PART VII

ADVANCED DATABASE TOPICS

CHAPTER 18

DISTRIBUTED DATABASE SYSTEMS

CHAPTER OBJECTIVES

- Understand the fundamental principles of distributed database systems
- Appreciate the motivation and goals for distributing data
- Examine the various types, configurations, and methods for data distribution
- Note how the client/server architecture enables distributed database systems
- Study the features and functions of a distributed database management system (DDBMS)
- Explore the principle of transparency and grasp its significance
- Learn the implications of query and transaction processing

Refer back to the evolution of database systems in the early 1970s from file-oriented data systems. Each application had its own set of files; each group of users accessed the files pertinent to their applications. As business conditions changed and the demand for information sharing escalated, file-oriented systems became inadequate. Each user group could not work in isolation. For the overall benefit of the organization, all user groups needed to share information and collaborate with one another to achieve the corporate business goals. Decentralization by applications in file-oriented systems no longer served useful purposes. A centralized database system emerged, and users could share information from the integrated database and work together. By the late 1980s, many organizations possessed huge, centralized databases. Tremendous growth in computing technology made this transition possible.

Database Design and Development: An Essential Guide for IT Professionals by Paulraj Ponniah
ISBN 0-471-21877-4

What has happened in the business scene since then? More and more corporations have become global in their operations. Business opportunities outside the local areas have provided growth and profitability. Think of the local bank where you opened a checking account about 10 years ago. To survive and be profitable, the bank has probably extended into a few more states and perhaps has even become international. Let us say that your progressive bank has become global and has opened branches in London, Paris, Tokyo, and Hong Kong in addition to a large number of domestic offices. The Paris branch primarily serves French customers. The users in the Paris branch access the data from the bank's database. But most of the data they need is about the local customers. Does it make sense for the Paris branch to go all the way to the centralized database stored in New York to access data about French customers?

All organizations expanding through mergers, acquisitions, and consolidations because of competitive pressures face similar questions. Each local office seeks autonomy, data ownership, and control for better service to its customers. The trend is to distribute the corporate data across the various dispersed locations so that each location can efficiently access the data it needs for running its portion of the overall business. This pattern of decentralization is not the same type of decentralization that existed in the case of the old file-oriented system. At that time, the decentralization based on applications perpetuated dispersal of redundant data with no information sharing. What you observe now is different. Database is being decentralized with geographic separation of data, but logical unity and integrity are preserved for proper information sharing—one logical database, but data not stored in just one location.

How is this becoming possible? Distributed database systems are emerging as a result of the merger between two major technologies: database and network. Recent advances in data communications and standardization of protocols such as Ethernet, TCP/IP, and ATM, along with the ubiquity of the Internet, promote physical distribution of corporate data across global locations yet preserving the integration and logical unity.

Database vendors have not yet come up with "pure" distributed database systems as envisioned in research studies. The technology is slowly maturing. Vendors are developing systems based on the client/server approach and the concept of a collection of active heterogeneous, geographically dispersed systems. But the momentum is strong and sustained.

FUNDAMENTAL PRINCIPLES

How is the push for decentralization of corporate data to be accomplished? Remember, the storing of relevant portions of the data has to be at each of the various geographic locations; but at the same time, users at each location must be able share data with users at any and all of the locations. Physical separation but logical integration—that is the underlying premise.

Examine this premise. What this implies is that when a user executes a query or a transaction accessing some data from the database, the user must be able to do so without knowing where the data resides and how it is going to be retrieved. The distribution of the data across the locations must be transparent to the user. The

user need not be concerned while coding his or her query. This distributed data independence is, in fact, a natural extension of the principles of logical and physical data independence.

Pursue further the consideration of a transaction executed from a certain location. The transaction may operate on pieces of data stored at different locations. If the transaction completes successfully and commits, then all changes to data by the transaction must persist; if the transaction aborts, none of the changes must remain in the database. In other words, the atomicity of the transaction in the distributed environment must be ensured as if it executed in a centralized database system.

Let us define a distributed database system. We will explore a few fundamental concepts and then review the goals of distributed database systems. That will lead into a review of the advantages and disadvantages.

What is a Distributed Database?

Let us begin with a standard definition:

A distributed database

is a related collection of shared data of an organization along with the description of that data, logically integrated and interrelated, by physically distributed across locations over a computer network.

Figure 18-1 shows an illustration of a distributed database.

A distributed computing system consists of a number of processing units interconnected by a communications network. These processing units need not be homogeneous, but they have to cooperate with one another in processing queries and

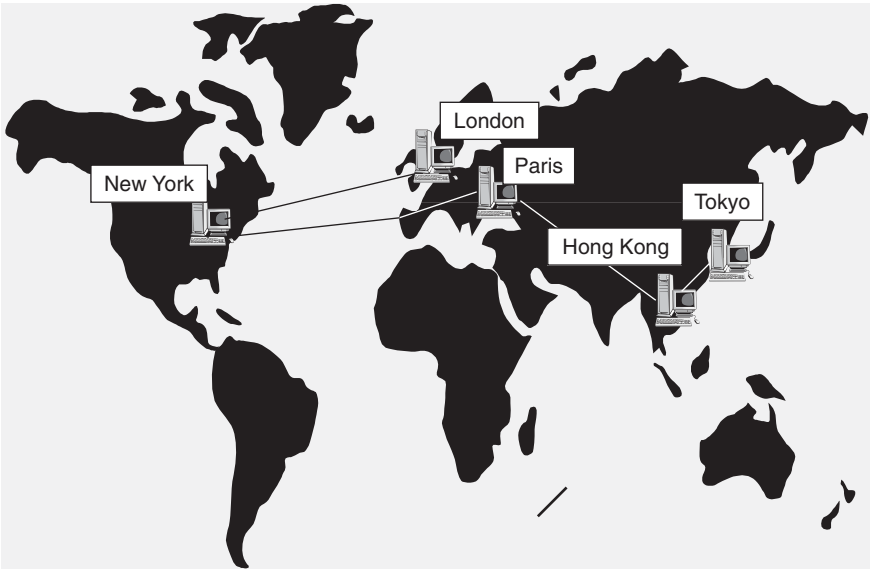


Figure 18-1 Distributed database.

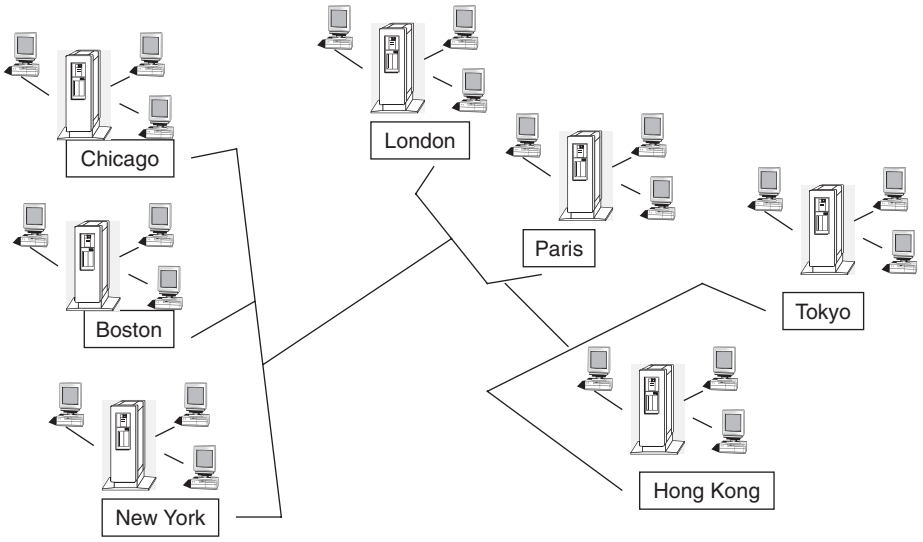


Figure 18-2 Distributed database system for a bank.

transactions. A distributed database is not stored in its entirety at a single physical location but spread across several locations on a computer network.

Let us also give a formal definition for a distributed database management system (DDBMS).

A DDBMS

is a software system that manages the distributed database and makes the distribution of data transparent to users.

Basic Concepts

Let us go over the basic concepts of a distributed database system by considering an example. Figure 18-2 presents a distributed database system for a bank that has domestic and international branches.

Observe the figure, note the following, and understand the underlying concepts:

- The distributed database system is a single, logical database.
- The database is split into a number of fragments.
- Each fragment is stored separately at a location, also known as a site or a node.
- All the sites or nodes, storing parts of the database, are linked by a communication network.
- The data at each site constitute a physical database managed by its own DBMS.
- A distributed database system is a network of loosely coupled sites not sharing physical components.
- Each site has substantial autonomy over the data and rarely depends on any type of centralized control or service.

- A distributed database system may be thought of as a collaboration of participating remote sites storing parts of the logical database.
- Each site, however, may participate in transaction execution when the transaction needs to access data from that site.
- Applications at a site not requiring data from any other sites are known as local applications. Applications that do need data from other sites are called global applications. If all applications at every site are local applications, then there is no necessity for a DDBMS.

Motivation and Goals

At the beginning of this chapter, we discussed broadly the evolution and purposes of distributed database systems. Let us now formalize the motivation for these systems. Why distributed databases? Here are the major reasons.

Efficient access of local data. In many global organizations or in those with a number of geographically dispersed locations, most of the database operations are on the local data. Maintaining the pertinent data at the local site allows efficient and immediate access to local data.

Improved customer service. Global organizations must be flexible enough to meet the needs of local customers. Each location, although part of the overall organization, must primarily look after the customers at that location. Local ownership and control of the local data are essential.

Enhanced reliability. If the computer system at one site fails or if the communication link between two locations goes down, presumably the other sites can continue working. Furthermore, if replication techniques are used to store copies of the same data at more than one site, then other operating sites may still supply the required data.

Availability of global data. Each site in a global organization, although autonomous, is still part of the overall organization. Therefore, users at a site would require data from other sites as well through their global applications. Global data must also be available at every site.

Advantages and Disadvantages

You have been introduced to the concept of distributed database systems. You have understood the motivation that has guided the development of these systems. Let us review a list of advantages provided by distributed database systems. From the list of advantages, you will note that, in practice, distributed database systems are fulfilling the aspirations that prompted their development. These systems also have a few disadvantages.

We will get into in-depth discussions of distributed data systems in the next sections. We will cover the components, the types, and the configurations. DDBMSs have to be more sophisticated than DBMSs. You will learn about

DDBMS—its functions and features. Later, we will consider all aspects of processing queries and transactions in a distributed database environment. But, before covering all of these, first let us summarize the advantages and disadvantages, and then move on.

Advantages Here is the list of major advantages:

Suitable structure. A modern enterprise has users in dispersed locations who need to access primarily local data and, less frequently, global data. Now note the structure of a distributed database with data spread across dispersed locations and yet any data available from anywhere. This structure is well suited for many of today's corporations.

Desirable local autonomy. Users at each site can have autonomy in owning and controlling their local data.

Preserved shareability. Data sharing, a principal motivation for database systems, is still available in a distributed database. All data in the distributed database may be shared by users at all sites.

Improved availability. If the computer system where a centralized database resides goes down, all the data become unavailable to users. However, in a distributed environment, only the part of the data that resides in the failed site becomes unavailable.

Enhanced reliability. Reliability ensures continuous operation of a database system. Even when one site is down, other sites can continue operation, and, in special circumstances, other sites may pick up the operations of the failed site until it is restored.

Better efficiency. As you know, users at each site demand local data much more than global data. Local data may be accessed quite efficiently.

Reduced transmission costs. For global enterprises, compared to centralized databases, distributed databases incur lower data communication costs.

Easier growth. A distributed database provides for modular growth. When a new office is opened at a remote location, simply add a new site or node in your distribution network.

Disadvantages Note the following potential disadvantages presented by distributed database systems. However, it is expected that, as distributed database products mature and become more robust, some of the disadvantages are likely to be less troublesome.

Increased complexity. Making the nature of distribution transparent to users, fragmenting and replicating of data, routing of queries and transactions—all these and other similar issues add to the complexity of distributed database systems.

More complex design. The design of a distributed database system, at both the logical and physical levels, is more complicated than that of a centralized database system.

Difficult integrity control. Distributed query and transaction processing, despite the benefits they offer, are more prone to problems relating to concurrency conflicts, deadlocks, and recovery from failures.

Involved security systems. Security protection must be duplicated at every site.

Added maintenance cost. Maintenance cost must increase to include maintenance of local computer systems, local DBMSs in addition to the overall DDBMS, and the communications network.

Lack of standards. Standards for data communications and data access, critical in distributed database environments, are only emerging now at a slow pace.

Limited acceptance. Despite the numerous advantages, general-purpose distributed database systems are not widely used. Therefore, there is not much industry experience.

DISTRIBUTED DATABASES

When you decide to distribute the corporate data and spread it across the various locations, you are faced with a number of design and implementation issues. You need to plan what type of distributed system you must have. How about the database software at each location? Then how are you going create segments of data and determine which segment gets stored in which location? The next consideration relates to the features necessary in the overall database software to manage the distribution. What about moving data from one site to another whenever needed? What are the characteristics of the communications network?

We will explore these issues in this section. You will learn about the variations available for distributing data and managing data at each site. You will examine the components of DDBMS. You will note implementation options and review design issues.

Types and Configurations

Let us begin by looking at a basic configuration. Figure 18-3 shows a plain distributed database environment.

Look at the figure and note the following points that indicate what components and features are needed in the configuration to make it a distributed database environment.

- At each location, also known as site or node, there is a computer system.
- Local data are stored at each site.

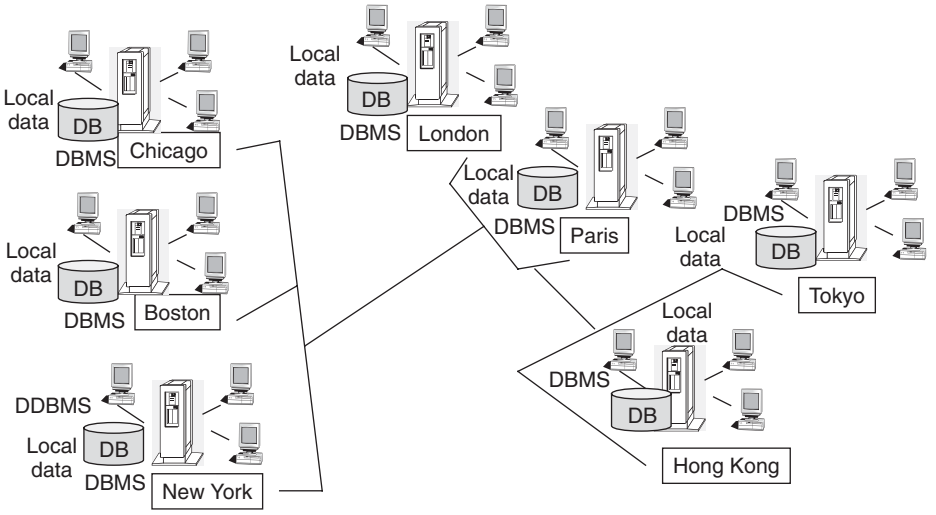


Figure 18-3 Basic distributed database configuration.

- Each site needs database software to manage its local data.
- Each site must have autonomy to own, control, and manage its data and operations.
- Each site must also collaborate with any other site that needs data stored at this site.
- Similarly, each site must also be able to access data from any other site.
- There must be some overall database software to enable cooperation among sites for data access.

Considering these factors, we find that distributed database systems fall into two major types: homogeneous and heterogeneous systems. Figure 18-4 illustrates how these two types are configured in addition to showing a centralized database system for comparison.

Homogeneous Systems The first feature of a homogeneous system is the degree of homogeneity and the second feature is the degree of autonomy. In a homogeneous system, all sites use identical software.

The server at each site uses the same DBMS as that in every other site. Clients at every site use identical software for data access. Each local DBMS is allowed to function as a stand-alone database management system, thus providing local autonomy.

Heterogeneous Systems In this case, different sites run under the control of different DBMSs. However, the sites are connected in such a manner to enable data access across different sites. A heterogeneous system, also known as a multidatabase system, provides a high degree of local autonomy. Because a heterogeneous system consists of a federation of autonomous database systems, it is also called a federated multidatabase system.

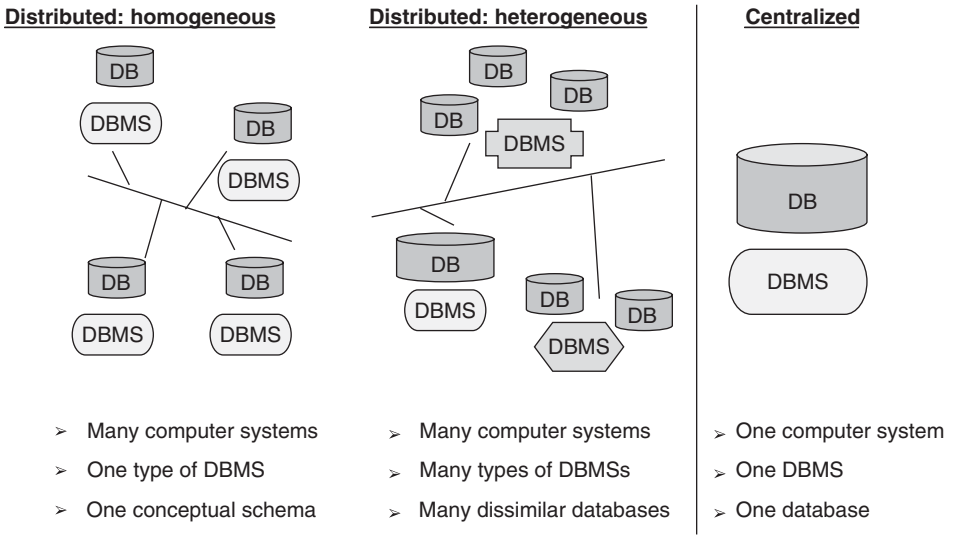


Figure 18-4 Two types of distributed database systems.

A configuration may be heterogeneous in several ways. The global schema must reconcile the variations. Heterogeneity can be manifested as follows:

Different data models. You may have relational databases at a few sites, and some hierarchical or network databases at other sites. In organizations with old legacy systems still in full operation, this configuration may be a practical method for implementing distributed database systems. However, you will have to deal with variations in data representations.

Different constraints. Different models implement data integrity constraints in different ways. For example, the referential integrity constraint in relational models governs relationships between entities in the relational model.

Different naming conventions. Each database may name the same data item in different ways, assigning different data types.

Different data content. The same data element may have different attributes at different sites.

Different data languages. Each site may have its own data definition and data manipulation languages.

DDBMS

The DDBMS constitutes the software that manages the distributed database as a logical whole. Although parts of the data are stored at different sites and managed as local data through the local database management system, DDBMS envelops all of the parts of data at different sites and provides global data access.

When managing global data access, a primary goal for DDBMS is transparency. All aspects of data distribution must be transparent to the users. Users at any site must not be concerned about the location of the data requested by them, nor should they need to know how the data are obtained. Because of the significance of transparency, a later section covers the topic in detail. Let us now examine the components of DDBMS and see how they fit together.

Architecture and Components As you know, for a centralized database, the DBMS supports three schema levels:

- External schema representing the set of user views of the database
- Conceptual schema representing the entire database
- Internal schema consisting of the physical representation of the entire database as files, blocks, and records

These are clearly identifiable and applicable to all centralized database systems. In the case of distributed database systems, however, variations on the types of schemas are possible. We will present two such variations, one for a homogeneous system and the other for a heterogeneous or federated system.

Homogeneous system: schema levels Figure 18-5 shows a configuration with the following schemas:

- A collection of global external schemas
- A single global conceptual schema
- A fragmentation schema

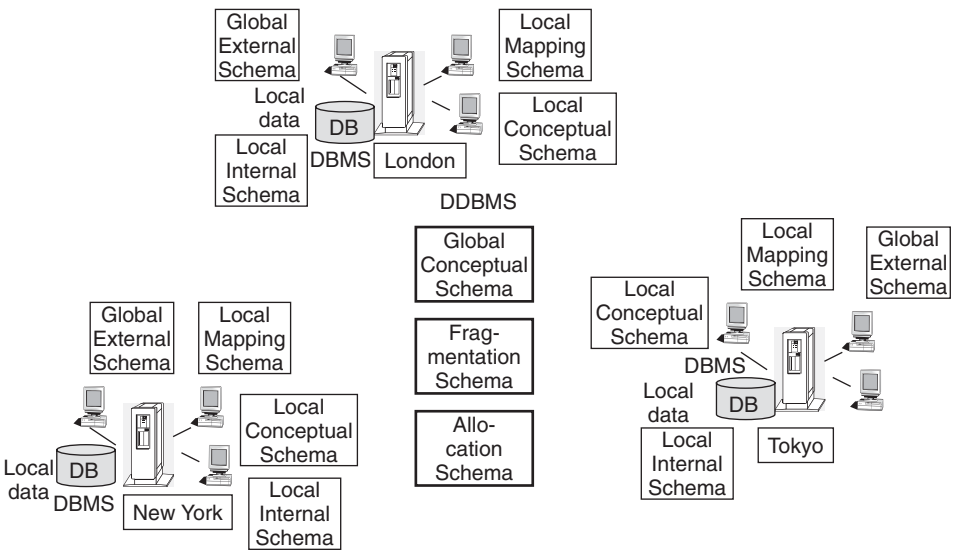


Figure 18-5 Homogeneous system: schema levels.

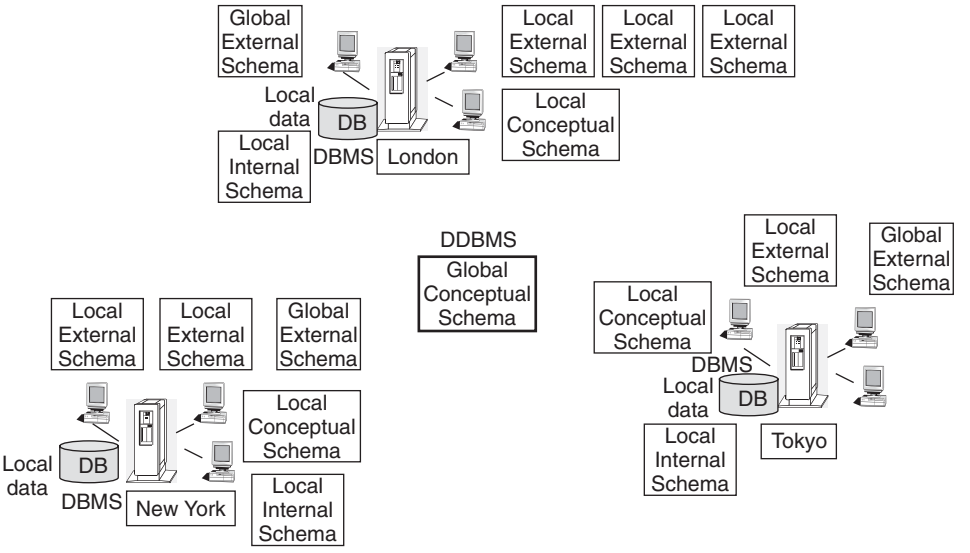


Figure 18-6 Heterogeneous system: schema levels.

- An allocation schema
- For each site, a set of local mapping schema, local conceptual schema, local internal schema

The global conceptual schema logically describes the entire database. It is a union of all the local conceptual schemas. Global external schemas represent user views containing data from different sites. The local mapping schema maps portions of data allocated to different sites to the local external schemas.

The fragmentation schema indicates how data are divided up through partitioning or replication. The allocation schema indicates where fragmented parts are to be located. We will cover fragmentation and allocation further in a later subsection.

Heterogeneous system: schema levels Figure 18-6 shows a configuration with the following schemas:

- A collection of global external schemas
- A single global conceptual schema
- For each site, a set of local external schemas
- For each site, a local conceptual schema
- For each site, a local internal schema

The global conceptual schema in this case is not necessarily a complete union of all the local conceptual schemas. It is a subset of the collection of all the local schemas. It consists of only those parts of the local data that each site agrees to share.

Functions and Features As a starting point, a DDBMS must have the features and functions of a regular database management system. In addition, a DDBMS is responsible for keeping track of the data allocations, routing data access requests to appropriate sites, and transmitting results correctly to the requesting sites.

Here is a summary of the additional functions and features of a DDBMS:

- Enable data fragmentation and allocation to different sites
- Extend system catalog or data dictionary functions to keep track of how data are distributed
- Manage distributed query processing including query optimization, translation of data access requests between sites, and routing of requests and results
- Manage distributed transaction processing including translation of data access requests between sites, routing of requests, and processing of database operations
- Provide concurrency control to maintain consistency of data replicated in multiple locations
- Provide locking and resolve deadlocks in a multisite database system
- Provide distributed database recovery from failures of individual sites and data communication links

Catalog Management As you know, the data dictionary, also known as the system catalog or system directory, holds information for the system to translate high-level requests for database objects from queries and transactions into appropriate low-level operations to be executed on physical data items. Usually, the system catalog in a DBMS contains the usual schema information, security authorizations, statistical data, programs, integrity constraints, and so on.

This type of information in a system catalog, although useful, is hardly sufficient in a distributed database environment. The system ought to know how database relations are fragmented and which parts of the relations reside where. For a given database object, the system catalog must record at which site or multiple sites the object is stored. The catalog must also enable the system to refer to the object using the correct name, taking into consideration the site where it was created and where it is stored now.

We will consider two fundamental issues about catalog management: where to keep the catalog and how to name the database objects. First, about the placement of the catalog, a few possible approaches are indicated:

At a central site. Only one copy exists in the system, and that is held at a single central site. All sites other than where the catalog is kept do not have full autonomy. Each site will have to inform the central site of any schema changes relating to local data at the site. Even for referring to local data at the sites, they have to go to the central site. Another more serious concern: What happens when the centralized site goes down? The catalog will become unavailable, virtually halting all database operations.

Full catalog at all sites. A full copy of the catalog is kept at every site. This approach addresses the vulnerability of losing the whole catalog when the central site goes

down. However, this does not solve the local autonomy problem. Every change to a local catalog must be broadcast to all sites.

Partial catalog at every site. Each site is responsible for its own data; each site maintains its own catalog for the local data. In this case, the total catalog is the union of all partial catalogs maintained at all the sites. This option provides site autonomy. For operations on local data, it works well. However, for requests for global data, the system must go to other sites to get the catalog entries.

Optimal approach. This is a combined approach that removes the vulnerability of a central site and also ensures site autonomy. Each site maintains the catalog for the local data. The catalog at the birth site of a relation, that is, where the relation was created, keeps track of where copies of the relation are held and also precise details of what parts of the relation are held at these other sites.

Now, let us look at the issue of naming database objects. Consider a relation called CUSTOMER. In a centralized database, there can be only one relation with that name. However, in a distributed database system, it is possible for two or more sites to have objects called CUSTOMER whose contents and purpose may not be same. If the CUSTOMER relation at a given site is kept purely local and private, then no other site will be referring to that relation. However, as you can imagine, in a distributed environment users in many sites would want to share the information contained in the CUSTOMER relation. So how do you name business objects? How does the DDBMS know which CUSTOMER table is required?

Database objects are assigned names with four components:

Creator Id	ID of user who created the database object (user ID unique within the site)
Creator Site Id	ID of site at which object was created (sites have unique IDs across entire system)
Local Name	name of object assigned by creator (unique within objects of this type created by this user at the birth site)
Birth Site Id	ID of site at which object is initially stored (sites have unique IDs across entire system)

Here is an example of an object following these principles:

MILLER @ MUNICH – ORDER @ PARIS

The name refers to the database object with the local name ORDER created by MILLER of the MUNICH site, with ORDER stored initially at the PARIS site.

How does a user refer to this object in a query? The user at a given site may refer to the object using the global name with all four of its components. As you know, this is cumbersome; you cannot expect users to refer to objects by their global names. The general practice is to define a synonym for the global name of the object. Each site will maintain a set of synonym tables for users at that site. If MARY at the LONDON site needs to refer to the above ORDER relation, an entry will be

made in a synonym table at the LONDON site for MARY to refer to the ORDER relation as M-ORD. Mary's queries will use the simpler name of M-ORD. When we mention queries here, we do not confine data access requests only to queries. The discussion applies equally to all transactions.

Network Component

Effective implementation of a distributed database depends on a good data communications network. A distributed database system is built over a communications network. The various sites have to be linked up properly so that pertinent portions of the database may be stored at the respective sites. The communications network must enable efficient flow of data from one site to another.

Data communications have made great progress over the past few years. We are not purely interested in the various communications protocols and their specifications. However, we want to examine a few aspects of data communications as they pertain to distribution of data in a distributed database environment. Two major aspects are of importance from this standpoint: the linking of sites and the routing of data.

Linking of Sites On the basis of whether the sites to be linked are within a short distance or further apart, data communications networks are classified as follows:

Local area network (LAN). Designed for connecting computers within the same premises.

Wide area network (WAN). Intended for linking computers or LANs at vast distances from one another.

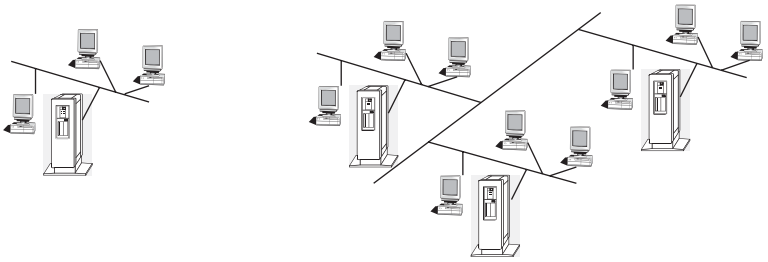
Figure 18-7 presents a comparison of the important features of LAN and WAN arrangements.

Routing of Data Routing implies choosing a path on the network for the movement of data. For this purpose, we may design a network either as a point-to-point network or a broadcast network. For sending a single message to more than one site in a point-to-point network, the sending site must send separate messages to each of the receiving sites. In a broadcast network, the ID of the destination site is attached as a prefix to the message. Virtually each message is sent around so that all sites may listen in. The site for which the message is intended picks up the message.

Configuration Options Figure 18-8 shows the common options for configuring a network for a distributed database. The nature of the organization, the distribution of data, and the data access pattern dictate the choice of the configuration.

Let us quickly review the configurations.

Fully connected. Each site is connected to every other site. Highly reliable and flexible, but expensive.



LAN	WAN
• Covers short distances	• Covers long distances
• Simple Protocol	• Complex Protocol
• Network managed locally	• Network managed by carriers
• Regular topologies	• Irregular topologies
• Links collaborating systems	• Links disparate systems
• Low data rate	• High data rate
• Low error rate	• High error rate
• Broadcasting common	• Point-to-point connections

Figure 18-7 LAN and WAN: features.

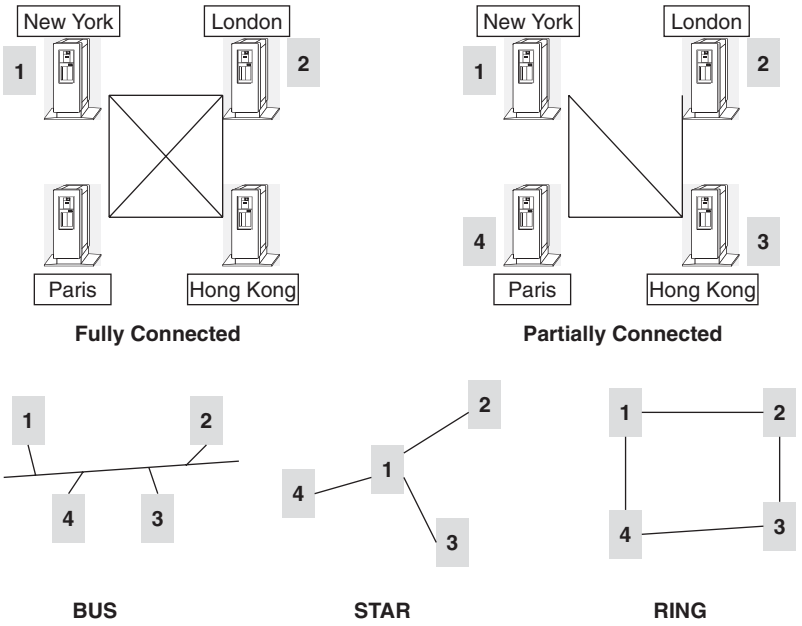


Figure 18-8 Network configurations.

Partially connected. Only certain sites are linked. Sites with high volumes of data traffic are usually connected. Less expensive than fully connected configuration.

Bus. All sites connected to a backbone link. Simple, easily extendable, and cost-effective. Useful in small network with low traffic volumes.

Star. Used for connecting sites with the use of the concept of a central site. The configuration fails if the central site goes down.

Ring. Connects sites in a closed loop. Generally used for high-performance networks.

Data Distribution

All along we have been saying that parts of the database are stored and managed at the various sites of an organization. We looked at the levels of schemas that must be present at the sites for representing the database. We also discussed the network configurations that could enable moving of data between sites as needed. Now the question remains, How exactly do you break up the database into portions that can be kept at different sites? How do you divide the data content and distribute it among the sites?

Two basic methods are commonly used to break up the database for the purpose of distribution among the sites. Our discussions will be based on the relational data model. Nevertheless, the principle applies equally well to the other data models such as hierarchical or network. Let us consider just one relation, an EMPLOYEE relation, to present the methods.

Figure 18-9 illustrates the following two methods using the EMPLOYEE relation as the contents of the database:

- Data fragmentation
- Data replication

Data Fragmentation When using the data fragmentation or data partitioning method, you break up a relation into smaller fragments or partitions in one of two ways. Look at the data in the EMPLOYEE relation. What is the data content made of? Rows and columns. You know the significance of rows and columns in a relation. How can you partition the relation into fragments? You can break up the relation horizontally into groups of rows or vertically into groups of columns. You can then store these fragments at the various sites.

Horizontal fragmentation Each fragment consists of a subset of rows or tuples of a relation. How can you use this method to create fragments from the EMPLOYEE relation? Let us say that three sites exist on your distributed system network—New York, London, and Milan. Naturally, the Milan site would be interested in dealing with the Italian employees, the London site with the British employees, and the New York site with the American employees. So it makes sense to break up the

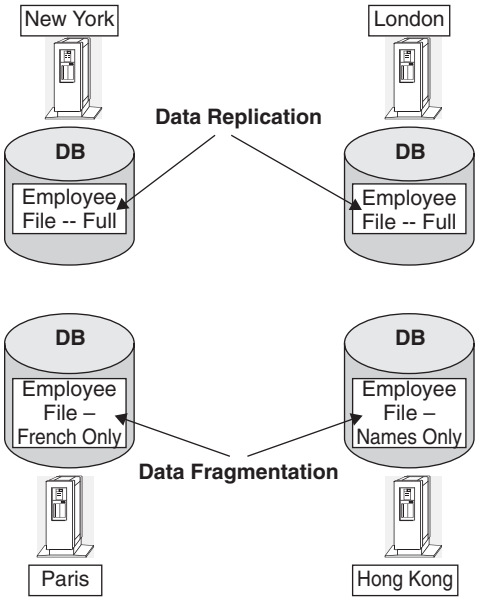


Figure 18-9 Data fragmentation and data replication.

EMPLOYEE relation into subsets of rows relating to employees in these three regions.

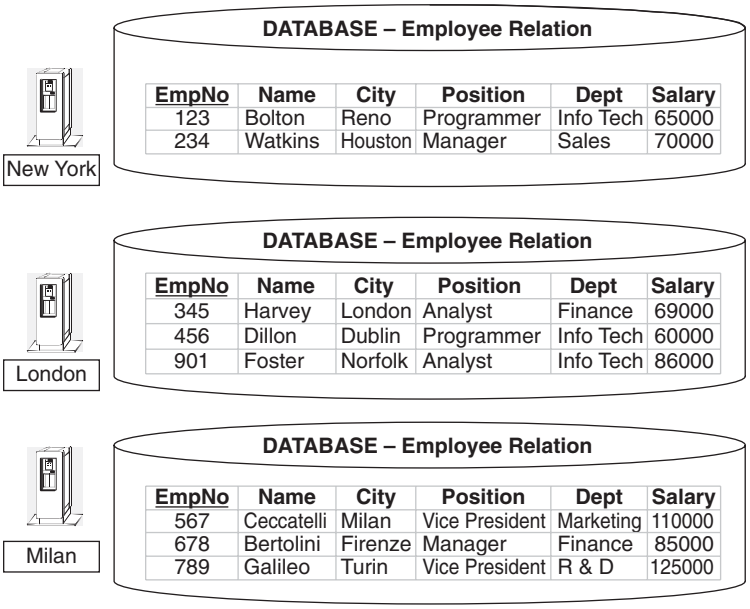
Figure 18-10 demonstrates horizontal fragmentation. Note how the subsets are stored at the three sites.

Vertical fragmentation Each fragment consists of a subset of columns of a relation. Each subset must also contain the key column in addition to the other selected columns. Sometimes, a tuple-ID is added to each tuple to uniquely identify it. How can you use this method to create fragments from the EMPLOYEE relation? Let us say that for your organization the human resources functions are localized in Boston and the accounting functions, including payroll, are localized in New York. When you examine the columns of the EMPLOYEE relation, you will note that there are a few columns that are of interest only for payroll and the remaining columns are desirable for the human resources division. Break up the EMPLOYEE relation into two subsets of columns to be stored at the two sites.

Figure 18-11 demonstrates vertical fragmentation. Note how the subsets are stored at the two sites.

Advantages and disadvantages Fragmentation improves efficiency in data access. Data fragments are stored at the sites where they are used most. Local query and transaction performance gets better. At the same time, you get enhanced data security because it is easier to protect the local data.

On the other hand, when any site needs data from other sites for any types of queries or transactions, access speeds become inconsistent. Queries on global data

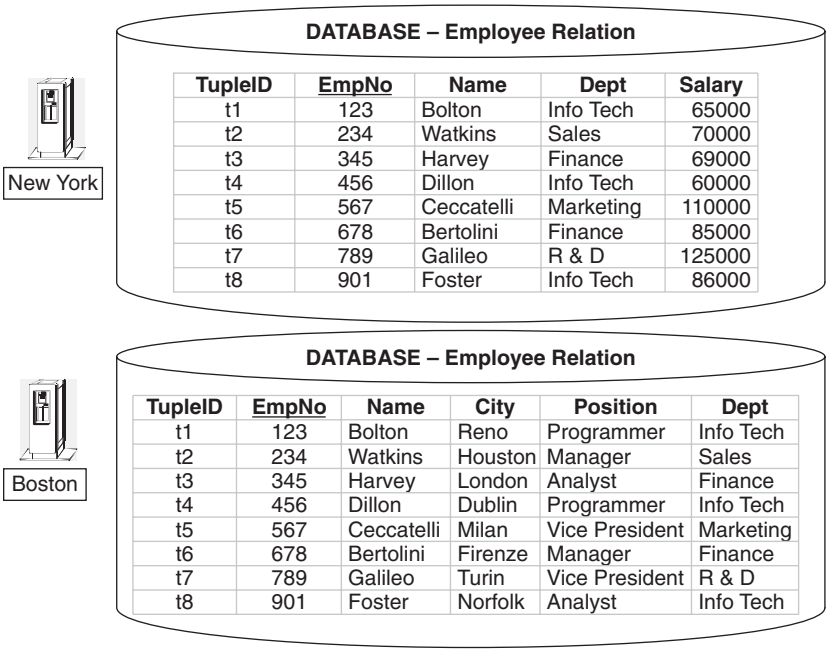


EmpNo	Name	City	Position	Dept	Salary
123	Bolton	Reno	Programmer	Info Tech	65000
234	Watkins	Houston	Manager	Sales	70000

EmpNo	Name	City	Position	Dept	Salary
345	Harvey	London	Analyst	Finance	69000
456	Dillon	Dublin	Programmer	Info Tech	60000
901	Foster	Norfolk	Analyst	Info Tech	86000

EmpNo	Name	City	Position	Dept	Salary
567	Ceccatelli	Milan	Vice President	Marketing	110000
678	Bertolini	Firenze	Manager	Finance	85000
789	Galileo	Turin	Vice President	R & D	125000

Figure 18-10 EMPLOYEE relation: horizontal fragmentation.



TupleID	EmpNo	Name	Dept	Salary
t1	123	Bolton	Info Tech	65000
t2	234	Watkins	Sales	70000
t3	345	Harvey	Finance	69000
t4	456	Dillon	Info Tech	60000
t5	567	Ceccatelli	Marketing	110000
t6	678	Bertolini	Finance	85000
t7	789	Galileo	R & D	125000
t8	901	Foster	Info Tech	86000

TupleID	EmpNo	Name	City	Position	Dept
t1	123	Bolton	Reno	Programmer	Info Tech
t2	234	Watkins	Houston	Manager	Sales
t3	345	Harvey	London	Analyst	Finance
t4	456	Dillon	Dublin	Programmer	Info Tech
t5	567	Ceccatelli	Milan	Vice President	Marketing
t6	678	Bertolini	Firenze	Manager	Finance
t7	789	Galileo	Turin	Vice President	R & D
t8	901	Foster	Norfolk	Analyst	Info Tech

Figure 18-11 EMPLOYEE relation: vertical fragmentation.

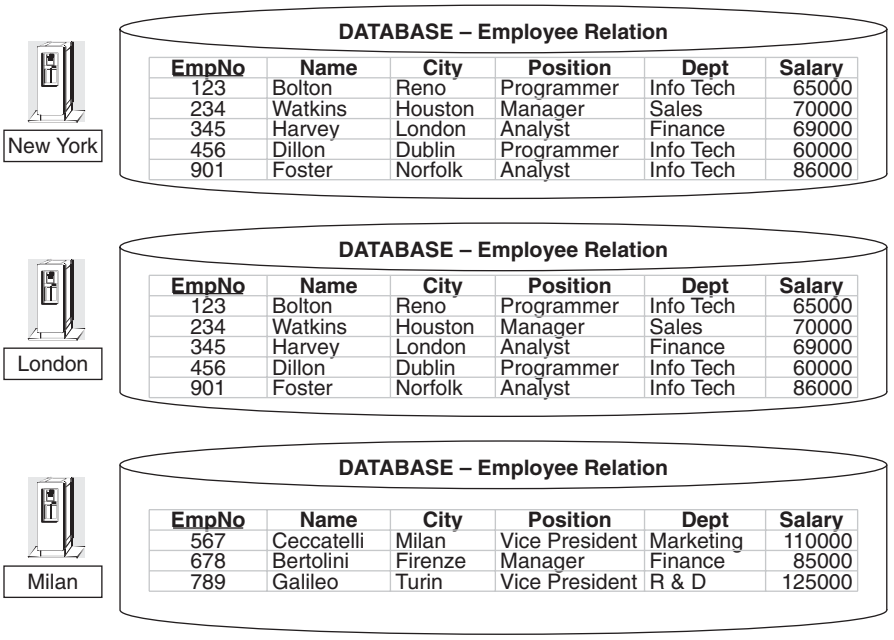


Figure 18-12 EMPLOYEE relation: replication.

run a lot slower than queries on local data. Also, fragmentation presents a problem with failures. When one site fails, the data at that site become inaccessible throughout the organization.

Data Replication Suppose that, in your environment, every site needs the full EMPLOYEE relation. What is the best method of providing for users at every site? Keep a full copy of the relation at every site. Then every user in the organization can have fast and easy access to employee data. The data replication method means that we store either full copies of a relation or copies of appropriate fragments at several sites. Figure 18-12 illustrates data replication. Observe the copies stored at different sites.

Advantages and disadvantages Replication improves data availability. If one of the sites containing a copy of a relation goes down, another copy may be accessed from another site. In full replication, because each site gets a full copy, the data constitute local data at every site. Therefore, you can get fast response to queries against the relation at every site.

However, in practice, implementation of replication is complex, especially for keeping all the copies synchronized whenever updates take place. Also, if each site stores a full copy of a large relation with thousands of rows and several columns, storage requirements for the distributed database can be quite extensive.

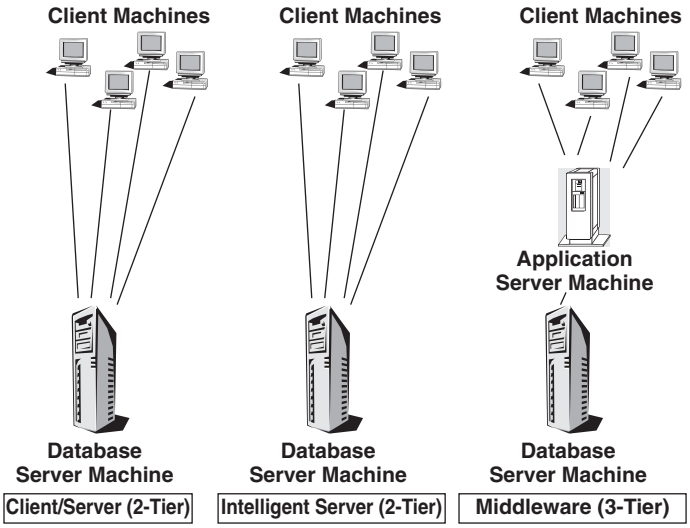


Figure 18-13 Architectural options.

Synchronous and asynchronous replication In our example of replication of the EMPLOYEE relation, suppose some transaction at the London site wants to update the copy of the relation at a certain time. How do we include the effects of the update in all the copies? One method is for the transaction to update all copies of the EMPLOYEE relation before it commits. This method is known as *synchronous replication*. If there are too many copies spread around at different sites, then update transactions will take a long time to synchronize all copies and commit.

Instead of updating all copies each time an update transaction executes, we can wait to update the copies at a later time. This method is called *asynchronous replication*. At specific intervals, copies of modified relations are updated. Obviously, anyone using a copy that has not yet been updated will not obtain the correct results. Many commercial systems implement this method.

Architectural Options

Let us now turn our attention to options for the types of computing architectures that are used for implementing distributed databases. You know that data are distributed among several sites of an organization. You have also learned the techniques for dividing the database and distributing the fragments to the various sites. You have examined the communications network options for linking the sites. What types of architectures are suitable at the sites?

We will explore three common options. Figure 18-13 presents the three options. Note how data access is provided in each option. We will now briefly describe the three architectural options.

Client/Server Architecture (Two Tier) As an information technology professional, you are familiar with client/server systems. Let us examine how this architecture works for distributed database systems in very simple terms. At each site

you have a database server machine and as many client machines as needed. The server on one tier and the clients on the other tier form this two-tier architecture. The server and the clients are usually linked on a LAN.

The following briefly describes how a two-tier system may apply to distributed databases.

- The server performs all database functions. Based on query requests, the server retrieves the requested data and passes them on to the requesting client. It also updates the database as required by executing transactions.
- The client manages the user interface, interprets queries and transactions, and routes them to the appropriate servers. When data is received from the server, the client presents the information to the user. If a query requires data from multiple sites, the client breaks up the query into appropriate subqueries and routes each subquery to the proper server. When results are received from multiple servers, the client consolidates the result sets and presents the consolidated result to the user.

The two-tier architecture clearly demarcates the functions of the server and the client. Each side can be suitably configured for high performance. The server can be made robust enough with sufficient memory, storage, and computing power to carry out its database functions. But, when you look at the configuration of the client, it must also be powerful and have software to interpret queries, create subqueries, and route them properly. And every client machine must be so configured. This is not an inexpensive option.

Intelligent Server Architecture (Two Tier) This is also a two-tier option. Let us go over this option and inspect how this can work in a distributed database environment. As in the previous case, at each site you have a database server machine and as many client machines as needed. The server on one tier and the clients on the other tier form this two-tier architecture. The server and the clients are usually linked on a LAN.

The server and the clients serve the following functions.

- The server performs all database functions. Based on query requests, the server retrieves the requested data and passes them on to the requesting client. It also updates the database as required by executing transactions. If a query requires data from multiple sites, the server breaks up the query into appropriate subqueries and routes each subquery to the proper server. When results are received from multiple servers, the local server consolidates the result sets and sends the merged result to the requesting client.
- The client manages the user interface, receives query and transaction input, and sends them to the local server at that site. When data are received from the server, the client presents the information to the user.

Although the two-tier architecture clearly demarcates the functions of the server and the client, we have just shifted some functions to the server from the clients. This relieves the clients from being too heavy. However, now the servers must

perform routing and consolidation functions in addition to the large array of database functions. This option is slightly less expensive because we have to duplicate routing and consolidation functionality only on the servers, not on the numerous clients.

Architecture with Middleware (Three Tier) This is a practical and efficient approach to relieve the server and the clients of the additional routing and consolidation functionality. You create a middle tier between the server and client tiers. You install software, called middleware, with routing and consolidation capability in the middle tier. Let us review how this architecture works for distributed database systems. At each site you have a database server machine, a middleware machine, and as many client machines as needed. The server on one tier, the clients on the other tier, and the middleware on the third tier form this three-tier architecture. The server, middleware, and the clients are usually linked on a LAN.

The following briefly describes how a three-tier system may apply to distributed databases.

- The server performs all database functions. Based on query requests, the server retrieves the requested data and passes them on to the middle-tier. It also updates the database as required by executing transactions.
- The client manages the user interface, interprets queries and transactions, and routes them to the middle tier. When data are received from the middle tier, the client presents the information to the user.
- The middle tier examines each query and routes it to the appropriate server. If a query requires data from multiple sites, the middle tier breaks up the query into appropriate subqueries and routes each subquery to the proper server. When results are received from multiple servers, the middle tier consolidates the result sets and sends the consolidated result to the requesting client.

The three-tier architecture proves to be a straightforward and efficient option. The machines at each tier can be configured correctly for the functions they intend to support. In practice, the systems tend to be multitier systems with other tiers such as application servers added to the mix.

Design and Implementation Issues

By now, you are proficient at the design and implementation process for centralized database systems. You are quite familiar with the phases of the database development life cycle. After the initial planning and feasibility study phases, you move into the requirements definition phase. For developing distributed database systems, you do the same thing. The requirements definition phase forms the basis for the design and implementation phases.

In the requirements phase, you gather information requirements for performing logical and physical design. While designing a centralized database, you know that the entire database will reside in one location, perhaps on a single database server. This, however, is not the case with a distributed database. You do not store all the

data in one place. This is a major difference you encounter while designing and implementing a distributed system.

Therefore, your design should include considerations for placement of data at the various sites. How can you make the design and implementation decisions on what parts of data must be kept at which points? Of course, you need to find out the usage patterns and match them up with data content. This expands the scope of your requirements definition phase. In that phase, you must gather details to address two basic issues:

- How to fragment the database
- How to allocate fragments to sites

We will address these issues for a relational data model. Let us list a few suggestions on these issues and indicate some design and implementation steps:

- On the basis of the requirement definition, proceed through the design phase and come up with a global schema for the entire database. This schema will include all the relations.
- Next comes the decision point on how to divide the database and then which segments to store where. Once you are able to determine the location of parts of the data, then the local schema at each site will represent the part of the data stored at that site.
- Consider each relation for the possibility of fragmenting it horizontally or vertically.
- When you are examining a potential subset for fragmentation, evaluate data sharing as it applies to the subset. Determine which site is likely to use the subset most. If you can establish the site that is expected to use the subset most, then assign the fragment to that site.
- Sometimes a number of sites are likely to use a subset or a complete relation with almost equal frequency. If so, consider full or partial replication and place copies at all such sites.
- Decisions on fragmentation and replication are never easy and clear-cut. That is why a lot of preliminary work must be done in the requirements definition phase itself. Analysts and designers must "know" their data extremely well.
- Remember that usage access patterns change over time. Business conditions change. New applications are added. The initial set of applications is enhanced. So the data allocation plan must be reviewed periodically after deployment and adjusted as necessary.
- Another key factor in the allocation plan is the placement and management of the global system catalog. Consider alternatives.

Figure 18-14 presents an example of a data allocation scheme. Note how the database is fragmented and fragments are placed at different sites. Also note the replicated copies stored at different sites.

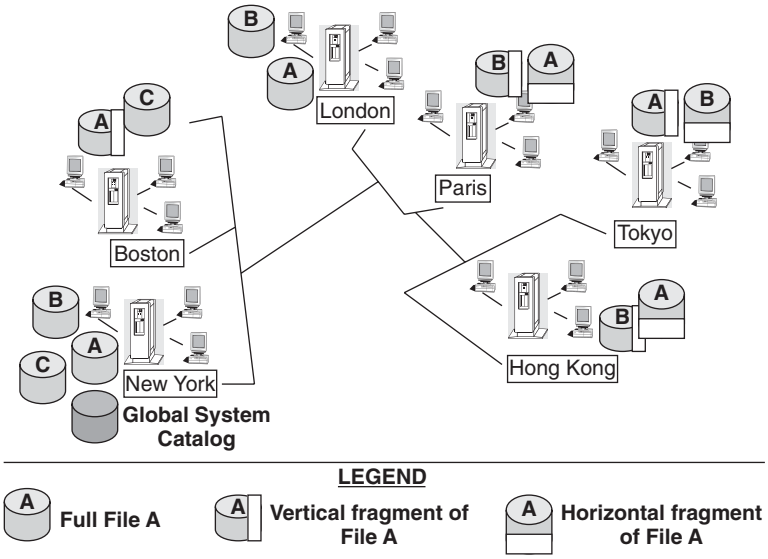


Figure 18-14 Sample data allocation scheme.

TRANSPARENCIES

When we introduced DDBMS in an earlier section, we mentioned in passing that a primary goal for DDBMS is transparency. We will now describe the concept further. Please note the motivation for transparency and why it must be a primary goal for DDBMS.

Transparency: Key Ideal

By now, you have realized that a distributed database is a lot more complicated than a centralized database in design, implementation, architecture, and software support. By and large, the complexity arises out of the way data are distributed across various sites. When a user initiates a query or an application executes a transaction performing database operations, should the user or the application be concerned about how and where the data are stored, how data will be accessed, and by exactly what means the results must be consolidated and presented? Not at all.

The DDBMS must insulate users from these complexities. The DDBMS must make the data locations and access routes transparent to users. Data access and processing must appear to the users as though all data are stored at one site. The overall ideal goal the DDBMS must strive for is to portray a distributed database to users as though it is a centralized database.

The DDBMS may provide transparencies at different levels. The actual techniques used by the DDBMS to provide for each level of transparency are quite intricate. We will not get into all the technical details. Instead, we will broadly discuss each level of transparency so that you can appreciate its purpose and significance.

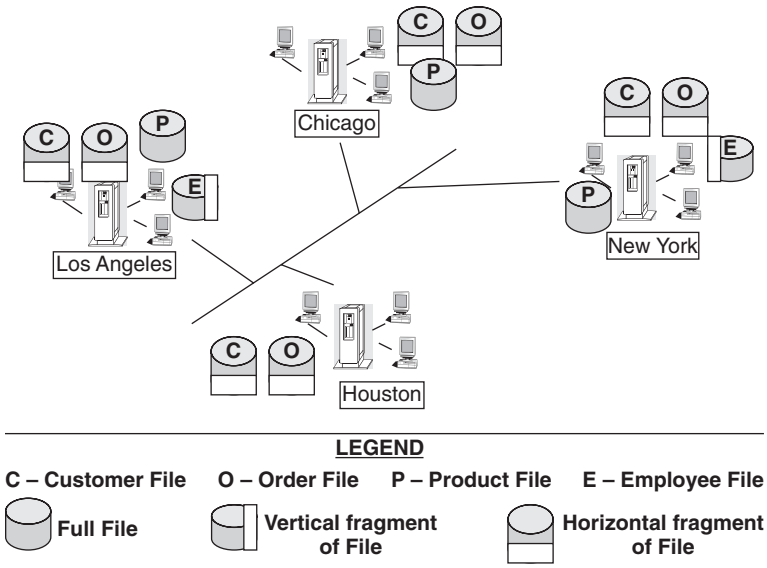


Figure 18-15 Sample distributed database system.

For the purpose of our discussion, let us assume a distributed database system with four sites at New York, Los Angeles, Chicago, and Houston. The database consists of four relations: CUSTOMER, ORDER, PRODUCT, and EMPLOYEE. Figure 18-15 presents this sample distributed database system and indicates how the data are distributed. Using the figure, let us examine the levels of desired transparencies.

Note the four relations and observe how they are fragmented. The CUSTOMER and ORDER relations are partitioned horizontally, and each site stores the partitions pertinent to it. The EMPLOYEE relation is partitioned vertically. The partition relating to human resources functions is kept in Los Angeles, and that relating to payroll is stored in New York. The PRODUCT relation is replicated, and full copies are stored in New York, Los Angeles, and Chicago. Houston has only occasional sales.

Fragmentation Transparency

You have noted how the database is divided and parts are allocated to the various sites. However, a user at any site does not know whether the database is stored in its entirety or is divided into parts. For example, let us say that a user in Chicago wants a report showing the complete details of all employees. The user query is submitted as though employee data are stored as whole. You know that all employee data are not kept together and that the EMPLOYEE relation is partitioned vertically. The DDBMS enables the query to execute by retrieving the information from both vertical partitions and to produce a consolidated result.

This is fragmentation transparency. The DDBMS hides the complexities of how the horizontal and vertical partitions are created. It makes the partition scheme completely transparent to the user.

Replication Transparency

You know that the PRODUCT relation is replicated and that full copies are stored in three sites. Assume that a user in Houston wants a product list. When the query is submitted, the user has no knowledge of whether there is one copy or more copies of product. In fact, the user need not be concerned. The DDBMS hides the details of replication from the user and provides replication transparency.

The DDBMS knows that there is no local copy of product data in Houston. So it selects a copy from one of the other three sites, enables the query to execute, and produces the result. When there are copies at multiple sites, the DDBMS chooses the best possible candidate that will execute the query efficiently. This takes in consideration network traffic and other factors. If one of the sites, namely, Los Angeles, is not operational at that time, the DDBMS will go to other sites even though they are further away by comparison.

Location Transparency

Through the property of location transparency, the DDBMS hides from the user the details of where parts of the database are stored. To the user, it appears as though all data are stored at the local site. How many sites there are and how the database is spread to any sites—these are totally transparent to the user.

For example, assume that the Chicago site is given the responsibility of analyzing all the orders for the past six months. A user in Chicago charged with this responsibility submits a query to retrieve selected data from the entire ORDER and CUSTOMER relations. You know that these two relations are partitioned horizontally and the data are spread across all sites. However, the query from the Chicago user need not be concerned about where all of these data are coming from. The DDBMS hides all the distribution details from the query. The coding of the query is exactly as though it is coded for a centralized database. The DDBMS enables data to be retrieved from all the partitions at different sites and consolidated to produce the result.

Network Transparency

In an earlier subsection, we discussed the various methods of configuring the communications network. We looked at LANs and WANs. We mentioned fully connected and partially connected networks. We also considered a few topologies such as bus, star, and ring. Now suppose that a user in New York needs to create a confidential employee report printing full details. Should the user be concerned whether New York and any other site that may store employee data are connected directly? Should the network configuration affect the way the user query must be coded?

The DDBMS makes the data communications network, its protocols, and its configuration transparent to the user. The user does not even need to be concerned whether there is a network at all. The user's query will be coded as though all data will be retrieved locally without any complex networks to support the retrieval.

Naming Transparency

While covering catalog management above, we dealt with the naming of database objects in a distributed database system. As in a centralized database, database objects in a distributed database must also have unique names. You have noted how the names are made unique. You attach three different identifiers to the local name of the object to make the name unique.

The DDBMS hides the complications of extending the local object names from the users. User queries can refer to database objects by local names. DDBMS provides naming transparency through a method of using synonyms. Please refer back to the subsection on catalog management.

Failure Transparency

You know that, in a centralized database system, a transaction must preserve its atomicity and durability. That is, in the event of failures, either every operation of a transaction is completed successfully or none of the operations of the transaction is completed at all. Then atomicity of transactions is maintained. Furthermore, if a transaction completes successfully and commits, no failure should be able undo any database update performed by the committed transaction. The database changes must be durable.

A distributed database environment is vulnerable to other types of failures related to communication links. The DDBMS ensures that atomicity and durability of a transaction will still be protected in the distributed environment as well. This property is failure transparency.

DISTRIBUTED PROCESSING

The complexities of processing queries and transactions in a distributed database environment far exceed those in a centralized database system. When a query is submitted in a distributed database system, the DDBMS has to take into account a number of factors to prepare an optimal query plan to execute the query. Do the data elements needed by the query reside locally at that site? If not, are the data elements stored elsewhere? Are the data elements parts of fragments? Is there any replication scheme to be considered? If so, which is the optimal copy to be used? Similar questions arise while processing transactions that perform insert, update, and deletion operations.

Concurrency control gets more complicated in a distributed environment. Where do you keep the locks? What types of locking protocols are effective? How do you resolve deadlocks? Could transactions from multiple sites be in a deadlock situation? Recovery from failures poses greater challenges as well. What happens when a site fails? Can the other sites keep operating while the failed site is recovered?

We will consider the essential basics in this section. We will highlight the underlying complexities and present how distributed processing is generally performed. We will present some recovery methods. However, more than cursory coverage of

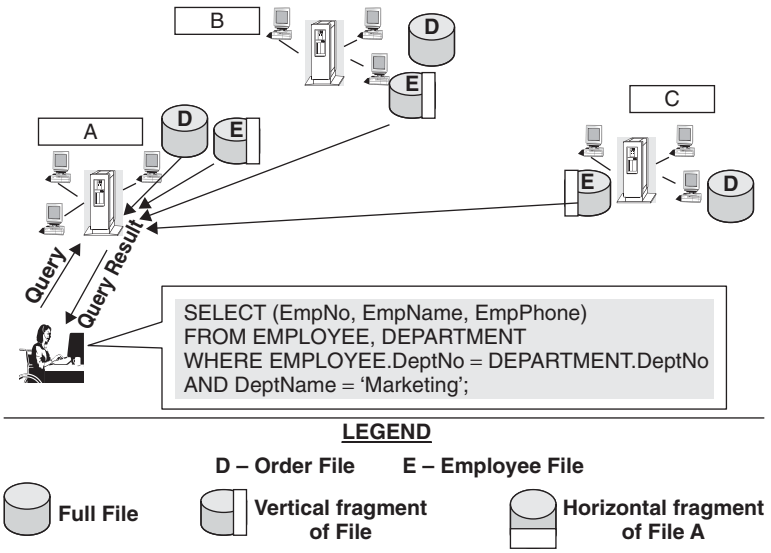


Figure 18-16 Query processing example.

the topic of distributed processing is beyond our scope. Nevertheless, you will survey and learn the fundamental principles and concepts.

Query Processing

For the sake of simplicity, let us say that our tiny distributed database consists of just the following two relations and the system comprises only three sites at A, B, and C.

DEPARTMENT (DeptNo, DeptName, DeptLocation, DeptManager)
EMPLOYEE (EmpNo, SocSecNo, EmpName, EmpAddr, EmpPhone, Salary, DeptNo)
Foreign Key: DeptNo REFERENCES DEPARTMENT

Figure 18-16 illustrates one version of the data distribution showing some fragments and replicas at the three sites. Also, note the typical query.

Query Transformation Let us take a very simple query to list all the rows of the DEPARTMENT relation, submitted by a user at site A.

```
SELECT *
    FROM DEPARTMENT
```

How is this simple query processed? That depends on the fragmentation and replication schemes used for distributing the data. Depending on the distribution, the DDBMS must transform the query into subqueries, execute each subquery at

the appropriate place, combine the results through union or join operations, and present the result to the requestor. While decomposing a query and executing subqueries the optimizer examines the options and chooses the most optimal execution. Let us consider some possible scenarios of query decomposition.

Horizontal Partitioning Assume that the DEPARTMENT relation is partitioned horizontally into three fragments, DEP-H-A, DEP-H-B, DEP-H-C, with each site storing the appropriate fragment. The DDBMS may transform the query into three subqueries: Subquery-A, Subquery-B, Subquery-C. For example, Subquery-A is as follows:

```
SELECT *
    FROM DEP-H-A
```

We are neither getting into the exact technical details of how the DDBMS looks up the system catalog and transforms the original query nor getting into the exact format of the query. However, please note the principles of query transformation. The DDBMS processes Subquery-A at the local site A and dispatches the other two subqueries to sites B and C, respectively. When the results are received from sites B and C, the results are consolidated through union operation and presented to the user at site A.

Vertical Partitioning Assume that the DEPARTMENT relation is partitioned vertically into two fragments, DEP-V-B and DEP-V-C, with each site storing the appropriate fragment. The DDBMS may transform the query into two subqueries: Subquery-B, Subquery-C. For example, Subquery-B is as follows:

```
SELECT *
    FROM DEP-V-B
```

No department data are stored at the local site of the user. The DDBMS dispatches the two subqueries to sites B and C, respectively. When the results are received from sites B and C, the results are consolidated through union operation and presented to the user at site A.

Hybrid Partitioning In this case, assume that the DEPARTMENT relation is first partitioned horizontally into two fragments, DEP-H-A and DEP-H-BC, and that site A stores fragment DEP-H-A. Fragment DEP-H-BC is portioned further vertically into two vertical partitions, DEP-H-BC-V-B and DEP-H-BC-V-C, stored at sites B and C, respectively. The DDBMS may transform the original query into three subqueries, each executing on the appropriate fragments at sites A, B, and C. The results from sites B and C are first put together through a join operation, and then the result of this join is combined with the result from site A through a union operation.

Replication Let us consider one case with replication. Assume that the DEPARTMENT relation is not partitioned but replicated. Full copies are stored at sites B

and C. In this case, the DDBMS must choose between sites B and C based on optimum processing conditions. Let us say that site C is chosen. The DDBMS transforms the query to execute on the copy at site C, changing the name of the relation to the local name at site C. The result of the query is received at site A and presented to the user there. No union or join operations are necessary.

What we have considered here simply illustrates the principle of query transformation, decomposition, and routing. The details are involved. Again, the DDBMS has a number of options to transform and decompose a query based on the partitioning and replication schemes. It makes the determination based on query optimization techniques by considering all costs in the distributed database environment.

Nonjoin Queries In the above subsection, we considered a very simple query and reviewed the query transformation process. Let us go over a few more queries with selection criteria, but without any explicit joins. We will present a few cases and make general comments on the nonjoin queries.

Example 1

Query from site B: List of employees from departments with department numbers 24 and 43.

Data distribution: EMPLOYEE relation partitioned horizontally. Employees with department number 24 at site A and those with department number 43 at site C.

Action by DDBMS: Transform query incorporating local database object names, execute query at sites A and C, receive results at site B, combine results through union operation.

Example 2

Query from site A: List of employees from department number 31.

Data distribution: EMPLOYEE relation partitioned horizontally. Employees with department number 31 at site B and all others at site C.

Action by DDBMS: Recognize that data for employees from department number 31 are stored at site B. Transform query incorporating local database object names, execute query at site B, receive results at site A.

Example 3

Query from site B: Find the average salary of employees from departments with department numbers 24 and 43; average computed taking employees from both departments together.

Data distribution: EMPLOYEE relation partitioned horizontally. Employees with department number 24 at site A and those with department number 43 at site C.

Action by DDBMS: Recognize that averages cannot be computed individually at each of the sites A and C and combined by union operation. Transform

query into two subqueries, one for site A and the other for site C. Change the queries to compute count of employees and sum of salaries. From these two queries, receive sum of salaries and count of employees at each site. Now calculate average by adding up the sums of salaries and dividing by the total number of employees. This case illustrates the potential complexity of some query transformations.

Example 4

Query from site B: List all data for employees from departments with department numbers 24 and 43.

Data distribution: EMPLOYEE relation first partitioned vertically, keeping salary data for all employees at site B. The rest of the data in the relation partitioned horizontally. Horizontal partition for employees with department number 24 at site A and those with department number 43 at site C.

Action by DDBMS: Recognize the hybrid partitioning scheme. Transform and decompose query into three subqueries. Execute two subqueries appropriately at sites A and C where horizontal partitions are stored. Receive results at B, and put the results together through union operation. Execute the other subquery at B. Combine this result with earlier consolidation of results from sites A and C through union operation.

Join Queries Processing of queries with join conditions in a distributed database environment can be quite complicated and difficult. The complexity varies with the manner in which base relations are distributed across the sites. Many of the typical queries in any database environment contain join conditions. For optimizing join queries, the DDBMS has to examine a large number of processing options.

Let us take a simple example and scrutinize the processing options. We will use the following relations for our example. This discussion is modeled after a presentation by C. J. Date, an acknowledged database expert and an eminent author, in his book mentioned in the References section.

PROJECT (ProjNo, ProjDesc, ProjManager, ProjLocation)
EMPLOYEE (EmpNo, SocSecNo, EmpName, EmpAddr, EmpPhone, Salary, DeptNo)
ASSIGNMENT (ProjNo, EmpNo, HoursWorked)
Foreign Keys: ProjNo REFERENCES PROJECT
EmpNo REFERENCES EMPLOYEE

Assume that database is distributed to two sites, A and B, as follows:

At site A PROJECT relation (500 rows)
ASSIGNMENT relation (50,000 rows)
At site B EMPLOYEE relation (10,000)

Manager Jones wants to know the names of employees on his projects who make more than $100,000.

```
SELECT E.EmpName
  FROM PROJECT P, EMPLOYEE E, ASSIGNMENT A
    WHERE P.ProjNo = A.ProjNo
    AND E.EmpNo = A.EmpNo
    AND P.ProjManager = 'Jones'
    AND E.Salary > 100000
```

This is not a very complicated query. As you know, in terms of relational algebra, the result is obtained by two select operations, two union operations, and one project operation. In a centralized database system, all of these operations are performed on the database at a single site. That is not the case with a distributed database system. There are numerous ways that data can be moved around to perform these operations in the most optimal manner. The DDBMS is expected to examine these ways and choose the least expensive option.

Even in our simple distributed database system with just three relations and three sites, several options are conceivable. Just to give you an idea of the complexity of join query processing, let us list a few options. After the result is obtained, the result is transmitted to the site where manager Jones is. Review the following list:

- Perform select operation at site B on EMPLOYEE for salary > 100000. Move result to site A. Complete all remaining operations at site A.
- Move entire EMPLOYEE relation to site A. Perform all operations at site A.
- Perform join operation on PROJECT and ASSIGNMENT relations at site A. Perform select operation on the result for 'Jones.' For each of the rows in this selection, send a message to B to check whether the corresponding EMPLOYEE row shows salary > 100000. Get responses back from B. Select only such EMPLOYEE rows. Perform project operation and obtain final result.
- Move PROJECT and ASSIGNMENT relations to site B. Perform all operations at site B.
- Perform select operation at site A on PROJECT relation for 'Jones.' Perform join operation on this result and ASSIGNMENT relation at site A. Perform select operation at site B on EMPLOYEE relation for salary > 100000. For each of the selected rows send a message to site A to verify whether the selected row relates to 'Jones.' Select only such rows. Perform project operation and obtain final result.
- Perform join operation on PROJECT and ASSIGNMENT relations at site A. Perform select operation on result for 'Jones.' Move result to site B. Complete remaining operations at site B.

The total data transmission time for each of these options can be computed based on the number of rows in each relation, the number of qualifying rows in the EMPLOYEE and PROJECT relations, the size of the records, and data transfer rates between the sites.

Optimization Considerations From our discussion in the previous subsection, you realize how complex query optimization would be in a distributed database environment. The more intricate the data distribution, the more complicated query optimization turns out to be. You have seen how, even in a very simple example of EMPLOYEE, PROJECT, and ASSIGNMENT relations distributed across two sites, several plausible options exist for query processing. Can you imagine how these would be multiplied in a real-life distributed database environment?

We discussed earlier the issues and factors involved in optimizing queries in a centralized database system. We need to include the challenges caused by the following additional factors in a distributed database environment:

- Scope, extent, and composition of the data fragmentation scheme
- Data replication scheme requirement decisions on which copies to use
- Options for performing primitive operations on the data fragments at different sites
- Creation and execution of subqueries
- Consolidation of results of subqueries
- Transmission of final result to the requesting user
- Network configuration and data transmission costs
- Necessity for the DDBMS to preserve the autonomy of local DBMSs

Transaction Processing

Consider a transaction in a centralized database system. Each transaction performs database operations as an atomic unit of work. In its execution, a transaction may perform operations on several database objects. Even as a transaction proceeds through its execution reading, updating, inserting, and deleting database items, it executes in a manner such that its ACID properties are maintained.

In a distributed database system, the parameters of transaction execution do change. A transaction must execute in such a manner as to preserve its ACID properties. Either all the updates the transaction makes to database items take place, or none at all. Still, as in the case of a centralized environment, a transaction must leave the overall distributed database in a correct and consistent state whether it commits or aborts. A transaction in a distributed environment is expected to perform as efficiently as in a centralized environment.

One major difference, however, is that all the data objects a transaction operates on may not be at a single site in a distributed environment. In fact, in almost all cases of transactions, they operate on database objects dispersed across different physical locations. Figure 18-17 provides a transaction processing example.

In this example, a transaction has to perform updates to employee records stored at three sites. The DDBMS must coordinate the execution of the transaction in cooperation with the local DBMSs at the three sites. Let us trace the execution of the transaction.

- Transaction requests locks or whatever other means available in the local DBMSs to gain shared or exclusive control over the database objects accessed.

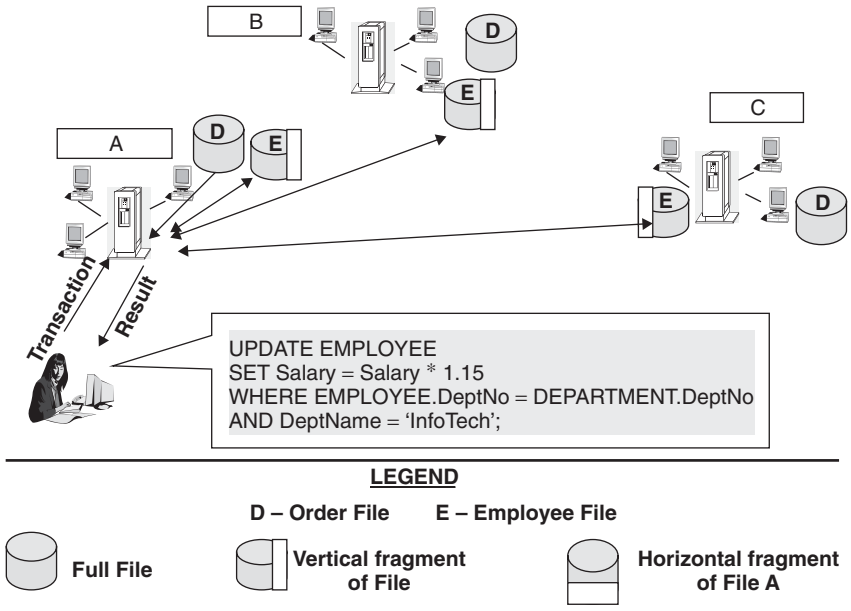


Figure 18-17 Transaction processing example.

- Completes database transactions of reads, writes, and so on.
- When ready to commit, sends "prepare to commit" message to all cooperating DBMSs where the database objects reside.
- Also, sends "prepare to commit" message to all other sites where replicated copies of the updated database objects are held.
- All sites where database updates are being made and those sites with replicated copies send back response messages to transaction initiating site (each response is either "ready to commit" or "not prepared to commit, aborted").
- The execution protocol may allow a transaction to commit only if all participating sites can commit or may permit a transaction to commit if a sufficient number of sites can commit. The coordinating transaction examines the responses and determines the course of action.
- If the coordinating transaction decides to commit, it issues a "commit" message to participating sites.
- Coordinator waits for "commit" responses from participating sites. When all responses are received, coordinator completes processing of the transaction.
- Sends "commit complete" message to participating sites.

If the coordinating transaction decides to abort, it issues messages to abort to all participating sites. The sequence to abort and roll back at every participating site and to complete the abort process is similar to the sequence following the decision to commit.

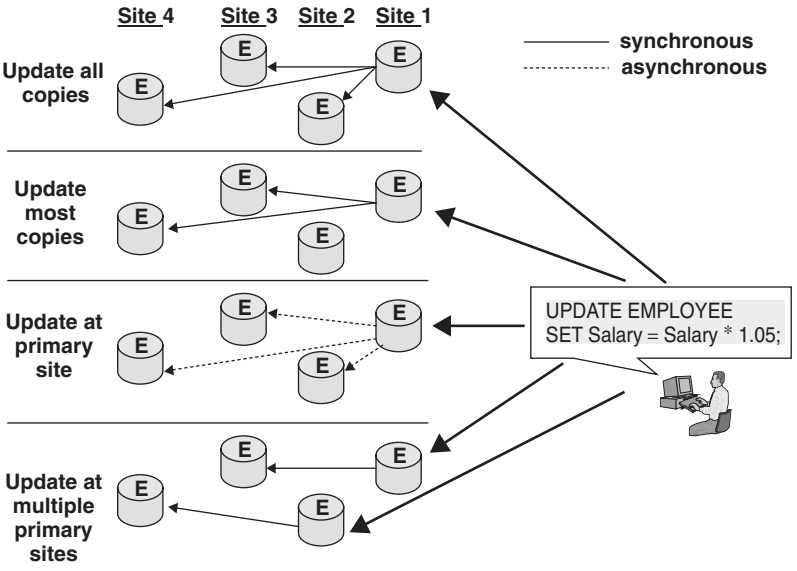


Figure 18-18 Update propagation techniques.

Update Propagation to Replicas In an earlier subsection on replication, we briefly mentioned two major methods for keeping replicated copies of database objects synchronized when updates take place. We covered synchronous and asynchronous replication techniques with an example of updates to an EMPLOYEE relation replicated at different sites. Now let us explore the topic a little further and study a few techniques. Figure 18-18 illustrates four common techniques.

Note the following brief descriptions of the update propagation techniques.

1. *Update all copies*
   This is a synchronous replication method. Locking is managed centrally. The central lock manager coordinates all local lock managers. All copies of the database object are updated before the transaction commits. This method imposes excessive overhead on the central site. This site has to handle all the traffic to coordinate the locking and unlocking tasks.
2. *Update most copies*
   This method is a slight variation of the synchronous replication method. As the central or coordinator site prepares to update its copy synchronously, it issues update requests to all sites where replicated copies are held. If sufficient number of sites respond positively, the coordinator completes the update process. This is similar to majority voting to authorize a process. A number of schemes exist to determine how many sites constitute a majority in a given situation.
3. *Update at primary site*
   One site is designated as the primary site where updates take place. The replicated copy at the primary site is called the primary or master copy. All other

copies at other sites are secondary copies. Only the primary copy is updated synchronously. Transactions cannot update secondary copies. The primary site propagates the updates to secondary copies at a later time. This method reduces the type of traffic at the central site in synchronous replication. Still, the primary site is subject to heavy traffic. Another shortcoming of this method relates to the fact that only data at the primary site are absolutely correct at all times; the other sites are constantly being synchronized with the primary site.

4. *Update at multiple primary sites*
   All or most of the sites holding replicated copies are designated as primary sites. Each of these designated sites is the primary site for some portion of the database being replicated. By spreading the responsibility around, the traffic at each primary site becomes reduced and manageable. However, this method adds another task to the transaction—for each update, the primary site must first be identified before the transaction can proceed.

Concurrency Control

In the above discussion on transaction processing, we concentrated on the execution of a single transaction and how the execution gets coordinated across participating sites. Now let us introduce other transactions and examine concurrency control in a distributed database environment. Concurrency control becomes more complex in a distributed database environment because two or more concurrent transactions may be attempting to simultaneously update two or more copies of the same database object. Therefore, a concurrency control technique must take into account the fragmentation, replication, and data allocation schemes and also how replicated data are being kept synchronized.

Recall the concept of serializability discussed when we were dealing with concurrency control in a centralized database system. If two transactions execute one after the other, this is serial execution. If the effects on the database produced by two concurrent transactions are the same as though they executed serially, then the concurrency control techniques ensure serializability. Locking mechanisms in distributed database systems are also based on the same principle. The concurrency control protocol dictates how locks are obtained and released. In our previous discussion on transaction processing, we ignored locking and unlocking issues and confined the discussion to propagation of updates. Now let us examine lock management in a distributed database environment. Which site manages the locks?

Lock Management Locking and unlocking functions are generally handled in the ways indicated below. As you will note, each option has its advantages and disadvantages.

Locking at primary site. One site is chosen as a primary, central site where locks are held. The lock manager at this site is responsible for all locking and unlocking functions. This is a straightforward option. However, transaction processing at any site will halt if the primary site is down. The whole system is dependent on a particular site being operational.

Locking primary copy. One of the replicated copies of a database object is selected as the primary copy. Locking and unlocking of any copy of the object is the responsibility of the lock manager at the site where the primary copy is held. In this option, if the primary copy is held in one site and the locking is for a copy at a different site, the processing of the relevant transaction requires it to communicate not just to the site where the read or update is intended to take place, but also to the site where the primary copy is held. However, all transaction processing need not stop if one site goes down.

Locking distributed copies. The lock manager at each site is responsible for locking and unlocking of all database objects at that site. When a transaction wants to lock a copy of a database object, it need not communicate with another site just for the locking and unlocking.

Deadlock Resolution Chapter 15, covering the important topic of data integrity, discusses deadlock resolution in a centralized database environment. In that chapter, you learned about the two methods for resolving deadlocks as part of concurrency control: deadlock prevention and deadlock detection.

Recall that deadlock prevention, a more involved technique, is generally applicable to environments where transactions run long and the number of update transactions far exceeds the number of queries that contain read-only operations. A typical database environment used for operating an organization's business generally has an even mixture of update transactions and read-only queries. Furthermore, not too many conflicts arise among concurrent transactions because the applications are streamlined and updates are orderly.

Deadlock detection, on the other hand, is simpler to implement. One of the contending transactions is forced out, and the deadlock gets resolved. The DBMS uses a wait-for graph to detect a deadlock situation and aborts one of the transactions, based on defined criteria, if a deadlock is sensed. You have studied how the DBMS maintains wait-for graphs and uses the technique to detect deadlocks. The other method used by DBMSs to detect deadlocks relies on a timeout technique.

In practice, deadlock detection with wait-for graphs is widely used. In Chapter 15, we discussed this deadlock detection scheme with wait-for graphs for the centralized database system. Now let us reexamine the scheme and note its applicability to the distributed database environment.

In a centralized database environment, there is only one site for the database and the wait-for graph is maintained and used at that site. Transactions execute at only one site. However, in a distributed database environment, transactions operate on database objects at multiple sites. When you inspect the concurrent transactions executing at a single site, some of these emanated from other sites and the rest were initiated locally. Maintenance of wait-for graphs—where and how—poses the greatest challenge.

Let us begin with the idea of each site maintaining a wait-for graph. As you know, the nodes of the graph represent all transactions, both local and otherwise, waiting on local database objects held by other transactions. Assuming a two-site distributed database system, Figure 18-19 presents an example of wait-for graphs at the two sites.

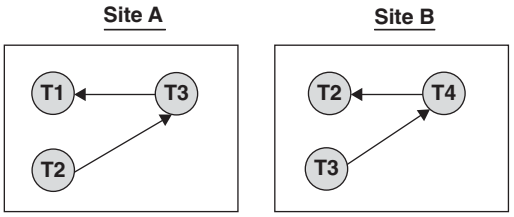


Figure 18-19 Example of local wait-for graph.

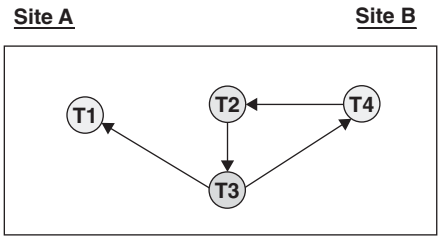


Figure 18-20 Example of global wait-for graph.

Study the two local wait-for graphs carefully. Observe that transactions T2 and T3 appear in both wait-for graphs. That means that these two concurrent transactions wait for database objects at both sites. You note each graph by itself it does not show a cycle, and, therefore, indicates that no deadlock has occurred. But what happens when you combine the two graphs and look at the union of the two? Figure 18-20 presents the global wait-for graphs combining the two individual graphs.

The global wait-for graph readily discloses a cycle in the graph and, therefore, a deadlock condition. But the DDBMS could discern and detect a deadlock only when the local wait-for graphs are combined. How could the wait-for graphs be organized and maintained to detect every deadlock without fail? Let us explore two major schemes for organizing wait-for graphs in a distributed database environment.

Centralized Scheme At specified short intervals, transmit all local wait-for graphs to a centralized site, nominated as the deadlock coordinator. A global wait-for graph is generated at the coordinator site by taking the union of all local wait-for graphs. If the global wait-for graph discloses any cycles, the DDBMS detects a deadlock and takes action. When the DDBMS detects a deadlock and decides on a victim transaction to be aborted, the coordinator informs all concerned sites about who the victim is so that all sites may abort the transaction and roll back any database changes.

Figure 18-20 is an example of a global wait-for graph generated at a deadlock coordinator site.

Occasionally, an additional transaction may be aborted unnecessarily after a deadlock has been detected and a victim chosen. For example, in Figure 18-20 suppose that the DBMS at site A has decided to abort T2 for some unrelated reason

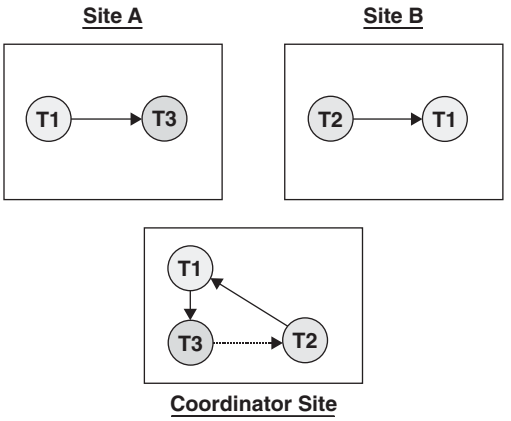


Figure 18-21 Example of a false cycle.

and, at the same time, the coordinator has picked T3 as the victim to be aborted. Then both T2 and T3 will be rolled back, although it is enough that T2 alone is aborted.

Also, sometimes false cycles may be sensed and deadlock resolution may be started unnecessarily. For example, refer to Figure 18-21 illustrating a false cycle in the global wait-for graph. False cycles indicate phantom deadlocks.

Let us say that T3 releases the resource it is holding in site A. A message to "delete" arrow from T1 to T3 is sent to the coordinator. At the same time, assume that T3 requests for a resource held by T2 at site B. This results in an "insert" arrow from T3 to T2 message to be sent to the coordinator. If the "delete" message arrives at the coordinator site a little later than the arrival of the "insert" message, a false cycle is recognized, causing unnecessary initiation of deadlock resolution.

Distributed Scheme As the name of this scheme implies, no global wait-for graph exists at a designated central site. Every site maintains its own wait-for graph. The collection of all the wait-for graphs at the various sites forms the total graph for the distributed database system. If at any time a deadlock is encountered, it is expected that it will show up in at least one of the several wait-for graphs. This is the underlying principle of the distributed scheme.

A wait-for graph under this scheme differs from that under a centralized scheme in one component. A wait-for graph under the distributed scheme contains one extra transaction TO representing transactions that hold resources in other sites. For example, an arrow T1→TO exists in a graph if T1 is waiting for a resource held in another site by any transaction. Similarly, the arrow TO→T1 represents that any transaction at another site is waiting for a resource in the current site held by T1. Figure 18-22 represents the wait-for graphs at sites A and B when you add transaction TO to the wait-for graphs shown in Figure 18-19.

If a local wait-for graphs records a cycle not involving the TO transaction, then you know it represents a deadlock condition. However, what does a cycle with TO transaction as part of it represent? It only implies the possibility of a deadlock, not

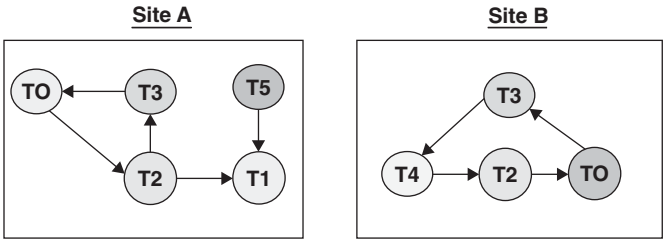


Figure 18-22 Local wait-for graphs with TO transaction.

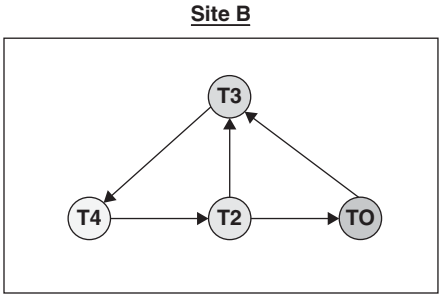


Figure 18-23 Updated local wait-for graph showing deadlock condition.

a certainty. The deadlock detection algorithm in the distributed scheme must ascertain whether a deadlock really exists. Let us see how the algorithm works.

Consider the local wait-for graphs shown in Figure 18-22. At site A, the deadlock detection algorithm senses a cycle containing a TO transaction. On analysis, a determination is made that T3 is waiting for a resource at site B. Site A sends a message to site B with information about the cycle. Site B updates its wait-for graph with the information from site A and creates the wait-for cycle shown in Figure 18-23.

The updated wait-for graph at site B displays a cycle without TO as part of it, indicating a definite deadlock. The technique described here enables you to understand the general idea behind the distributed scheme of wait-for graphs just enough for the scope of our study here.

Distributed Recovery

As expected, recovery from failures has added dimensions of complexity. A distributed database environment faces new kinds of failures not found in centralized systems. One or more remote sites may go down, and these may be the sites at which subqueries or subtransactions are executing. The DDBMS must continue to operate the other sites while the failed sites are brought back up. The communication network forms a major component in the distributed system, and the communications network, completely or in part, may fail and cause interruptions.

When the recovery manager software completes recovery, the distributed database, all parts of it and the replicas, must be left in a consistent and correct state. In

a distributed database environment, typically a transaction gets divided into subtransactions and executed at different sites. So each subtransaction must behave like a complete transaction within a site. To preserve the atomicity of the entire transaction, either all subtransactions should commit or none at all. Whether any failure occurs or not, the atomicity and durability properties must be ensured. These are ensured by means of a proper commit protocol. The method for a centralized environment must be enhanced. The two-phase commit is commonly used as the standard for the distributed environment.

As you know, in a centralized database system the log file contains all the details that can be used to recover from failures. In a distributed database system, each site maintains a separate log file. These log files contain details of the actions performed as part of the commit protocol in addition to the operations of the transactions as in a centralized system. Recovery from failures in a distributed environment depends on such additional details.

We will briefly discuss the commit protocol and the recovery process. We will provide a summarized discussion without getting into too much technical detail. The site from which a transaction originates is known as the coordinator site for the transaction, and all other sites where subtransactions of the original transaction execute are called subordinate or participant sites.

Two-Phase Commit Protocol (2PC) As the name implies, actions in this protocol take place in two phases: a voting phase and a decision or termination phase. The underlying principle is that the coordinator decides to commit a transaction only when all participants vote to commit.

Now let us walk through the execution of a transaction while adopting the protocol.

Voting Phase The user transaction completes processing and decides to commit. Write *begin-commit* record in the log file. Send *prepare* message to each participant.

Termination Phase On receiving a *prepare* message, each participant is either ready to commit or abort based on local conditions. Each participant either writes a *ready-commit* record in the local log file and returns a *ready-commit* message to the coordinator or writes an *abort* message in the local log file and returns an *abort* message to the coordinator.

Case 1. Any participant returns an *abort* vote based on local processing.

Write *abort* record in the log file. Send *global-abort* message to each participant. The participant writes an *abort* record in the local log file and sends an *acknowledgement* to the coordinator. The coordinator, on receiving all *acknowledgement* messages, writes *end-transaction* record in its log file.

Case 2. All participants return *ready-commit* votes.

Write *commit* record in the log file. Send *global-commit* message to each participant. The participant writes a *commit* record in the local log file, commits the subtransaction, releases all locks, and sends an *acknowledgement* to the coordinator.

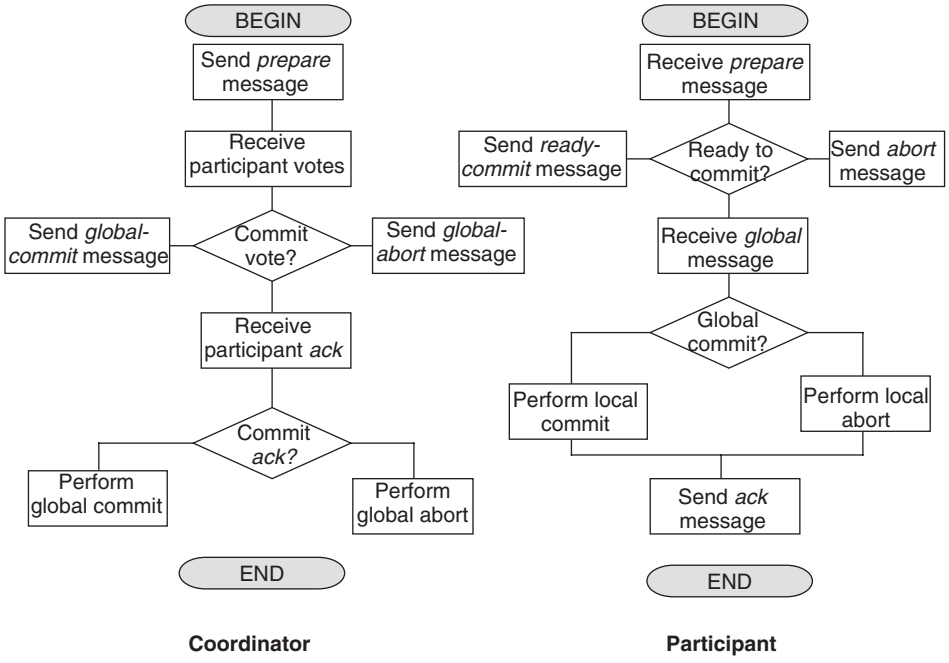


Figure 18-24 Two-phase commit: action states.

The coordinator, on receiving all *acknowledgement* messages, writes *end-transaction* record in its log file.

While these actions take place, the coordinator and the participants go through four states as illustrated in Figure 18-24.

Recovery Consider a site coming back up after a crash. The recovery process of the DDBMS gets initiated, and it reads the local log file. It takes suitable recovery actions by processing all the log records for the transactions that were executing the two-phase commit protocol at the time of the crash. Note that this site could have been the coordinator for some of these transactions and a participant for others.

The following indicates the recovery process in a general manner. This is not a detailed discussion. Assume that transaction T is one of those for which the recovery process applies.

Case 1. *Commit* or *abort* record exists in the log file.

If *commit* record exists, redo T; if *abort* record exists, undo T. If this site happens to be the coordinator for T, resend *commit* or *abort* message to all participants because there may be other site or link failures. When *acknowledgements* are received back from all participants, write *end-transaction* record in the log file.

Case 2. *Prepare* record exists, but no *commit* or *abort* record in the log file.

This site is a participant. Find coordinator site from *prepare* record and contact coordinator for status of T. If coordinator responds with a *commit* message, write

corresponding log record, and redo T. If coordinator responds with an *abort* message, write corresponding log record, and undo T. Then write *end* log record for T.

Case 3. No *prepare*, *commit*, or *abort* record in the log file.

Abort and undo T. Write end log record for T. In this case, we safely assume the site could not have voted to commit T before the crash.

Three-Phase Commit Protocol (3PC) Think of the effect of a crash on the participants if the coordinator fails. In the above discussion of transaction T, participants that have voted to commit cannot decide on the course of action—whether to commit or abort—until the coordinating site recovers. That means T is blocked. However, the active participants can talk to one another and check whether at least one of them has an *abort* or *commit* log record.

The three-phase commit protocol is intended to avoid blocking even if the coordinator site fails during recovery. This protocol postpones the decision to commit until it is ensured that a sufficient number of sites know about the decision to commit. The idea is that these participant sites could communicate among themselves and find out about the decision to commit or abort, even if the coordinator site fails. Here, briefly, is how the protocol works:

- Coordinator sends prepare message to each participant.
- When coordinator receives *ready-commit* messages back from all participants, it sends out *precommit* messages instead of *commit* messages.
- When coordinator receives back sufficient number of *acknowledgements* back from the participants, it writes a *commit* record in its log file and sends out *commit* messages.

CHAPTER SUMMARY

- Distributed database systems are emerging as a result of the merger of two major technologies: database and network.
- A distributed database features geographic separation of data but preserves logical unity and integrity. Each data fragment is stored at the site where it is most frequently used.
- A distributed database management system (DDBMS) is a software system that manages the distributed database and makes the distribution transparent to the user.
- Motivation and goals of a distributed system: efficient access of local data, improved customer service, enhanced reliability, availability of global data. A distributed system offers several advantages but also has a few shortcomings.
- Two broad types of distributed database systems: homogeneous, in which all sites run under the same DBMS; heterogeneous, in which a different DBMS may manage the data at each site.
- For a distributed database, DDBMS supports several schema levels: global external, global conceptual, data fragmentation, data allocation, local external, local conceptual, and local internal.

- The system catalog or data dictionary resides in one place for a centralized database. However, for a distributed database, a few options are available for placing the catalog: at a central site, at all sites fully or partially, and a hybrid option.
- Data distribution methods: data fragmentation (horizontal and vertical partitioning) and data replication.
- Computing architectural options: two-tier client/server, two-tier intelligent server, and three-tier architecture with middleware.
- Design and implementation issues focus on fragmentation and allocation of data to the various sites.
- The key motivation for the DDBMS is transparency or hiding complexities of data distribution and making it transparent to users. Transparencies provided at different levels: fragmentation, replication, location, network, naming, and failure.
- A query in a distributed environment typically gets divided into subqueries and executed at the sites where the pertinent data are stored. Numerous additional factors must be considered for query optimization.
- Transaction execution and propagation of updates to various replicas are more difficult. Lock management and concurrency control pose many challenges.
- The two-phase commit protocol (2PC) is the industry standard to enable distributed recovery.

REVIEW QUESTIONS

1. Briefly describe the basic concepts of a distributed database system.
2. List five advantages and five disadvantages of distributed databases.
3. Compare and contrast homogeneous and heterogeneous systems.
4. What are the options for storing the system catalog or data dictionary?
5. Name any three of the network configurations for a distributed database system. Describe one of these.
6. What is data fragmentation? Describe it using an example of a CUSTOMER relation.
7. Name three types of transparencies provided by a DDBMS. Describe any one.
8. List the four common techniques for update propagation.
9. Describe the essential differences between centralized and distributed schemes for using wait-for graphs.
10. What is the underlying principle of the two-phase commit protocol?

EXERCISES

1. Indicate whether true or false:
   A. In a distributed database system, each site must have autonomy to manage its data.

B. In a three-tier architecture, middleware performs the database functions.
C. Location transparency relates to how data are fragmented for storing at locations.
D. Hybrid data partitioning combines horizontal and vertical partitioning methods.
E. Locking primary copy technique—all locks for all database objects are kept at one site.
F. A global wait-for graph is created at the coordinator site.
G. The two-phase commit protocol requires a transaction to commit in the voting phase.
H. Atomicity and durability properties of a transaction must be preserved in a distributed database environment.
I. A heterogeneous system must have a different DBMS at every site.
J. The ring network configuration connects sites in a closed loop.

2. An international bank has 10 domestic sites and 6 international sites. A homogeneous distributed database system supports the bank's business. Consider options to distribute the customer data and customer account data among the bank's sites. Also, discuss the schema definitions at the various levels for the distributed database.
3. Using an example, describe synchronous and asynchronous data replication in a distributed database environment.
4. How are locks for concurrency control managed in a distributed database? Discuss the options.
5. You are the senior DBA for an international distributor of electronic components with 60 domestic sales offices and 10 international offices in the European Union countries. Develop a very general plan for a distributed database system. Without getting into technical details, include data distribution strategy, network configuration, system architecture, concurrency control techniques, and recovery methods.