12

TWELVE

Distributed Database Management Systems

In this chapter, you will learn:

- What a distributed database management system (DDBMS) is and what its components are
- How database implementation is affected by different levels of data and process distribution
- How transactions are managed in a distributed database environment
- How database design is affected by the distributed database environment

Preview

In this chapter, you learn that a single database can be divided into several fragments. The fragments can be stored on different computers within a network. Processing, too, can be dispersed among several different network sites, or nodes. The multisite database forms the core of the distributed database system.

The growth of distributed database systems has been fostered by the dispersion of business operations across the country and the world, along with the rapid pace of technological change that has made local and wide area networks practical and more reliable. The network-based distributed database system is very flexible: it can serve the needs of a small business operating two stores in the same town while at the same time meeting the needs of a global business.

Although a distributed database system requires a more sophisticated DBMS, the end user should not be burdened by increased operational complexity. That is, the greater complexity of a distributed database system should be transparent to the end user.

The distributed database management system (DDBMS) treats a distributed database as a single logical database; therefore, the basic design concepts you learned in earlier chapters apply. However, although the end user need not be aware of the distributed database's special characteristics, the distribution of data among different sites in a computer network clearly adds to a system's complexity. For example, the design of a distributed database must consider the location of the data and the partitioning of the data into database fragments. You examine such issues in this chapter.

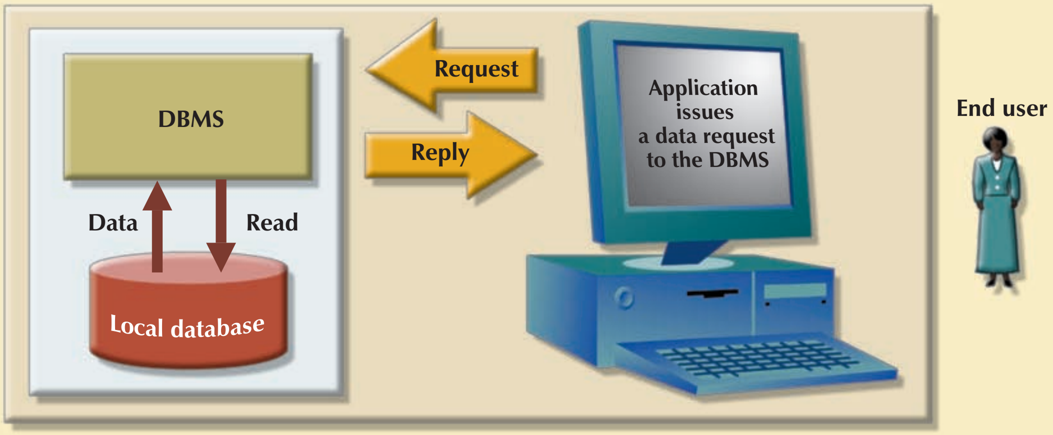
12.1 THE EVOLUTION OF DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

A **distributed database management system** (**DDBMS**) governs the storage and processing of logically related data over interconnected computer systems in which both data and processing functions are distributed among several sites. To understand how and why the DDBMS is different from the DBMS, it is useful to briefly examine the changes in the business environment that set the stage for the development of the DDBMS.

During the 1970s, corporations implemented centralized database management systems to meet their structured information needs. Structured information is usually presented as regularly issued formal reports in a standard format. Such information, generated by procedural programming languages, is created by specialists in response to precisely channeled requests. Thus, structured information needs are well served by centralized systems.

The use of a centralized database required that corporate data be stored in a single central site, usually a mainframe computer. Data access was provided through dumb terminals. The centralized approach, illustrated in Figure 12.1, worked well to fill the structured information needs of corporations, but it fell short when quickly moving events required faster response times and equally quick access to information. The slow progression from information request to approval, to specialist, to user simply did not serve decision makers well in a dynamic environment. What was needed was quick, unstructured access to databases, using ad hoc queries to generate on-the-spot information.

FIGURE 12.1 Centralized database management system



Database management systems based on the relational model could provide the environment in which unstructured information needs would be met by employing ad hoc queries. End users would be given the ability to access data when needed. Unfortunately, the early relational model implementations did not yet deliver acceptable throughput when compared to the well-established hierarchical or network database models.

The last two decades gave birth to a series of crucial social and technological changes that affected database development and design. Among those changes were:

- Business operations became decentralized.
- Competition increased at the global level.
- Customer demands and market needs favored a decentralized management style.

- Rapid technological change created low-cost computers with mainframe-like power, impressive multifunction handheld portable wireless devices with cellular phone and data services, and increasingly complex and fast networks to connect them. As a consequence, corporations have increasingly adopted advanced network technologies as the platform for their computerized solutions.
- The large number of applications based on DBMSs and the need to protect investments in centralized DBMS software made the notion of data sharing attractive. Data realms are converging in the digital world more and more. As a result, single applications manage multiple different types of data (voice, video, music, images, etc.), and such data are accessed from multiple geographically dispersed locations.

Those factors created a dynamic business environment in which companies had to respond quickly to competitive and technological pressures. As large business units restructured to form leaner and meaner, quickly reacting, dispersed operations, two database requirements became obvious:

- *Rapid ad hoc data access* became crucial in the quick-response decision-making environment.
- *The decentralization of management structures* based on the decentralization of business units made decentralized multiple-access and multiple-location databases a necessity.

During recent years, the factors just described became even more firmly entrenched. However, the way those factors were addressed was strongly influenced by:

- *The growing acceptance of the Internet as the platform for data access and distribution*. The World Wide Web (WWW or just the Web) is, in effect, the *repository* for distributed data.
- *The wireless revolution*. The widespread use of wireless digital devices, such as personal digital assistants (PDAs) like Palm and BlackBerry and multipurpose "smart phones" like the iPhone, has created high demand for data access. Such devices access data from geographically dispersed locations and require varied data exchanges in multiple formats (data, voice, video, music, pictures, etc.) Although distributed data access does not necessarily imply distributed databases; performance and failure tolerance requirements often make use of data replication techniques similar to the ones found in distributed databases.
- *The accelerated growth of companies providing "application as a service" type of services*. This new type of service provides remote application services to companies wanting to outsource their application development, maintenance, and operations. The company data is generally stored on central servers and is not necessarily distributed. Just as with wireless data access, this type of service may not require fully distributed data functionality; however, other factors such as performance and failure tolerance often require the use of data replication techniques similar to the ones found in distributed databases.
- *The increased focus on data analysis that led to data mining and data warehousing*. Although a data warehouse is not usually a distributed database, it does rely on techniques such as data replication and distributed queries that facilitate data extraction and integration. (Data warehouse design, implementation, and use are discussed in Chapter 13, Business Intelligence and Data Warehouses.)

ONLINE CONTENT

To learn more about the Internet's impact on data access and distribution, see **Appendix I** in the Student Online Companion, **Databases in Electronic Commerce.**

At this point, the long-term impact of the Internet and the wireless revolution on *distributed* database design and management is still unclear. Perhaps the success of the Internet and wireless technologies will foster the use of distributed databases as bandwidth becomes a more troublesome bottleneck. Perhaps the resolution of bandwidth problems will simply confirm the centralized database standard. In any case, distributed databases exist today and many distributed database operating concepts and components are likely to find a place in future database developments.

The decentralized database is especially desirable because centralized database management is subject to problems such as:

- *Performance degradation* due to a growing number of remote locations over greater distances.

- *High costs* associated with maintaining and operating large central (mainframe) database systems.
- *Reliability problems* created by dependence on a central site (single point of failure syndrome) and the need for data replication.
- *Scalability problems* associated with the physical limits imposed by a single location (power, temperature conditioning, and power consumption.)
- *Organizational rigidity* imposed by the database might not support the flexibility and agility required by modern global organizations.

The dynamic business environment and the centralized database's shortcomings spawned a demand for applications based on accessing data from different sources at multiple locations. Such a multiple-source/multiple-location database environment is best managed by a distributed database management system (DDBMS).

12.2 DDBMS ADVANTAGES AND DISADVANTAGES

Distributed database management systems deliver several advantages over traditional systems. At the same time, they are subject to some problems. Table 12.1 summarizes the advantages and disadvantages associated with a DDBMS.

TABLE 12.1 Distributed DBMS Advantages and Disadvantages

ADVANTAGES	DISADVANTAGES
• *Data are located near the greatest demand site*. The data in a distributed database system are dispersed to match business requirements. • *Faster data access*. End users often work with only a locally stored subset of the company's data. • *Faster data processing*. A distributed database system spreads out the systems workload by processing data at several sites. • *Growth facilitation*. New sites can be added to the network without affecting the operations of other sites. • *Improved communications*. Because local sites are smaller and located closer to customers, local sites foster better communication among departments and between customers and company staff. • *Reduced operating costs*. It is more cost-effective to add workstations to a network than to update a mainframe system. Development work is done more cheaply and more quickly on low-cost PCs than on mainframes. • *User-friendly interface*. PCs and workstations are usually equipped with an easy-to-use graphical user interface (GUI). The GUI simplifies training and use for end users. • *Less danger of a single-point failure*. When one of the computers fails, the workload is picked up by other workstations. Data are also distributed at multiple sites. • *Processor independence*. The end user is able to access any available copy of the data, and an end user's request is processed by any processor at the data location.	• *Complexity of management and control*. Applications must recognize data location, and they must be able to stitch together data from various sites. Database administrators must have the ability to coordinate database activities to prevent database degradation due to data anomalies. • *Technological difficulty*. Data integrity, transaction management, concurrency control, security, backup, recovery, query optimization, access path selection, and so on, must all be addressed and resolved. • *Security*. The probability of security lapses increases when data are located at multiple sites. The responsibility of data management will be shared by different people at several sites. • *Lack of standards*. There are no standard communication protocols *at the database level*. (Although TCP/IP is the de facto standard at the network level, there is no standard at the application level.) For example, different database vendors employ different—and often incompatible—techniques to manage the distribution of data and processing in a DDBMS environment. • *Increased storage and infrastructure requirements*. Multiple copies of data are required at different sites, thus requiring additional disk storage space. • *Increased training cost*. Training costs are generally higher in a distributed model than they would be in a centralized model, sometimes even to the extent of offsetting operational and hardware savings. • *Costs*. Distributed databases require duplicated infrastructure to operate (physical location, environment, personnel, software, licensing, etc.)

Distributed databases are used successfully but have a long way to go before they will yield the full flexibility and power of which they are theoretically capable. The inherently complex distributed data environment increases the urgency for standard protocols governing transaction management, concurrency control, security, backup, recovery, query optimization, access path selection, and so on. Such issues must be addressed and resolved before DDBMS technology is widely embraced.

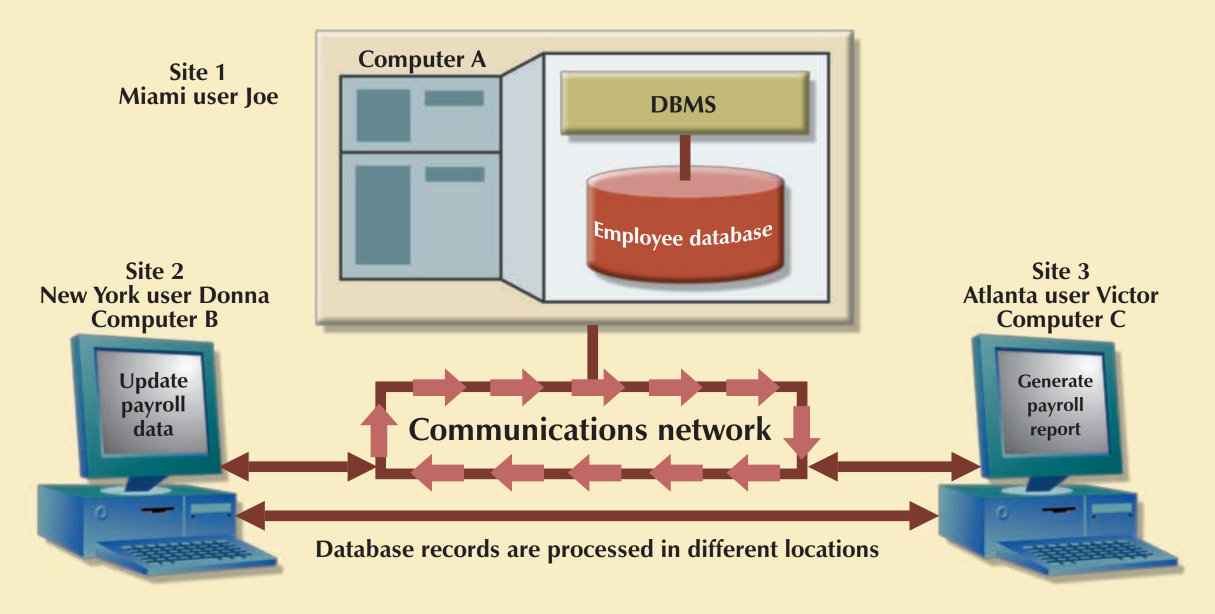
The remainder of this chapter explores the basic components and concepts of the distributed database. Because the distributed database is usually based on the relational database model, relational terminology is used to explain the basic concepts and components of a distributed database.

12.3 DISTRIBUTED PROCESSING AND DISTRIBUTED DATABASES

In **distributed processing**, a database's logical processing is shared among two or more physically independent sites that are connected through a network. For example, the data input/output (I/O), data selection, and data validation might be performed on one computer, and a report based on that data might be created on another computer.

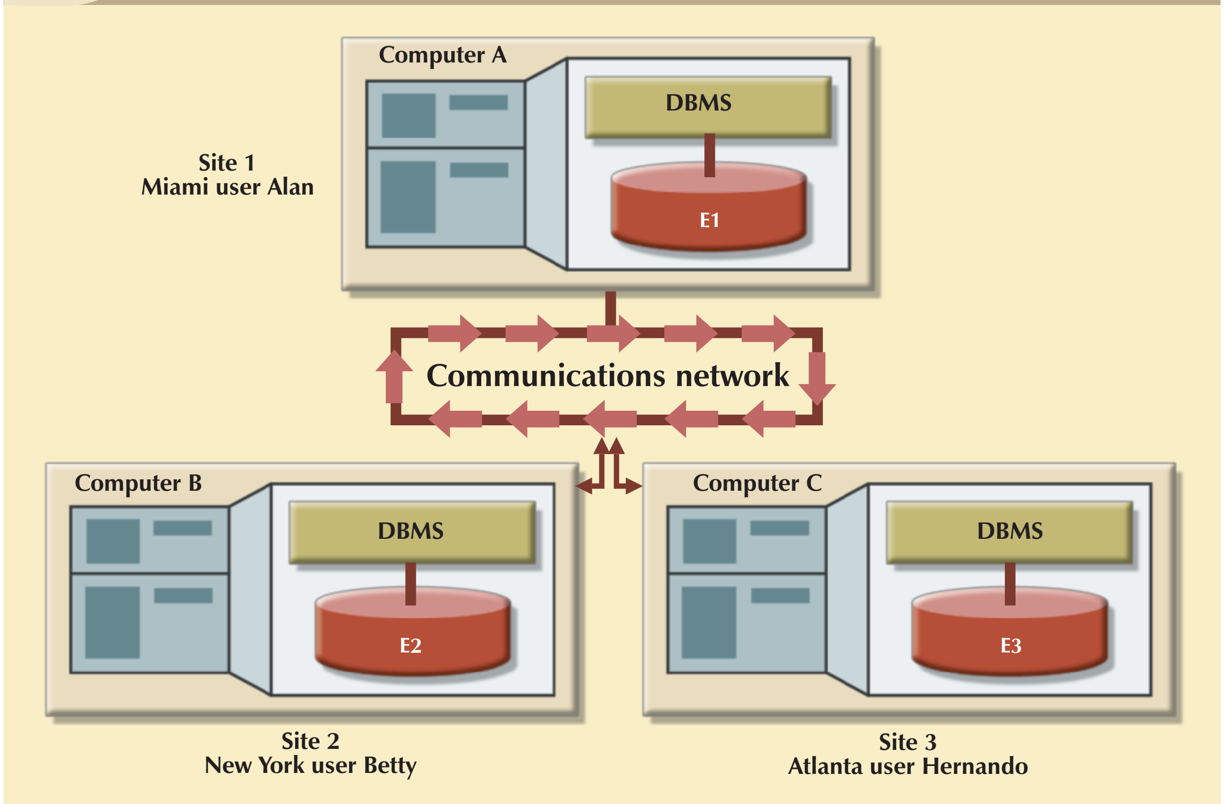
A basic distributed processing environment is illustrated in Figure 12.2, which shows that a distributed processing system shares the database processing chores among three sites connected through a communications network. Although the database resides at only one site (Miami), each site can access the data and update the database. The database is located on Computer A, a network computer known as the *database server*.

FIGURE 12.2 Distributed processing environment



A **distributed database**, on the other hand, stores a logically related database over two or more physically independent sites. The sites are connected via a computer network. In contrast, the distributed processing system uses only a single-site database but shares the processing chores among several sites. In a distributed database system, a database is composed of several parts known as **database fragments**. The database fragments are located at different sites and can be replicated among various sites. Each database fragment is, in turn, managed by its local database process. An example of a distributed database environment is shown in Figure 12.3.

FIGURE 12.3 Distributed database environment



The database in Figure 12.3 is divided into three database fragments (E1, E2, and E3) located at different sites. The computers are connected through a network system. In a fully distributed database, the users Alan, Betty, and Hernando do not need to know the name or location of each database fragment in order to access the database. Also, the users might be located at sites other than Miami, New York, or Atlanta, and still be able to access the database as a single logical unit.

As you examine Figures 12.2 and 12.3, you should keep the following points in mind:

- Distributed processing does not require a distributed database, but a distributed database requires distributed processing (each database fragment is managed by its own local database process).
- Distributed processing may be based on a single database located on a single computer. For the management of distributed data to occur, copies or parts of the database processing functions must be distributed to all data storage sites.
- Both distributed processing and distributed databases require a network to connect all components.

12.4 CHARACTERISTICS OF DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

A DDBMS governs the storage and processing of logically related data over interconnected computer systems in which both data and processing functions are distributed among several sites. A DBMS must have at least the following functions to be classified as distributed:

- *Application interface* to interact with the end user, application programs, and other DBMSs within the distributed database.
- *Validation* to analyze data requests for syntax correctness.
- *Transformation* to decompose complex requests into atomic data request components.
- *Query optimization* to find the best access strategy. (Which database fragments must be accessed by the query, and how must data updates, if any, be synchronized?)
- *Mapping* to determine the data location of local and remote fragments.
- *I/O interface* to read or write data from or to permanent local storage.
- *Formatting* to prepare the data for presentation to the end user or to an application program.
- *Security* to provide data privacy at both local and remote databases.
- *Backup and recovery* to ensure the availability and recoverability of the database in case of a failure.
- *DB administration features* for the database administrator.
- *Concurrency control* to manage simultaneous data access and to ensure data consistency across database fragments in the DDBMS.
- *Transaction management* to ensure that the data moves from one consistent state to another. This activity includes the synchronization of local and remote transactions as well as transactions across multiple distributed segments.

A fully distributed database management system must perform all of the functions of a centralized DBMS, as follows:

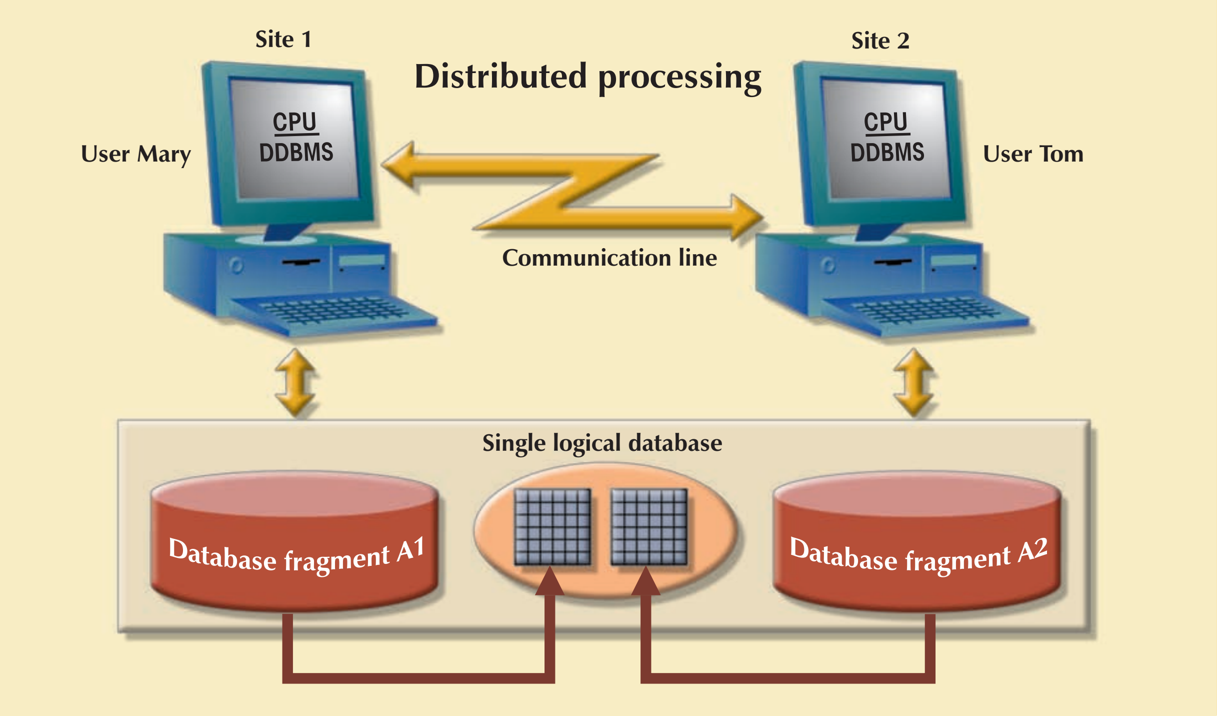
1. Receive an application's (or an end user's) request.
2. Validate, analyze, and decompose the request. The request might include mathematical and/or logical operations such as the following: Select all customers with a balance greater than $1,000. The request might require data from only a single table, or it might require access to several tables.
3. Map the request's logical-to-physical data components.
4. Decompose the request into several disk I/O operations.
5. Search for, locate, read, and validate the data.
6. Ensure database consistency, security, and integrity.
7. Validate the data for the conditions, if any, specified by the request.
8. Present the selected data in the required format.

In addition, a distributed DBMS must handle all necessary functions imposed by the distribution of data and processing. And it must perform those additional functions *transparently* to the end user. The DDBMS's transparent data access features are illustrated in Figure 12.4.

The single logical database in Figure 12.4 consists of two database fragments, A1 and A2, located at sites 1 and 2, respectively. Mary can query the database as if it were a local database; so can Tom. Both users "see" only one logical database and *do not need to know the names of the fragments*. In fact, the end users do not even need to know that the database is divided into fragments, *nor do they need to know where the fragments are located*.

To better understand the different types of distributed database scenarios, let's first define the distributed database system's components.

FIGURE 12.4 A fully distributed database management system



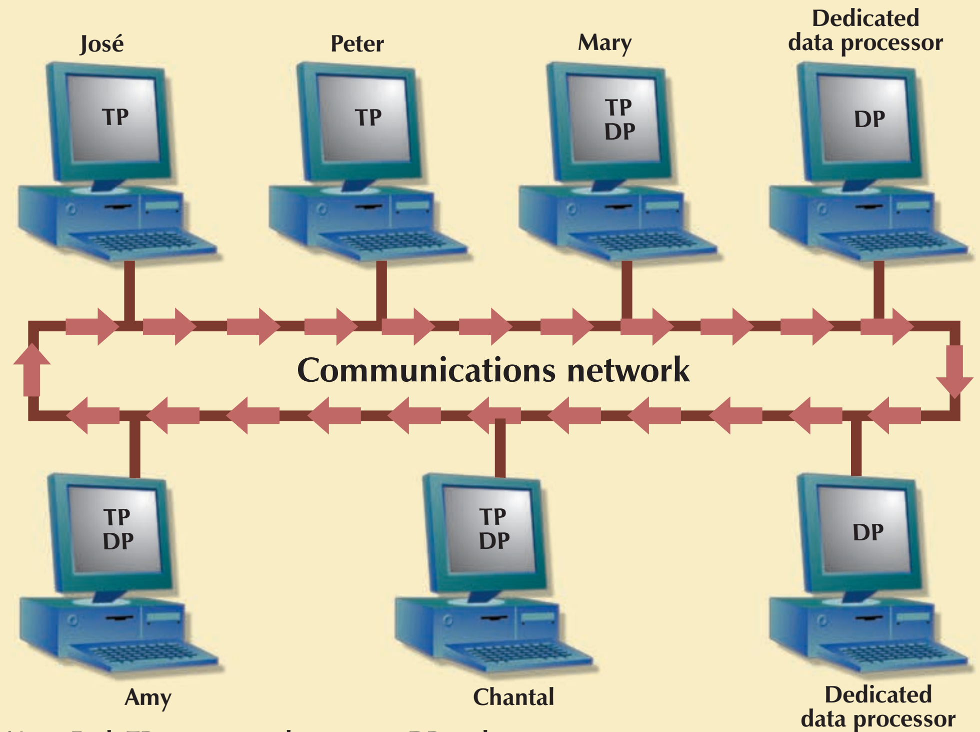
12.5 DDBMS COMPONENTS

The DDBMS must include at least the following components:

- *Computer workstations* (sites or nodes) that form the network system. The distributed database system must be independent of the computer system hardware.
- *Network hardware and software* components that reside in each workstation. The network components allow all sites to interact and exchange data. Because the components—computers, operating systems, network hardware, and so on—are likely to be supplied by different vendors, it is best to ensure that distributed database functions can be run on multiple platforms.
- *Communications media* that carry the data from one workstation to another. The DDBMS must be communications-media-independent; that is, it must be able to support several types of communications media.
- The **transaction processor** (**TP**), which is the software component found in each computer that requests data. The transaction processor receives and processes the application's data requests (remote and local). The TP is also known as the **application processor** (**AP**) or the **transaction manager** (**TM**).
- The **data processor** (**DP**), which is the software component residing on each computer that stores and retrieves data located at the site. The DP is also known as the **data manager** (**DM**). A data processor may even be a centralized DBMS.

Figure 12.5 illustrates the placement of the components and the interaction among them. The communication among TPs and DPs shown in Figure 12.5 is made possible through a specific set of rules, or *protocols*, used by the DDBMS.

FIGURE 12.5 Distributed database system management components



Note: Each TP can access data on any DP, and each DP handles all requests for local data from any TP.

The protocols determine how the distributed database system will:

- Interface with the network to transport data and commands between data processors (DPs) and transaction processors (TPs).
- Synchronize all data received from DPs (TP side) and route retrieved data to the appropriate TPs (DP side).
- Ensure common database functions in a distributed system. Such functions include security, concurrency control, backup, and recovery.

DPs and TPs can be added to the system without affecting the operation of the other components. A TP and a DP can reside on the same computer, allowing the end user to access local as well as remote data transparently. In theory, a DP can be an independent centralized DBMS with proper interfaces to support remote access from other independent DBMSs in the network.

12.6 LEVELS OF DATA AND PROCESS DISTRIBUTION

Current database systems can be classified on the basis of how process distribution and data distribution are supported. For example, a DBMS may store data in a single site (centralized DB) or in multiple sites (distributed DB) and may support data processing at a single site or at multiple sites. Table 12.2 uses a simple matrix to classify database systems according to data and process distribution. These types of processes are discussed in the sections that follow.

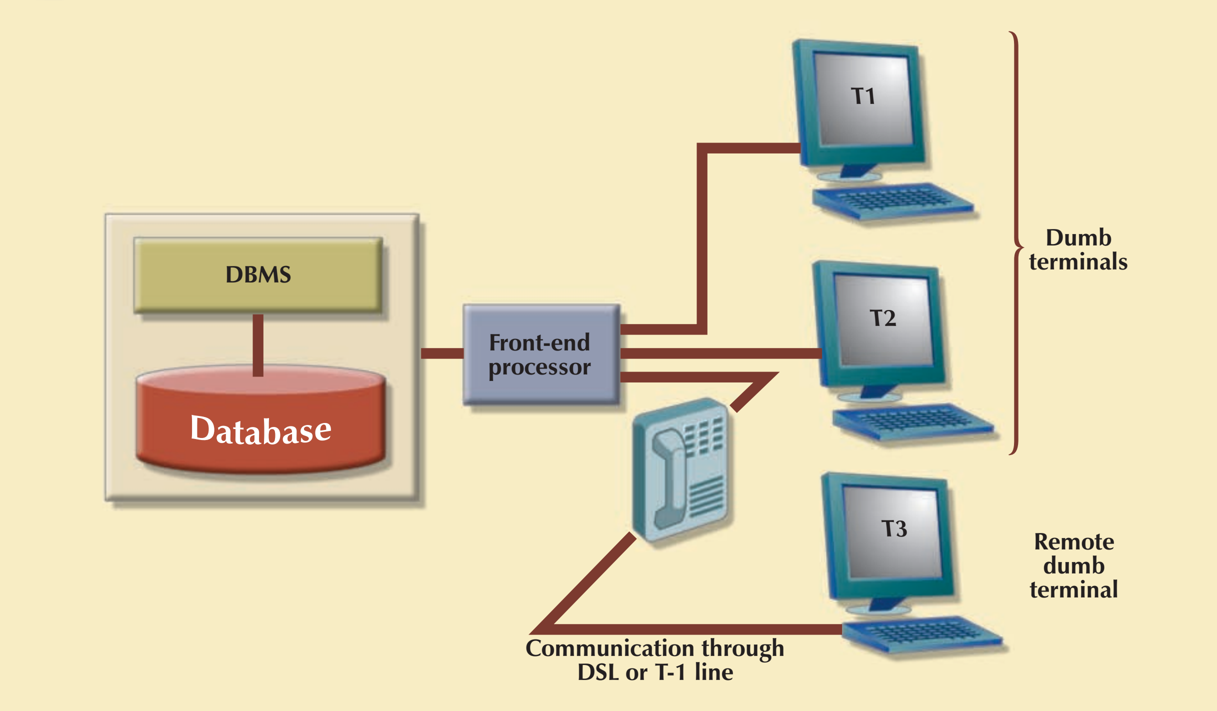
TABLE 12.2 Database Systems: Levels of Data and Process Distribution

	SINGLE-SITE DATA	MULTIPLE-SITE DATA
Single-site process	Host DBMS	Not applicable (Requires multiple processes)
Multiple-site process	File server Client/server DBMS (LAN DBMS)	Fully distributed Client/server DDBMS

12.6.1 Single-Site Processing, Single-Site Data (SPSD)

In the **single-site processing**, **single-site data** (**SPSD**) scenario, all processing is done on a single host computer (single-processor server, multiprocessor server, mainframe system) and all data are stored on the host computer's local disk system. Processing cannot be done on the end user's side of the system. Such a scenario is typical of most mainframe and midrange server computer DBMSs. The DBMS is located on the host computer, which is accessed by dumb terminals connected to it. See Figure 12.6. This scenario is also typical of the first generation of single-user microcomputer databases.

FIGURE 12.6 Single-site processing, single-site data (centralized)

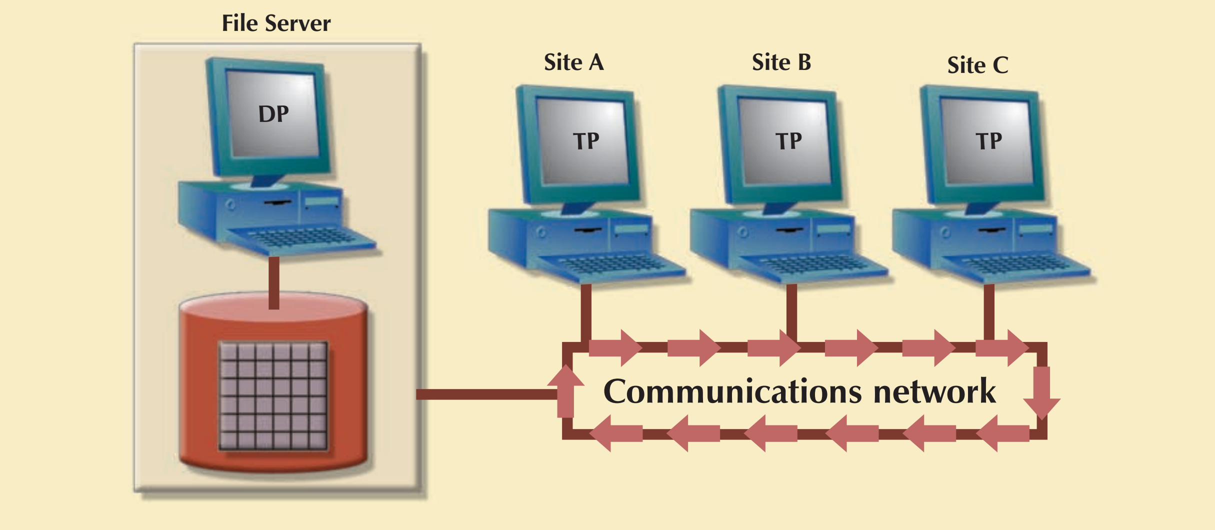


Using Figure 12.6 as an example, you can see that the functions of the TP and the DP are embedded within the DBMS located on a single computer. The DBMS usually runs under a time-sharing, multitasking operating system, which allows several processes to run concurrently on a host computer accessing a single DP. All data storage and data processing are handled by a single host computer.

12.6.2 Multiple-Site Processing, Single-Site Data (MPSD)

Under the **multiple-site processing**, **single-site data** (**MPSD**) scenario, multiple processes run on different computers sharing a single data repository. Typically, the MPSD scenario requires a network file server running conventional applications that are accessed through a network. Many multiuser accounting applications running under a personal computer network fit such a description. (See Figure 12.7.)

FIGURE 12.7 Multiple-site processing, single-site data



As you examine Figure 12.7, note that:

- The TP on each workstation acts only as a redirector to route all network data requests to the file server.
- The end user sees the file server as just another hard disk. Because only the data storage input/output (I/O) is handled by the file server's computer, the MPSD offers limited capabilities for distributed processing.
- The end user must make a direct reference to the file server in order to access remote data. All record- and file-locking activities are done at the end-user location.
- All data selection, search, and update functions take place at the workstation, thus requiring that entire files travel through the network for processing at the workstation. Such a requirement increases network traffic, slows response time, and increases communication costs.

The inefficiency of the last condition can be illustrated easily. For example, suppose the file server computer stores a CUSTOMER table containing 10,000 data rows, 50 of which have balances greater than $1,000. Suppose site A issues the following SQL query:

```
SELECT     *
FROM       CUSTOMER
WHERE      CUS_BALANCE > 1000;
```

All 10,000 CUSTOMER rows must travel through the network to be evaluated at site A.

A variation of the multiple-site processing, single-site data approach is known as client/server architecture. **Client/server architecture** is similar to that of the network file server *except that all database processing is done at the server site, thus reducing network traffic*. Although both the network file server and the client/server systems perform multiple-site processing, the latter's processing is distributed. Note that the network file server approach requires the database to be located at a single site. In contrast, the client/server architecture is capable of supporting data at multiple sites.

ONLINE CONTENT

Appendix F, Client/Server Systems, is located in the Student Online Companion for this book.

12.6.3 Multiple-Site Processing, Multiple-Site Data (MPMD)

The **multiple-site processing**, **multiple-site data** (**MPMD**) scenario describes a fully distributed DBMS with support for multiple data processors and transaction processors at multiple sites. Depending on the level of support for various types of centralized DBMSs, DDBMSs are classified as either homogeneous or heterogeneous.

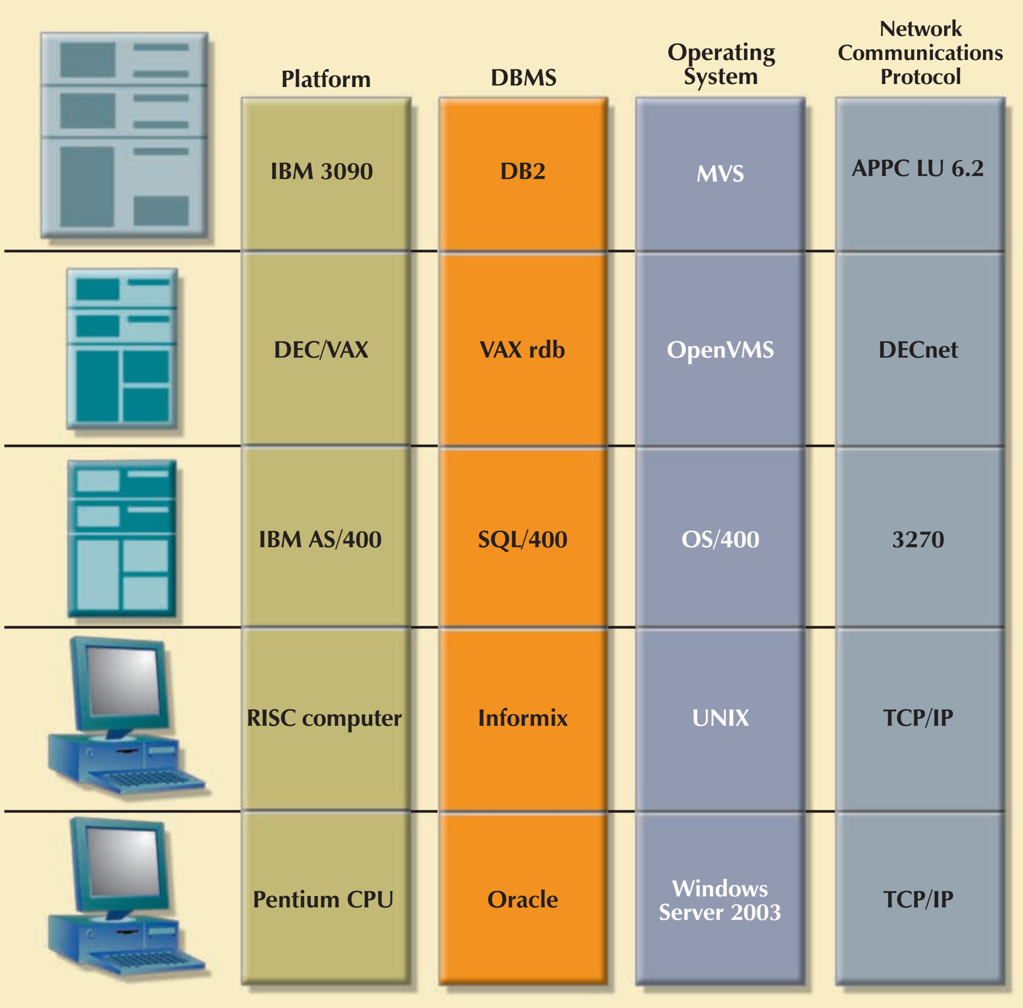
Homogeneous DDBMSs integrate only one type of centralized DBMS over a network. Thus, the same DBMS will be running on different server platforms (single processor server, multi-processor server, server farms, or server blades). In contrast, **heterogeneous DDBMSs** integrate different types of centralized DBMSs over a network. See Figure 12.8. A **fully heterogeneous DDBMS** will support different DBMSs that may even support different data models (relational, hierarchical, or network) running under different computer systems, such as mainframes and PCs.

Some DDBMS implementations support several platforms, operating systems, and networks and allow remote data access to another DBMS. However, such DDBMSs still are subject to certain restrictions. For example:

- Remote access is provided on a read-only basis and does not support write privileges.
- Restrictions are placed on the number of remote tables that may be accessed in a single transaction.
- Restrictions are placed on the number of distinct databases that may be accessed.
- Restrictions are placed on the database model that may be accessed. Thus, access may be provided to relational databases but not to network or hierarchical databases.

The preceding list of restrictions is by no means exhaustive. The DDBMS technology continues to change rapidly, and new features are added frequently. Managing data at multiple sites leads to a number of issues that must be addressed and understood. The next section will examine several key features of distributed database management systems.

FIGURE 12.8 Heterogeneous distributed database scenario



Platform	DBMS	Operating System	Network Communications Protocol
IBM 3090	DB2	MVS	APPC LU 6.2
DEC/VAX	VAX rdb	OpenVMS	DECnet
IBM AS/400	SQL/400	OS/400	3270
RISC computer	Informix	UNIX	TCP/IP
Pentium CPU	Oracle	Windows Server 2003	TCP/IP

12.7 DISTRIBUTED DATABASE TRANSPARENCY FEATURES

A distributed database system requires functional characteristics that can be grouped and described as transparency features. DDBMS transparency features have the common property of allowing the end user to feel like the database's only user. In other words, the user believes that (s)he is working with a centralized DBMS; all complexities of a distributed database are hidden, or transparent, to the user.

The DDBMS transparency features are:

- **Distribution transparency**, which allows a distributed database to be treated as a single logical database. If a DDBMS exhibits distribution transparency, the user does not need to know:
  - That the data are partitioned—meaning the table's rows and columns are split vertically or horizontally and stored among multiple sites.
  - That the data can be replicated at several sites.
  - The data location.

- **Transaction transparency**, which allows a transaction to update data at more than one network site. Transaction transparency ensures that the transaction will be either entirely completed or aborted, thus maintaining database integrity.
- **Failure transparency**, which ensures that the system will continue to operate in the event of a node failure. Functions that were lost because of the failure will be picked up by another network node.
- **Performance transparency**, which allows the system to perform as if it were a centralized DBMS. The system will not suffer any performance degradation due to its use on a network or due to the network's platform differences. Performance transparency also ensures that the system will find the most cost-effective path to access remote data.
- **Heterogeneity transparency**, which allows the integration of several different local DBMSs (relational, network, and hierarchical) under a common, or global, schema. The DDBMS is responsible for translating the data requests from the global schema to the local DBMS schema.

Distribution, transaction, and performance transparency features will be examined in greater detail in the next few sections.

12.8 DISTRIBUTION TRANSPARENCY

Distribution transparency allows a physically dispersed database to be managed as though it were a centralized database. The level of transparency supported by the DDBMS varies from system to system. Three levels of distribution transparency are recognized:

- **Fragmentation transparency** is the highest level of transparency. The end user or programmer does not need to know that a database is partitioned. Therefore, neither fragment names nor fragment locations are specified prior to data access.
- **Location transparency** exists when the end user or programmer must specify the database fragment names but does not need to specify where those fragments are located.
- **Local mapping transparency** exists when the end user or programmer must specify both the fragment names and their locations.

Transparency features are summarized in Table 12.3.

TABLE 12.3 A Summary of Transparency Features

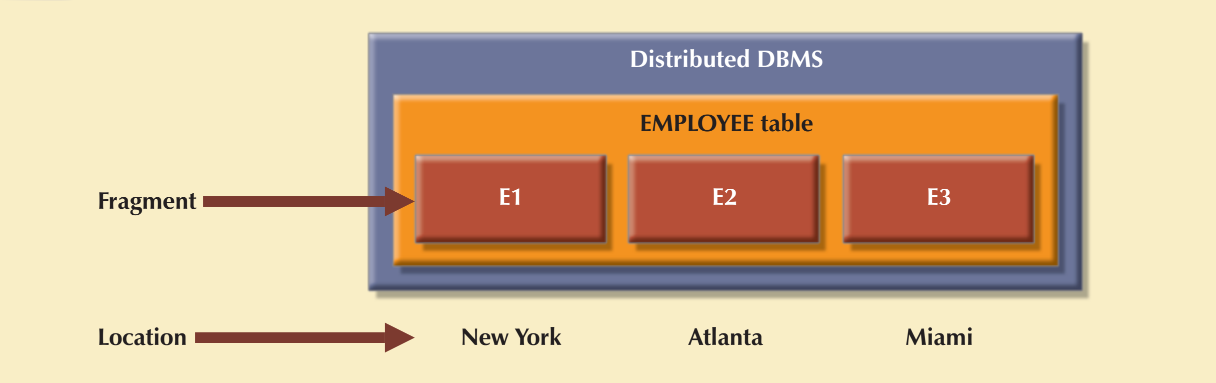
IF THE SQL STATEMENT REQUIRES:			
FRAGMENT NAME?	LOCATION NAME?	THEN THE DBMS SUPPORTS	LEVEL OF DISTRIBUTON TRANSPARENCY
Yes	Yes	Local mapping	Low
Yes	No	Location transparency	Medium
No	No	Fragmentation transparency	High

As you examine Table 12.3, you might ask why there is no reference to a situation in which the fragment name is "No" and the location name is "Yes." The reason for not including that scenario is simple: you cannot have a location name that fails to reference an existing fragment. (If you don't need to specify a fragment name, its location is clearly irrelevant.)

To illustrate the use of various transparency levels, suppose you have an EMPLOYEE table containing the attributes EMP_NAME, EMP_DOB, EMP_ADDRESS, EMP_DEPARTMENT, and EMP_SALARY. The EMPLOYEE data are distributed over three different locations: New York, Atlanta, and Miami. The table is divided by location; that is, New

York employee data are stored in fragment E1, Atlanta employee data are stored in fragment E2, and Miami employee data are stored in fragment E3. See Figure 12.9.

FIGURE 12.9 Fragment locations



Now suppose the end user wants to list all employees with a date of birth prior to January 1, 1960. To focus on the transparency issues, also suppose the EMPLOYEE table is fragmented and each fragment is unique. The **unique fragment** condition indicates that each row is unique, regardless of the fragment in which it is located. Finally, assume that no portion of the database is replicated at any other site on the network.

Depending on the level of distribution transparency support, you may examine three query cases.

Case 1: The Database Supports Fragmentation Transparency

The query conforms to a nondistributed database query format; that is, it does not specify fragment names or locations. The query reads:

```
SELECT    *
FROM      EMPLOYEE
WHERE     EMP_DOB < '01-JAN-1960';
```

Case 2: The Database Supports Location Transparency

Fragment names must be specified in the query, but fragment location is not specified. The query reads:

```
SELECT    *
FROM      E1
WHERE     EMP_DOB < '01-JAN-1960';
UNION
SELECT    *
FROM      E2
WHERE     EMP_DOB < '01-JAN-1960';
UNION
SELECT    *
FROM      E3
WHERE     EMP_DOB < '01-JAN-1960';
```

Case 3: The Database Supports Local Mapping Transparency

Both the fragment name and location must be specified in the query. Using pseudo-SQL:

```
SELECT    *
FROM      El NODE NY
WHERE     EMP_DOB < '01-JAN-1960';
UNION
SELECT    *
FROM      E2 NODE ATL
WHERE     EMP_DOB < '01-JAN-1960';
UNION
SELECT    *
FROM      E3 NODE MIA
WHERE     EMP_DOB < '01-JAN-1960';
```

NOTE

NODE indicates the location of the database fragment. NODE is used for illustration purposes and is not part of the standard SQL syntax.

As you examine the preceding query formats, you can see how distribution transparency affects the way end users and programmers interact with the database.

Distribution transparency is supported by a **distributed data dictionary** (**DDD**), or a **distributed data catalog** (**DDC**). The DDC contains the description of the entire database as seen by the database administrator. The database description, known as the **distributed global schema**, is the common database schema used by local TPs to translate user requests into subqueries (remote requests) that will be processed by different DPs. The DDC is itself distributed, and it is replicated at the network nodes. Therefore, the DDC must maintain consistency through updating at all sites.

Keep in mind that some of the current DDBMS implementations impose limitations on the level of transparency support. For instance, you might be able to distribute a database, but not a table, across multiple sites. Such a condition indicates that the DDBMS supports location transparency but not fragmentation transparency.

12.9 TRANSACTION TRANSPARENCY

Transaction transparency is a DDBMS property that ensures that database transactions will maintain the distributed database's integrity and consistency. Remember that a DDBMS database transaction can update data stored in many different computers connected in a network. Transaction transparency ensures that the transaction will be completed only when all database sites involved in the transaction complete their part of the transaction.

Distributed database systems require complex mechanisms to manage transactions and to ensure the database's consistency and integrity. To understand how the transactions are managed, you should know the basic concepts governing remote requests, remote transactions, distributed transactions, and distributed requests.

12.9.1 Distributed Requests and Distributed Transactions[1]

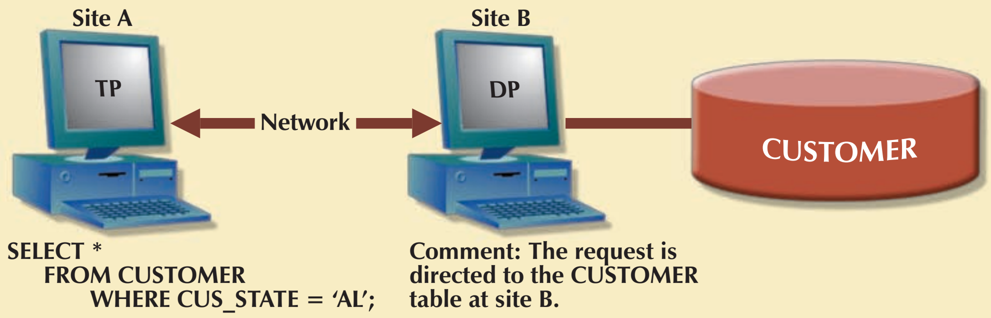
Whether or not a transaction is distributed, it is formed by one or more database requests. The basic difference between a nondistributed transaction and a distributed transaction is that the latter can update or request data from

[1] The details of distributed requests and transactions were originally described in David McGoveran and Colin White, "Clarifying Client/Server," *DBMS* 3(12), November 1990, pp. 78–89.

several different remote sites on a network. To better illustrate the distributed transaction concepts, let's begin by establishing the difference between remote and distributed transactions, using the BEGIN WORK and COMMIT WORK transaction format. Assume the existence of location transparency to avoid having to specify the data location.

A **remote request**, illustrated in Figure 12.10, lets a single SQL statement access the data that are to be processed by a single remote database processor. In other words, the SQL statement (or request) can reference data at only one remote site.

FIGURE 12.10 A remote request

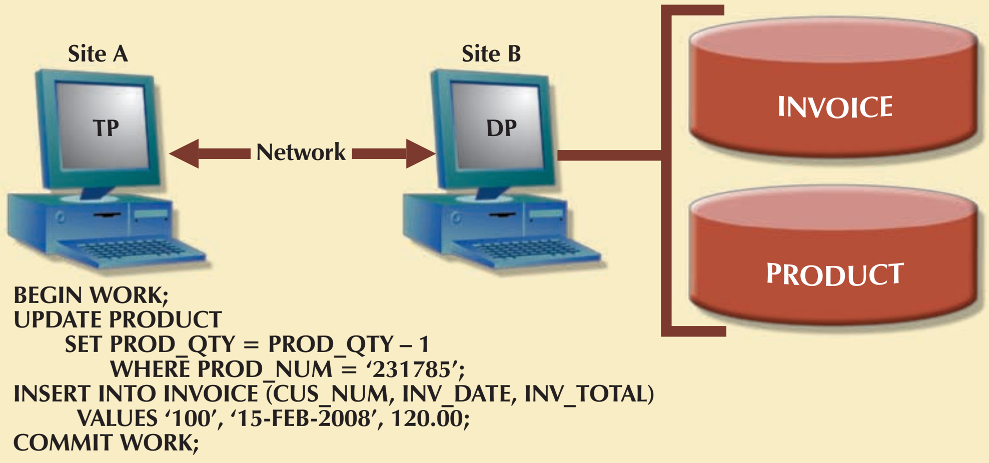


```
SELECT *
    FROM CUSTOMER
        WHERE CUS_STATE = 'AL';
```

Comment: The request is directed to the CUSTOMER table at site B.

Similarly, a **remote transaction**, composed of several requests, accesses data at a single remote site. A remote transaction is illustrated in Figure 12.11.

FIGURE 12.11 A remote transaction



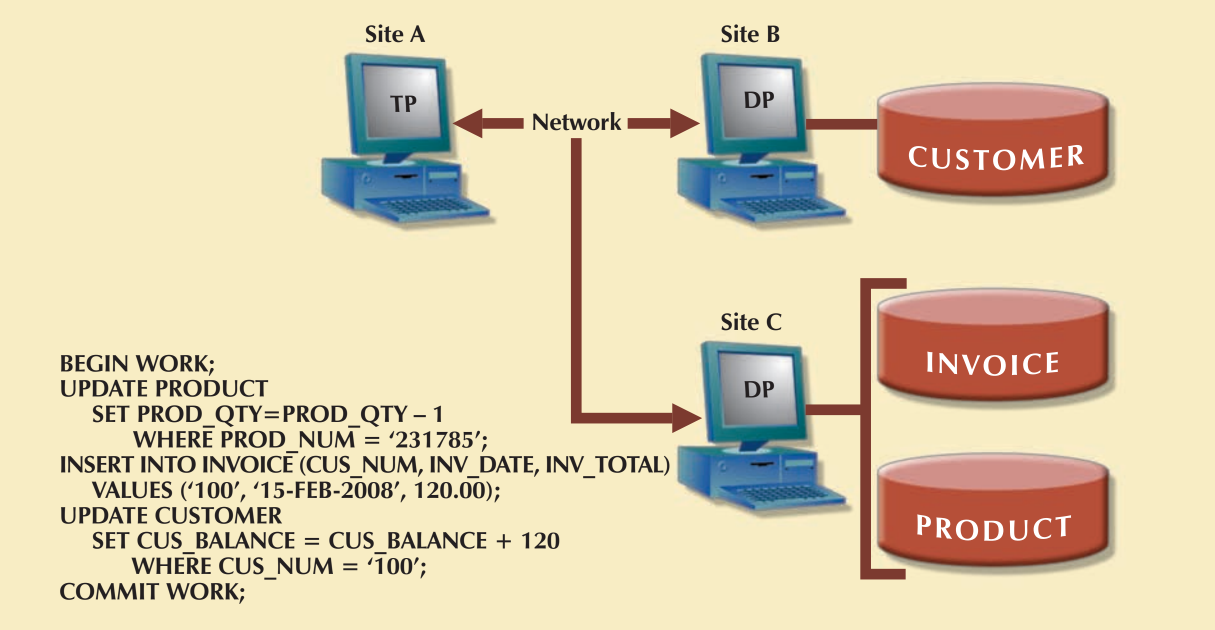
```
BEGIN WORK;
UPDATE PRODUCT
    SET PROD_QTY = PROD_QTY – 1
        WHERE PROD_NUM = '231785';
INSERT INTO INVOICE (CUS_NUM, INV_DATE, INV_TOTAL)
        VALUES '100', '15-FEB-2008', 120.00;
COMMIT WORK;
```

As you examine Figure 12.11, note the following remote transaction features:

- The transaction updates the PRODUCT and INVOICE tables (located at site B).
- The remote transaction is sent to and executed at the remote site B.
- The transaction can reference only one remote DP.
- Each SQL statement (or request) can reference only one (the same) remote DP at a time, and the entire transaction can reference and be executed at only one remote DP.

A **distributed transaction** allows a transaction to reference several different local or remote DP sites. Although each single request can reference only one local or remote DP site, the transaction as a whole can reference multiple DP sites because each request can reference a different site. The distributed transaction process is illustrated in Figure 12.12.

FIGURE 12.12 A distributed transaction



```
BEGIN WORK;
UPDATE PRODUCT
    SET PROD_QTY=PROD_QTY – 1
        WHERE PROD_NUM = ‘231785’;
INSERT INTO INVOICE (CUS_NUM, INV_DATE, INV_TOTAL)
    VALUES (‘100’, ‘15-FEB-2008’, 120.00);
UPDATE CUSTOMER
    SET CUS_BALANCE = CUS_BALANCE + 120
        WHERE CUS_NUM = ‘100’;
COMMIT WORK;
```

Note the following features in Figure 12.12:

- The transaction references two remote sites (B and C).
- The first two requests (UPDATE PRODUCT and INSERT INTO INVOICE) are processed by the DP at the remote site C, and the last request (UPDATE CUSTOMER) is processed by the DP at the remote site B.
- Each request can access only one remote site at a time.

The third characteristic may create problems. For example, suppose the table PRODUCT is divided into two fragments, PRODl and PROD2, located at sites B and C, respectively. Given that scenario, the preceding distributed transaction cannot be executed because the request:

```
SELECT      *
FROM        PRODUCT
WHERE       PROD_NUM = &'231785';
```

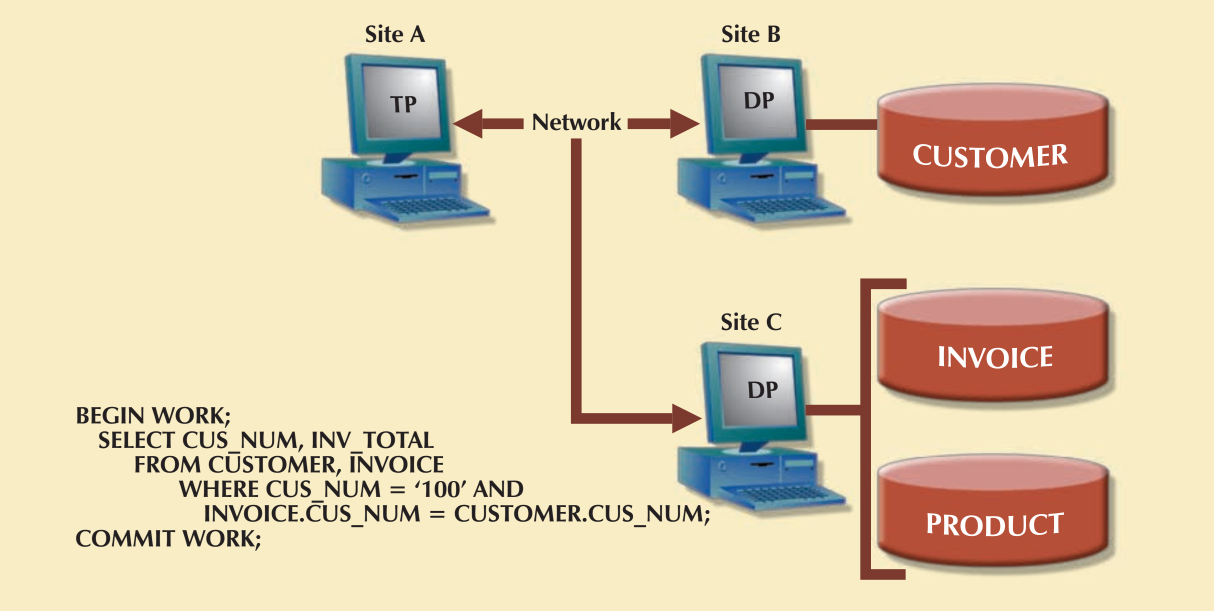
cannot access data from more than one remote site. Therefore, the DBMS must be able to support a distributed request.

A **distributed request** lets a single SQL statement reference data located at several different local or remote DP sites. Because each request (SQL statement) can access data from more than one local or remote DP site, a transaction can access several sites. The ability to execute a distributed request provides fully distributed database processing capabilities because of the ability to:

- Partition a database table into several fragments.
- Reference one or more of those fragments with only one request. In other words, there is fragmentation transparency.

The location and partition of the data should be transparent to the end user. Figure 12.13 illustrates a distributed request. As you examine Figure 12.13, note that the transaction uses a single SELECT statement to reference two tables, CUSTOMER and INVOICE. The two tables are located at two different sites, B and C.

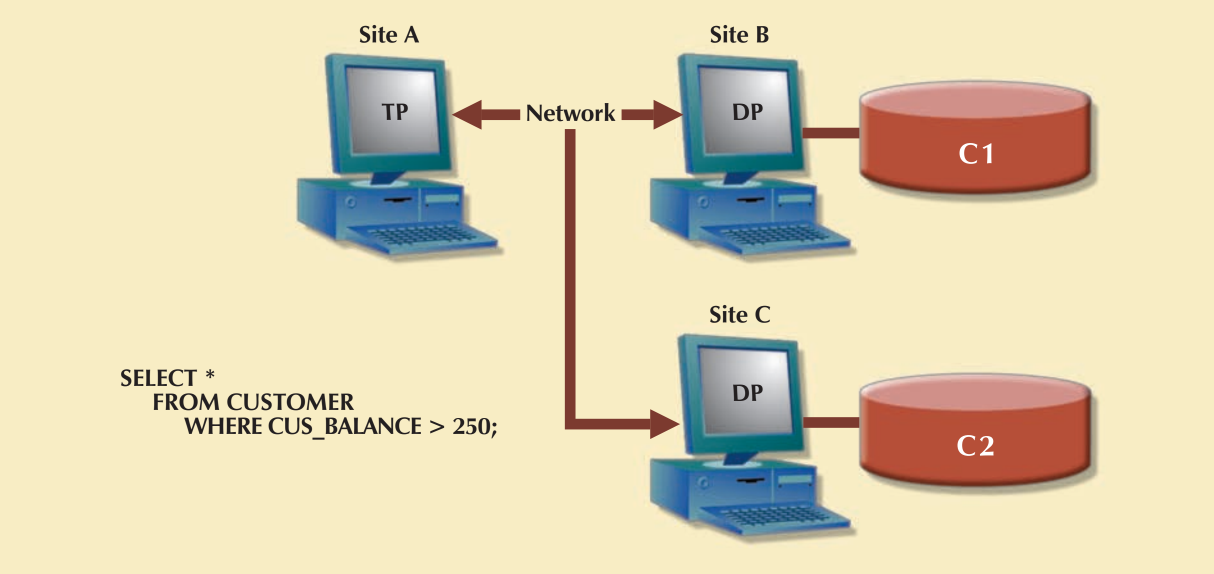
FIGURE 12.13 A distributed request



```
BEGIN WORK;
   SELECT CUS_NUM, INV_TOTAL
        FROM CUSTOMER, INVOICE
              WHERE CUS_NUM = '100' AND
                 INVOICE.CUS_NUM = CUSTOMER.CUS_NUM;
COMMIT WORK;
```

The distributed request feature also allows a single request to reference a physically partitioned table. For example, suppose a CUSTOMER table is divided into two fragments, C1 and C2, located at sites B and C, respectively. Further suppose the end user wants to obtain a list of all customers whose balances exceed $250. The request is illustrated in Figure 12.14. Full fragmentation transparency support is provided only by a DDBMS that supports distributed requests.

FIGURE 12.14 Another distributed request



```
SELECT *
    FROM CUSTOMER
        WHERE CUS_BALANCE > 250;
```

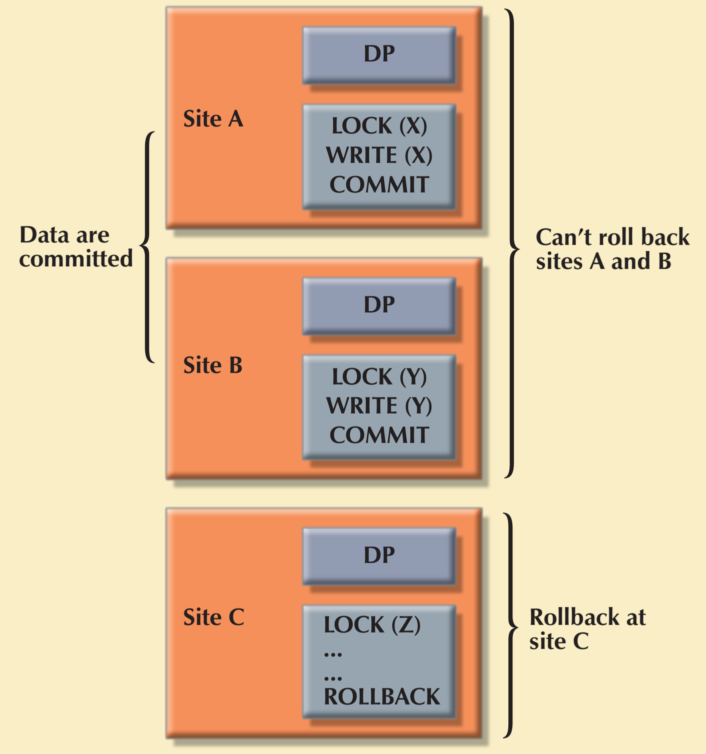
Understanding the different types of database requests in distributed database systems helps you address the transaction transparency issue more effectively. Transaction transparency ensures that distributed transactions are treated as centralized transactions, ensuring the serializability of transactions. (Review Chapter 10, Transaction Management and Concurrency Control, if necessary.) That is, the execution of concurrent transactions, whether or not they are distributed, will take the database from one consistent state to another.

12.9.2 Distributed Concurrency Control

Concurrency control becomes especially important in the distributed database environment because multisite, multiple-process operations are more likely to create data inconsistencies and deadlocked transactions than single-site systems are. For example, the TP component of a DDBMS must ensure that all parts of the transaction are completed at all sites before a final COMMIT is issued to record the transaction.

Suppose each transaction operation was committed by each local DP, but one of the DPs could not commit the transaction's results. Such a scenario would yield the problems illustrated in Figure 12.15: the transaction(s) would yield an inconsistent database, with its inevitable integrity problems, because committed data cannot be uncommitted! The solution for the problem illustrated in Figure 12.15 is a *two-phase commit protocol*, which you will explore next.

FIGURE 12.15 The effect of a premature COMMIT



12.9.3 Two-Phase Commit Protocol

Centralized databases require only one DP. All database operations take place at only one site, and the consequences of database operations are immediately known to the DBMS. In contrast, distributed databases make it possible for a transaction to access data at several sites. A final COMMIT must not be issued until all sites have committed their parts of the transaction. The **two-phase commit protocol** guarantees that if a portion of a transaction operation cannot be committed, all changes made at the other sites participating in the transaction will be undone to maintain a consistent database state.

Each DP maintains its own transaction log. The two-phase commit protocol requires that the transaction entry log for each DP be written before the database fragment is actually updated. (See Chapter 10.) Therefore, the two-phase commit protocol requires a DO-UNDO-REDO protocol and a write-ahead protocol.

The **DO-UNDO-REDO protocol** is used by the DP to roll back and/or roll forward transactions with the help of the system's transaction log entries. The DO-UNDO-REDO protocol defines three types of operations:

- DO performs the operation and records the "before" and "after" values in the transaction log.
- UNDO reverses an operation, using the log entries written by the DO portion of the sequence.
- REDO redoes an operation, using the log entries written by the DO portion of the sequence.

To ensure that the DO, UNDO, and REDO operations can survive a system crash while they are being executed, a write-ahead protocol is used. The **write-ahead protocol** forces the log entry to be written to permanent storage before the actual operation takes place.

The two-phase commit protocol defines the operations between two types of nodes: the **coordinator** and one or more **subordinates**, or *cohorts*. The participating nodes agree on a coordinator. Generally, the coordinator role is assigned to the node that initiates the transaction. However, different systems implement various, more sophisticated election methods. The protocol is implemented in two phases:

Phase 1: Preparation

The coordinator sends a PREPARE TO COMMIT message to all subordinates.

1. The subordinates receive the message; write the transaction log, using the write-ahead protocol; and send an acknowledgment (YES/PREPARED TO COMMIT or NO/NOT PREPARED) message to the coordinator.
2. The coordinator makes sure that all nodes are ready to commit, or it aborts the action.

If all nodes are PREPARED TO COMMIT, the transaction goes to phase 2. If one or more nodes reply NO or NOT PREPARED, the coordinator broadcasts an ABORT message to all subordinates.

Phase 2: The Final COMMIT

1. The coordinator broadcasts a COMMIT message to all subordinates and waits for the replies.
2. Each subordinate receives the COMMIT message, and then updates the database using the DO protocol.
3. The subordinates reply with a COMMITTED or NOT COMMITTED message to the coordinator.

If one or more subordinates did not commit, the coordinator sends an ABORT message, thereby forcing them to UNDO all changes.

The objective of the two-phase commit is to ensure that each node commits its part of the transaction; otherwise, the transaction is aborted. If one of the nodes fails to commit, the information necessary to recover the database is in the transaction log, and the database can be recovered with the DO-UNDO-REDO protocol. (Remember that the log information was updated using the write-ahead protocol.)

12.10 PERFORMANCE TRANSPARENCY AND QUERY OPTIMIZATION

One of the most important functions of a database is its ability to make data available. Because all data reside at a single site in a centralized database, the DBMS must evaluate every data request and find the most efficient way to access the local data. In contrast, the DDBMS makes it possible to partition a database into several fragments, thereby rendering the query translation more complicated, because the DDBMS must decide which fragment of the database to access. In addition, the data may also be replicated at several different sites. The data replication makes the access problem even more complex, because the database must decide which copy of the data to access. The DDBMS uses query optimization techniques to deal with such problems and to ensure acceptable database performance.

The objective of a query optimization routine is to minimize the total cost associated with the execution of a request. The costs associated with a request are a function of the:

- Access time (I/O) cost involved in accessing the physical data stored on disk.
- Communication cost associated with the transmission of data among nodes in distributed database systems.
- CPU time cost associated with the processing overhead of managing distributed transactions.

Although costs are often classified as either communication or processing costs, it is difficult to separate the two. Not all query optimization algorithms use the same parameters, and all algorithms do not assign the same weight to each parameter. For example, some algorithms minimize total time; others minimize the communication time; and still others do not factor in the CPU time, considering it insignificant relative to other cost sources.

NOTE

Chapter 11, Database Performance Tuning and Query Optimization, provides additional details about query optimization.

To evaluate query optimization, keep in mind that the TP must receive data from the DP, synchronize it, assemble the answer, and present it to the end user or an application. Although that process is standard, you should consider that a particular query may be executed at any one of several different sites. The response time associated with remote sites cannot be easily predetermined because some nodes are able to finish their part of the query in less time than others.

One of the most important characteristics of query optimization in distributed database systems is that it must provide distribution transparency as well as *replica* transparency. (Distribution transparency was explained earlier in this chapter.) **Replica transparency** refers to the DDBMS's ability to hide the existence of multiple copies of data from the user.

Most of the algorithms proposed for query optimization are based on two principles:

- The selection of the optimum execution order.
- The selection of sites to be accessed to minimize communication costs.

Within those two principles, a query optimization algorithm can be evaluated on the basis of its *operation mode* or the *timing of its optimization*.

Operation modes can be classified as manual or automatic. **Automatic query optimization** means that the DDBMS finds the most cost-effective access path without user intervention. **Manual query optimization** requires that the optimization be selected and scheduled by the end user or programmer. Automatic query optimization is clearly more desirable from the end user's point of view, but the cost of such convenience is the increased overhead that it imposes on the DDBMS.

Query optimization algorithms can also be classified according to when the optimization is done. Within this timing classification, query optimization algorithms can be classified as static or dynamic.

- **Static query optimization** takes place at compilation time. In other words, the best optimization strategy is selected when the query is compiled by the DBMS. This approach is common when SQL statements are embedded in procedural programming languages such as C# or Visual Basic .NET. When the program is submitted to the DBMS for compilation, it creates the plan necessary to access the database. When the program is executed, the DBMS uses that plan to access the database.

- **Dynamic query optimization** takes place at execution time. Database access strategy is defined when the program is executed. Therefore, access strategy is dynamically determined by the DBMS at run time, using the most up-to-date information about the database. Although dynamic query optimization is efficient, its cost is measured by run-time processing overhead. The best strategy is determined every time the query is executed; this could happen several times in the same program.

Finally, query optimization techniques can be classified according to the type of information that is used to optimize the query. For example, queries may be based on statistically based or rule-based algorithms.

- A **statistically based query optimization algorithm** uses statistical information about the database. The statistics provide information about database characteristics such as size, number of records, average access time, number of requests serviced, and number of users with access rights. These statistics are then used by the DBMS to determine the best access strategy.
- The statistical information is managed by the DDBMS and is generated in one of two different modes: dynamic or manual. In the **dynamic statistical generation mode**, the DDBMS automatically evaluates and updates the statistics after each access. In the **manual statistical generation mode**, the statistics must be updated periodically through a user-selected utility such as IBM's RUNSTAT command used by DB2 DBMSs.
- A **rule-based query optimization algorithm** is based on a set of user-defined rules to determine the best query access strategy. The rules are entered by the end user or database administrator, and they typically are very general in nature.

12.11 DISTRIBUTED DATABASE DESIGN

Whether the database is centralized or distributed, the design principles and concepts described in Chapter 3, The Relational Database Model; Chapter 4, Entity Relationship Modeling; and Chapter 5, Normalization of Database Tables, are still applicable. However, the design of a distributed database introduces three new issues:

- How to partition the database into fragments.
- Which fragments to replicate.
- Where to locate those fragments and replicas.

Data fragmentation and data replication deal with the first two issues, and data allocation deals with the third issue.

12.11.1 Data Fragmentation

Data fragmentation allows you to break a single object into two or more segments or fragments. The object might be a user's database, a system database, or a table. Each fragment can be stored at any site over a computer network. Information about data fragmentation is stored in the distributed data catalog (DDC), from which it is accessed by the TP to process user requests.

Data fragmentation strategies, as discussed here, are based at the table level and consist of dividing a table into logical fragments. You will explore three types of data fragmentation strategies: horizontal, vertical, and mixed. (Keep in mind that a fragmented table can always be re-created from its fragmented parts by a combination of unions and joins.)

- **Horizontal fragmentation** refers to the division of a relation into subsets (fragments) of tuples (rows). Each fragment is stored at a different node, and each fragment has unique rows. However, the unique rows all have the same attributes (columns). In short, each fragment represents the equivalent of a SELECT statement, with the WHERE clause on a single attribute.
- **Vertical fragmentation** refers to the division of a relation into attribute (column) subsets. Each subset (fragment) is stored at a different node, and each fragment has unique columns—with the exception of the key column, which is common to all fragments. This is the equivalent of the PROJECT statement in SQL.
- **Mixed fragmentation** refers to a combination of horizontal and vertical strategies. In other words, a table may be divided into several horizontal subsets (rows), each one having a subset of the attributes (columns).

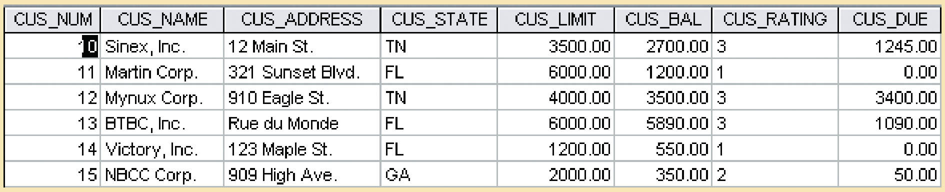
To illustrate the fragmentation strategies, let's use the CUSTOMER table for the XYZ Company, depicted in Figure 12.16. The table contains the attributes CUS_NUM, CUS_NAME, CUS_ADDRESS, CUS_STATE, CUS_LIMIT, CUS_BAL, CUS_RATING, and CUS_DUE.

ONLINE CONTENT

The databases used to illustrate the material in this chapter are found in the Student Online Companion for this book.

FIGURE 12.16 A sample CUSTOMER table

Table name: CUSTOMER



CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	Sinex, Inc.	12 Main St.	TN	3500.00	2700.00	3	1245.00
11	Martin Corp.	321 Sunset Blvd.	FL	6000.00	1200.00	1	0.00
12	Mynux Corp.	910 Eagle St.	TN	4000.00	3500.00	3	3400.00
13	BTBC, Inc.	Rue du Monde	FL	6000.00	5890.00	3	1090.00
14	Victory, Inc.	123 Maple St.	FL	1200.00	550.00	1	0.00
15	NBCC Corp.	909 High Ave.	GA	2000.00	350.00	2	50.00

Horizontal Fragmentation

Suppose XYZ Company's corporate management requires information about its customers in all three states, but company locations in each state (TN, FL, and GA) require data regarding local customers only. Based on such requirements, you decide to distribute the data by state. Therefore, you define the horizontal fragments to conform to the structure shown in Table 12.4.

TABLE 12.4 Horizontal Fragmentation of the Customer Table by State

FRAGMENT NAME	LOCATION	CONDITION	NODE NAME	CUSTOMER NUMBERS	NUMBER OF ROWS
CUST_H1	Tennessee	CUS_STATE = 'TN'	NAS	10, 12	2
CUST_H2	Georgia	CUS_STATE = 'GA'	ATL	15	1
CUST_H3	Florida	CUS_STATE = 'FL'	TAM	11, 13, 14	3

Each horizontal fragment may have a different number of rows, but each fragment *must* have the same attributes. The resulting fragments yield the three tables depicted in Figure 12.17.

Vertical Fragmentation

You may also divide the CUSTOMER relation into vertical fragments that are composed of a collection of attributes. For example, suppose the company is divided into two departments: the service department and the collections department. Each department is located in a separate building, and each has an interest in only a few of the CUSTOMER table's attributes. In this case, the fragments are defined as shown in Table 12.5.

FIGURE 12.17 Table fragments in three locations

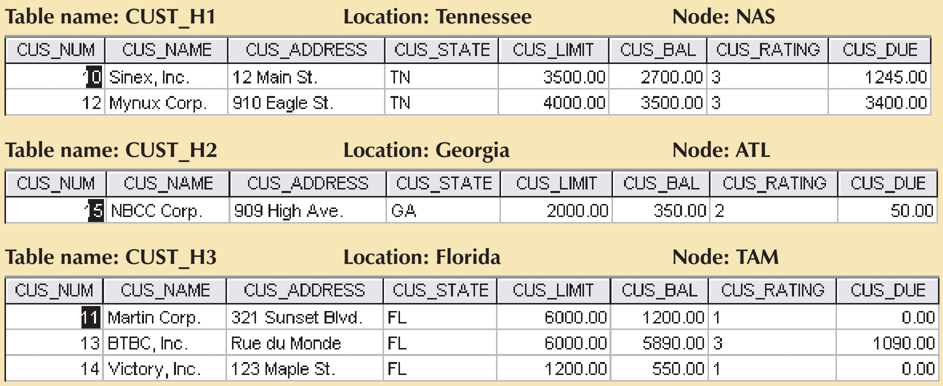


Table name: CUST_H1 Location: Tennessee Node: NAS

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	Sinex, Inc.	12 Main St.	TN	3500.00	2700.00	3	1245.00
12	Mynux Corp.	910 Eagle St.	TN	4000.00	3500.00	3	3400.00

Table name: CUST_H2 Location: Georgia Node: ATL

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
15	NBCC Corp.	909 High Ave.	GA	2000.00	350.00	2	50.00

Table name: CUST_H3 Location: Florida Node: TAM

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
11	Martin Corp.	321 Sunset Blvd.	FL	6000.00	1200.00	1	0.00
13	BTBC, Inc.	Rue du Monde	FL	6000.00	5890.00	3	1090.00
14	Victory, Inc.	123 Maple St.	FL	1200.00	550.00	1	0.00

TABLE 12.5 Vertical Fragmentation of the Customer Table

FRAGMENT NAME	LOCATION	NODE NAME	ATTRIBUTE NAMES
CUST_V1	Service Bldg.	SVC	CUS_NUM, CUS_NAME, CUS_ADDRESS, CUS_STATE
CUST_V2	Collection Bldg.	ARC	CUS_NUM, CUS_LIMIT, CUS_BAL, CUS_RATING, CUS_DUE

Each vertical fragment must have the same number of rows, but the inclusion of the different attributes depends on the key column. The vertical fragmentation results are displayed in Figure 12.18. Note that the key attribute (CUS_NUM) is common to both fragments CUST_V1 and CUST_V2.

FIGURE 12.18 Vertically fragmented table contents

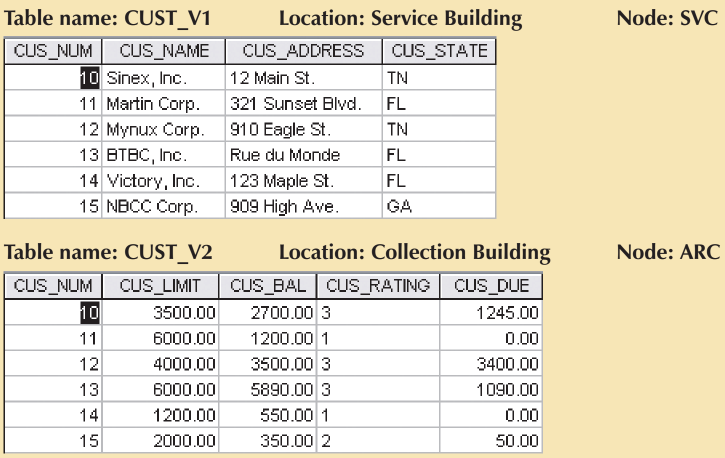


Table name: CUST_V1 Location: Service Building Node: SVC

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
10	Sinex, Inc.	12 Main St.	TN
11	Martin Corp.	321 Sunset Blvd.	FL
12	Mynux Corp.	910 Eagle St.	TN
13	BTBC, Inc.	Rue du Monde	FL
14	Victory, Inc.	123 Maple St.	FL
15	NBCC Corp.	909 High Ave.	GA

Table name: CUST_V2 Location: Collection Building Node: ARC

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	3500.00	2700.00	3	1245.00
11	6000.00	1200.00	1	0.00
12	4000.00	3500.00	3	3400.00
13	6000.00	5890.00	3	1090.00
14	1200.00	550.00	1	0.00
15	2000.00	350.00	2	50.00

Mixed Fragmentation

The XYZ Company's structure requires that the CUSTOMER data be fragmented horizontally to accommodate the various company locations; within the locations, the data must be fragmented vertically to accommodate the two departments (service and collection). In short, the CUSTOMER table requires mixed fragmentation.

Mixed fragmentation requires a two-step procedure. First, horizontal fragmentation is introduced for each site based on the location within a state (CUS_STATE). The horizontal fragmentation yields the subsets of customer tuples (horizontal fragments) that are located at each site. Because the departments are located in different buildings, vertical fragmentation is used within each horizontal fragment to divide the attributes, thus meeting each department's information needs at each subsite. Mixed fragmentation yields the results displayed in Table 12.6.

TABLE 12.6 Mixed Fragmentation of the Customer Table

FRAGMENT NAME	LOCATION	HORIZONTAL CRITERIA	NODE NAME	RESULTING ROWS AT SITE	VERTICAL CRITERIA ATTRIBUTES AT EACH FRAGMENT
CUST_M1	TN-Service	CUS_STATE = ’TN’	NAS-S	10, 12	CUS_NUM, CUS_NAME CUS_ADDRESS, CUS_STATE
CUST_M2	TN-Collection	CUS_STATE = ’TN’	NAS-C	10, 12	CUS_NUM, CUS_LIMIT, CUS_BAL, CUS_RATING, CUS_DUE
CUST_M3	GA-Service	CUS_STATE = ‘GA’	ATL-S	15	CUS_NUM, CUS_NAME CUS_ADDRESS, CUS_STATE
CUST_M4	GA-Collection	CUS_STATE = ‘GA’	ATL-C	15	CUS_NUM, CUS_LIMIT, CUS_BAL, CUS_RATING, CUS_DUE
CUST_M5	FL-Service	CUS_STATE = ’FL’	TAM-S	11, 13, 14	CUS_NUM, CUS_NAME CUS_ADDRESS, CUS_STATE
CUST_M6	FL-Collection	CUS_STATE = ’FL’	TAM-C	11, 13, 14	CUS_NUM, CUS_LIMIT, CUS_BAL, CUS_RATING, CUS_DUE

Each fragment displayed in Table 12.6 contains customer data by state and, within each state, by department location, to fit each department's data requirements. The tables corresponding to the fragments listed in Table 12.6 are shown in Figure 12.19.

FIGURE 12.19 Table contents after the mixed fragmentation process

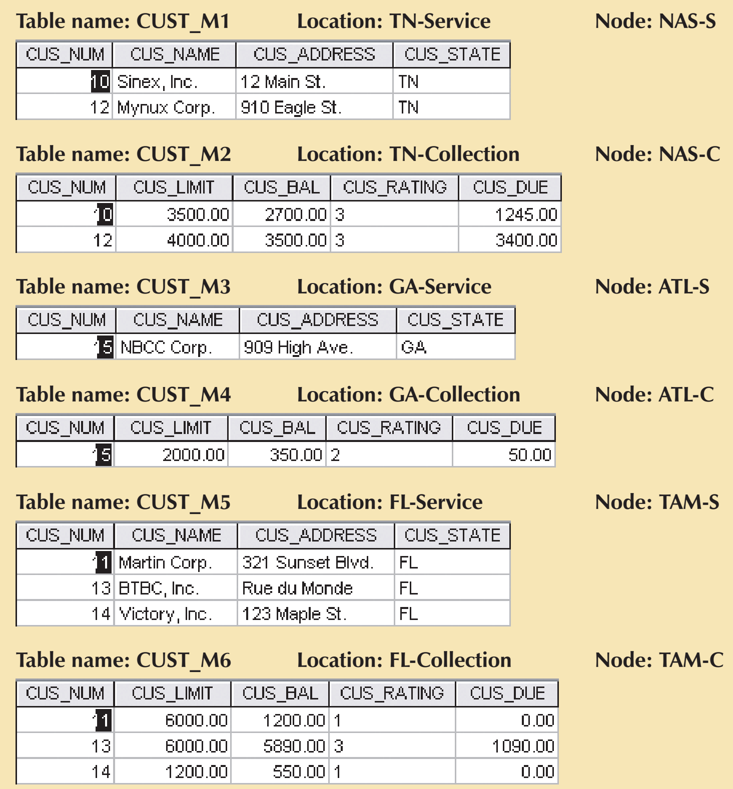


Table name: CUST_M1 **Location: TN-Service** **Node: NAS-S**

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
10	Sinex, Inc.	12 Main St.	TN
12	Mynux Corp.	910 Eagle St.	TN

Table name: CUST_M2 **Location: TN-Collection** **Node: NAS-C**

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	3500.00	2700.00	3	1245.00
12	4000.00	3500.00	3	3400.00

Table name: CUST_M3 **Location: GA-Service** **Node: ATL-S**

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
15	NBCC Corp.	909 High Ave.	GA

Table name: CUST_M4 **Location: GA-Collection** **Node: ATL-C**

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
15	2000.00	350.00	2	50.00

Table name: CUST_M5 **Location: FL-Service** **Node: TAM-S**

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
11	Martin Corp.	321 Sunset Blvd.	FL
13	BTBC, Inc.	Rue du Monde	FL
14	Victory, Inc.	123 Maple St.	FL

Table name: CUST_M6 **Location: FL-Collection** **Node: TAM-C**

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
11	6000.00	1200.00	1	0.00
13	6000.00	5890.00	3	1090.00
14	1200.00	550.00	1	0.00

12.11.2 Data Replication

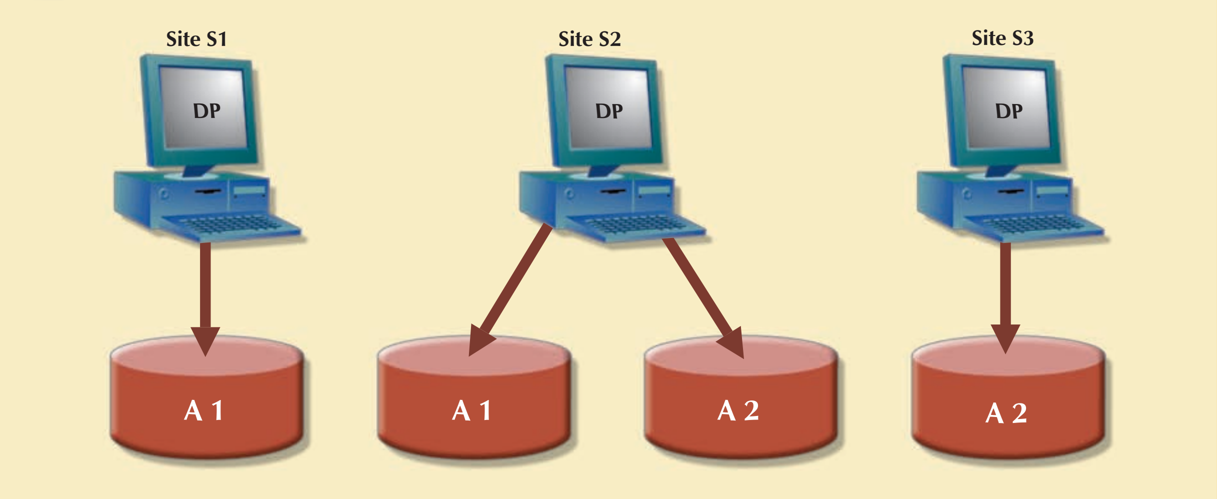
Data replication refers to the storage of data copies at multiple sites served by a computer network. Fragment copies can be stored at several sites to serve specific information requirements. Because the existence of fragment copies can enhance data availability and response time, data copies can help to reduce communication and total query costs.

Suppose database A is divided into two fragments, A1 and A2. Within a replicated distributed database, the scenario depicted in Figure 12.20 is possible: fragment A1 is stored at sites S1 and S2, while fragment A2 is stored at sites S2 and S3.

Replicated data are subject to the mutual consistency rule. The **mutual consistency rule** requires that all copies of data fragments be identical. Therefore, to maintain data consistency among the replicas, the DDBMS must ensure that a database update is performed at all sites where replicas exist.

Although replication has some benefits (such as improved data availability, better load distribution, improved data failure-tolerance, and reduced query costs), it also imposes additional DDBMS processing overhead— because each data copy must be maintained by the system. Furthermore, because the data are replicated at another site, there are

FIGURE 12.20 Data replication



associated storage costs and increased transaction times (as data must be updated at several sites concurrently to comply with the mutual consistency rule). To illustrate the replica overhead imposed on a DDBMS, consider the processes that the DDBMS must perform to use the database.

- If the database is fragmented, the DDBMS must decompose a query into *subqueries* to access the appropriate fragments.
- If the database is replicated, the DDBMS must decide which copy to access. A READ operation selects the *nearest copy* to satisfy the transaction. A WRITE operation requires that *all copies* be selected and updated to satisfy the mutual consistency rule.
- The TP sends a data request to each selected DP for execution.
- The DP receives and executes each request and sends the data back to the TP.
- The TP assembles the DP responses.

The problem becomes more complex when you consider additional factors such as network topology and communication throughputs.

Three replication scenarios exist: a database can be fully replicated, partially replicated, or unreplicated.

- A **fully replicated database** stores multiple copies of *each* database fragment at multiple sites. In this case, all database fragments are replicated. A fully replicated database can be impractical due to the amount of overhead it imposes on the system.
- A **partially replicated database** stores multiple copies of *some* database fragments at multiple sites. Most DDBMSs are able to handle the partially replicated database well.
- An **unreplicated database** stores each database fragment at a single site. Therefore, there are no duplicate database fragments.

Several factors influence the decision to use data replication:

- Database size. The amount of data replicated will have an impact on the storage requirements and also on the data transmission costs. Replicating large amounts of data requires a window of time and higher network bandwidth that could affect other applications.

- Usage frequency. The frequency of data usage determines how frequently the data needs to be updated. Frequently used data needs to be updated more often, for example, than large data sets that are used only every quarter.
- Costs, including those for performance, software overhead, and management associated with synchronizing transactions and their components vs. fault-tolerance benefits that are associated with replicated data.

When the usage frequency of remotely located data is high and the database is large, data replication can reduce the cost of data requests. Data replication information is stored in the distributed data catalog (DDC), whose contents are used by the TP to decide which copy of a database fragment to access. The data replication makes it possible to restore lost data.

12.11.3 Data Allocation

Data allocation describes the process of deciding where to locate data. Data allocation strategies are as follows:

- With **centralized data allocation**, the entire database is stored at one site.
- With **partitioned data allocation**, the database is divided into two or more disjointed parts (fragments) and stored at two or more sites.
- With **replicated data allocation**, copies of one or more database fragments are stored at several sites.

Data distribution over a computer network is achieved through data partition, through data replication, or through a combination of both. Data allocation is closely related to the way a database is divided or fragmented. Most data allocation studies focus on one issue: *which* data to locate *where*.

Data allocation algorithms take into consideration a variety of factors, including:

- Performance and data availability goals.
- Size, number of rows, and number of relations that an entity maintains with other entities.
- Types of transactions to be applied to the database and the attributes accessed by each of those transactions.
- Disconnected operation for mobile users. In some cases, the design might consider the use of loosely disconnected fragments for mobile users, particularly for read-only data that does not require frequent updates and for which the replica update windows (the amount of time available to perform a certain data processing task that cannot be executed concurrently with other tasks) may be longer.

Some algorithms include external data, such as network topology or network throughput. No optimal or universally accepted algorithm exists yet, and very few algorithms have been implemented to date.

12.12 CLIENT/SERVER VS. DDBMS

Because the trend toward distributed databases is firmly established, many database vendors have used the "client/server" label to indicate distributed database capability. However, distributed databases do not always accurately reflect the characteristics implied by the client/server label.

Client/server architecture refers to the way in which computers interact to form a system. The client/server architecture features a *user* of resources, or a client, and a *provider* of resources, or a server. The client/server architecture can be used to implement a DBMS in which the client is the TP and the server is the DP.

Client/server interactions in a DDBMS are carefully scripted. The client (TP) interacts with the end user and sends a request to the server (DP). The server receives, schedules, and executes the request, *selecting only those records that are needed by the client*. The server then sends the data to the client *only* when the client requests the data.

Client/server applications offer several advantages.

- Client/server solutions tend to be less expensive than alternate minicomputer or mainframe solutions in terms of startup infrastructure requirements.
- Client/server solutions allow the end user to use the microcomputer's GUI, thereby improving functionality and simplicity. In particular, using the ubiquitous Web browser in conjunction with Java and .NET frameworks provides a familiar end-user interface.
- More people in the job market have PC skills than mainframe skills. The majority of new generation students are learning Java and .NET programming skills.
- The PC is well established in the workplace. In addition, the increased use of the Internet as a business channel, coupled with security advances (SSL, Virtual Private Networks, multifactor authentication, etc.) provide a more reliable and secure platform for business transactions.
- Numerous data analysis and query tools exist to facilitate interaction with many of the DBMSs that are available in the PC market.
- There is a considerable cost advantage to offloading applications development from the mainframe to powerful PCs.

Client/server applications are also subject to some disadvantages.

- The client/server architecture creates a more complex environment in which different platforms (LANs, operating systems, and so on) are often difficult to manage.
- An increase in the number of users and processing sites often paves the way for security problems.
- The client/server environment makes it possible to spread data access to a much wider circle of users. Such an environment increases the demand for people with a broad knowledge of computers and software applications. The burden of training increases the cost of maintaining the environment.

ONLINE CONTENT

Refer to **Appendix F, Client/Server Systems,** for complete coverage of client/server computing concepts, components, and managerial implications.

12.13 C. J. DATE'S TWELVE COMMANDMENTS FOR DISTRIBUTED DATABASES

The notion of distributed databases has been around for at least 20 years. With the rise of relational databases, most vendors implemented their own versions of distributed databases, generally highlighting their respective product's strengths. To make the comparison of distributed databases easier, C. J. Date formulated twelve "commandments" or basic principles of distributed databases.[2] Although no current DDBMS conforms to all of them, they constitute a useful target. The twelve rules are as follows:

1. *Local site independence.* Each local site can act as an independent, autonomous, centralized DBMS. Each site is responsible for security, concurrency control, backup, and recovery.
2. *Central site independence.* No site in the network relies on a central site or any other site. All sites have the same capabilities.
3. *Failure independence.* The system is not affected by node failures. The system is in continuous operation even in the case of a node failure or an expansion of the network.
4. *Location transparency.* The user does not need to know the location of data in order to retrieve those data.

[2] Date, C. J. "Twelve Rules for a Distributed Database," *Computer World*, June 8, 1987, 2(23) pp. 77–81.

5. *Fragmentation transparency.* Data fragmentation is transparent to the user, who sees only one logical database. The user does not need to know the name of the database fragments in order to retrieve them.
6. *Replication transparency.* The user sees only one logical database. The DDBMS transparently selects the database fragment to access. To the user, the DDBMS manages all fragments transparently.
7. *Distributed query processing.* A distributed query may be executed at several different DP sites. Query optimization is performed transparently by the DDBMS.
8. *Distributed transaction processing.* A transaction may update data at several different sites, and the transaction is executed transparently.
9. *Hardware independence.* The system must run on any hardware platform.
10. *Operating system independence.* The system must run on any operating system platform.
11. *Network independence.* The system must run on any network platform.
12. *Database independence.* The system must support any vendor's database product.

SUMMARY

- A distributed database stores logically related data in two or more physically independent sites connected via a computer network. The database is divided into fragments, which can be horizontal (a set of rows) or vertical (a set of attributes). Each fragment can be allocated to a different network node.
- Distributed processing is the division of logical database processing among two or more network nodes. Distributed databases require distributed processing. A distributed database management system (DDBMS) governs the processing and storage of logically related data through interconnected computer systems.
- The main components of a DDBMS are the transaction processor (TP) and the data processor (DP). The transaction processor component is the software that resides on each computer node that requests data. The data processor component is the software that resides on each computer that stores and retrieves data.
- Current database systems can be classified by the extent to which they support processing and data distribution. Three major categories are used to classify distributed database systems: (1) single-site processing, single-site data (SPSD); (2) multiple-site processing, single-site data (MPSD); and (3) multiple-site processing, multiple-site data (MPMD).
- A homogeneous distributed database system integrates only one particular type of DBMS over a computer network. A heterogeneous distributed database system integrates several different types of DBMSs over a computer network.
- DDBMS characteristics are best described as a set of transparencies: distribution, transaction, failure, heterogeneity, and performance. All transparencies share the common objective of making the distributed database behave as though it were a centralized database system; that is, the end user sees the data as part of a single logical centralized database and is unaware of the system's complexities.
- A transaction is formed by one or more database requests. An undistributed transaction updates or requests data from a single site. A distributed transaction can update or request data from multiple sites.
- Distributed concurrency control is required in a network of distributed databases. A two-phase COMMIT protocol is used to ensure that all parts of a transaction are completed.
- A distributed DBMS evaluates every data request to find the optimum access path in a distributed database. The DDBMS must optimize the query to reduce access, communications, and CPU costs associated with the query.
- The design of a distributed database must consider the fragmentation and replication of data. The designer must also decide how to allocate each fragment or replica to obtain better overall response time and to ensure data availability to the end user.
- A database can be replicated over several different sites on a computer network. The replication of the database fragments has the objective of improving data availability, thus decreasing access time. A database can be partially, fully, or not replicated. Data allocation strategies are designed to determine the location of the database fragments or replicas.
- Database vendors often label software as client/server database products. The client/server architecture label refers to the way in which two computers interact over a computer network to form a system.

Key Terms

- application processor (AP), 484
- automatic query optimization, 498
- client/server architecture, 488
- coordinator, 497
- data allocation, 505
  - centralized, 505
  - partitioned, 505
  - replicated, 505
- database fragments, 481
- data fragmentation, 499
  - horizontal, 499
  - mixed, 499
  - vertical, 499
- data manager (DM), 484
- data processor (DP), 484
- data replication, 503
- distributed database, 481
- distributed database management system (DDBMS), 478
- distributed data catalog (DDC), 492
- distributed data dictionary (DDD), 492
- distributed global schema, 492
- distributed processing, 481
- distributed request, 494
- distributed transaction, 494
- distribution transparency, 489
- DO-UNDO-REDO protocol, 497
- dynamic query optimization, 499
- dynamic statistical generation mode, 499
- failure transparency, 490
- fragmentation transparency, 490
- fully heterogeneous DDBMS, 488
- fully replicated database, 504
- heterogeneity transparency, 490
- heterogeneous DDBMS, 488
- homogeneous DDBMS, 488
- local mapping transparency, 490
- location transparency, 490
- manual query optimization, 498
- manual statistical generation mode, 499
- multiple-site processing, multiple-site data (MPMD), 488
- multiple-site processing, single-site data (MPSD), 487
- mutual consistency rule, 503
- partially replicated database, 504
- performance transparency, 490
- remote request, 493
- remote transaction, 493
- replica transparency, 498
- rule-based query optimization algorithm, 499
- single-site processing, single-site data (SPSD), 487
- static query optimization, 498
- statistically based query optimization algorithm, 499
- subordinates, 497
- transaction manager (TM), 484
- transaction processor (TP), 484
- transaction transparency, 490
- two-phase commit protocol, 496
- unique fragment, 491
- unreplicated database, 504
- write-ahead protocol, 497

Online Content

Answers to selected Review Questions and Problems for this chapter are contained in the Student Online Companion for this book.

Review Questions

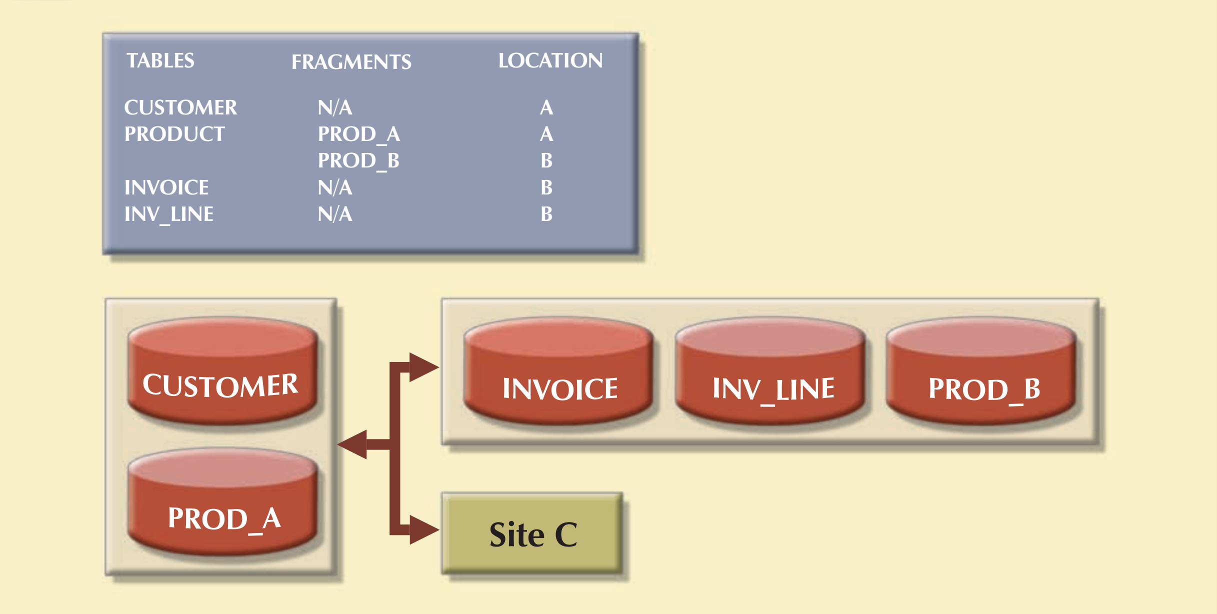
1. Describe the evolution from centralized DBMSs to distributed DBMSs.
2. List and discuss some of the factors that influenced the evolution of the DDBMS.
3. What are the advantages of the DDBMS?
4. What are the disadvantages of the DDBMS?
5. Explain the difference between a distributed database and distributed processing.
6. What is a fully distributed database management system?
7. What are the components of a DDBMS?
8. List and explain the transparency features of a DDBMS.
9. Define and explain the different types of distribution transparency.

10. Describe the different types of database requests and transactions.
11. Explain the need for the two-phase commit protocol. Then describe the two phases.
12. What is the objective of query optimization functions?
13. To which transparency feature are the query optimization functions related?
14. What are the different types of query optimization algorithms?
15. Describe the three data fragmentation strategies. Give some examples of each.
16. What is data replication, and what are the three replication strategies?
17. Explain the difference between distributed databases and client/server architecture.

PROBLEMS

The first problem is based on the DDBMS scenario in Figure P12.1.

FIGURE P12.1 **The DDBMS scenario for Problem 1**



TABLES	FRAGMENTS	LOCATION
CUSTOMER	N/A	A
PRODUCT	PROD_A	A
	PROD_B	B
INVOICE	N/A	B
INV_LINE	N/A	B

1. Specify the minimum type(s) of operation(s) the database must support (remote request, remote transaction, distributed transaction, or distributed request) to perform the following operations:

At site C

```
a. SELECT    *
   FROM      CUSTOMER;
b. SELECT    *
   FROM      INVOICE
   WHERE     INV_TOT > 1000;
c. SELECT    *
   FROM      PRODUCT
   WHERE     PROD_ QOH < 10;
```

```
d. BEGIN WORK;
   UPDATE     CUSTOMER
   SET        CUS_BAL = CUS_BAL + 100
   WHERE      CUS_NUM = '10936';
   INSERT     INTO INVOICE(INV_NUM, CUS_NUM, INV_DATE, INV_TOTAL)
              VALUES ('986391', '10936', '15-FEB-2008', 100);
   INSERT     INTO LINE(INV_NUM, PROD_NUM, LINE_PRICE)
              VALUES('986391', '1023', 100);
   UPDATE     PRODUCT
   SET        PROD_QOH = PROD_ QOH –1
   WHERE      PROD_NUM = '1023'; COMMIT WORK;
```

```
e. BEGIN WORK;
   INSERT     INTO CUSTOMER(CUS_NUM, CUS_NAME, CUS_ADDRESS, CUS_BAL)
              VALUES ('34210', 'Victor Ephanor', '123 Main St.', 0.00);
   INSERTINTO INVOICE(INV_NUM, CUS_NUM, INV_DATE, INV_TOTAL)
              VALUES ('986434', '34210', '10-AUG-2007', 2.00);
   COMMIT WORK;
```

At site A

```
f. SELECT     CUS_NUM,CUS_NAME,INV_TOTAL
   FROM       CUSTOMER, INVOICE
   WHERE      CUSTOMER.CUS_NUM = INVOICE.CUS_NUM;
```

```
g. SELECT     *
   FROM       INVOICE
   WHERE      INV_TOTAL > 1000;
```

```
h. SELECT     *
   FROM       PRODUCT
   WHERE      PROD_QOH < 10;
```

At site B

```
i. SELECT     *
   FROM       CUSTOMER;
```

```
j. SELECT     CUS_NAME, INV_TOTAL
   FROM       CUSTOMER, INVOICE
   WHERE      INV_TOTAL > 1000
              AND CUSTOMER.CUS_NUM = INVOICE.CUS_NUM;
```

```
k. SELECT     *
   FROM       PRODUCT
   WHERE      PROD_QOH < 10;
```

2. The following data structure and constraints exist for a magazine publishing company:
   a. The company publishes one regional magazine in each region: Florida (FL), South Carolina (SC), Georgia (GA), and Tennessee (TN).
   b. The company has 300,000 customers (subscribers) distributed throughout the four states listed in Part a.
   c. On the first of each month, an annual subscription INVOICE is printed and sent to each customer whose subscription is due for renewal. The INVOICE entity contains a REGION attribute to indicate the state (FL, SC, GA, TN) in which the customer resides:

   CUSTOMER (CUS_NUM, CUS_NAME, CUS_ADDRESS, CUS_CITY, CUS_ZIP, CUS_SUBSDATE)
   INVOICE (INV_NUM, INV_REGION, CUS_NUM, INV_DATE, INV_TOTAL)

   The company's management is aware of the problems associated with centralized management and has decided to decentralize management of the subscriptions into the company's four regional subsidiaries. Each subscription site will handle its own customer and invoice data. The management at company headquarters, however, will have access to customer and invoice data to generate annual reports and to issue ad hoc queries such as:

   - List all current customers by region.
   - List all new customers by region.
   - Report all invoices by customer and by region.

   Given those requirements, how must you partition the database?

3. Given the scenario and the requirements in Question 2, answer the following questions:
   a. What recommendations will you make regarding the type and characteristics of the required database system?
   b. What type of data fragmentation is needed for each table?
   c. What criteria must be used to partition each database?
   d. Design the database fragments. Show an example with node names, location, fragment names, attribute names, and demonstration data.
   e. What type of distributed database operations must be supported at each remote site?
   f. What type of distributed database operations must be supported at the headquarters site?