partitioning of relations is formally known as **fragmentation.** However, in the database literature, the term **disjoint fragmentation** is used to denote partitioning, and the term fragmentation refers to either disjoint or nondisjoint fragmentation.

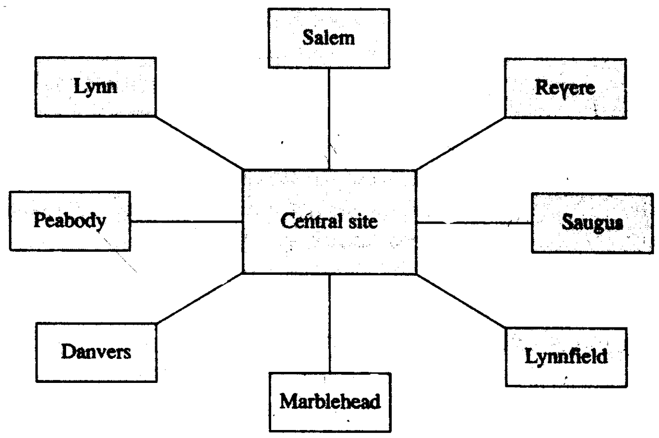
A distributed database system insulates the user from knowledge of data fragmentation. This characteristic of distributed database systems is called **fragmentation transparency.** However, from our discussions on networks and data distribution, we realize that it may not always be possible to access all the data when communication link and node failures occur. The user may sense that some data is unavailable and consequently realize that data is partitioned.

To achieve locality of reference and reduced communication and redundancy costs, data is often fragmented. Fragmentation allows a subset of the relation's attributes or the subset of the relation's tuples to be defined at a given site to satisfy local applications. The idea of data fragmentation is displayed in Figure 15.5 and examples are given in Examples 15.3 and 15.7 and Figure E.

**Example 15.3**

Consider the MUC library system shown in Figure B. It has a number of branches and maintains a central acquisition, cataloging, and distribution center. A central catalog contains the title and a detailed description of each item. However, each branch maintains a local catalog and has access to the central catalog, as well as catalogs at other branches. In a manual system, the index card for items are duplicated at the central site and sent to each branch where they are stored in their local catalogs. Access to the central catalog or the catalog of another branch can only be had by calling on these locations. An alternate solution to this problem would be to include in each index card a list of all the branches at which a copy is maintained, and have a copy of the entire catalog stored at all branches. In a computerized distributed database system, the catalog is fragmented and maintained in a database at each branch.

**Figure B** The MUC library system.



■

## 15.3.1 Fragmentation

A relation R defined on the scheme **R** can be broken down into the fragments $R_1$, $R_2, \ldots, R_n$ defined on the schemes $\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R_n}$ such that it is always possible to obtain R from the fragments $R_1, R_2, \ldots, R_n$. The fragmentation could be vertical, horizontal, or mixed, as described below.

### Vertical Fragmentation

**Vertical fragmentation** is the projection of the original relation on different sets of attributes. Relations may be fragmented by decomposing the scheme of **R**, such that

$$\mathbf{R} = \bigcup_{i=1}^{n} \mathbf{R_i}$$

and

$$R_i = \pi_{\mathbf{R}_i}(R), \text{ for } i = 1, 2, \ldots, n$$

The original relation R can be reconstructed by a join of the fragments:

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

It should be clear that for the original relation to be reconstructible, either:

1. For all fragments $R_i$ ($i = 1, 2, \ldots, n$), there must exist another fragment $R_j$ ($i \neq j, j = 1, 2, \ldots, n$), such that if we represent $R_i \cap R_j$ by X, then X is either a key of $R_i$ or of $R_j$; or
2. System-generated TIDs (tuple identifiers) of the original relation must be duplicated in all fragments.

Examples 15.4 and 15.5 illustrate these methods of deriving the original relation from its fragments.

**Example 15.4** Consider the relation EMPLOYEE(*Employee#, Name, Department, Degree, Phone#, Salary_Rate, Start_Date*). This relation can be partitioned into the vertical fragments EMPLOYEE_QUALIFICATION(*Employee#, Name, Degree, Phone#*) and EMPLOYEE_PAY(*Employee#, Name, Salary Rate, Start_Date*). The fragments are not disjoint because the *Employee#* and *Name* attributes are common in the fragments. If *Employee#* is a primary key of the original relation, we can derive the original relation by a natural join of these fragments, followed by the elimination of the duplicate *Name* attribute. ■

**Example 15.5** Consider the relation MODULE_USE given in Figure C. Two vertical fragments of this relation, MODULE and USES, include the system-supplied attribute TIDs and could be joined to derive the original relation.

**Figure C** TIDs in vertical fragmentation.

relation: MODULE_USE

| TID | MODULE | USES |
|---|---|---|
| t1 | Query processor | SORT |
| t2 | User interface | SORT |

fragment: MODULE

| TID | MODULE |
|---|---|
| t1 | Query processor |
| t2 | User interface |

fragment: USES

| TID | USES |
|---|---|
| t1 | SORT |
| t2 | SORT |

■

TIDs may be used by the DDBMS as a physical pointer and are not visible to users. If the TIDs are visible, a user may use them in some manner and this constrains the DDBMS from changing the TIDs, for instance, when the data is reorganized. As a result, data independence, a goal of database systems, is compromised.

Note that with fragmentation, duplicate tuples may in reality be part of distinct tuples of the unfragmented relation. Such duplicate tuples should not be deleted from a fragment or, alternatively, the TIDs of the deleted tuples should somehow be maintained to reconstruct the original tuples. For example, consider the relation and its fragments given in Figure C. It is obvious that if one of the tuples in the fragment USES is deleted, say the tuple with TID t2, a join with the fragment MODULE will not result in the original relation. The reconstructed relation would lack the fact that the SORT module is also used by the user interface module. If we include the TIDs in the fragmented relation, there is no possibility of duplicate tuples. The original relation can be obtained using a join on the TIDs.

## Horizontal Fragmentation

In **horizontal fragmentation** the tuples of a relation are assigned to different fragments, such that

$$R = \bigcup_{i=1}^{n} R_i$$

where each $R_i \doteqdot \sigma_{C_i}(R)$ each $C_i$ is some selection condition, and $\mathbf{R} = \mathbf{R_1} = \mathbf{R_2} = \ldots = \mathbf{R_n}$.

**Example 15.6** In Figure D we graphically show a relation that is fragmented into a number of disjoint horizontal fragments, which are replicated and stored at a number of sites. The original relation could be obtained by a union operation.

**Figure D** Replications of disjoint horizontal fragments.

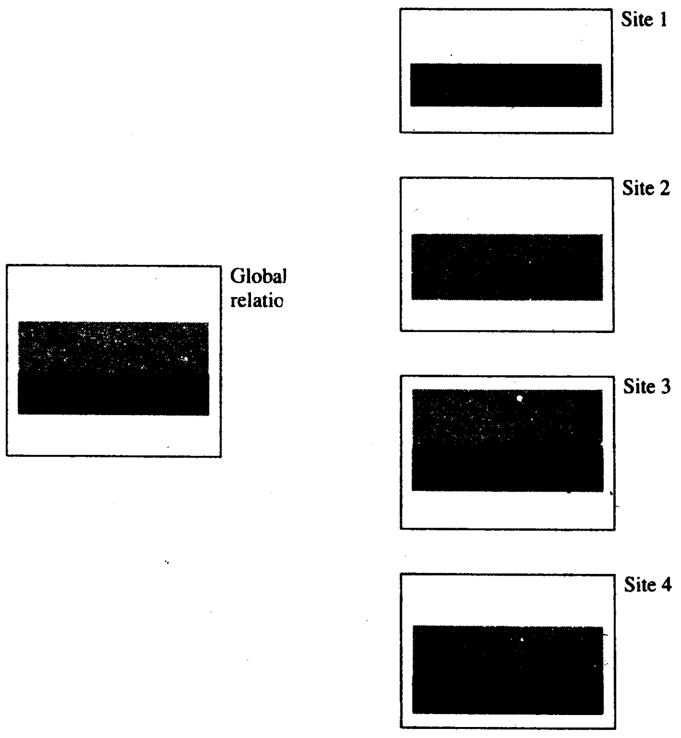


**Figure 15.6** Data fragmentation tree.

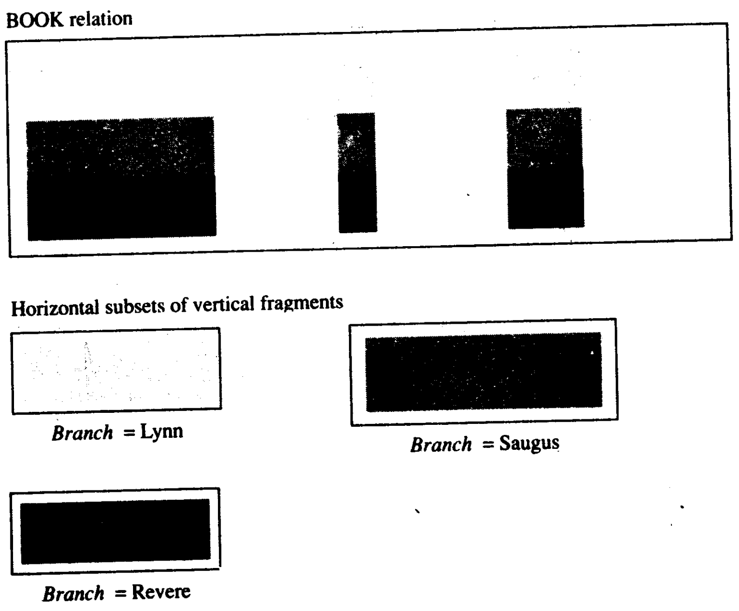
$$\begin{tikzcd}
 & & \text{Relation R} \arrow[dll, no head] \arrow[dl, no head] \arrow[dr, no head] \arrow[drr, no head] & & & \\
\pi_X(R) & \pi_Y(R) \arrow[dl, no head] \arrow[dr, no head] & & \sigma_{Z=z}(R) & \sigma_{X=x}(R) \arrow[dl, no head] \arrow[dr, no head] & \\
\sigma_{A=a}(\pi_Y(R)) & & \sigma_{B=b}(\pi_Y(R)) & \pi_W(\sigma_{X=x}(R)) & & \pi_V(\sigma_{X=x}(R))
\end{tikzcd}$$

## Mixed Fragmentation

Horizontal (or vertical) fragmentation of a relation, followed by further vertical (or horizontal) fragmentation of some of the fragments, is called **mixed fragmentation.** The original relation is obtained by a combination of join and union operations. Figure 15.6 illustrates a data fragmentation tree for a mixed fragmentation.

**Example 15.7** The BOOK relation in the library database can be made up of the following attributes. *Book#, Call#, Copy#, First_Author_Name, Title, Volume, Publisher, Place_of_Publication, Date, Binding, Size, Number_of_Pages, Date_Acquired, Branch,* and *Cost*. Note that the attribute *Book#* is unique

**Figure E** Mixed fragmentation.



and forms a primary key of the relation. The attributes *Call#* and *Copy#* also form a key of the relation. The portion of this relation of interest to the general public is limited to *Call#*, *Copy#*, *First_Author_Name*, *Title*, and *Branch;* it forms a vertical fragment of the BOOK relation. The collection at a given branch forms the horizontal subset of this vertical fragment. This is illustrated graphically in Figure E. ■

## Disjoint Fragmentation

In **disjoint vertical fragmentation** there are either no common attributes between any two vertical fragments or one fragment contains all the attributes of another, i.e., $R_i \cap R_j = \emptyset$ or $R_i$ for all i and j. In **disjoint horizontal fragmentation** there are either no common tuples in any two fragments or one fragment contains all the tuples contained in another fragment, i.e., $R_i \cap R_j = \{\ \}$ or $R_i$ for all i and j.

There is no partial overlap between the fragments. Replicas of a complete fragment are allowed in disjoint fragmentation. We point out that in disjoint vertical fragmentation with $\mathbf{R_i} \cap \mathbf{R_j} = \emptyset$, it is not possible to reconstruct the original relation unless each fragment contains the system generated TID.

## Nondisjoint Fragmentation

In **nondisjoint horizontal fragmentation,** a tuple may be assigned to more than one fragment. With **nondisjoint vertical fragmentation,** an attribute may be assigned to more than one fragment. This differs from replication. Replicate fragments are exact modification of attributes. In distributed databases, certain statistics pertaining to the characteristics of the data play an important role in determining access and query evaluation plans. These statistics, maintained in the system catalog, are likely to change regularly, entailing catalog updates. Some typical approaches to catalog distribution and maintenance problems are discussed below.

The R* system uses a distributed catalog. Local catalogs keep information on locally stored objects, including any fragments and replicates. The catalog at the **birth_site** of an object (site at which the object was first created) maintains the current storage sites of that object. Object movement causes this information to be updated. This scheme maintains complete site autonomy and is a type 1 scheme.

Distributed INGRES differentiates between local and global relations. Only global relations are accessible from all sites. A catalog of all global relations, the so-called **global catalog,** is maintained at all sites. The creation of a global relation requires its name and location to be broadcast to all sites. This is a type 2 scheme.

In the SDD-1 system, the catalog is a single relation that can be fragmented and replicated, allowing the entries to be distributed at data module sites. It is possible for local objects to have their catalog entries at a remote site. Consequently data definition operations may be nonlocal. This is a type 3 scheme. However, a fully replicated locator catalog is required at each site to keep track of the database catalog. A **locator catalog** contains information on the global scheme and details concerning fragmentation and replication.

Catalog details such as local-to-global name mappings, physical details concerning file organization and access methods, general and integrity constraint details, and database statistics could be stored locally. A site needing remote catalog information requests such information and stores it for later use. This scheme is called **caching the remote catalog.** It is not a replication of the remote catalog insofar as no attempt is made to maintain the consistency between the cached catalog and the remote one. The two are identical at the time of caching and this is indicated by both having identical version numbers. However, over time the remote catalog could be modified and its version number could change. This inconsistency is revealed when a query processed with a cached catalog is executed. At that time it is discovered that an out-of-date catalog has been used. This causes the query plan to be abandoned and the updated catalog to be transmitted to and cached at the site in question. The query is then reprocessed with the up-do-date remote catalog. SSD-1 and Distributed INGRES use this scheme of remote catalog caching.

## 15.4 Object Naming

In a distributed database system, we want to share data but we don't want too many restrictions on the user's choice of names. The system can adopt a **global naming** scheme such that all names are unique throughout the system. Two sites or users cannot use the same name for different data objects. This requirement for unique names can cause problems when a new site with an existing database is being integrated into the DDBMS. A unique name criterion would entail renaming objects in the database to be integrated as well as in the application programs that access them.

A drawback of the global name requirement is the loss of local autonomy, which allows users to choose appropriate local names even for global data items. Another deterrent is the bottleneck that would be created with the use of a single global name server, which has to be consulted for each name that is to be introduced in the database. The reliability of the system would also be compromised, since the entire system would be dependent on a single name server site for resolving naming conflicts.

For these reasons we stay away from a global naming scheme or the requirement that users choose systemwide unique names. Lifting such restrictions make it possible for different names to be used for the same data object, or the same name for different data objects. Although objects may not have unique names in the database, the DDBMS is required to differentiate between the objects.

Names used in queries or application programs are chosen by the end-users. To keep programming and query specification simple and invariant, regardless of the site from which they are executed, the network details must be transparent to the user. For instance, user A can enter the same query at site 1 or site 2 and anticipate the same results. Names selected by users have to be converted into system-unique names. This is done by consulting the local and/or the remote site catalog.

System R* maps end-user names (called **print names**) to internal systemwide names (SWNs). An SWN has the form:

creator@creator_site.object_name@birth_site

The birth_site is the site at which the object was first created, and because site names are chosen to be unique, an SWN is unique. An object X that was created in Washington by user John will have the SWN of:

John@Washington.X@Washington.

The same user could create, from Washington, an object named X at Montreal and this would receive the SWN of:

John@Washington.X@Montreal.

Note that the second data item is distinct from the first one. Also note that the user name is local; John@Washington is distinct from John@Montreal. In addition the name of an object includes its birth_site but this need not be its actual location. The data item could be moved to another site and be replicated at a number of sites.

To allow users to use print names, which are names of their choice for global data items, System R* creates these print names as synonyms for the corresponding SWNs. The synonyms are stored in the local catalog. The synonym-mapping scheme allows different print names for the same object and different objects having the same print names. The local catalog entry for an object includes its SWN, among other things. To find the catalog entry for an object, search the local catalog, followed by the birth_site catalog, then the site indicated by the birth_site catalog as currently holding the object.

Internal names can also be used to differentiate between fragments and replicates. If each fragment and replicate is assigned a number, these numbers can be concatenated with the name@birth_site to distinguish the different fragments or copies.

## 15.5 Distributed Query Processing

A query in a DDBMS may require data from more than one site. The transmission of this data entails communication costs. If some of the query operations can be executed at the site of the data, they may be performed in parallel. Section 15.5.1

**Figure F** Obtaining a join using a semijoin.

STUDENT

| *Std#* | *Std_Name* |
|---|---|
| 1234567 | Jim |
| 7654321 | Jane |
| 2345678 | San |
| 8765432 | Ram |
| 3920137 | John |
| 4729435 | Ron |
| 3927942 | Aron |
| 1934681 | Rodney |
| 8520183 | Maria |

Site 1

REGISTRATION

| *Std#* | *Course#* |
|---|---|
| 1234567 | COMP353 |
| 1234567 | COMP443 |
| 2345678 | COMP201 |
| 8765432 | COMP353 |
| 8765432 | COMP441 |
| 7654321 | COMP441 |

Site 2

X = $\pi_{Std\#}$ (REGISTRATION)

| *Std#* |
|---|
| 1234567 |
| 2345678 |
| 8765432 |
| 7654321 |

Y = STUDENT ⋉ REGISTRATION = STUDENT ⋈ X

| *Std#* | *Std_Name* |
|---|---|
| 1234567 | Jim |
| 7654321 | Jane |
| 2345678 | San |
| 8765432 | Ram |

STUDENT ⋈ REGISTRATION = Y ⋈ REGISTRATION

| *Std#* | *Course#* | *Std_Name* |
|---|---|---|
| 1234567 | COMP353 | Jim |
| 1234567 | COMP443 | Jim |
| 2345678 | COMP201 | San |
| 8765432 | COMP353 | Ram |
| 8765432 | COMP441 | Ram |
| 7654321 | COMP441 | Jane |

prepared, which involved joining the two relations. The join could be performed by first projecting REGISTRATION on *Std#* and transmitting the result, $\pi_{std\#}$ (REGISTRATION), to site 1. At site 1, we select those tuples of STUDENT that have the same value for the attribute *Std#* as a tuple in $\pi_{std\#}$ (REGISTRATION) by a join. The entire operation of first projecting the REGISTRATION and then performing this join is called a semijoin and denoted by ⋉. However, we do not obtain the desired result after the ⋉ operation. The semijoin operation reduces the number of tuples of STUDENT that have to be transmitted to site 2. The final result is obtained by a join of the reduced STUDENT relation and REGISTRATION. These steps are illustrated in Figure F. The class list can be obtained by sorting the resulting relation on *Course#*.

Note: It may be worthwhile to compute Y = STUDENT ⋉ REGISTRATION and Z = REGISTRATION ⋉ STUDENT and then obtain the final result by X ⋈ Z. ■

To reduce the communication cost in performing a join, the **semijoin** ($\ltimes$) operator has been introduced. Let P be the result of the semijoin:

$$P = R \ltimes S$$

Then P represents the set of tuples of R that join with some tuple(s) in S. P does not contain tuples of R that do not join with any tuple in S, thus P represents the reduced R that can be transmitted to a site of S for a join with it. If the join of R and S is highly selective, the size of P would only be a small proportion of the size of R. To get the join of R and S, we now join P with S, i.e.,

$$\begin{aligned} T &= P \bowtie S \\ &= (R \ltimes S) \bowtie S \\ &= (S \ltimes R) \bowtie R \\ &= (R \ltimes S) \bowtie (S \ltimes R) \end{aligned}$$

The semijoin is a reduction operator; R $\ltimes$ S can be read as R semijoin S or the reduction of R by S. Note that the semijoin operation is not associative. In Example 15.9, STUDENT $\ltimes$ REGISTRATION is not the same as REGISTRATION $\ltimes$ STUDENT. The former produces a reduction in the number of tuples of STUDENT; however, the latter is the same relation as REGISTRATION!

In distributed query processing, communication cost reduction is one of the objectives. The semijoin operation can be introduced to reduce the cardinality of large relations that are to be transmitted. Reduction in the number of tuples reduces the number and total size of the transmission and the total cost of communication.

It is wrong to assume that if $|R| > |S|$, then R should be reduced, as we shall see below.

To compute the join of R and S, we first compute the semijoin and then the join of one of the reduced relations with the other. The evaluation of the semijoin R $\ltimes$ S requires that we transmit $\pi_{R \cap S}(S)$ to the site of R. We do not need to transmit the whole of S. Let us refer to this projection of S and S′ and the size of the projected S as s′.

We use S′ to reduce R by computing R $\ltimes$ S′. Let us refer to the reduced R as R′ and the size of reduced R as r′. R′ is then transmitted to the site of S to compute the join (R′$\bowtie$ S). The communication cost incurred is:

$$2 * c_0 + c_1 * (s' + r')$$

Without the semijoin, we would have sent the whole of R to the site of S and the cost would have been:

$$c_0 + c_1 * |R| * R_{sz}$$

Therefore, the benefit of using the semijoin is:

$$c_1 * (|R| * R_{sz} - s' - r') - c_0$$

If the benefit is greater than zero, we prefer the semijoin over the traditional join.

The decision as to whether to reduce R or S can only be made after comparing the benefit of reducing R with that of reducing S. (We can also choose to reduce both.) We have already calculated the cost of reducing R; now let us do the same for S. As before, let us represent the size of $\pi_{R \cap S}(R)$ as r″ and the size of the reduced

As we discussed in Chapter 12, transactions are said to possess certain properties:

- **Consistency:** A transaction transforms a consistent database state into another consistent database state.
- **Atomicity:** All operations of the transaction are performed or none are performed.
- **Serializability:** If several transactions are executed concurrently, the result must be the same as if they were executed serially in some order.
- **Durability:** Once a transaction has been committed the results are guaranteed not to be lost.
- **Isolation:** An incomplete transaction cannot reveal its results.

These properties of a transaction are assured by using certain concurrency control and recovery techniques. Chapters 11 and 12 covered such techniques for centralized DBMSs. In the next two sections we briefly cover some techniques used in distributed DBMSs.

## 15.7 Concurrency Control

Concurrency control in a DDBMS has to take into account the existence of fragmentation and replication of data. Variations of the schemes used in centralized DBMSs are used in distributed concurrency control. A number of such schemes based on the locking and timestamp approaches are presented in this section.

Locking is the simplest concurrency control method. Locking enforces serial access to data. In centralized DBMSs, the lock requests go to a single lock manager, which can arbitrate any conflicts. In distributed systems, a centralized lock manager is not desirable due to the bottlenecks created at the central site. A centralized lock manager at a single site, furthermore, is vulnerable to failure, leading to the disruption of the entire system.

The locking scheme must be well formed. In other words, no transactions can access (read or write) a data item that it has not locked.

### 15.7.1 Distributed Locking

As discussed in Chapter 12, the different locking types can be applied to distributed locking. A centralized lock manager at a single site is relatively simple to implement. Here a transaction sends a message to the lock manager site requesting appropriate locks on specific data items. If the request for the locks could be granted immediately, the lock manager replies granting the request. If the request is incompatible with the current state of locking of the requested data items, the request is delayed. In the case of a read lock request, the data item from any site containing a copy of it, is locked in the share mode and then read. In the case of a write, all copies of the data items have to be modified and are locked in the exclusive mode. With a centralized lock manager, the detection of deadlock is straightforward, requiring the generation of a global wait-for graph (GWFG). The disadvantage of this scheme, in addition to the bottlenecks it creates, is the disruption of the entire system in case of the failure of the centralized lock manager site.

In the distributed method each lock manager is responsible for locking certain data items. The problem this scheme creates, however, is that of detection of deadlocks. Since lock requests are directed to a number of different sites, the nonexistence of a cycle in the **local wait-for graph** at each lock manager is not sufficient to conclude the absence of a deadlock. It is still necessary to generate a **global wait-for graph** to detect a deadlock.

Example 15.11 illustrates the type of locking required in a distributed system where data is fragmented as well as replicated.

**Example 15.11**

Consider transactions $T_1$ and $T_2$ of Figure G. Suppose the data is replicated and three copies of A are stored at sites $S_1$, $S_2$, and $S_3$. To execute these transactions, each spawns three local subtransactions, $T_{1S1}$, $T_{1S2}$, $T_{1S3}$, and $T_{2S1}$, $T_{2S2}$, $T_{2S3}$ to be executed at sites $S_1$, $S_2$, and $S_3$, respectively. A possible execution schedule for these transactions is given in Figure H. As we see from Figure H, the final result obtained is incorrect because the schedule is not serializable. If each subtransaction of $T_1$ had run to completion before those of transaction $T_2$, the values in each replicate of A would have been 200. If each subtransaction of $T_2$ had run to completion before those of

**Figure G** Two modifying transactions.

| Transaction $T_1$ | Transaction $T_2$ |
|---|---|
| **Lockx(A)** | **Lockx(A)** |
| A := 100 | A := 200 |
| **Write(A)** | **Write(A)** |
| **Unlock(A)** | **Unlock(A)** |

**Figure H** A schedule for the transactions in Figure G.

| | | site $S_1$ | | site $S_2$ | | site $S_3$ |
|---|---|---|---|---|---|---|
| Time | Transaction $T_{1S1}$ | Transaction $T_{2S1}$ | Transaction $T_{1S2}$ | Transaction $T_{2S2}$ | Transaction $T_{1S3}$ | Transaction $T_{2S3}$ |
| $t_1$ | **Lockx(A)** | | | **Lockx(A)** | **Lockx(A)** | |
| $t_2$ | A := 100 | | | A := 200 | A := 100 | |
| $t_3$ | **Write(A)** | | | **Write(A)** | **Write(A)** | |
| $t_4$ | **Unlock(A)** | | | **Unlock(A)** | **Unlock(A)** | |
| $t_5$ | | **Lockx(A)** | **Lockx(A)** | | | **Lockx(A)** |
| $t_6$ | | A := 100 | A := 200 | | | A := 100 |
| $t_7$ | | **Write(A)** | **Write(A)** | | | **Write(A)** |
| $t_8$ | | **Unlock(A)** | **Unlock(A)** | | | **Unlock(A)** |

**Figure J** A schedule for the transactions of Figure I.

| | site $S_1$ | | site $S_2$ | |
|---|---|---|---|---|
| Time | Transaction $T_{1S1}$ | Transaction $T_{2S1}$ | Transaction $T_{1S2}$ | Transaction $T_{2S2}$ |
| $t_1$ | **Lockx**(A) | | | **Lockx**(B) |
| $t_2$ | A := 100 | | | B := 2000 |
| $t_3$ | **Write**(A) | | | **Write**(B) |
| $t_4$ | **Unlock**(A) | | | **Unlock**(B) |
| $t_5$ | | **Lockx**(A) | **Lockx**(B) | |
| $t_6$ | | A := 200 | B := 1000 | |
| $t_7$ | | **Write**(A) | **Write**(B) | |
| $t_8$ | | **Unlock**(A) | **Unlock**(B) | |

respectively, to be executed at sites $S_1$ and $S_2$. A possible execution schedule for these transactions is given in Figure J. As we see from Figure J, the final result obtained is incorrect since the schedule is not serializable. If transaction $T_1$ had run to completion before transaction $T_2$, the values of A and B would have been 200 and 2000, respectively. Had transaction $T_2$ run to completion before transaction $T_1$, A and B would have the values 100 and 1000, respectively. ■

As in centralized two-phase locking, serializability requires that the locking in the distributed system also be two-phase. Recall that the two-phase locking scheme is required to have growing and shrinking phases. All lock requests made by a transaction or any of its subtransactions should be made in the growing phase and released in the shrinking phrase. Whenever a transaction issues an unlock instruction the shrinking phase starts indicating that all required locks are obtained. Where data is replicated, all subtransactions of a transaction that would modify the replicated data item would have to observe the two-phase locking protocol. Therefore, we cannot have one subtransaction release a lock and subsequently have another subtransaction request another lock. This requires that each subtransaction of a transaction notify all other subtransactions that it has acquired all its locks. The shrinking phase can start once all subtransactions have acquired all their locks.

In establishing the fact that all subtransactions have finished their growing phase, the number of messages involved is high. The possibility of failure in nodes and communication links and that of a rollback of some subtransactions in case of failure of others to complete normally indicates that the unlocking operations should be delayed until the distributed commit point of all subtransactions.

The distributed commit requires the exchange of a number of messages between the sites of subtransactions. It is done using a two-phase commit protocol discussed in Section 15.8.

## 15.7.2 Timestamp-Based Concurrency Control

Locking schemes suffer from two serious disadvantages: deadlock and low level of concurrency. Timestamp methods have been advocated as an alternative to locking.

The timestamp methods discussed in Chapter 12 can be extended to the distributed case. As in the case of the centralized timestamp methods, each copy of a data item in the distributed approach contains two timestamp values: the read timestamp and the write timestamp. Also, each transaction in the system is assigned a timestamp value that determines its serializability order. A transaction T with a timestamp value of t ensures that it does not read a value from the future (that is, the write timestamp of the data item must not be greater than value t) nor write a value that was already read by a younger transaction (i.e., the read timestamp of the data item must not be greater than value t). If the write timestamp of the data item to be read is greater than value t (written by a younger transaction) or if the read timestamp of the data item to be written is greater than value t (read by a younger transaction), transaction T must be aborted and restarted. If transaction T attempts to write a data item but finds that the read timestamp of the data item is less than t (an older transaction had read the value) and the write timestamp of the data item is greater than t (a younger transaction had already written a new value), transaction T is not required to be aborted. However, it does not update the data item (it was too slow to change the value of the data item). When more than one copy of a data item exists, a new value must be written in all of its copies. In this case, the two-phase commit protocol discussed in Section 15.8 must be used to make the new value permanent.

As in the centralized database system, a number of different timestamp-based schemes can be used. In these schemes a timestamp is used to associate some value with a transaction and give it an order in the set of all transactions being executed. In the serial execution of transactions, time plays an important role and timestamping seems to be the natural solution to the serializability problem.

If a system assigns a unique timestamp to a transaction, the timestamp identifies the transaction. The generation of timestamps in a centralized system requires the use of some monotonically increasing numbers. In distributed systems, each site generates a local timestamp and concatenates it with the site identifier. If the local timestamp is unique, its concatenation with the unique site identifier would make the (global) timestamp unique across the network. The site identifier must be the least significant digits of the timestamp so that the events can be ordered according to their occurrence and not their location, as illustrated in Example 15.13.

**Example 15.13** Let two events be assigned the timestamps 200100 and 100200, where the first three digits of the timestamp identify the site and the last three digits the time at which the event occurred. Now even though the event with timestamp 100200 occurred later than the event with timestamp 200100, the timestamp comparison states otherwise. ■

The local timestamp can be generated by some local clock or counter. In the event a counter is used, a relatively busy site would rapidly outrun slower sites. The local clocks at different sites can also get out of step. These local timestamp-generating schemes can be kept fairly well synchronized by including the timestamp in the messages sent between sites. On receiving a message, a site compares its clock or counter with the timestamp contained in the message. If it finds its clock or counter to be slower, it sets it to some value greater than the message timestamp. In this way, an inactive site's counter or a slower clock will become synchronized with the others at the first message interaction with another site.

readiness to commit or abort. In the **decision phase,** the decision as to whether all subtransactions should commit or abort is made and carried out. The transactions at a site interact with the transaction manager of the site, cooperating in the exchange of messages.

It is more convenient to use the process concept rather than the transaction concept in discussing the two-phase commit and deadlocks. Just like a transaction, a process is capable of requesting data items and releasing them. However, they have a better knowledge of their environment, including knowledge about the identity of the processes that are blocking them. The pseudocode for the processes at the participant and coordinator sites is given below. Note that part of the code belongs to the transaction manager (TM) and the remaining to the subtransactions or the coordinator.

The coordinator process starts by spawning a number, n, of subtransactions. Some of these would be at remote sites and others could be at the same site as the coordinator. The only difference is that a subtransaction at the same site does not have to communicate via the network. These subtransactions are run along with the respective TM as participant processes at a number of sites.

**Participant Process**

```
begin
acquire locks and make local changes
if normal end
  then status := okay to commit
  else status := should abort;
set timeout;
while (not request from coordinator for voting or not timeout)
    do {nothing};
if timeout
    then write recovery log, release all locks, and abort
if request from coordinator for voting
    then if status := should abort
        then begin
                send abort to coordinator
                write status on recovery log, release all locks, and abort
                end;
            else begin {status := okay to commit}
                send ready to commit, write status on recovery log
                set timeout
                while (not second_signal from coordinator or not
                timeout)
                        do nothing;
                if receive commit from coordinator
                  then write recovery log, commit,
                            release all locks, and send
                                acknowledge to coordinator
                if receive abort from coordinator
                  then write recovery log, release all locks,
                            abort, and send acknowledge to coordinator
                if timeout {blocked}
                  then begin
```

```
                    send SOS_second_signal and wait for response
                    activate recovery
                    end
               end
          end
```

**Coordinator Process**

```
  begin
    spawn n subtransactions
    write into recovery log request for voting, send to all
        subtransactions request for voting
    messages := 0;
    abortall := false;
    set timeout;
    while (messages ≠ n or not timeout or not abortall)
      do begin
           if receive ready to commit
             then messages := messages + 1;
           if receive abort
             then abortall := true;
           end;
    if timeout or abortall
      then begin
           second_signal := abort
           write global abort in log
           end
      else begin
           second_signal := commit
           write global commit in log
           end;
    send second_signal to all subtransactions;
    set timeout;
    acknowledge := 0;
    while (acknowledge ≠ n or not timeout)
      do begin
           if receive acknowledge from participant
             then acknowledge := acknowledge + 1;
           end;
      if timeout and acknowledge ≠ n
        then spawn SOS (second_signal) response process
        else write transaction complete in log
      end;
```

When the participant processes execute, they know whether the tasks assigned to them were completed successfully or not. If successful, they are willing to commit, otherwise they have to abort. Recall that the assigned database update is done only on a copy of the data items in each process's own workspace. These participant processes wait for a voting request from the coordinator process. If such a request is not received by a participant process, after a predetermined time period (timeout) it aborts after writing an appropriate recovery log. No changes are made to any data items in the database. If a request for voting is received before timeout, the participant process sends the appropriate status signal (okay to commit or should abort) to the coordinator process. On receipt of okay to commit signals from all the participant processes, the coordinator process sends a second signal to the participants to commit. On receipt of this commit signal, the participant process writes appropriate recovery log and commit markers onto stable storage at its site. Following this, it makes the changes permanent in the database.

If a participant process does not receive the second signal within a predetermined time period, it is said to be blocked. This could happen if the site goes down and then the recovery operation restores it and finds that the second signal was not received before the crash. A blocked participant process sends out an SOS signal, which is responded to by an SOS process. Such an SOS process could have been spawned by the coordinator process to help in the recovery of any site that failed after the vote was taken to commit or abort, but before the site could actually commit or abort. The SOS signal would also be emitted by a participant process if it did not receive the second signal (to commit or abort) from the coordinator, the signal being lost in the network.

A participant process that does not receive a request from the coordinator process for voting within a predefined time period will timeout. Timeouts result in the participant process having to write the recovery log, release all locks, and abort. In case the request for a voting message from the coordinator was lost, the coordinator would not receive any signal from such aborted participant processes. The coordinator process timeouts and therefore aborts all the other participant processes.

## 15.8.2 Recovery with Two-Phase Commit

The recovery log, in addition to the type of information indicated in the centralized case, includes the log of the messages transmitted between sites. Such a record would enable the recovery system to decide, when the site is reconnected to the network, on the extent of the site's interaction with the rest of the system. The recovery system would also be able to determine the fate of the subtransactions running at the site. It can then determine which subtransactions were committed, aborted, or blocked. Regarding the committed subtransactions, the recovery system would ensure that the changes are reflected in the database at the site. In the case of aborted transactions, any partial updates would be undone. As for the transactions that were blocked, an SOS signal would be sent to determine whether it should be committed or aborted.

Communication link failures in certain cases can result in the database system becoming partitioned. Each of the partitioned systems could operate by marking the sites in the other partitions as being down. A moment's thought should tell us that there may be no possibility of a smooth recovery from such a partitioning. In this case, the complete system has to be restarted from the period before the partitioning occurred with a manual assist to recover subsequent database modifications.

### Site Recovery

When a failed site resumes operation, it consults the recovery log to find the transactions that were active at the time of the failure. For strictly local transactions, recovery actions similar to a centralized database requiring a simple undo or redo would be called for. Global transactions would be of two types: coordinator or participant.

Regarding all participant type transactions, if the log indicated that it had not sent the status message to the coordinator, then the latter would have aborted all subtransactions. The recovery operation would ensure that such participant transactions be aborted and no changes be reflected by such transactions in the database. Suppose the log for a participant type transaction indicates that it send an okay to commit status to the coordinator. This means that the global transaction could have been either committed or aborted. The recovery operation would ensure that the participant transaction, on restart, would send a SOS message to learn its fate from the SOS process. Once it receives the signal either to commit or abort, the recovery process performs a redo or undo operation. In the case of a participant for which the log indicates the receipt of a second signal from the coordinator (to commit or abort), the recovery process can take appropriate action and ensure that an acknowledge signal be sent to the coordinator.

For a coordinating transaction at the failed site, the recovery process examines the log to determine its status. If no request for a voting message was sent before the site failure, all participants would have aborted, whereupon the coordinating transaction can be aborted as well. If the coordinator sent a request for voting before the crash, the recovery process must retransmit this request for voting. Even though the pseudocode of the participant processes given above does not indicate this, they should treat the second request for voting as the first and proceed as if this were the first request for voting. The global transaction can then be completed as if nothing had happened. If the site failed after the coordinator sent the second signal for commit or abort, the recovery process would entail resending this signal. Participant sites that received this signal and acted accordingly would treat this as a repeat message, ensure that appropriate actions were taken (from their recovery logs), and send the required acknowledge signal. Participants that did not receive this second signal would be blocked and attempt to recover via SOS. The coordinator would not receive acknowledgement from these participants and therefore would spawn the SOS process, which would respond to these SOS signals and conclude the global transaction.

If the site failed after the coordinator wrote a complete transaction marker in the log, no further actions would be called for.

### Lost Message

The type of recovery operation to be performed depends on the message that was lost. If the request to vote from the coordinator is lost, the participant would abort, which would eventually lead to the abortion of the global transaction. If the status message from any one of the participants is lost, the coordinator would timeout and abort the global transaction, including all the participant transactions. Should the second signal be lost, a participant would timeout and attempt a recovery via the SOS message. In the event that one of the acknowledge messages is lost, the coordinator would spawn the SOS response process. The coordinator would not know if the transaction is complete. An alternative approach is to have the coordinator send a request to the participants to retransmit the acknowledgements.

### Communication Link Failure

Suppose the failure of the communication link occurs in such a way that a subset of the participant sites are partitioned without a coordinator. In this case, as far as the coordinator is concerned, this is equivalent to the failure of a number of participant sites. If the failure occurs before the partitioned participants were sent the voting message, the coordinator would have aborted the global transaction, including all nonpartitioned subtransactions. The partitioned participants would also abort after a timeout. If the failure occurs after the participants have reported their status, the coordinator would have decided either to commit or abort. The partitioned sites could recover, on reconnection, by sending an SOS.

## 15.9 Deadlocks in Distributed Systems

As in the case of a centralized system, deadlocks can occur in a distributed system, as illustrated in Example 15.15.

**Example 15.15** Consider the transactions of Figure L, where data item A is resident at site $S_1$ and data item B is resident at site $S_2$. The schedule for the execution of the transactions is given in Figure M. The transactions are using two-phase

**Figure L** Two modifying transactions.

| Transaction $T_1$ | Transaction $T_2$ |
|---|---|
| **Lockx**(A) | **Lockx**(B) |
| **Read**(A) | **Read**(B) |
| A := A − 100 | B := B * 1.1 |
| **Write**(A) | **Write**(B) |
| **Lockx**(B) | **Lockx**(A) |
| **Read**(B) | **Read**(A) |
| B := B + 100 | A := A * 1.1 |
| **Write**(B) | **Write**(A) |
| **Unlock**(A) | **Unlock**(B) |
| **Unlock**(B) | **Unlock**(A) |

**Figure M** A schedule for the transactions in Figure L.

| | site $S_1$ | | site $S_2$ | |
|---|---|---|---|---|
| Step | Transaction $T_{1S1}$ | Transaction $T_{2S1}$ | Transaction $T_{2S2}$ | Transaction $T_{1S2}$ |
| $s_1$ | **Lockx**(A) | | **Lockx**(B) | |
| $s_2$ | **Read**(A) | | **Read**(B) | |
| $s_3$ | A := A − 100 | | B := B * 1.1 | |
| $s_4$ | **Write**(A) | | **Write**(B) | |
| $s_5$ | | **Lockx**(A) | | **Lockx**(B) |