of r appear in more than one of the r_i. This form of data replication is discussed further at the end of this section.

18.1.2.2 Vertical Fragmentation

In its simplest form, vertical fragmentation is the same as decomposition (see Chapter 7). Vertical fragmentation of $r(R)$ involves the definition of several subsets of attributes $R_1, R_2, \ldots, R_n$ of the schema R such that

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

Each fragment r_i of r is defined by

$$r_i = \Pi_{R_i}(r)$$

branch-name	*account-number*	*customer-name*	*balance*
Hillside	A-305	Lowman	500
Hillside	A-226	Camp	336
Valleyview	A-177	Camp	205
Valleyview	A-402	Kahn	10000
Hillside	A-155	Kahn	62
Valleyview	A-408	Kahn	1123
Valleyview	A-639	Green	750

Figure 18.3 Sample *deposit* relation.

The fragmentation should be done such that we can reconstruct relation r from the fragments by taking the natural join

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots \bowtie r_n$$

One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each of the R_i. More generally, any superkey can be used. It is often convenient to add a special attribute, called a *tuple-id*, to the schema R. The tuple-id value of a tuple is a unique value, used to distinguish the tuple from all other tuples. The tuple-id attribute thus serves as a candidate key for the augmented schema, and is included in each of the R_is. The physical or logical address for a tuple can be used as a tuple-id, since each tuple has a unique address.

To illustrate vertical fragmentation, we consider for our bank database an alternative database design that includes the schema[†]

Deposit-schema = (*branch-name*, *account-number*, *customer-name*, *balance*)

Figure 18.3 shows the *deposit* relation for our example. In Figure 18.4, we show the relation *deposit'*: the *deposit* relation of Figure 18.3 with tuple-ids

branch-name	*account-number*	*customer-name*	*balance*	*tuple-id*
Hillside	A-305	Lowman	500	1
Hillside	A-226	Camp	336	2
Valleyview	A-177	Camp	205	3
Valleyview	A-402	Kahn	10000	4
Hillside	A-155	Kahn	62	5
Valleyview	A-408	Kahn	1123	6
Valleyview	A-639	Green	750	7

Figure 18.4 The *deposit* relation of Figure 18.3 with tuple-ids.

[†] Although the highly normalized database design that we use elsewhere in this text can be vertically fragmented, such fragmentation is not particularly useful. Vertical fragmentation is more meaningful for a schema such as the one we use here.

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1$

account-number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2$

Figure 18.5 Vertical fragmentation of relation *deposit*.

added. Figure 18.5 shows a vertical decomposition of the schema *Deposit-schema* ∪ {*tuple-id*} into

Deposit-schema-1 = (*branch-name*, *customer-name*, *tuple-id*)
Deposit-schema-2 = (*account-number*, *balance*, *tuple-id*)

The two relations shown in Figure 18.5 result from computing

$$deposit_1 = \Pi_{Deposit\text{-}schema\text{-}1}(deposit')$$
$$deposit_2 = \Pi_{Deposit\text{-}schema\text{-}2}(deposit')$$

To reconstruct the original *deposit* relation from the fragments, we compute

$$\Pi_{Deposit\text{-}schema}(deposit_1 \bowtie deposit_2)$$

Note that the expression

$$deposit_1 \bowtie deposit_2$$

is a special form of natural join. The join attribute is *tuple-id*. Although the *tuple-id* attribute facilitates the implementation of vertical partitioning, it must not be visible to users, since it is an internal artifact of the implementation, and violates data independence — which is one of the main virtues of the relational model.

18.1.2.3 Mixed Fragmentation

The relation r is divided into a number of fragment relations $r_1, r_2, ..., r_n$. Each fragment is obtained as the result of application of either the horizontal-fragmentation or vertical-fragmentation scheme on relation r, or on a fragment of r that was obtained previously.

As an illustration, suppose that the relation r is the *deposit* relation of Figure 18.3. This relation is divided initially into the fragments $deposit_1$ and $deposit_2$, as defined previously. We can now further divide fragment $deposit_1$, using the horizontal-fragmentation scheme, into the following two fragments:

$$deposit_{1a} = \sigma_{branch\text{-}name\ =\ \text{"Hillside"}}(deposit_1)$$
$$deposit_{1b} = \sigma_{branch\text{-}name\ =\ \text{"Valleyview"}}(deposit_1)$$

Thus, relation r is divided into three fragments: $deposit_{1a}$, $deposit_{1b}$, and $deposit_2$. Each of these fragments may reside in a different site.

18.1.3 Data Replication and Fragmentation

The techniques described in Sections 18.1.1 and 18.1.2 for data replication and data fragmentation can be applied successively to the same relation. That is, a fragment can be replicated, replicas of fragments can be fragmented further, and so on. For example, consider a distributed system consisting of sites S_1, $S_2, ..., S_{10}$. We can fragment *deposit* into $deposit_{1a}$, $deposit_{1b}$, and $deposit_2$, and, for example, store a copy of $deposit_{1a}$ at sites S_1, S_3, and S_7; a copy of $deposit_{1b}$ at sites S_7 and S_{10}; and a copy of $deposit_2$ at sites S_2, S_8, and S_9.

18.2 Network Transparency

In Section 18.1, we saw that a relation r can be stored in a variety of ways in a distributed database system. It is essential that the system minimize the degree to which a user needs to be aware of how a relation is stored. As we shall see, a system can hide the details of the distribution of data in the network. We call this hiding *network transparency*, and define it as the degree to which system users can remain unaware of the details of how and where the data items are stored in a distributed system.

We shall consider the issues of transparency from the points of view of

- Naming of data items
- Replication of data items
- Fragmentation of data items
- Location of fragments and replicas

18.2.1 Naming of Data Items

Data items—such as relations, fragments, and replicas — must have unique names. This property is easy to ensure in a centralized database. In a distributed database,

however, we must take care to ensure that two sites do not use the same name for distinct data items.

One solution to this problem is to require all names to be registered in a central *name server*. The name server helps to ensure that the same name does not get used for different data items. We can also use the name server to locate a data item, given the name of the item. This approach, however, suffers from two major disadvantages. First, the name server may become a performance bottleneck when data items are located via their names, resulting in poor performance. Second, if the name server crashes, it may not be possible for any site in the distributed system to continue to run.

An alternative approach is to require that each site prefix its own site identifier to any name that it generates. This approach ensures that no two sites generate the same name (since each site has a unique identifier). Furthermore, no central control is required. This solution, however, fails to achieve network transparency, since site identifiers are attached to names. Thus, the *account* relation might be referred to as *site*17.*account*, rather than as simply *account*.

To overcome this problem, the database system can create a set of alternative names or *aliases* for data items. A user may thus refer to data items by simple names that are translated by the system to complete names. The mapping of aliases to the real names can be stored at each site. With aliases, the user can be unaware of the physical location of a data item. Furthermore, the user will be unaffected if the database administrator decides to move a data item from one site to another.

Each replica of a data item and each fragment of a data item must also have a unique name. It is important that the system be able to determine those replicas that are replicas of the same data item and those fragments that are fragments of the same data item. We adopt the convention of postfixing ".*f*1", ".*f*2", . . . , ".*fn*" to fragments of a data item, and ".*r*1", ".*r*2", . . . , ".*rn*" to replicas. Thus

*site*17.*account*.*f*3.*r*2

refers to replica 2 of fragment 3 of *account*, and tells us that this item was generated by site 17.

It is undesirable to expect users to refer to a specific replica of a data item. Instead, the system should determine which replica to reference on a **read** request, and should update all replicas on a **write** request. We can ensure that it does so by maintaining a catalog table, which the system uses to determine all replicas for the data item.

Similarly, a user should not be required to know how a data item is fragmented. As we observed earlier, vertical fragments may contain *tuple-ids*. Horizontal fragments may involve complicated selection predicates. Therefore, a distributed database system should allow requests to be stated in terms of the unfragmented data items. This requirement presents no major difficulty, since it is always possible to reconstruct the original data item from its fragments. However, it may be inefficient to reconstruct data from fragments. Returning to our horizontal fragmentation of *account*, consider the query

$$\sigma_{branch\text{-}name = \text{"Hillside"}}\ (account)$$


```
if name appears in the alias table
    then expression := map (name)
    else expression := name;

function map (n)
    if n appears in the replica table
        then result := name of a replica of n;
    if n appears in the fragment table
        then begin
                result := expression to construct fragment;
                for each n' in result do begin
                    replace n' in result with map (n');
                end
            end
    return result;
```

Figure 18.6 Name-translation algorithm.

We could answer this query using only the $account_1$ fragment. However, fragmentation transparency requires that the user not be aware of the existence of fragments $account_1$ and $account_2$. If we reconstruct *account* prior to processing the query, we obtain the expression

$$\sigma_{branch\text{-}name = \text{"Hillside"}}\ (account_1 \cup account_2)$$

The optimization of this expression is left to the query optimizer (see Section 18.3).

Figure 18.6 shows the complete translation scheme for a given data-item name. To illustrate the operation of the scheme, we consider a user located in the Hillside branch (site S_1). This user uses the alias *local-account* for the local fragment *account.f1* of the *account* relation. When this user references *local-account*, the query-processing subsystem looks up *local-account* in the alias table, and replaces *local-account* with *S*1.*account.f1*. It is possible that *S*1.*account.f1* is replicated. If so, the system must consult the replica table in order to choose a replica. This replica could itself be fragmented, requiring examination of the fragmentation table. In most cases, only one or two tables must be consulted. However, the name-translation scheme of Figure 18.6 is sufficiently general to deal with any combination of successive replication and fragmentation of relations.

18.2.2 Transparency and Updates

Providing transparency for users that update the database is somewhat more difficult than is providing transparency for readers. The main problems are ensuring that all replicas of a data item are updated, and that all affected fragments are updated.

In its full generality, the update problem for replicated and fragmented data is related to the problem of view maintenance—that is, to the problem of keep-

ing materialized views up-to-date when the database relations are updated (Section 3.7.1). Consider our example of the *account* relation, and the insertion of the tuple

("Valleyview", A-733, 600)

If *account* is fragmented horizontally, there is a predicate P_i associated with the ith fragment. We apply P_i to the tuple ("Valleyview", A-733, 600) to test whether that tuple must be inserted in the ith fragment. Using our example of *account* being fragmented into

$$account_1 = \sigma_{branch\text{-}name = \text{"Hillside"}} (account)$$
$$account_2 = \sigma_{branch\text{-}name = \text{"Valleyview"}} (account)$$

the tuple would be inserted into $account_2$.

Now consider a vertical fragmentation of deposit into $deposit_1$ and $deposit_2$. The tuple ("Valleyview", A-733, "Jones", 600) must be split into two fragments: one to be inserted into $deposit_1$, and one to be inserted into $deposit_2$.

If an update is made to a replicated relation, the update must be applied to all replicas. This requirement presents a problem if there is concurrent access to the relation, since it is possible that one replica will be updated earlier than another. We consider this problem in Section 18.7.

18.3 Distributed Query Processing

In Chapter 12, we saw that there is a variety of methods for computing the answer to a query. We examined several techniques for choosing a strategy for processing a query that minimize the amount of time that it takes to compute the answer. For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses. In a distributed system, we must take into account several other matters, including

- The cost of data transmission over the network
- The potential gain in performance from having several sites process parts of the query in parallel

The relative cost of data transfer over the network and data transfer to and from disk varies widely depending on the type of network and on the speed of the disks. Thus, in general, we cannot focus solely on disk costs or on network costs. Rather, we must find a good tradeoff between the two.

18.3.1 Query Transformation

Let us consider an extremely simple query: "Find all the tuples in the *account* relation." Although the query is simple — indeed, trivial—processing of this query is not trivial, since the *account* relation may be fragmented, replicated, or both, as we saw in Section 18.1. If the *account* relation is replicated, we have a choice of replica to make. If no replicas are fragmented, we choose the replica for which the transmission cost is lowest. However, if a replica is fragmented, the choice is not so easy to make, since we need to compute several joins or unions to reconstruct the *account* relation. In this case, the number of strategies for our simple example may be large. Query optimization by exhaustive enumeration of all alternative strategies may not be practical in such situations.

Fragmentation transparency implies that a user may write a query such as

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (account)$$

Since *account* is defined as

$$account_1 \cup account_2$$

the expression that results from the name translation scheme is

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (account_1 \cup account_2)$$

Using the query-optimization techniques of Chapter 12, we can simplify the preceding expression automatically. The result is the expression

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (account_1) \cup \sigma_{branch\text{-}name = \text{"Hillside"}} (account_2)$$

which includes two subexpressions. The first involves only $account_1$, and thus can be evaluated at the Hillside site. The second involves only $account_2$, and thus can be evaluated at the Valleyview site.

There is a further optimization that can be made in evaluating

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (account_1)$$

Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation. In evaluating

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (account_2)$$

we can apply the definition of the $account_2$ fragment to obtain

$$\sigma_{branch\text{-}name = \text{"Hillside"}} (\sigma_{branch\text{-}name = \text{"Valleyview"}} (account))$$

This expression is the empty set, regardless of the contents of the *account* relation.

Thus, our final strategy is for the Hillside site to return $account_1$ as the result of the query.

18.3.2 Simple Join Processing

As we saw in Chapter 12, a major aspect of the selection of a query-processing strategy is choosing a join strategy. Consider the following relational-algebra expression:

$$account \bowtie depositor \bowtie branch$$

Assume that the three relations are neither replicated nor fragmented, and that *account* is stored at site S_1, *depositor* at S_2, and *branch* at S_3. Let S_I denote the site at which the query was issued. The system needs to produce the result at site S_I. Among the possible strategies for processing this query are the following:

- Ship copies of all three relations to site S_I. Using the techniques of Chapter 12, choose a strategy for processing the entire query locally at site S_I.
- Ship a copy of the *account* relation to site S_2, and compute $temp_1 = account \bowtie depositor$ at S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3. Ship the result $temp_2$ to S_I.
- Devise strategies similar to the previous one, with the roles of S_1, S_2, S_3 exchanged.

No one strategy is always the best one. Among the factors that must be considered are the volume of data being shipped, the cost of transmitting a block of data between a pair of sites, and the relative speed of processing at each site. Consider the first two strategies listed. If we ship all three relations to S_I, and indices exist on these relations, we may need to recreate these indices at S_I. This recreation of indices entails extra processing overhead and extra disk accesses. However, the second strategy has the disadvantage that a potentially large relation ($customer \bowtie account$) must be shipped from S_2 to S_3. This relation repeats the address data for a customer once for each account that the customer has. Thus, the second strategy may result in extra network transmission, as compared with the first strategy.

18.3.3 Semijoin Strategy

Suppose that we wish to evaluate the expression $r_1 \bowtie r_2$, where r_1 and r_2 are stored at sites S_1 and S_2, respectively. Let the schemas of r_1 and r_2 be R_1 and R_2. Suppose that we wish to obtain the result at S_1. If there are many tuples of r_2 that do not join with any tuple of r_1, then shipping r_2 to S_1 entails shipping tuples that fail to contribute to the result. It is desirable to remove such tuples before shipping data to S_1, particularly if network costs are high.

A strategy can be implemented as follows:

1. Compute $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ at S_1.
2. Ship $temp_1$ from S_1 to S_2.
3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at S_2.
4. Ship $temp_2$ from S_2 to S_1.
5. Compute $r_1 \bowtie temp_2$ at S_1. The resulting relation is the same as $r_1 \bowtie r_2$.

Before considering the efficiency of this strategy, let us verify that the strategy computes the correct answer. In step 3, $temp_2$ has the result of $r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$. In step 5, we compute

$$r_1 \bowtie r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$$

Since join is associative and commutative, we can rewrite this expression as

$$(r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1)) \bowtie r_2$$

Since $r_1 \bowtie \Pi_{(R_1 \cap R_2)}(r_1) = r_1$, the expression is, indeed, equal to $r_1 \bowtie r_2$.

This strategy is particularly advantageous when relatively few tuples of r_2 contribute to the join. This situation is likely to occur if r_1 is the result of a relational-algebra expression involving selection. In such a case, $temp_2$ may have significantly fewer tuples than r_2. The cost savings of the strategy result from having to ship only $temp_2$, rather than all of r_2, to S_1. Additional cost is incurred in shipping $temp_1$ to S_2. If a sufficiently small fraction of tuples in r_2 contribute to the join, the overhead of shipping $temp_1$ will be dominated by the savings of shipping only a fraction of the tuples in r_2.

This strategy is called a *semijoin strategy*, after the semijoin operator of the relational algebra, denoted $\ltimes$. The semijoin of r_1 with r_2, denoted $r_1 \ltimes r_2$, is

$$\Pi_{R_1}(r_1 \bowtie r_2)$$

Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$. In step 3, $temp_2 = r_2 \ltimes r_1$.

For joins of several relations, this strategy can be extended to a series of semijoin steps. A substantial body of theory has been developed regarding the use of semijoins for query optimization. Some of this theory is referenced in the bibliographic notes.

18.3.4 Join Strategies that Exploit Parallelism

Implementing intraoperation parallelism by redistributing tuples is generally not considered viable in a distributed system, due to the small degree of parallelism and the high cost of communication. However, interoperation parallelism, including pipelined parallelism and independent parallelism (Section 17.6), can be useful in a distributed system.

For example, consider a join of four relations:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

where relation r_i is stored at site S_i. Assume that the result must be presented at site S_1. There are many possible strategies for parallel evaluation; for example, any of the strategies described in Section 17.6 may be used. In one such strategy, r_1 is shipped to S_2, and $r_1 \bowtie r_2$ computed at S_2. At the same time, r_3 is shipped to S_4, and $r_3 \bowtie r_4$ computed at S_4. Site S_2 can ship tuples of $(r_1 \bowtie r_2)$ to S_1 as they are produced, rather than waiting for the entire join to be computed. Similarly, S_4 can ship tuples of $(r_3 \bowtie r_4)$ to S_1. Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1, the computation of $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ can begin, with the pipelined join technique of Section 12.8.2.2. Thus, computation of the final join result at S_1 can be done in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2, and with the computation of $(r_3 \bowtie r_4)$ at S_4.

18.4 Distributed Transaction Model

Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties (Section 13.1). There are two types of transaction that we need to consider. The *local* transactions are those that access and update data in only one local database; the *global*

transactions are those that access and update data in several local databases. Ensuring the ACID properties of the local transactions can be done in a manner similar to that discussed in Chapters 13, 14, and 15. However, in the case of global transactions, this task is much more complicated, since several sites may be participating in execution. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

18.4.1 System Structure

Each site has its own *local transaction manager*, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. To understand how such a manager can be implemented, we define an abstract model of a transaction system. Each site of the system contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.

The overall system architecture is depicted in Figure 18.7.

The structure of a transaction manager is similar in many respects to the structure used in the centralized-system case. Each transaction manager is responsible for

- Maintaining a log for recovery purposes
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site

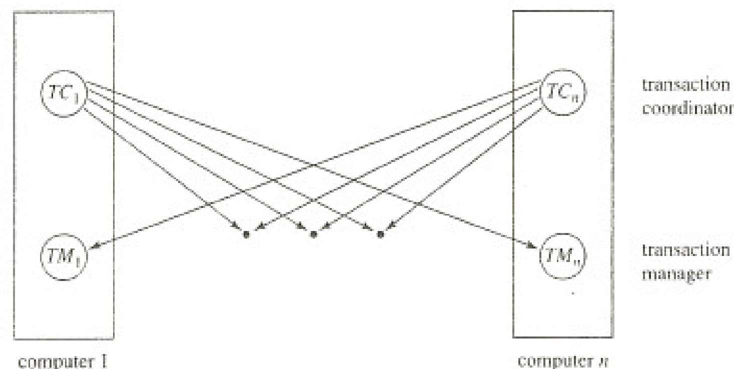

Figure 18.7 System architecture.

As we shall see, we need to modify both the recovery and concurrency schemes to accommodate the distribution of transactions.

The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for

- Starting the execution of the transaction
- Breaking the transaction into a number of subtransactions, and distributing these subtransactions to the appropriate sites for execution
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites

18.4.2 System Failure Modes

A distributed system may suffer from the same types of failure that a centralized system does (for example, software errors, hardware errors, or disk crashes). There are, however, additional types of failure with which we need to deal in a distributed environment. The basic failure types are

- Failure of a site
- Loss of messages
- Failure of a communication link
- Network partition

The loss or corruption of messages is always a possibility in a distributed system. The system uses transmission-control protocols, such as TCP/IP, to handle such errors. Information about such protocols may be found in standard textbooks on networking (see the bibliographic notes).

To understand the effect of failures of communication links, and of network partition, we must first understand how sites in a distributed system are interconnected. The sites in the system can be connected physically in a variety of ways. Some of the most common configurations are depicted in Figure 18.8.

Each configuration has advantages and disadvantages. The configurations can be compared with one another, based on the following criteria:

- **Installation cost.** The cost of physically linking the sites in the system
- **Communication cost.** The cost in time and money to send a message from site A to site B
- **Availability.** The degree to which data can be accessed despite the failure of some links or sites

The various topologies are depicted in Figure 18.8 as graphs whose nodes correspond to sites. An edge from node A to node B corresponds to a direct

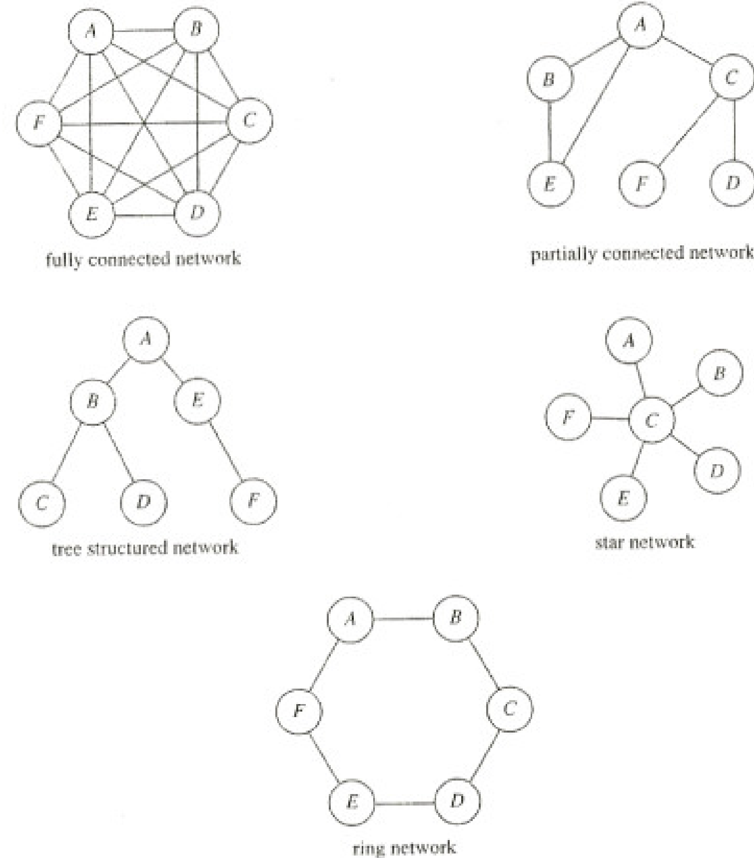

Figure 18.8 Network topology.

communication link between the two sites. In a fully connected network, each site is directly connected to every other site. However, the number of links grows as the square of the number of sites, resulting in a huge installation cost. Therefore, fully connected networks are impractical in any large system.

In a *partially connected network*, direct links exist between some—but not all—pairs of sites. Hence, the installation cost of such a configuration is lower than that of the fully connected network. However, if two sites A and B are not directly connected, messages from one to the other must be *routed* through a sequence of communication links. This requirement results in a higher communication cost.

If a communication link fails, messages that would have been transmitted across the link must be rerouted. In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination. In other cases, a failure may result in there being no connection between some pairs of sites. A system is *partitioned* if it has been split into two (or more) subsystems, called *partitions*, that lack any connection between them. Note that, under this definition, a subsystem may consist of a single node.

The different partially connected network types shown in Figure 18.8 have different failure characteristics and installation and communication costs. Installation and communication costs are relatively low for a tree-structured network. However, the failure of a single link in a tree-structured network can result in the network becoming partitioned. In a ring network, at least two links must fail for partition to occur. Thus, the ring network has a higher degree of availability than does a tree structured network. However, the communication cost is high, since a message may have to cross a large number of links. In a star network, the failure of a single link results in a network partition, but one of the partitions has only a single site. Such a partition can be treated as a single-site failure. The star network also has a low communication cost, since each site is at most two links away from every other site. However, the failure of the central site results in every site in the system becoming disconnected.

18.4.3 Robustness

For a distributed system to be robust, it must *detect* failures, *reconfigure* the system so that computation may continue, and *recover* when a processor or a link is repaired.

The different types of failures are handled in different ways. Message loss is handled by retransmission. Repeated retransmission of a message across a link, without receipt of an acknowledgment, is usually a symptom of a link failure. The network usually attempts to find an alternative route for the message. Failure to find such a route is usually a symptom of network partition.

It is generally not possible, however, to differentiate clearly between site failure and network partition. The system can usually detect that a failure has occurred, but it may not be able to identify the type of failure. For example, suppose that site S_1 is not able to communicate with S_2. It could be that S_2 has failed. However, another possibility is that the link between S_1 and S_2 has failed, resulting in network partition.

Suppose that site S_1 has discovered that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure, and to continue with the normal mode of operation.

- If replicated data are stored at the failed site, the catalog should be updated so that queries do not reference the copy at the failed site.
- If transactions were active at the failed site at the time of the failure, these transactions should be aborted. It is desirable to abort such transactions promptly, since they may hold locks on data at sites that are still active.
- If the failed site is a central server for some subsystem, an *election* must be held to determine the new server (see Section 18.6.2). Examples of central servers include a name server, a concurrency coordinator, or a global deadlock detector.

Since it is, in general, not possible to distinguish between network link failures and site failures, any reconfiguration scheme must be designed to work correctly in case of a partitioning of the network. In particular, the following situations must be avoided:

- Two or more central servers are elected in distinct partitions.
- More than one partition updates a replicated data item.

Reintegration of a repaired site or link into the system also requires care. When a failed site recovers, it must initiate a procedure to update its system tables to reflect changes made while it was down. If the site had replicas of any data items, it must obtain the current values of these data items and ensure that it receives all future updates. Reintegration of a site is more complicated than it may seem to be at first glance, since there may be updates to the data items processed during the time that the site is recovering. An easy solution is temporarily to halt the entire system while the failed site rejoins it. In most applications, however, such a temporary halt is unacceptably disruptive. Techniques have been developed to allow failed sites to reintegrate while allowing concurrent updates to data items. If a failed link recovers, two or more partitions can be rejoined. Since a partitioning of the network limits the allowable operations by some or all sites, it is desirable to inform all sites promptly of the recovery of the link. See the bibliographic notes for more information on recovery in distributed systems.

18.5 Commit Protocols

If we are to ensure atomicity, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites, or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a *commit protocol*.

Among the simplest and most widely used commit protocols is the *two-phase commit* protocol (2PC), which is described in Section 18.5.1. An alternative is the *three-phase commit* protocol (3PC), which avoids certain disadvantages of the 2PC protocol but adds to complexity and overhead. The 3PC protocol is described in Section 18.5.2.

18.5.1 Two-Phase Commit

Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i.

18.5.1.1 The Commit Protocol

When T completes its execution—that is, when all the sites at which T has executed inform C_i that T has completed—C_i starts the 2PC protocol.

- **Phase 1.** C_i adds the record <**prepare** T> to the log, and forces the log onto stable storage. It then sends a **prepare** T message to all sites at which T executed. On receiving such a message, the transaction manager at that site determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <**no** T> to the log, and then responds by sending an **abort** T message to C_i. If the answer is yes, it adds a record <**ready** T> to the log, and forces the log (with all the log records corresponding to T) onto stable storage. The transaction manager then replies with a **ready** T message to C_i.
- **Phase 2.** When C_i receives responses to the **prepare** T message from all the sites, or when a prespecified interval of time has elapsed since the **prepare** T message was sent out, C_i can determine whether the transaction T can be committed or aborted. Transaction T can be committed if C_i received a **ready** T message from all the participating sites. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <**commit** T> or a record <**abort** T> is added to the log and the log is forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this point, the coordinator sends either a **commit** T or an **abort** T message to all participating sites. When a site receives that message, it records the message in the log.

A site at which T executed can unconditionally abort T at any time prior to its sending the message **ready** T to the coordinator. The **ready** T message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T. The only means by which a site can make such a promise is if the needed information is stored in stable storage. Otherwise, if the site crashes after sending **ready** T, it may be unable to make good on its promise.

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one site responds **abort** T. Since the coordinator site S_i is one of the sites at which T executed, the coordinator can decide unilaterally to abort T. The final verdict regarding T is determined at the time that the coordinator writes that verdict (commit or abort) to the log and forces that verdict to stable storage. In some implementations of the 2PC protocol, a site sends an **acknowledge** T message to the coordinator at the end of the second phase of the protocol. When the coordinator receives the **acknowledge** T message from all the sites, it adds the record <**complete** T> to the log.

18.5.1.2 Handling of Failures

We now examine in detail how the 2PC protocol responds to various types of failures.

- **Failure of a participating site.** If the coordinator C_i detects that a site has failed, it takes the following actions. If the site fails before responding with a **ready** T message to C_i, it is assumed to have responded with an **abort** T message. If the site fails after the coordinator has received the **ready** T message from the site, the rest of the commit protocol is executed in the normal fashion, ignoring the failure of the site.

  When a participating site S_k recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst

of execution when the failure occurred. Let T be one such transaction. We consider each of the possible cases:

- The log contains a <**commit** T> record. In this case, the site executes **redo**(T).
- The log contains an <**abort** T> record. In this case, the site executes **undo**(T).
- The log contains a <**ready** T> record. In this case, the site must consult C_i to determine the fate of T. If C_i is up, it notifies S_k regarding whether T committed or aborted. In the former case, it executes **redo**(T); in the latter case, it executes **undo**(T). If C_i is down, S_k must try to find the fate of T from other sites. It does so by sending a **query-status** T message to all the sites in the system. On receiving such a message, a site must consult its log to determine whether T has executed there, and if T has, whether T committed or aborted. It then notifies S_k about this outcome. If no site has the appropriate information (that is, whether T committed or aborted), then S_k can neither abort nor commit T. The decision concerning T is postponed until S_k can obtain the needed information. Thus, S_k must periodically resend the **query-status** message to the other sites. It continues to do so until a site recovers that contains the needed information. Note that the site at which C_i resides always has the needed information.
- The log contains no control records (**abort, commit, ready**) concerning T. Thus, we know that S_k failed before responding to the **prepare** T message from C_i. Since the failure of S_k precludes the sending of such a response, by our algorithm C_i must abort T. Hence, S_k must execute **undo**(T).

- **Failure of the coordinator.** If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating sites must decide the fate of T. We shall see that, in certain cases, the participating sites cannot decide whether to commit or abort T, and therefore these sites must wait for the recovery of the failed coordinator.
  - If an active site contains a <**commit** T> record in its log, then T must be committed.
  - If an active site contains an <**abort** T> record in its log, then T must be aborted.
  - If some active site does *not* contain a <**ready** T> record in its log, then the failed coordinator C_i cannot have decided to commit T, because a site that does not have a <**ready** T> record in its log cannot have sent a **ready** T message to C_i. However, the coordinator may have decided to abort T, but not to commit T. Rather than wait for C_i to recover, it is preferable to abort T.
  - If none of the preceding cases holds, then all active sites must have a <**ready** T> record in their logs, but no additional control records (such as <**abort** T> or <**commit** T>). Since the coordinator has failed, it is impossible to determine whether a decision has been made, and if one has, what that decision is, until the coordinator recovers. Thus, the active sites must wait for C_i to recover. Since the fate of T remains in doubt, T may continue to hold system resources. For example, if locking is used, T may hold locks on data at active sites. Such a situation is undesirable, because it may be hours or days before C_i is again active. During this time, other transactions may be forced to wait for T. As a result, data items may be unavailable not only on the failed site (C_i), but on active sites as well. This situation is called the *blocking* problem, because T is blocked pending the recovery of site C_i.

- **Network partition.** When a network partitions, two possibilities exist:
  1. The coordinator and all its participants remain in one partition. In this case, the failure has no effect on the commit protocol.
  2. The coordinator and its participants belong to several partitions. From the viewpoint of the sites in one of the partitions, it appears that the sites in other partitions have failed. Sites that are not in the partition containing the coordinator simply execute the protocol to deal with failure of the coordinator. The coordinator and the sites that are in the same partition as the coordinator follow the usual commit protocol, assuming that the sites in the other partitions have failed.

Thus, the major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, where a decision either to commit or to abort T may have to be postponed until C_i recovers.

18.5.1.3 Recovery and Concurrency Control

When a failed site restarts, we can perform recovery using, for example, the recovery algorithm described in Section 15.9. To deal with distributed commit protocols (such as 2PC and 3PC), the recovery procedure must treat *in-doubt* transactions specially; in-doubt transactions are transactions for which a <**ready** T> log record is found, but neither a <**commit** T> log record, nor an <**abort** T> log record, is found. The recovering site must determine the commit–abort status of such transactions by contacting other sites, as described in Section 18.5.1.2.

If recovery is done as just described, however, normal transaction processing at the site cannot begin until all in-doubt transactions have been committed or rolled back. Finding the status of in-doubt transactions can be slow, since multiple sites may have to be contacted. Further, if the coordinator has failed, and no other site has information about the commit–abort status of an incomplete transaction, recovery potentially could become blocked if 2PC is used. As a result, the site performing restart recovery may remain unusable for a long period.

To circumvent this problem, recovery algorithms typically provide support for noting lock information in the log. (We are assuming here that locking is used for concurrency control.) Instead of a <**ready** T> log record being written, a <**ready** T, L> log record is written out, where L is a list of all locks held by the transaction T when the log record is written. At recovery time, after performance of local recovery actions, for every in-doubt transaction T, all the locks noted in the <**ready** T, L> log record (read from the log) are reacquired.

After lock reacquisition is complete for all in-doubt transactions, transaction processing can start at the site, even before the commit–abort status of the in-doubt

transactions is determined. The commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions. Thus, site recovery is faster, and never gets blocked. Note that new transactions that have a lock conflict with any in-doubt transactions will be unable to make progress until the conflicting in-doubt transactions have been committed or rolled back.

18.5.2 Three-Phase Commit

The 3PC protocol is designed to avoid the possibility of blocking in a restricted case of possible failures. The version of the 3PC protocol that we describe requires that

- No network partition can occur.
- At most K participating sites can fail while the 3PC protocol is being executed for a transaction. K is a parameter indicating the resiliency of the protocol to site failures.
- At any point, at least $K + 1$ sites must be up.

The protocol achieves the nonblocking property by adding an extra phase in which a preliminary decision is reached regarding the fate of T. The information made available to the participating sites as a result of this preliminary decision allows a decision to be made despite the failure of the coordinator.

18.5.2.1 The Commit Protocol

As before, let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i.

- **Phase 1**. This phase is identical to phase 1 of the 2PC protocol.
- **Phase 2**. If C_i receives an **abort** T message from a participating site, or if C_i receives no response within a prespecified interval from a participating site, then C_i decides to abort T. The abort decision is implemented in the same way as is the 2PC protocol. If C_i receives a **ready** T message from every participating site, C_i makes the preliminary decision to *precommit* T. Precommit differs from commit in that T may still be aborted eventually. The precommit decision allows the coordinator to inform each participating site that all participating sites are ready. C_i adds a record <**precommit** T> to the log and forces the log onto stable storage. Then, C_i sends a **precommit** T message to all participating sites. When a site receives a message from the coordinator (either **abort** T or **precommit** T), it records that message in its log, forces this information to stable storage, and sends a message **acknowledge** T to the coordinator.
- **Phase 3**. This phase is executed only if the decision in phase 2 was to precommit. After the **precommit** T messages are sent to all participating sites, the coordinator must wait until it receives at least K **acknowledge** T messages. Then, the coordinator reaches a commit decision. It adds a <**commit** T> record to its log, and forces the log to stable storage. Then, C_i sends a **commit** T message to all participating sites. When a site receives that message, it records the information in its log.

Just as in the 2PC protocol, a site at which T executed can unconditionally abort T at any time prior to sending the message **ready** T to the coordinator. The **ready** T message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T. In contrast to the 2PC protocol, in which the coordinator can unconditionally abort T at any time prior to sending the message **commit** T, the **precommit** T message in the 3PC protocol is a promise by the coordinator to follow the participant's order to commit T.

Since phase 3 always leads to a commit decision, it may seem to be of little use. The role of the third phase becomes apparent when we look at how the 3PC protocol handles failures.

In some implementations of the 3PC protocol, a site sends a message **ack** T to the coordinator upon receipt of the **commit** T message. (Note the use of **ack** to distinguish this term from the **acknowledge** messages that were used in phase 2.) When the coordinator receives the **ack** T message from all sites, it adds the record <**complete** T> to the log.

18.5.2.2 Handling of Failures

We now examine in detail how the 3PC protocol responds to various types of failures.

- **Failure of a participating site.** If the coordinator C_i detects that a site has failed, the actions that it takes are similar to the actions taken in 2PC. If the site fails before responding with a **ready** T message to C_i, it is assumed to have responded with an **abort** T message. Otherwise, the rest of the commit protocol is executed in the normal fashion, ignoring the failure of the site.

  When a participating site S_j recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Let T be one such transaction. We consider each of the possible cases:
  - The log contains a <**commit** T> record. In this case, the site executes **redo**(T).
  - The log contains an <**abort** T> record. In this case, the site executes **undo**(T).
  - The log contains a <**ready** T> record, but no <**abort** T> or <**precommit** T> record. In this case, the site attempts to consult C_i to determine the fate of T. If C_i responds with a message that T aborted, the site executes **undo**(T). If C_i responds with a message **precommit** T, the site (as in phase 2) records this information in its log, and resumes the protocol by sending an **acknowledge** T message to the coordinator. If C_i responds with a message that T committed, the site executes **redo**(T). In the event that C_i fails to respond within a prespecified interval, the site executes a coordinator failure protocol (see next list entry).

- The log contains a <**precommit** T> record, but no <**abort** T> or <**commit** T> record. As before, the site consults C_i. If C_i responds that T aborted or committed, the site executes **undo**(T) or **redo**(T), respectively. If C_i responds that T is still in the precommit state, the site resumes the protocol at this point. If C_i fails to respond within a prescribed interval, the site executes the coordinator-failure protocol.

- **Failure of the coordinator**. When a participating site fails to receive a response from the coordinator, for whatever reason, it executes the *coordinator-failure protocol*. This protocol results in the selection of a new coordinator. When the failed coordinator recovers, it does so in the role of a participating site. It no longer acts as coordinator; rather, it must determine the decision that has been reached by the new coordinator.

18.5.2.3 Coordinator-Failure Protocol

The coordinator-failure protocol is triggered by a participating site that fails to receive a response from the coordinator within a prespecified interval. Since we assume no network partition, the only possible cause for this situation is the failure of the coordinator.

1. The active participating sites select a new coordinator using an election protocol (see Section 18.6).
2. The new coordinator, C_{new}, sends a message to each participating site requesting the local status of T.
3. Each participating site, including C_{new}, determines the local status of T:
   - **Committed**. The log contains a <**commit** T> record.
   - **Aborted**. The log contains an <**abort** T> record.
   - **Ready**. The log contains a <**ready** T> record, but contains no <**abort** T> or <**precommit** T> record.
   - **Precommitted**. The log contains a <**precommit** T> record, but contains no <**abort** T> or <**commit** T> record.
   - **Not ready**. The log contains neither a <**ready** T> nor an <**abort** T> record.

   Each participating site sends its local status to C_{new}.
4. Depending on the responses received, C_{new} decides either to commit or abort T, or to restart the 3PC protocol:
   - If at least one site has local status = **committed**, then C_{new} commits T.
   - If at least one site has local status = **aborted**, then C_{new} aborts T. (Note that it is not possible for some site to have local status = **committed** while another has local status = **aborted**.)
   - If no site has local status = **aborted**, and no site has local status = **committed**, but at least one site has local status = **precommitted**, then C_{new} resumes the 3PC protocol by sending new **precommit** messages.
   - Otherwise, C_{new} aborts T.

The coordinator-failure protocol allows the new coordinator to obtain knowledge about the state of the failed coordinator, C_i.

If any active site has a <**commit** T> in its log, then C_i must have decided to commit T. If an active site has <**precommit** T> in its log, then C_i must have reached a preliminary decision to precommit T, and that means that all sites, including any that may have failed, have reached **ready** states. It is therefore safe to commit T. However, C_{new} does not commit T unilaterally; doing so would create the same blocking problem, if C_{new} fails, as in 2PC. It is for this reason that phase 3 is resumed by C_{new}.

Consider the case where no active site has received a precommit message from C_i. We must consider three possibilities:

1. C_i had decided to commit T prior to C_i failing.
2. C_i had decided to abort T prior to C_i failing.
3. C_i had not yet decided the fate of T.

We shall show that the first of these alternatives is not possible, and that, therefore, it is safe to abort T.

Suppose that C_i had decided to commit T. Then, at least K sites must have decided to precommit T and have sent acknowledge messages to C_i. Since C_i has failed, and we assume that at most K sites fail while the 3PC protocol is executed for a transaction, at least one of the K sites must be active, and hence at least one active site would inform C_{new} that it has received a precommit message. Thus, if no active site had received a precommit message, a commit decision certainly could not have been reached by C_i. Therefore, it is indeed safe to abort T. It is possible that C_i had not decided to abort T, so it may still be possible to commit T. However, detecting that C_i had not decided to abort T would require waiting for C_i (or for some other failed site that had received a precommit message) to recover. Hence, the protocol aborts T if no active site has received a precommit message.

In the preceding discussion, if more than K sites could fail while the 3PC protocol is executed for a transaction, it may not be possible for the surviving participants to determine the action taken by C_i prior to failing; this situation would force blocking to occur until C_i recovers. Although a large value for K is best from this standpoint, it forces a coordinator to wait for more responses before deciding to commit—thus delaying routine (failure-free) processing. Further, if fewer than K participants (in addition to the coordinator) are active, it may not be possible for the coordinator to complete the commit protocol, resulting in blocking. Thus, the choice of a value for K is crucial, as it determines the degree to which the protocol avoids blocking.

Our assumption of no network partitions is crucial to our discussion. As mentioned earlier, it is, in general, impossible to differentiate between network failure and site failure. Thus, network partitioning could lead to the election of two new coordinators (each of which believes that all sites in partitions other than its own have failed). The decisions of the two coordinators may not agree, resulting in the transaction being committed in some sites while being aborted in others.

18.5.3 Comparison of Protocols

The 2PC protocol is widely used, despite its potential for blocking. The probability of blocking occurring in practice is usually sufficiently low that the extra cost of the 3PC protocol is not justified. The vulnerability of 3PC to link failures is another practical issue. This disadvantage can be overcome by network-level protocols, but that solution adds overhead.

We can streamline both protocols to reduce the number of messages sent, and to reduce the number of times that records must be forced to stable storage. The 3PC protocol can be extended to allow more than K failures, provided that not more than K sites fail before the new coordinator makes a commit decision. The bibliographic notes contain references to several such techniques.

18.6 Coordinator Selection

Several of the algorithms that we have presented require the use of a coordinator. If the coordinator fails because of a failure of the site at which it resides, the system can continue execution only by restarting a new coordinator on another site. It can do so by maintaining a backup to the coordinator that is ready to assume responsibility if the coordinator fails. Or, it can choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called *election* algorithms.

18.6.1 Backup Coordinators

A *backup coordinator* is a site that, in addition to other tasks, maintains enough information locally to allow it to assume the role of coordinator with minimal disruption to the distributed system. All messages directed to the coordinator are received by both the coordinator and its backup. The backup coordinator executes the same algorithms and maintains the same internal state information (such as, for a concurrency coordinator, the lock table) as does the actual coordinator. The only difference in function between the coordinator and its backup is that the backup does not take any action that affects other sites. Such actions are left to the actual coordinator.

In the event that the backup coordinator detects the failure of the actual coordinator, it assumes the role of coordinator. Since the backup has all the information available to it that the failed coordinator had, processing can continue without interruption.

The prime advantage to the backup approach is the ability to continue processing immediately. If a backup were not ready to assume the coordinator's responsibility, a newly appointed coordinator would have to seek information from all sites in the system so that it could execute the coordination tasks. Frequently, the only source of some of the requisite information is the failed coordinator. In this case, it may be necessary to abort several (or all) active transactions, and to restart them under the control of the new coordinator.

Thus, the backup-coordinator approach avoids a substantial amount of delay while the distributed system recovers from a coordinator failure. The disadvantage

is the overhead of duplicate execution of the coordinator's tasks. Furthermore, a coordinator and its backup need to communicate regularly to ensure that their activities are synchronized.

In short, the backup-coordinator approach incurs overhead during normal processing to allow fast recovery from a coordinator failure. In Section 18.6.2, we consider a lower-overhead recovery scheme that requires somewhat more effort to recover from a failure.

18.6.2 Election Algorithms

Election algorithms require that a unique identification number be associated with each active site in the system. For ease of notation, we shall assume that the identification number of site S_i is i. Also, to simplify our discussion, we assume that the coordinator always resides at the site with the largest identification number. The goal of an election algorithm is to choose a site for the new coordinator. Hence, when a coordinator fails, the algorithm must elect the active site that has the largest identification number. This number must be sent to each active site in the system. In addition, the algorithm must provide a mechanism by which a site recovering from a crash can identify the current coordinator.

The various election algorithms usually differ in terms of the network configuration. In this section, we present one of these algorithms: the *bully* algorithm.

Suppose that site S_i sends a request that is not answered by the coordinator within a prespecified time interval T. In this situation, it is assumed that the coordinator has failed, and S_i tries to elect itself as the site for the new coordinator.

Site S_i sends an election message to every site that has a higher identification number. Site S_i then waits, for a time interval T, for an answer from any one of these sites. If it receives no response within time T, it assumes that all sites with numbers greater than i have failed, and it elects itself as the site for the new coordinator and sends a message to inform all active sites with identification numbers lower than i that it is the site at which the new coordinator resides.

If S_i does receive an answer, it begins a time interval T', to receive a message informing it that a site with a higher identification number has been elected. (Some other site is electing itself coordinator, and should report the results within time T'.) If no message is received within T', then the site with a higher number is assumed to have failed, and site S_i restarts the algorithm.

After a failed site recovers, it immediately begins execution of the same algorithm. If there are no active sites with higher numbers, the recovered site forces all sites with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number. It is for this reason that the algorithm is termed the *bully* algorithm.

18.7 Concurrency Control

In this section, we show how some of the concurrency-control schemes discussed earlier can be modified such that they can be used in a distributed environment.

We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.

18.7.1 Locking Protocols

The various locking protocols described in Chapter 14 can be used in a distributed environment. The only change that needs to be incorporated is in the way the lock manager is implemented. We present several possible schemes that are applicable to an environment where data can be replicated in several sites. As in Chapter 14, we shall assume the existence of the *shared* and *exclusive* lock modes.

18.7.1.1 Single-Lock-Manager Approach

The system maintains a *single* lock manager that resides in a *single* chosen site—say, S_i. All lock and unlock requests are made at site S_i. When a transaction needs to lock a data item, it sends a lock request to S_i. The lock manager determines whether the lock can be granted immediately. If the lock can be granted, the lock manager sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted, at which time a message is sent to the site at which the lock request was initiated. The transaction can read the data item from *any* one of the sites at which a replica of the data item resides. In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.

The scheme has the following advantages:

- **Simple implementation**. This scheme requires two messages for handling lock requests, and one message for handling unlock requests.
- **Simple deadlock handling**. Since all lock and unlock requests are made at one site, the deadlock-handling algorithms discussed in Chapter 14 can be applied directly to this environment.

The disadvantages of the scheme include the following:

- **Bottleneck**. The site S_i becomes a bottleneck, since all requests must be processed there.
- **Vulnerability**. If the site S_i fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme must be used so that a new site can take over lock management from S_i, as described in Section 18.6.

18.7.1.2 Multiple Coordinators

A compromise between the advantages and disadvantages just noted can be achieved through a *multiple-coordinator approach*, in which the lock-manager function is distributed over several sites.

Each lock manager administers the lock and unlock requests for a subset of the data items. Each lock manager resides in a different site. This approach reduces the degree to which the coordinator is a bottleneck, but it complicates

deadlock handling, since the lock and unlock requests are not made at a single site.

18.7.1.3 Majority Protocol

In a majority protocol, each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that site. When a transaction wishes to lock data item Q, which is not replicated and resides at site S_i, a message is sent to the lock manager at site S_i requesting a lock (in a particular lock mode). If data item Q is locked in an incompatible mode, then the request is delayed until it can be granted. Once it has determined that the lock request can be granted, the lock manager sends a message back to the initiator indicating that it has granted the lock request. The scheme has the advantage of simple implementation. It requires two message transfers for handling lock requests, and one message transfer for handling unlock requests. However, deadlock handling is more complex. Since the lock and unlock requests are no longer made at a single site, the various deadlock-handling algorithms discussed in Chapter 14 must be modified, as we shall discuss in Section 18.8.

If data item Q is replicated in n different sites, then a lock-request message must be sent to more than one-half of the n sites in which Q is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response is delayed until the request can be granted. The transaction does not operate on Q until it has successfully obtained a lock on a majority of the replicas of Q.

This scheme deals with replicated data in a decentralized manner, thus avoiding the drawbacks of central control. However, when there is replication of data, it suffers from the following disadvantages:

- **Implementation**. The majority protocol is more complicated to implement than are the previous schemes. It requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests.
- **Deadlock handling**. Since the lock and unlock requests are not made at one site, the deadlock-handling algorithms must be modified (see Section 18.8). In addition, it is possible for a deadlock to occur even if only one data item is being locked. As an illustration, consider a system with four sites and full replication. Suppose that transactions T_1 and T_2 wish to lock data item Q in exclusive mode. Transaction T_1 may succeed in locking Q at sites S_1 and S_3, while transaction T_2 may succeed in locking Q at sites S_2 and S_4. Each then must wait to acquire the third lock; hence, a deadlock has occurred. Luckily, we can avoid such deadlocks with relative ease, by requiring all sites to request locks on the replicas of a data item in the same predetermined order.

18.7.1.4 Biased Protocol

The *biased protocol* is based on a model similar to that of the majority protocol. The difference is that requests for shared locks are given more favorable treatment than requests for exclusive locks. The system maintains a lock manager at each

site. Each manager manages the locks for all the data items stored at that site. *Shared* and *exclusive* locks are handled differently.

- **Shared locks.** When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site that contains a replica of Q.
- **Exclusive locks.** When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites that contain a replica of Q.

As before, the response to the request is delayed until it can be granted.

The biased scheme has the advantage of imposing less overhead on **read** operations than does the majority protocol. This savings is especially significant in common cases in which the frequency of **read** is much greater than the frequency of **write**. However, the additional overhead on writes is a disadvantage. Furthermore, the biased protocol shares the majority protocol's disadvantage of complexity in handling deadlock.

18.7.1.5 Primary Copy

In the case of data replication, we can choose one of the replicas as the primary copy. Thus, for each data item Q, the primary copy of Q must reside in precisely one site, which we call the *primary site of Q*.

When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q. As before, the response to the request is delayed until it can be granted.

Thus, the primary copy enables concurrency control for replicated data to be handled in a manner similar to that for unreplicated data. This similarity allows for a simple implementation. However, if the primary site of Q fails, Q is inaccessible, even though other sites containing a replica may be accessible.

18.7.2 Timestamping

The principal idea behind the timestamping scheme discussed in Section 14.2 is that each transaction is given a *unique* timestamp that the system uses in deciding the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a scheme for generating unique timestamps. Then, our previous protocols can be applied directly to the nonreplicated environment.

There are two primary methods for generating unique timestamps, one centralized and one distributed. In the centralized scheme, a single site is chosen for distributing the timestamps. The site can use a logical counter or its own local clock for this purpose.

In the distributed scheme, each site generates a unique local timestamp using either a logical counter or the local clock. We obtain the unique global timestamp by concatenating the unique local timestamp with the site identifier, which also must be unique (Figure 18.9). The order of concatenation is important! We use the site identifier in the least significant position to ensure that the global timestamps

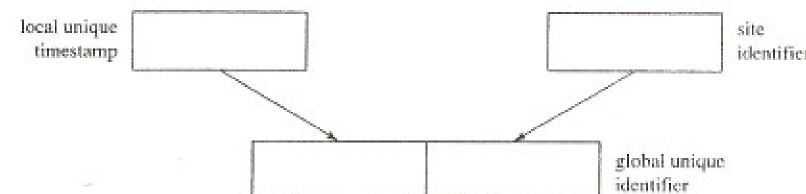

Figure 18.9 Generation of unique timestamps.

generated in one site are not always greater than those generated in another site. Compare this technique for generating unique timestamps with the one that we presented earlier for generating unique names.

We may still have a problem if one site generates local timestamps at a rate faster than that of the other sites. In such a case, the fast site's logical counter will be larger than that of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. What we need is a mechanism to ensure that local timestamps are generated fairly across the system. We define within each site S_i a *logical clock* (LC_i), which generates the unique local timestamp. The logical clock can be implemented as a counter that is incremented after a new local timestamp is generated. To ensure that the various logical clocks are synchronized, we require that a site S_i advance its logical clock whenever a transaction T_i with timestamp $<x,y>$ visits that site and x is greater than the current value of LC_i. In this case, site S_i advances its logical clock to the value $x + 1$.

If the system clock is used to generate timestamps, then timestamps are assigned fairly, provided that no site has a system clock that runs fast or slow. Since clocks may not be perfectly accurate, a technique similar to that used for logical clocks must be used to ensure that no clock gets far ahead of or behind another clock.

18.8 Deadlock Handling

The deadlock-prevention and deadlock-detection algorithms presented in Chapter 14 can be used in a distributed system, provided that modifications are made. For example, we can use the tree protocol by defining a *global* tree among the system data items. Similarly, the timestamp-ordering approach could be directly applied to a distributed environment, as we saw in Section 18.7.2.

Deadlock prevention may result in unnecessary waiting and rollback. Furthermore, certain deadlock-prevention techniques may require more sites to be involved in the execution of a transaction than would otherwise be the case.

If we allow deadlocks to occur and rely on deadlock detection, the main problem in a distributed system is deciding how to maintain the wait-for graph. Common techniques for dealing with this issue require that each site keep a *local* wait-for graph. The nodes of the graph correspond to all the transactions (local as well as nonlocal) that are currently either holding or requesting any of the items local to that site. For example, Figure 18.10 depicts a system consisting of two sites,

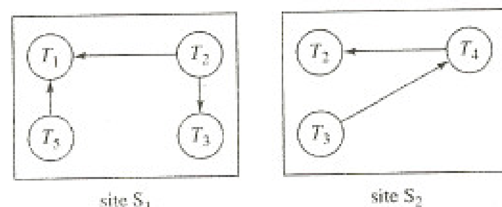

Figure 18.10 Local wait-for graphs.

each maintaining its local wait-for graph. Note that transactions T_2 and T_3 appear in both graphs, indicating that the transactions have requested items at both sites.

These local wait-for graphs are constructed in the usual manner for local transactions and data items. When a transaction T_i on site S_1 needs a resource in site S_2, a request message is sent by T_i to site S_2. If the resource is held by transaction T_j, an edge $T_i \rightarrow T_j$ is inserted in the local wait-for graph of site S_2.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. On the other hand, the fact that there are no cycles in any of the local wait-for graphs does not mean that there are no deadlocks. To illustrate this problem, we consider the local wait-for graphs of Figure 18.10. Each wait-for graph is acyclic; nevertheless, a deadlock exists in the system because the *union* of the local wait-for graphs contains a cycle. This graph is shown in Figure 18.11.

Several common schemes for organizing the wait-for graph in a distributed system are described in Sections 18.8.1 and 18.8.2.

18.8.1 Centralized Approach

In the *centralized approach*, a global wait-for graph (union of all the local graphs) is constructed and maintained in a *single* site: the deadlock-detection coordinator. Since there is communication delay in the system, we must distinguish between two types of wait-for graphs. The *real* graph describes the real but unknown state of the system at any instance in time, as would be seen by an omniscient observer. The *constructed* graph is an approximation generated by the controller during the execution of the controller's algorithm. Obviously, the constructed graph must be generated such that, whenever the detection algorithm is invoked, the reported results are correct in the sense that, if a deadlock exists, it is reported promptly, and if the system reports a deadlock, it is indeed in a deadlock state.

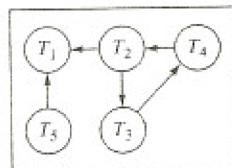

Figure 18.11 Global wait-for graph for Figure 18.10.

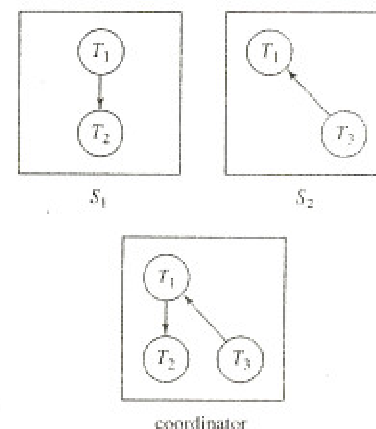

Figure 18.12 False cycles in the global wait-for graph.

The global wait-for graph can be constructed under these conditions:

- Whenever a new edge is inserted in or removed from one of the local wait-for graphs.
- Periodically, when a number of changes have occurred in a local wait-for graph.
- Whenever the coordinator needs to invoke the cycle-detection algorithm.

When the deadlock-detection algorithm is invoked, the coordinator searches its global graph. If a cycle is found, a victim is selected to be rolled back. The coordinator must notify all the sites that a particular transaction has been selected as victim. The sites, in turn, roll back the victim transaction.

We note that this scheme may produce unnecessary rollbacks, as a result of one of the following:

- *False cycles* may exist in the global wait-for graph. As an illustration, consider a snapshot of the system represented by the local wait-for graphs of Figure 18.12. Suppose that T_2 releases the resource that it is holding in site S_1, resulting in the deletion of the edge $T_1 \rightarrow T_2$ in S_1. Transaction T_2 then requests a resource held by T_3 at site S_2, resulting in the addition of the edge $T_2 \rightarrow T_3$ in S_2. If the **insert** $T_2 \rightarrow T_3$ message from S_2 arrives before the **remove** $T_1 \rightarrow T_2$ message from S_1, the coordinator may discover the false cycle $T_1 \rightarrow T_2 \rightarrow T_3$ after the **insert** (but before the **remove**). Deadlock recovery may be initiated, although no deadlock has occurred.

  Note that the preceding example could not occur under two-phase locking. Indeed, the likelihood of false cycles is usually sufficiently low that false cycles do not cause a serious performance problem.
- Unnecessary rollbacks may also result when a *deadlock* has indeed occurred and a victim has been picked, while one of the transactions was aborted for

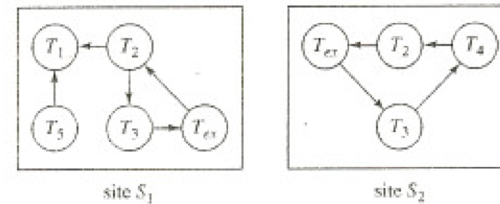

Figure 18.13 Local wait-for graphs.

reasons unrelated to the deadlock. For example, suppose that site S_1 in Figure 18.10 decides to abort T_2. At the same time, the coordinator has discovered a cycle, and has picked T_3 as a victim. Both T_2 and T_3 are now rolled back, although only T_2 needed to be rolled back.

18.8.2 Fully Distributed Approach

In the *fully distributed* deadlock-detection algorithm, all controllers share equally the responsibility for detecting deadlock. In this scheme, every site constructs a wait-for graph that represents a part of the total graph, depending on the dynamic behavior of the system. The idea is that, if a deadlock exists, a cycle will appear in (at least) one of the partial graphs. We present one such algorithm that involves construction of partial graphs in every site.

Each site maintains its own local wait-for graph. A local wait-for graph differs from the wait-for graph described previously in that we add one additional node T_{ex} to the graph. T_{ex} represents transactions that are external to the local site. An arc $T_i \rightarrow T_{ex}$ exists in the graph if T_i is waiting for a data item in another site that is being held by *any* transaction. Similarly, an arc $T_{ex} \rightarrow T_j$ exists in the graph if a transaction at another site is waiting to acquire a resource currently being held by T_j in this local site.

As an illustration, consider the two local wait-for graphs of Figure 18.10. The addition of the node T_{ex} in both graphs results in the local wait-for graphs shown in Figure 18.13.

If a local wait-for graph contains a cycle that does not involve node T_{ex}, then the system is in a deadlock state. However, the existence of a cycle involving T_{ex} implies that there is a *possibility* of a deadlock. To ascertain whether a deadlock really exists, we must invoke a distributed deadlock detection algorithm.

Suppose that site S_i contains in its local wait-for graph a cycle involving node T_{ex}. This cycle must be of the form

$$T_{ex} \rightarrow T_{k_1} \rightarrow T_{k_2} \rightarrow \ldots \rightarrow T_{k_n} \rightarrow T_{ex}$$

which indicates that transaction T_{k_n} in S_i is waiting to acquire a data item in some other site — say, S_j. On discovering this cycle, site S_i sends to site S_j a deadlock-detection message containing information about that cycle.

When site S_j receives this deadlock-detection message, it updates its local wait-for graph with the new information that it has obtained. Next, it searches

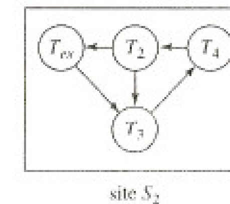

Figure 18.14 Local wait-for graph.

the newly constructed wait-for graph for a cycle not involving T_{ex}. If one exists, a deadlock is found, and an appropriate recovery scheme is invoked. If a cycle involving T_{ex} is discovered, then S_j transmits a deadlock-detection message to the appropriate site — say, S_k. Site S_k, in return, repeats the previous procedure. Thus, after a finite number of rounds, if a deadlock exists, it is discovered; if no deadlock exists, the deadlock-detection computation halts.

As an illustration, consider the local wait-for graphs of Figure 18.13. Suppose that site S_1 discovers the cycle

$$T_{ex} \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ex}$$

Since T_3 is waiting to acquire a data item in site S_2, a deadlock-detection message describing that cycle is transmitted from site S_1 to site S_2. When site S_2 receives this message, it updates its local wait-for graph, obtaining the wait-for graph of Figure 18.14. This graph contains the cycle

$$T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$$

which does not include node T_{ex}. Therefore, the system is in a deadlock state, and an appropriate recovery scheme must be invoked.

Note that the outcome would be the same if site S_2 discovered the cycle first in its local wait-for graph, and sent the deadlock-detection message to site S_1. In the worst case, both sites discover the cycle at about the same time, and two deadlock-detection messages are sent: one by S_1 to S_2, and another by S_2 to S_1. This unnecessary message transfer adds to overhead in updating the two local wait-for graphs and searching for cycles in both graphs.

To reduce message traffic, we assign to each transaction T_i a unique identifier, which we denote by $ID(T_i)$. When site S_k discovers that its local wait-for graph contains a cycle involving node T_{ex} of the form

$$T_{ex} \rightarrow T_{K_1} \rightarrow T_{K_2} \rightarrow \ldots \rightarrow T_{K_n} \rightarrow T_{ex}$$

it will send a deadlock-detection message to another site only if

$$ID(T_{K_n}) < ID(T_{K_1})$$

Otherwise, site S_k continues its normal execution, leaving the burden of initiating the deadlock-detection algorithm to some other site.

Consider again the wait-for graphs maintained at sites S_1 and S_2 in Figure 18.13. Suppose that

$$ID(T_1) < ID(T_2) < ID(T_3) < ID(T_4)$$

Assume that both sites discover these local cycles at about the same time. The cycle in site S_1 is of the form

$$T_{ex} \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ex}$$

Since $ID(T_3) > ID(T_2)$, site S_1 does not send a deadlock-detection message to site S_2.

The cycle in site S_2 is of the form

$$T_{ex} \rightarrow T_3 \rightarrow T_4 \rightarrow T_2 \rightarrow T_{ex}$$

Since $ID(T_2) < ID(T_3)$, site S_2 does send a deadlock-detection message to site S_1. On receiving the message, S_1 updates its local wait-for graph, searches for a cycle in the graph, and discovers that the system is in a deadlock state.

18.9 Multidatabase Systems

In recent years, new database applications have been developed that require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software environments. Manipulation of information located in a heterogeneous database requires an additional software layer on top of existing database systems. This software layer is called a *multidatabase system*. The local database systems may employ different logical models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms. A multidatabase system creates the illusion of logical database integration without requiring physical database integration.

Full integration of existing systems into a *homogeneous* distributed database–the kind of distributed environment we have considered up to this point–is often difficult or impossible:

1. **Technical difficulties.** The investment in application programs based on existing database systems may be huge, and the cost of converting these applications may be prohibitive.
2. **Organizational difficulties.** Even if integration is *technically* possible, it may not be *politically* possible, due to the existing database systems belonging to different corporations or organizations.

In the second case, it is important for a multidatabase system to allow the local database systems to retain a high degree of *autonomy* over the local database and transactions running against that data.

For these reasons, multidatabase systems offer significant advantages that outweigh their overhead. In this section, we provide an overview of the challenges faced in constructing a multidatabase environment from the standpoints of data definition and transaction management.

18.9.1 Unified View of Data

Each local DBMS may use a different data model. That is, some may employ modern data models, such as the relational model, whereas others may employ older data models, such as the network model (see Appendix A) or the hierarchical model (see Appendix B).

Since the multidatabase system is supposed to provide the illusion of a single, integrated database system, a common data model must be used. The natural choice is the relational model, with SQL as the common query language. Indeed, there are several systems available today that allow SQL queries to a nonrelational DBMS.

Another difficulty is the provision of a common conceptual schema. Each local DBMS provides its own conceptual schema. The multidatabase system must integrate these separate schemas into one common schema. Schema integration is a complicated task, mainly due to the semantic heterogeneity.

Schema integration is not simply straightforward translation between data-definition languages. The same attribute names may appear in different local DBMSs but with different meanings. The data types used in one system may not be supported by other systems, and translation between types may not be simple. Even for identical data types, problems may arise due to the physical representation of data. One system may use ASCII, another EBCDIC. Floating-point representations may differ. Integers may be represented in *big-endian* or *little-endian* form. At the semantic level, an integer value for length may be inches in one system and millimeters in another, thus creating an awkward situation in which equality of integers is only an approximate notion (as is always the case for floating-point numbers). The same name may appear in different languages in different systems. For example, a system based in the United States may refer to the city "Cologne," whereas one in Germany refers to it as "Köln".

All these seemingly minor distinctions must be properly recorded in the common global conceptual schema. Translation functions must be provided. Indices must be annotated for system-dependent behavior (for example, the sort order of nonalphanumeric characters is not the same in ASCII as in EBCDIC). As we noted earlier, the alternative of converting each database to a common format may not be feasible without obsoleting existing application programs.

Under these circumstances, query processing is complex. Query optimization at the global level is difficult. The usual solution is to rely on only local-level optimization.

18.9.2 Transaction Management

A multidatabase system supports two types of transactions:

1. **Local transactions**. These transactions are executed by each local DBMS, outside of the multidatabase system's control.
2. **Global transactions**. These transactions are executed under the multidatabase system's control.

The multidatabase system is aware of the fact that local transactions may run at the local sites, but it is not aware of what specific transactions are being executed, or of what data they may access.

Ensuring the local autonomy of each DBMS requires making no changes to the local DBMS software. A DBMS at one site thus is not able to communicate directly with a DBMS at any other sites to synchronize the execution of a global transaction active at several sites.

Since the multidatabase system has no control over the execution of local transactions, each local DBMS must use a concurrency-control scheme (for example, two-phase locking or timestamping) to ensure that its schedule is serializable. In addition, in case of locking, the local DBMS must be able to guard against the possibility of local deadlocks.

The guarantee of local serializability is not sufficient to ensure global serializability. As an illustration, consider two global transactions T_1 and T_2, each of which accesses and updates two data items, A and B, located at sites S_1 and S_2, respectively. Suppose that the local schedules are serializable. It is still possible to have a situation where, at site S_1, T_2 follows T_1, whereas, at S_2, T_1 follows T_2, resulting in a nonserializable global schedule. Indeed, even if there is no concurrency among global transactions (that is, a global transaction is submitted only after the previous one commits or aborts), local serializability is not sufficient to ensure global serializability (see Exercise 18.20).

There are many protocols for ensuring consistency despite concurrent execution of global and local transactions in multidatabase systems. Some are based on imposing sufficient conditions to ensure global serializability. Others ensure only a form of consistency weaker than serializability, but achieve this consistency by less restrictive means. We consider one of the latter schemes: *two-level serializability*. Section 20.4 describes further approaches to consistency without serializability; other approaches are cited in the bibliographic notes.

18.9.2.1 Two-Level Serializability

Two-level serializability (2LSR) ensures serializability at two levels of the system:

- Each local DBMS ensures local serializability among its local transactions, including those that are part of a global transaction.
- The multidatabase system ensures serializability among the global transactions alone — *ignoring the orderings induced by local transactions*.

Each of these serializability levels is simple to enforce. Local systems already offer guarantees of serializability; thus, the first requirement is easy to achieve. The second requirement applies to only a projection of the global schedule in which local transactions do not appear. Thus, the MBDS can ensure the second requirement using standard concurrency-control techniques (the precise choice of technique does not matter).

The two requirements of 2LSR are not sufficient to ensure global serializability. However, under the 2LSR-based approach, we adopt a requirement weaker

than serializability, called *strong correctness*:

1. Preservation of consistency as specified by a set of consistency constraints
2. Guarantee that the set of data items read by each transaction is consistent

It can be shown that certain restrictions on transaction behavior, combined with 2LSR, are sufficient to ensure strong correctness (although not necessarily to ensure serializability). We list several of these restrictions.

In each of the protocols, we distinguish between *local* and *global* data. Local data items belong to a particular site and are under the sole control of that site. Note that there cannot be any consistency constraints between local data items at distinct sites. Global data items belong to the multidatabase system, and, though they may be stored at a local site, are under the control of the multidatabase system.

The *global-read protocol* allows global transactions to read, but not to update, local data items, while disallowing all access to global data by local transactions. The global-read protocol ensures strong correctness if all the following hold:

1. Local transactions access only local data items.
2. Global transactions may access global data items, and may read local data items (although they must not write local data items).
3. There are no consistency constraints between local and global data items.

The *local-read protocol* grants local transactions read access to global data, but disallows all access to local data by global transactions. In this protocol, we need to introduce the notion of a *value dependency*. A transaction has a value dependency if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.

The local-read protocol ensures strong correctness if all the following hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).
2. Global transactions access only global data items.
3. No transaction may have a value dependency.

The *global-read–write/local-read protocol* is the most generous in terms of data access of the protocols that we have considered. It allows global transactions to read and write local data, and allows local transactions to read global data. However, it imposes both the value-dependency condition of the local-read protocol, and the condition from the global-read protocol that there be no consistency constraints between local and global data.

The global-read–write/local-read protocol ensures strong correctness if all of the following hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).

2. Global transactions may access global data items as well as local data items (that is, they may read and write all data).
3. There are no consistency constraints between local and global data items.
4. No transaction may have a value dependency.

18.9.2.2 Ensuring Global Serializability

Early multidatabase systems restricted global transactions to be read only. They thus avoided the possibility of global transactions introducing inconsistency to the data, but were not sufficiently restrictive to ensure global serializability. We leave it to the reader to demonstrate that it is indeed possible to get such global schedules, and to develop a scheme to ensure global serializability (Exercise 18.21).

There are a number of general schemes to ensure global serializability in an environment where update as well read-only transactions can execute. Several of these schemes are based on the idea of a *ticket*. A special data item called a ticket is created in each local DBMS. Every global transaction that accesses data at a site must write the ticket at that site. This requirement ensures that global transactions conflict directly at every site they visit. Furthermore, the global transaction manager can control the order in which global transactions are serialized, by controlling the order in which the tickets are accessed. References to such schemes appear in the bibliographic notes.

If we want to ensure global serializability in an environment where no direct local conflicts are generated in each site, some assumptions must be made about the schedules allowed by the local DBMSs. For example, if the local schedules are such that the commit order and serialization order are always identical, we can ensure serializability by controlling only the order in which transactions commit.

The problem with schemes that ensure global serializability is that they may restrict concurrency unduly. They are particularly likely to do so because most transactions submit SQL statements to the underlying DBMS, rather than submitting individual **read**, **write**, **commit**, and **abort** steps. Although it is still possible to ensure global serializability under this assumption, the level of concurrency may be such that other schemes, such as the two-level serializability technique discussed previously, are attractive alternatives.

18.10 Summary

A distributed database system consists of a collection of sites, each of which maintains a local database system. Each site is able to process local transactions: those transactions that access data in only that single site. In addition, a site may participate in the execution of global transactions: those transactions that access data in several sites. The execution of global transactions requires communication among the sites.

There are several reasons for building distributed database systems, including sharing of data, reliability and availability, and speedup of query processing. However, along with these advantages come several disadvantages, including higher software-development cost, greater potential for bugs, and increased processing

overhead. The primary disadvantage of distributed database systems is the added complexity required to ensure proper coordination among the sites.

There are several issues involved in storing a relation in the distributed database, including replication and fragmentation. It is essential that the system minimize the degree to which a user needs to be aware of how a relation is stored.

A distributed system may suffer from the same types of failure that can afflict a centralized system. There are, however, additional failures with which we need to deal in a distributed environment, including the failure of a site, the failure of a link, loss of a message, and network partition. Each of these problems needs to be considered in the design of a distributed recovery scheme. If the system is to be robust, therefore, it must detect any of these failures, reconfigure itself such that computation may continue, and recover when a processor or a link is repaired.

If we are to ensure atomicity, all the sites in which a transaction T executed must agree on the final outcome of the execution. T either commits at all sites or aborts at all sites. To ensure this property, the transaction coordinator of T must execute a commit protocol. The most widely used commit protocol is the two-phase commit protocol.

The two-phase commit protocol may lead to blocking: a situation in which the fate of a transaction cannot be determined until a failed site (the coordinator) recovers. To avoid blocking, we can use the three-phase commit protocol.

The various concurrency-control schemes that can be used in a centralized system can be modified for use in a distributed environment. In the case of locking protocols, the only change that needs to be incorporated is in the way that the lock manager is implemented. There is a variety of different approaches here. One or more central coordinators may be used. If, instead, a distributed approach is taken, replicated data must be treated specially. Protocols for handling replicated data include the majority, biased, and primary-copy protocols. In the case of timestamping and validation schemes, the only needed change is to develop a mechanism for generating unique global timestamps. We can develop one by either concatenating a local timestamp with the site identification or advancing local clocks whenever a message arrives with a larger timestamp.

The primary method for dealing with deadlocks in a distributed environment is deadlock detection. The main problem is deciding how to maintain the wait-for graph. Different methods for organizing the wait-for graph include a centralized approach, a hierarchical approach, and a fully distributed approach.

Some of the distributed algorithms require the use of a coordinator. If the coordinator fails owing to the failure of the site at which it resides, the system can continue execution only by restarting a new copy of the coordinator on some other site. It can do so by maintaining a backup to the coordinator that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called election algorithms.

A multidatabase system provides an environment in which new database applications can access data from a variety of preexisting databases located in various heterogeneous hardware and software environments. The local database systems

may employ different logical models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms. A multidatabase system creates the illusion of logical database integration, without requiring physical database integration.

Exercises

18.1. Discuss the relative advantages of centralized and distributed databases.

18.2. Explain how the following differ: fragmentation transparency, replication transparency, and location transparency.

18.3. How might a distributed database designed for a local-area network differ from one designed for a wide-area network?

18.4. When is it useful to have replication or fragmentation of data? Explain your answer.

18.5. Explain the notions of transparency and autonomy. Why are these notions desirable from a human-factors standpoint?

18.6. Consider a relation that is fragmented horizontally by *plant-number*:

employee (*name*, *address*, *salary*, *plant-number*)

Assume that each fragment has two replicas: one stored at the New York site, and one stored locally at the plant site. Describe a good processing strategy for the following queries entered at the San Jose site.

(*a*) Find all employees at the Boca plant.
(*b*) Find the average salary of all employees.
(*c*) Find the highest-paid employee at each of the following sites: Toronto, Edmonton, Vancouver, Montreal.
(*d*) Find the lowest-paid employee in the company.

18.7. Consider the relations

employee (*name*, *address*, *salary*, *plant-number*)
machine (*machine-number*, *type*, *plant-number*)

Assume that the *employee* relation is fragmented horizontally by *plant-number*, and that each fragment is stored locally at its corresponding plant site. Assume that the *machine* relation is stored in its entirety at the Armonk site. Describe a good strategy for processing each of the following queries.

(*a*) Find all employees at the plant that contains machine number 1130.
(*b*) Find all employees at plants that contain machines whose type is "milling machine."
(*c*) Find all machines at the Almaden plant.
(*d*) Find employee $\bowtie$ machine.

18.8. For each of the strategies of Exercise 18.7, state how your choice of a strategy depends on:

(*a*) The site at which the query was entered
(*b*) The site at which the result is desired

18.9. Compute $r \ltimes s$ for the following relations:

r

A	B	C
1	2	3
4	5	6
1	2	4
5	3	2
8	9	7

s

C	D	E
3	4	5
3	6	8
2	3	2
1	4	1
1	2	3

18.10. Is $r_i \ltimes r_j$ necessarily equal to $r_j \ltimes r_i$? Under what conditions does $r_i \ltimes r_j = r_j \ltimes r_i$ hold?

18.11. To build a robust distributed system, you must know what kinds of failures can occur.

(*a*) List possible types of failure in a distributed system.
(*b*) Which items in your list from part a are also applicable to a centralized system?

18.12. Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Exercise 18.11a, explain how 2PC ensures transaction atomicity despite the failure.

18.13. Repeat Exercise 18.12 for 3PC.

18.14. List those types of failure that 3PC cannot handle. Describe how failures of these types could be handled by lower-level protocols.

18.15. Consider a distributed deadlock-detection algorithm in which the sites are organized in a hierarchy. Each site checks for deadlocks local to the site, and for global deadlocks that involve descendant sites in the hierarchy. Give a detailed description of this algorithm. Argue that the algorithm detects all deadlocks. Compare the relative merits of this hierarchical scheme with those of the centralized scheme and of the fully distributed scheme.

18.16. Consider a distributed system with two sites, *A* and *B*. Can site *A* distinguish among the following?

- *B* goes down.
- The link between *A* and *B* goes down.
- *B* is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

18.17. If we apply a distributed version of the multiple-granularity protocol of Chapter 14 to a distributed database, the site responsible for the root of the DAG may become a bottleneck. Suppose we modify that protocol as follows:

- Only intention-mode locks are allowed on the root.
- All transactions are given all possible intention-mode locks on the root automatically.

Show that these modifications alleviate this problem without allowing any nonserializable schedules.

18.18. Discuss the advantages and disadvantages of the two methods that we presented in Section 18.7.2 for generating globally unique timestamps.

18.19. Consider the following deadlock-detection algorithm. When transaction T_i, at site S_1, requests a resource from T_j, at site S_3, a request message with timestamp n is sent. The edge (T_i, T_j, n) is inserted in the local wait-for of S_1. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_3 only if T_j has received the request message and cannot immediately grant the requested resource. A request from T_i to T_j in the same site is handled in the usual manner; no timestamps are associated with the edge (T_i, T_j). A central coordinator invokes the detection algorithm by sending an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. Note that such a graph contains all the local information that the site has about the state of the real graph. The wait-for graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.

When the controller has received a reply from each site, it constructs a graph as follows:

- The graph contains a vertex for every transaction in the system.
- The graph has an edge (T_i, T_j) if and only if
  - There is an edge (T_i, T_j) in one of the wait-for graphs.
  - An edge (T_i, T_j, n) (for some n) appears in more than one wait-for graph.

Show that, if there is a cycle in the constructed graph, then the system is in a deadlock state, and that, if there is no cycle in the constructed graph, then the system was not in a deadlock state when the execution of the algorithm began.

18.20. Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

(*a*) Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.

(*b*) Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

18.21. Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read-only.

(*a*) Show by example that nonserializable executions may result in such a system.

(*b*) Show how you could use ticket scheme to ensure global serializability.

Bibliographic Notes

Computer networks are discussed in Tanenbaum [1996] and Halsall [1992]. A survey paper discussing major issues concerning distributed database systems has been written by Rothnie and Goodman [1977]. Textbook discussions are offered by Bray [1982], Date [1983], Ceri and Pelagatti [1984], and Ozsu and Valduriez [1991].

Distributed query processing is discussed in Wong [1977], Epstein et al. [1978], Hevner and Yao [1979], Epstein and Stonebraker [1980], Apers et al. [1983], Ceri and Pelagatti [1983], and Wong [1983]. Selinger and Adiba [1980], and Daniels et al. [1982], discuss the approach to distributed query processing taken by R* (a distributed version of System R). Mackert and Lohman [1986] provide a performance evaluation of query-processing algorithms in R*. The performance results also serve to validate the cost model used in the R* query optimizer. Theoretical results concerning semijoins are presented by Bernstein and Chiu [1981], Chiu and Ho [1980], Bernstein and Goodman [1981b], and Kambayashi et al. [1982].

The implementation of the transaction concept in a distributed database are presented by Gray [1981], Traiger et al. [1982], Spector and Schwarz [1983], and Eppinger et al. [1991]. The 2PC protocol was developed by Lampson and Sturgis [1976], and by Gray [1978]. The three-phase commit protocol is from Skeen [1981]. Mohan and Lindsay [1983] discuss two modified versions of 2PC, called *presume commit* and *presume abort*, that reduce the overhead of 2PC by defining default assumptions regarding the fate of transactions.

The bully algorithm presented in Section 18.6.2 is from Garcia-Molina [1982]. Distributed clock synchronization is discussed in Lamport [1978].

Papers covering distributed concurrency control are offered by Rosenkrantz et al. [1978], Bernstein et al. [1978, 1980a], Menasce et al. [1980], Bernstein and Goodman [1980a, 1981a, 1982], and Garcia-Molina and Wiederhold [1982]. The transaction manager of R* is described in Mohan et al. [1986].

Concurrency control for replicated data that is based on the concept of voting is presented by Gifford [1979] and Thomas [1979]. Validation techniques for distributed concurrency-control schemes are described by Schlageter [1981], Ceri and Owicki [1983], and Bassiouni [1988]. Discussions concerning semantic-based transaction-management techniques are offered by Garcia-Molina [1983], and by Kumar and Stonebraker [1988]. Recently, the problem of concurrent update to replicated data has re-emerged as an important research issue in the context of data warehouses. Problems in this environment are discussed in Gray et al. [1996].

Attar et al. [1984] discuss the use of transactions in distributed recovery in database systems with replicated data. A survey of techniques for recovery in distributed database systems is presented by Kohler [1981].

Distributed deadlock-detection algorithms are presented by Gray [1978], Rosenkrantz et al. [1978], Menasce and Muntz [1979], Gligor and Shattuck [1980], Chandy and Misra [1982], and Chandy et al. [1983]. Knapp [1987] surveys the distributed deadlock-detection literature. The algorithm presented in Section 18.8.2 comes from Obermark [1982]. Exercise 18.19 is from Stuart et al. [1984].

Transaction processing in multidatabase systems is discussed in Breitbart [1990], Breitbart et al. [1991, 1992], Soparkar et al. [1991], and Mehrotra et al. [1992a, 1992b]. The ticket scheme is presented in Georgakopoulos et al. [1994]. 2LSR is introduced in Mehrotra et al. [1991]. An earlier approach, called *quasi-serializability*, is presented in Du and Elmagarmid [1989].

CHAPTER 19

SPECIAL TOPICS

In earlier chapters, we covered the basic principles of database design and implementation. In this and subsequent chapters, we briefly cover a number of special topics in the area of database systems. In Chapter 20, we discuss advanced transaction processing schemes. In Chapter 21, we consider new applications of database systems, and the challenges that they pose to database system design. The bibliographic notes at the end of these chapters provide references for each topic; they can serve as a starting point for detailed study.

19.1 Security and Integrity

The data stored in the database need to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In Chapter 6, we saw how integrity constraints can be specified. In Chapter 7, we saw how databases can be designed to facilitate checking of integrity constraints. In Chapters 13, 14, and 15, we saw how to preserve integrity despite failures, crashes, and potential anomalies from concurrent processing. In Chapters 17 and 18, we saw how to preserve integrity in parallel and distributed systems. Until now, we have considered only how to prevent the accidental loss of data integrity. In this section, we examine the ways in which data may be misused or intentionally made inconsistent. We then present mechanisms to guard against such occurrences.

19.1.1 Security and Integrity Violations

Misuse of the database can be categorized as being either intentional (malicious) or accidental. Accidental loss of data consistency may result from

- Crashes during transaction processing