

Distributed Databases

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **distributed database, decentralized database, location transparency, local autonomy, synchronous distributed database, asynchronous distributed database, local transaction, global transaction, replication transparency, transaction manager, failure transparency, commit protocol, two-phase commit, concurrency transparency, time-stamping, and semijoin.**
- Explain the business conditions that are drivers for the use of distributed databases in organizations.
- Describe the salient characteristics of the variety of distributed database environments.
- Explain the potential advantages and risks associated with distributed databases.
- Explain four strategies for the design of distributed databases, options within each strategy, and the factors to consider in selection among these strategies.
- State the relative advantages of synchronous and asynchronous data replication and partitioning as three major approaches for distributed database design.
- Outline the steps involved in processing a query in a distributed database and several approaches used to optimize distributed query processing.
- Explain the salient features of several distributed database management systems.

INTRODUCTION

When an organization is geographically dispersed, it may choose to store its databases on a central computer or to distribute them to local computers (or a combination of both). A **distributed database** is a single logical database that is spread physically across computers in multiple locations that are connected by a data communications network. We emphasize that a distributed database is truly a database, not a loose collection of files. The distributed database is still centrally administered as

a corporate resource while providing local flexibility and customization. The network must allow the users to share the data; thus a user (or program) at location A must be able to access (and perhaps update) data at location B. The sites of a distributed system may be spread over a large area (such as the United States or the world) or over a small area (such as a building or campus). The computers may range from microcomputers to large-scale computers or even supercomputers.

Distributed database: A single logical database that is spread physically across computers in multiple locations that are connected by a data communication link.

Decentralized database: A database that is stored on computers at multiple locations; these computers are not interconnected by network and database software that make the data appear in one logical database.

A distributed database requires multiple database management systems, running at each remote site. The degree to which these different DBMSs cooperate, or work in partnership, and whether there is a master site that coordinates requests involving data from multiple sites distinguish different types of distributed database environments.

It is important to distinguish between distributed and decentralized databases. A **decentralized database** is also stored on computers at multiple locations; however, the computers are not interconnected by network and database software that make the data appear to be in one logical database. Thus, users at the various sites cannot share data. A decentralized database is best regarded as a collection of independent databases, rather than having the geographical distribution of a single database.

Various business conditions encourage the use of distributed databases:

- *Distribution and autonomy of business units* Divisions, departments, and facilities in modern organizations are often geographically (and possibly internationally) distributed. Often each unit has the authority to create its own information systems, and often these units want local data over which they can have control. Business mergers and acquisitions often create this environment.
- *Data sharing* Even moderately complex business decisions require sharing data across business units, so it must be convenient to consolidate data across local databases on demand.
- *Data communications costs and reliability* The cost to ship large quantities of data across a communications network or to handle a large volume of transactions from remote sources can be high. It is often more economical to locate data and applications close to where they are needed. Also, dependence on data communications can be risky, so keeping local copies or fragments of data can be a reliable way to support the need for rapid access to data across the organization.
- *Multiple application vendor environment* Today, many organizations purchase packaged application software from several different vendors. Each “best in breed” package is designed to work with its own database, and possibly with different database management systems. A distributed database can possibly be defined to provide functionality that cuts across the separate applications.
- *Database recovery* Replicating data on separate computers is one strategy for ensuring that a damaged database can be quickly recovered and users can have access to data while the primary site is being restored. Replicating data across multiple computer sites is one natural form of a distributed database.
- *Satisfying both transaction and analytical processing* As you learned in Chapter 11, the requirements for database management vary across OLTP and OLAP applications. Yet, the same data are in common between the two databases supporting each type of application. Distributed database technology can be helpful in synchronizing data across OLTP and OLAP platforms.

The ability to create a distributed database has existed since at least the 1980s. As you might expect, a variety of distributed database options exist (Bell and Grimson, 1992). Figure 13-1 outlines the range of distributed database environments. These environments are briefly explained by the following:

- I. *Homogeneous* The same DBMS is used at each node.
 - A. *Autonomous* Each DBMS works independently, passing messages back and forth to share data updates.
 - B. *Nonautonomous* A central, or master, DBMS coordinates database access and update across the nodes.

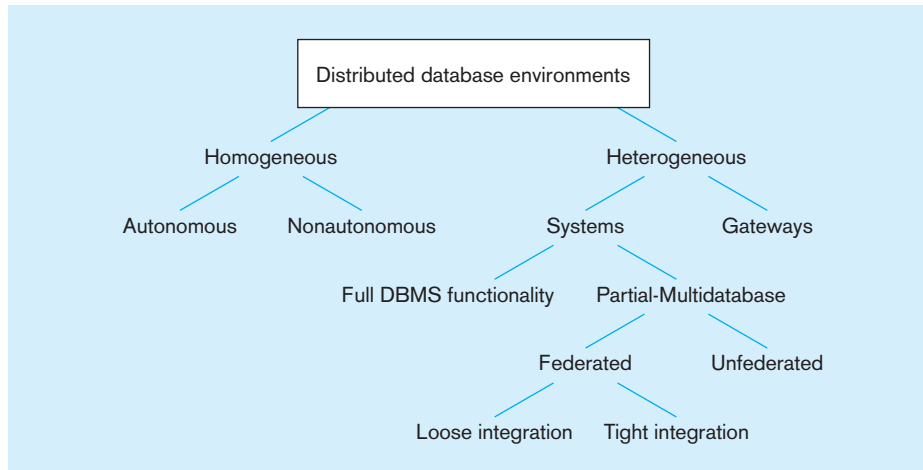


Figure 13-1
Distributed database environments
(adapted from Bell and Grimson, 1992)

- II. *Heterogeneous* Potentially different DBMSs are used at each node.
- A. *Systems* Supports some or all of the functionality of one logical database.
 1. *Full DBMS Functionality* Supports all of the functionality of a distributed database, as discussed in the remainder of this chapter.
 2. *Partial-Multidatabase* Supports some features of a distributed database, as discussed in the remainder of this chapter.
 - a. *Federated* Supports local databases for unique data requests.
 - i. *Loose Integration* Many schemas exist, for each local database, and each local DBMS must communicate with all local schemas.
 - ii. *Tight Integration* One global schema exists that defines all the data across all local databases.
 - b. *Unfederated* Requires all access to go through a central coordinating module.
 - B. *Gateways* Simple paths are created to other databases, without the benefits of one logical database.

A homogeneous distributed database environment is depicted in Figure 13-2. This environment is typically defined by the following characteristics (related to the nonautonomous category described above):

- Data are distributed across all the nodes.
- The same DBMS is used at each location.
- All data are managed by the distributed DBMS (so there are no exclusively local data).
- All users access the database through one global schema or database definition.
- The global schema is simply the union of all the local database schemas.

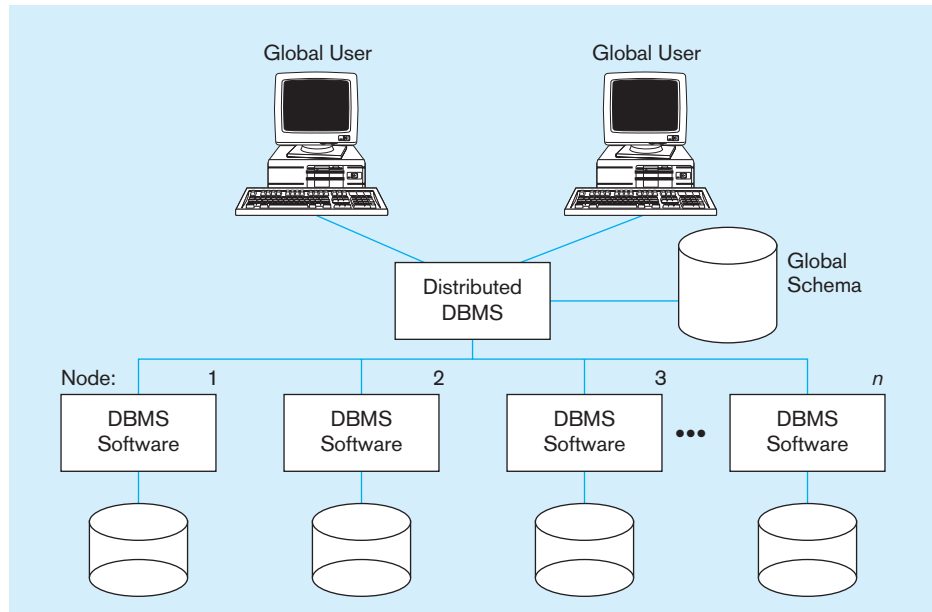
It is difficult in most organizations to force a homogeneous environment, yet heterogeneous environments are much more difficult to manage.

As listed previously, there are many variations of heterogeneous distributed database environments. In the remainder of the chapter, however, a heterogeneous environment will be defined by the following characteristics (as depicted in Figure 13-3):

- Data are distributed across all the nodes.
- Different DBMSs may be used at each node.
- Some users require only local access to databases, which can be accomplished by using only the local DBMS and schema.
- A global schema exists, which allows local users to access remote data.

Figure 13-2

Homogeneous distributed database environment (adapted from Bell and Grimson, 1992)



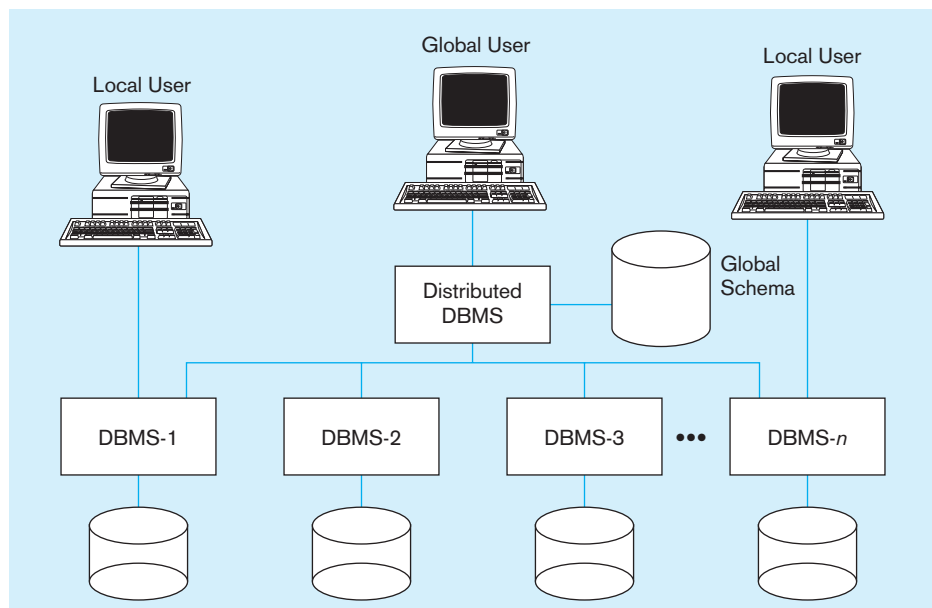
Objectives and Trade-Offs

A major objective of distributed databases is to provide ease of access to data for users at many different locations. To meet this objective, the distributed database system must provide what is called **location transparency**, which means that a user (or user program) using data for querying or updating need not know the location of the data. Any request to retrieve or update data from any site is automatically forwarded by the system to the site or sites related to the processing request. Ideally, the user is unaware of the distribution of data, and all data in the network appear as a single logical database stored at one site. In this ideal case, a single query can join data from tables in multiple sites as if the data were all in one site.

Location transparency: A design goal for a distributed database, which says that a user (or user program) using data need not know the location of the data.

Figure 13-3

Heterogeneous distributed database environment (adapted from Bell and Grimson, 1992)



A second objective of distributed databases is **local autonomy**, which is the capability to administer a local database and to operate independently when connections to other nodes have failed (Date, 1995). With local autonomy, each site has the capability to control local data, administer security, and log transactions and recover when local failures occur and to provide full access to local data to local users when any central or coordinating site cannot operate. In this case, data are locally owned and managed, even though they are accessible from remote sites. This implies that there is no reliance on a central site.

A significant trade-off in designing a distributed database environment is whether to use synchronous or asynchronous distributed technology. With **synchronous distributed database** technology, all data across the network are continuously kept up to date so that a user at any site can access data anywhere on the network at any time and get the same answer. With synchronous technology, if any copy of a data item is updated anywhere on the network, the same update is immediately applied to all other copies or it is aborted. Synchronous technology ensures data integrity and minimizes the complexity of knowing where the most recent copy of data are located. Synchronous technology can result in unsatisfactorily slow response time because the distributed DBMS is spending considerable time checking that an update is accurately and completely propagated across the network.

Asynchronous distributed database technology keeps copies of replicated data at different nodes so that local servers can access data without reaching out across the network. With asynchronous technology, there is usually some delay in propagating data updates across the remote databases, so some degree of at least temporary inconsistency is tolerated. Asynchronous technology tends to have acceptable response time because updates happen locally and data replicas are synchronized in batches and predetermined intervals, but may be more complex to plan and design to ensure exactly the right level of data integrity and consistency across the nodes.

Compared with centralized databases, either form of a distributed database has numerous advantages. The most important are the following:

- *Increased reliability and availability* When a centralized system fails, the database is unavailable to all users. A distributed system will continue to function at some reduced level, however, even when a component fails. The reliability and availability will depend (among other things) on the way the data are distributed (discussed in the following sections).
- *Local control* Distributing the data encourages local groups to exercise greater control over “their” data, which promotes improved data integrity and administration. At the same time, users can access nonlocal data when necessary. Hardware can be chosen for the local site to match the local, not global, data processing work.
- *Modular growth* Suppose that an organization expands to a new location or adds a new work group. It is often easier and more economical to add a local computer and its associated data to the distributed network than to expand a large central computer. Also, there is less chance of disruption to existing users than is the case when a central computer system is modified or expanded.
- *Lower communication costs* With a distributed system, data can be located closer to their point of use. This can reduce communication costs, compared with a central system.
- *Faster response* Depending on the way data are distributed, most requests for data by users at a particular site can be satisfied by data stored at that site. This speeds up query processing since communication and central computer delays are minimized. It may also be possible to split complex queries into subqueries that can be processed in parallel at several sites, providing even faster response.

Local autonomy: A design goal for a distributed database, which says that a site can independently administer and operate its database when connections to other nodes have failed.

Synchronous distributed database: A form of distributed database technology in which all data across the network are continuously kept up to date so that a user at any site can access data anywhere on the network at any time and get the same answer.

Asynchronous distributed database: A form of distributed database technology in which copies of replicated data are kept at different nodes so that local servers can access data without reaching out across the network.

A distributed database system also faces certain costs and disadvantages:

- *Software cost and complexity* More complex software (especially the DBMS) is required for a distributed database environment. We discuss this software later in the chapter.
- *Processing overhead* The various sites must exchange messages and perform additional calculations to ensure proper coordination among data at the different sites.
- *Data integrity* A by-product of the increased complexity and need for coordination is the additional exposure to improper updating and other problems of data integrity.
- *Slow response* If the data are not distributed properly according to their usage, or if queries are not formulated correctly, response to requests for data can be extremely slow. These issues are discussed later in the chapter.

OPTIONS FOR DISTRIBUTING A DATABASE

How should a database be distributed among the sites (or nodes) of a network? We discussed this important issue of physical database design in Chapter 6, which introduced an analytical procedure for evaluating alternative distribution strategies. In that chapter, we noted that there are four basic strategies for distributing databases:

1. Data replication
2. Horizontal partitioning
3. Vertical partitioning
4. Combinations of the above

We will explain and illustrate each of these approaches using relational databases. The same concepts apply (with some variations) for other data models, such as hierarchical and network models.

Suppose that a bank has numerous branches located throughout a state. One of the base relations in the bank's database is the Customer relation. Figure 13-4 shows the format for an abbreviated version of this relation. For simplicity, the sample data in the relation apply to only two of the branches (Lakeview and Valley). The primary key in this relation is account number (Acct_Number). Branch_Name is the name of the branch where customers have opened their accounts (and therefore where they presumably perform most of their transactions).

Data Replication

An increasingly popular option for data distribution as well as for fault tolerance of any database is to store a separate copy of the database at each of two or more sites.

Figure 13-4
Customer relation for a bank

Acct_Number	Customer_Name	Branch_Name	Balance
200	Jones	Lakeview	1000
324	Smith	Valley	250
153	Gray	Valley	38
426	Dorman	Lakeview	796
500	Green	Valley	168
683	McIntyre	Lakeview	1500
252	Elmore	Lakeview	330

Replication may allow an IS organization to move a database off a centralized mainframe onto less expensive, departmental or location-specific servers, close to end users (Koop, 1995). Replication may use either synchronous or asynchronous distributed database technologies, although asynchronous technologies are more typical in a replicated environment. The Customer relation in Figure 13-4 could be stored at Lakeview or Valley, for example. If a copy is stored at every site, we have the case of full replication, which may be impractical except for only relatively small databases.

There are five advantages to data replication:

1. *Reliability* If one of the sites containing the relation (or database) fails, a copy can always be found at another site without network traffic delays. Also, available copies can all be updated as soon as transactions occur, and unavailable nodes will be updated once they return to service.
2. *Fast response* Each site that has a full copy can process queries locally, so queries can be processed rapidly.
3. *Possible avoidance of complicated distributed transaction integrity routines* Replicated databases are usually refreshed at scheduled intervals, so most forms of replication are used when some relaxing of synchronization across database copies is acceptable.
4. *Node decoupling* Each transaction may proceed without coordination across the network. Thus, if nodes are down, busy, or disconnected (e.g., in the case of mobile personal computers), a transaction is handled when the user desires. In the place of real-time synchronization of updates, a behind-the-scenes process coordinates all data copies.
5. *Reduced network traffic at prime time* Often updating data happens during prime business hours, when network traffic is highest and the demands for rapid response greatest. Replication, with delayed updating of copies of data, moves network traffic for sending updates to other nodes to non-prime-time hours.

Replication has two primary disadvantages:

1. *Storage requirements* Each site that has a full copy must have the same storage capacity that would be required if the data were stored centrally. Each copy requires storage space (the cost for which is constantly decreasing), and processing time is required to update each copy on each node.
2. *Complexity and cost of updating* Whenever a relation is updated, it must (eventually) be updated at each site that holds a copy. Synchronizing updating in near real time can require careful coordination, as will be clear later under the topic of commit protocol.

For these reasons, data replication is favored where most process requests are read-only and where the data are relatively static, as in catalogs, telephone directories, train schedules, and so on. CD-ROM and DVD storage technology has promise as an economical medium for replicated databases. Replication is used for “noncollaborative data,” where one location does not need a real-time update of data maintained by other locations (Thé, 1994). In these applications, data eventually need to be synchronized, as quickly as is practical. Replication is not a viable approach for online applications such as airline reservations, ATM transactions, and other financial activities—applications for which each user wants data about the same, non-sharable resource.

Snapshot Replication Different schemes exist for updating data copies. Some applications, such as those for decision support and data warehousing or mining, which often do not require current data, are supported by simple table copying or

periodic snapshots. This might work as follows, assuming multiple sites are updating the same data. First, updates from all replicated sites are periodically collected at a master or primary site, where all the updates are made to form a consolidated record of all changes. With some distributed DBMSs, this list of changes is collected in a snapshot log, which is a table of row identifiers for the records to go into the snapshot. Then a read-only snapshot of the replicated portion of the database is taken at the master site. Finally, the snapshot is sent to each site where there is a copy. (It is often said that these other sites “subscribe” to the data owned at the primary site.) This is called a full refresh of the database (Buretta, 1997; Edelstein, 1995a). Alternatively, only those pages that have changed since the last snapshot can be sent, which is called a differential or incremental refresh. In this case, a snapshot log for each replicated table is joined with the associated base to form the set of changed rows to be sent to the replicated sites.

Some forms of replication management allow dynamic ownership of data, in which the right to update replicated data moves from site to site, but at any point in time, only one site owns the right. Dynamic ownership would be appropriate as business activities move across time zones or when the processing of data follows a work flow across business units supported by different database servers.

A final form of replication management allows shared ownership of data. Shared updates introduce significant issues for managing update conflicts across sites. For example, what if tellers at two bank branches try to update a customer’s address at the same time? Asynchronous technology will allow conflicts to exist temporarily. This may be fine as long as the updates are not critical to business operations, and such conflicts can be detected and resolved before real business problems arise.

The cost to perform a snapshot refresh may depend on whether the snapshot is simple or complex. A simple snapshot is one that references all or a portion of only one table. A complex snapshot involves multiple tables, usually from transactions that involve joins, such as the entry of a customer order and associated line items. With some distributed DBMSs, a simple snapshot can be handled by a differential refresh whereas complex snapshots require more time-consuming full refreshes. Some distributed DBMSs support only simple snapshots.

Near Real-Time Replication For near real-time requirements, store and forward messages for each completed transaction can be broadcast across the network informing all nodes to update data as soon as possible, without forcing a confirmation to the originating node (as is the case with a coordinated commit protocol, discussed later) before the database at the originating node is updated. One way to generate such messages is by using triggers (discussed in Chapter 8). A trigger can be stored at each local database so that when a piece of replicated data is updated, the trigger executes corresponding update commands against remote database replicas (Edelstein, 1993). With the use of triggers, each database update event can be handled individually and transparently to programs and users. If network connections to a node are down or the node is busy, these messages informing the node to update its database are held in a queue to be processed when possible.

Pull Replication The schemes just presented for synchronizing replicas are all examples of push strategies. Pull strategies also exist. In a pull strategy, the target, not the source node, controls when a local database is updated. With pull strategies, the local database determines when it needs to be refreshed and requests a snapshot or the emptying of an update message queue. Pull strategies have the advantage that the local site controls when it needs and can handle updates. Thus, synchronization is less disruptive and occurs only when needed by each site, not when a central master site thinks it is best to update.

Database Integrity with Replication For both periodic and near real-time replication, consistency across the distributed, replicated database is compromised.

Whether delayed or near real-time, the DBMS managing replicated databases still must ensure the integrity of the database. Decision support applications permit synchronization on a table-by-table basis, whereas near real-time applications require transaction-by-transaction synchronization. But in both cases, the DBMS must ensure that copies are synchronized per application requirements.

The difficulty of handling updates with a replicated database also depends on the number of nodes at which updates may occur (Froemming, 1996). In a single-updater environment, updates will usually be handled by periodically sending read-only database snapshots of updated database segments to the nonupdater nodes. In this case, the effects of multiple updates are effectively batched for the read-only sites. This would be the situation for product catalogs, price lists, and other reference data for a mobile sales force. In a multiple-updater environment, the most obvious issue is data collisions. Data collisions arise when the independently operating updating nodes are each attempting to update the same data at the same time. In this case, the DBMS must include mechanisms to detect and handle data collisions. For example, the DBMS must decide if processing at nodes in conflict should be suspended until the collision is resolved.

When to Use Replication Whether replication is a viable alternative design for a distributed database depends on several factors (Froemming, 1996):

1. *Data timeliness* Applications that can tolerate out-of-date data (whether this be for a few seconds or a few hours) are better candidates for replication.
2. *DBMS capabilities* An important DBMS capability is whether it will support a query that references data from more than one node. If not, the replication is a better candidate than the partitioning schemes, which are discussed in the following sections.
3. *Performance implications* Replication means that each node is periodically refreshed. When this refreshing occurs, the distributed node may be very busy handling a large volume of updates. If the refreshing occurs by event triggers (e.g., when a certain volume of changes accumulate), refreshing could occur at a time when the remote node is busy doing local work.
4. *Heterogeneity in the network* Replication can be complicated if different nodes use different operating systems and DBMSs, or, more commonly, use different database designs. Mapping changes from one site to n other sites could mean n different routines to translate the changes from the originating node into the scheme for processing at the other nodes.
5. *Communications network capabilities* Transmission speeds and capacity in a data communications network may prohibit frequent, complete refreshing of very large tables. Replication does not require a dedicated communications connection, however, so less expensive, shared networks could be used for database snapshot transmissions.

Horizontal Partitioning

With *horizontal partitioning* (see Chapter 6 for a description of different forms of table partitioning), some of the rows of a table (or relation) are put into a base relation at one site, and other rows are put into a base relation at another site. More generally, the rows of a relation are distributed to many sites.

Figure 13-5 shows the result of taking horizontal partitions of the Customer relation. Each row is now located at its home branch. If customers actually conduct most of their transactions at the home branch, the transactions are processed locally and response times are minimized. When a customer initiates a transaction at another branch, the transaction must be transmitted to the home branch for processing and the response transmitted back to the initiating branch. (This is the normal pattern

Figure 13-5
Horizontal partitions
(a) Lakeview Branch

Acct_Number	Customer_Name	Branch_Name	Balance
200	Jones	Lakeview	1000
426	Dorman	Lakeview	796
683	McIntyre	Lakeview	1500
252	Elmore	Lakeview	330

(b) Valley Branch

Acct_Number	Customer_Name	Branch_Name	Balance
324	Smith	Valley	250
153	Gray	Valley	38
500	Green	Valley	168

for persons using ATMs.) If a customer's usage pattern changes (perhaps because of a move), the system may be able to detect this change and dynamically move the record to the location where most transactions are being initiated. In summary, horizontal partitions for a distributed database have four major advantages:

1. *Efficiency* Data are stored close to where they are used and separate from other data used by other users or applications.
2. *Local optimization* Data can be stored to optimize performance for local access.
3. *Security* Data not relevant to usage at a particular site are not made available.
4. *Ease of querying* Combining data across horizontal partitions is easy because rows are simply merged by unions across the partitions.

Thus, horizontal partitions are usually used when an organizational function is distributed, but each site is concerned with only a subset of the entity instances (frequently based on geography).

Horizontal partitions also have two primary disadvantages:

1. *Inconsistent access speed* When data from several partitions are required, the access time can be significantly different from local-only data access.
2. *Backup vulnerability* Because data are not replicated, when data at one site become inaccessible or damaged, usage cannot switch to another site where a copy exists; data may be lost if proper backup is not performed at each site.

Vertical Partitioning

With the *vertical partitioning* approach (again, see Chapter 6), some of the columns of a relation are projected into a base relation at one of the sites, and other columns are projected into a base relation at another site (more generally, columns may be projected to several sites). The relations at each of the sites must share a common domain so that the original table can be reconstructed.

To illustrate vertical partitioning, we use an application for the manufacturing company shown in Figure 13-6. Figure 13-7 shows the Part relation with Part_Number as the primary key. Some of these data are used primarily by manufacturing, whereas others are used mostly by engineering. The data are distributed to the respective departmental computers using vertical partitioning, as shown in Figure 13-8. Each of the partitions shown in Figure 13-8 is obtained by taking projections (i.e., selected columns) of the original relation. The original relation in turn can be obtained by taking natural joins of the resulting partitions.

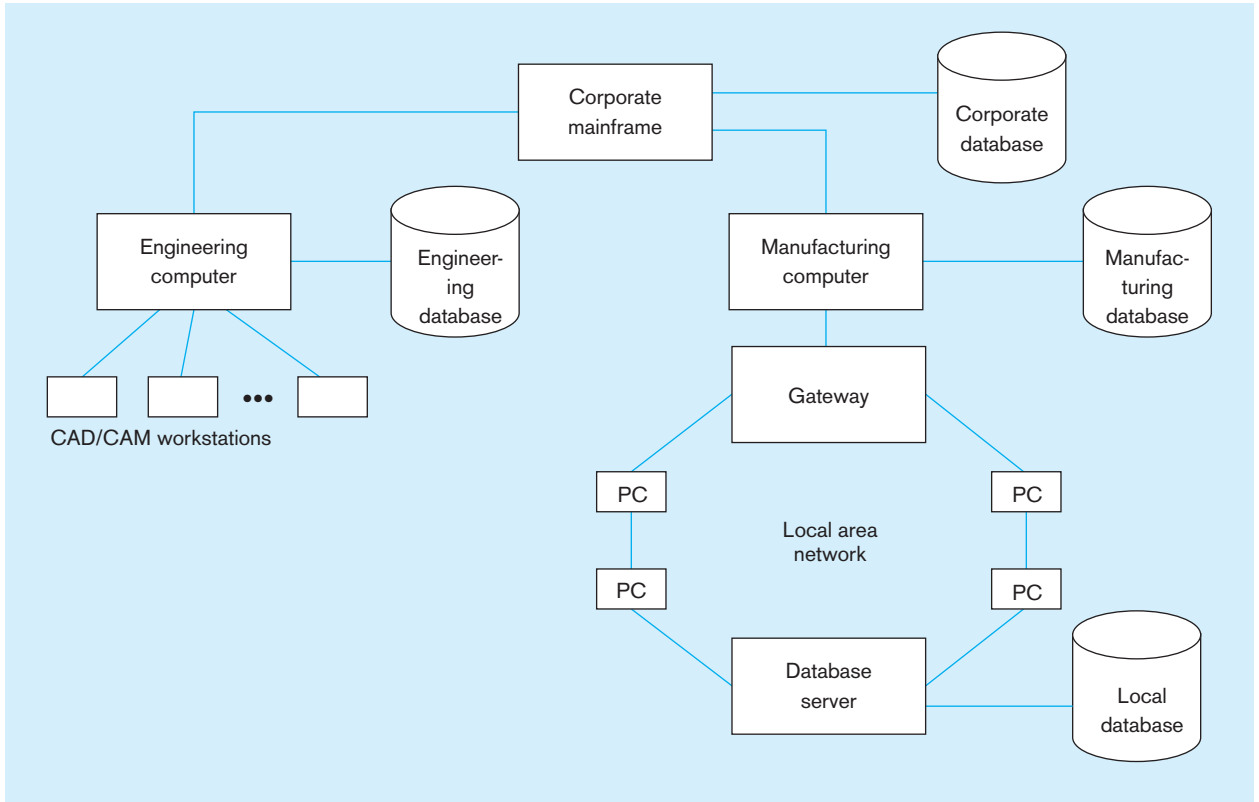


Figure 13-6
Distributed processing system for a manufacturing company

Part_Number	Name	Cost	Drawing_Number	Qty_On_Hand
P2	Widget	100	123-7	20
P7	Gizmo	550	621-0	100
P3	Thing	48	174-3	0
P1	Whatsit	220	416-2	16
P8	Thumzer	16	321-0	50
P9	Bobbit	75	400-1	0
P6	Nailit	125	129-4	200

Figure 13-7
Part relation

Part_Number	Drawing_Number
P2	123-7
P7	621-0
P3	174-3
P1	416-2
P8	321-0
P9	400-1
P6	129-4

Part_Number	Name	Cost	Qty_On_Hand
P2	Widget	100	20
P7	Gizmo	550	100
P3	Thing	48	0
P1	Whatsit	220	16
P8	Thumzer	16	50
P9	Bobbit	75	0
P6	Nailit	125	200

Figure 13-8
Vertical partitions

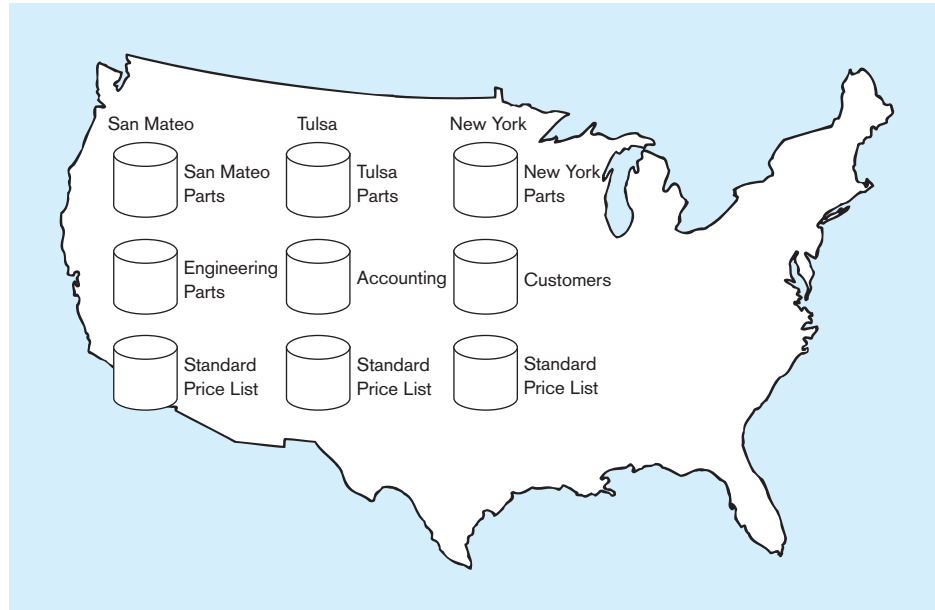
(a) Engineering

(b) Manufacturing

Figure 13-9

Hybrid data distribution strategy

(Source: Copyright © *Database Programming & Design*, April 1989, Vol. 2, No. 4. Reprinted by permission of Miller Freeman Publications.)



In summary, the advantages and disadvantages of vertical partitions are identical to those for horizontal partitions, with the exception that combining data across vertical partitions is more difficult than across horizontal partitions. This difficulty arises from the need to match primary keys or other qualifications to join rows across partitions. Horizontal partitions support an organizational design in which functions are replicated, often on a regional basis, whereas vertical partitions are typically applied across organizational functions with reasonably separate data requirements.

Combinations of Operations

To complicate matters further, there are almost unlimited combinations of the preceding strategies. Some data may be stored centrally, whereas other data are replicated at the various sites. Also, for a given relation, both horizontal and vertical partitions may be desirable for data distribution. Figure 13-9 is an example of a combination strategy:

1. Engineering Parts, Accounting, and Customer data are each centralized at different locations.
2. Standard parts data are partitioned (horizontally) among the three locations.
3. The Standard Price List is replicated at all three locations.

The overriding principle in distributed database design is that data should be stored at the sites where they will be accessed most frequently (although other considerations, such as security, data integrity, and cost, are also likely to be important). The data administrator plays a critical and central role in organizing a distributed database to make it distributed, not decentralized.

Selecting the Right Data Distribution Strategy

Based on the prior sections, a distributed database can be organized in five unique ways:

1. Totally centralized at one location accessed from many geographically distributed sites
2. Partially or totally replicated across geographically distributed sites, with each copy periodically updated with snapshots

3. Partially or totally replicated across geographically distributed sites, with near real-time synchronization of updates
4. Partitioned into segments at different geographically distributed sites, but still within one logical database and one distributed DBMS
5. Partitioned into independent, nonintegrated segments spanning multiple computers and database software

None of these five approaches is always best. Table 13-1 summarizes the comparative features of these five approaches using the key dimensions of reliability, expandability for adding nodes, communications overhead or demand on communications networks, manageability, and data consistency. A distributed database designer needs to balance these factors to select a good strategy for a given distributed database environment. The choice of which strategy is best in a given situation depends on several factors.

- *Organizational forces* Funding availability, autonomy of organizational units, and the need for security.
- *Frequency and locality or clustering of reference to data* In general, data should be located close to the applications that use those data.
- *Need for growth and expansion* The availability of processors on the network will influence where data may be located and applications can be run and may indicate the need for expansion of the network.
- *Technological capabilities* Capabilities at each node and for DBMSs, coupled with the costs for acquiring and managing technology, must be considered. Storage costs tend to be low, but the costs for managing complex technology can be great.
- *Need for reliable service* Mission-critical applications and very frequently required data encourage replication schemes.

Table 13-1 Comparison of Distributed Database Design Strategies

Strategy	Reliability	Expandability	Communications Overhead	Manageability	Data Consistency
Centralized	POOR: Highly dependent on central server	POOR: Limitations are barriers to performance	VERY HIGH: High traffic to one site	VERY GOOD: One, monolithic site requires little coordination	EXCELLENT: All users always have same data
Replicated with snapshots	GOOD: Redundancy and tolerated delays	VERY GOOD: Cost of additional copies may be less than linear	LOW to MEDIUM: Not constant, but periodic snapshots can cause bursts of network traffic	VERY GOOD: Each copy is like every other one	MEDIUM: Fine as long as delays are tolerated by business needs
Synchronized replication	EXCELLENT: Redundancy and minimal delays	VERY GOOD: Cost of additional copies may be low and synchronization work only linear	MEDIUM: Messages are constant but some delays are tolerated	MEDIUM: Collisions add some complexity to manageability	MEDIUM to VERY GOOD: Close to precise consistency
Integrated partitions	VERY GOOD: Effective use of partitioning and redundancy	VERY GOOD: New nodes get only data they need without changes in overall database design	LOW to MEDIUM: Most queries are local but queries which require data from multiple sites can cause a temporary load	DIFFICULT: Especially difficult for queries that need data from distributed tables, and updates must be tightly coordinated	VERY POOR: Considerable effort; and inconsistencies not tolerated
Decentralized with independent partitions	GOOD: Depends on only local database availability	GOOD: New sites independent of existing ones	LOW: Little if any need to pass data or queries across the network (if one exists)	VERY GOOD: Easy for each site, until there is a need to share data across sites	LOW: No guarantees of consistency, in fact pretty sure of inconsistency

DISTRIBUTED DBMS

To have a distributed database, there must be a database management system that coordinates the access to data at the various nodes. We will call such a system a *distributed DBMS*. Although each site may have a DBMS managing the local database at that site, a distributed DBMS will perform the following functions (Buretta, 1997; Elmasri and Navathe, 1989):

1. Keep track of where data are located in a distributed data dictionary. This means, in part, presenting one logical database and schema to developers and users.
2. Determine the location from which to retrieve requested data and the location at which to process each part of a distributed query without any special actions by the developer or user.
3. If necessary, translate the request at one node using a local DBMS into the proper request to another node using a different DBMS and data model and return data to the requesting node in the format accepted by that node.
4. Provide data management functions such as security, concurrency and deadlock control, global query optimization, and automatic failure recording and recovery.
5. Provide consistency among copies of data across the remote sites (e.g., by using multiphase commit protocols).
6. Present a single logical database that is physically distributed. One ramification of this view of data is global primary key control, meaning that data about the same business object are associated with the same primary key no matter where in the distributed database the data are stored, and different objects are associated with different primary keys.
7. Be scalable. Scalability is the ability to grow, reduce in size, and become more heterogeneous as the needs of the business change. Thus, a distributed database must be dynamic and be able to change within reasonable limits without having to be redesigned. Scalability also means that there are easy ways for new sites to be added (or to subscribe) and to be initialized (e.g., with replicated data).
8. Replicate both data and stored procedures across the nodes of the distributed database. The need to distribute stored procedures is motivated by the same reasons for distributing data.
9. Transparently use residual computing power to improve the performance of database processing. This means, for example, the same database query may be processed at different sites and in different ways when submitted at different times, depending on the particular workload across the distributed database at the time of query submission.
10. Permit different nodes to run different DBMSs. Middleware (see Chapter 9) can be used by the distributed DBMS and each local DBMS to mask the differences in query languages and nuances of local data.
11. Allow different versions of application code to reside on different nodes of the distributed database. In a large organization with multiple, distributed servers, it may not be practical to have each server/node running the same version of software.

Not all distributed DBMSs are capable of performing all of the functions described above. The first six functions are present in almost every viable distributed DBMS. We have listed the remaining functions in approximately decreasing order of importance and how often they are provided by current technologies.

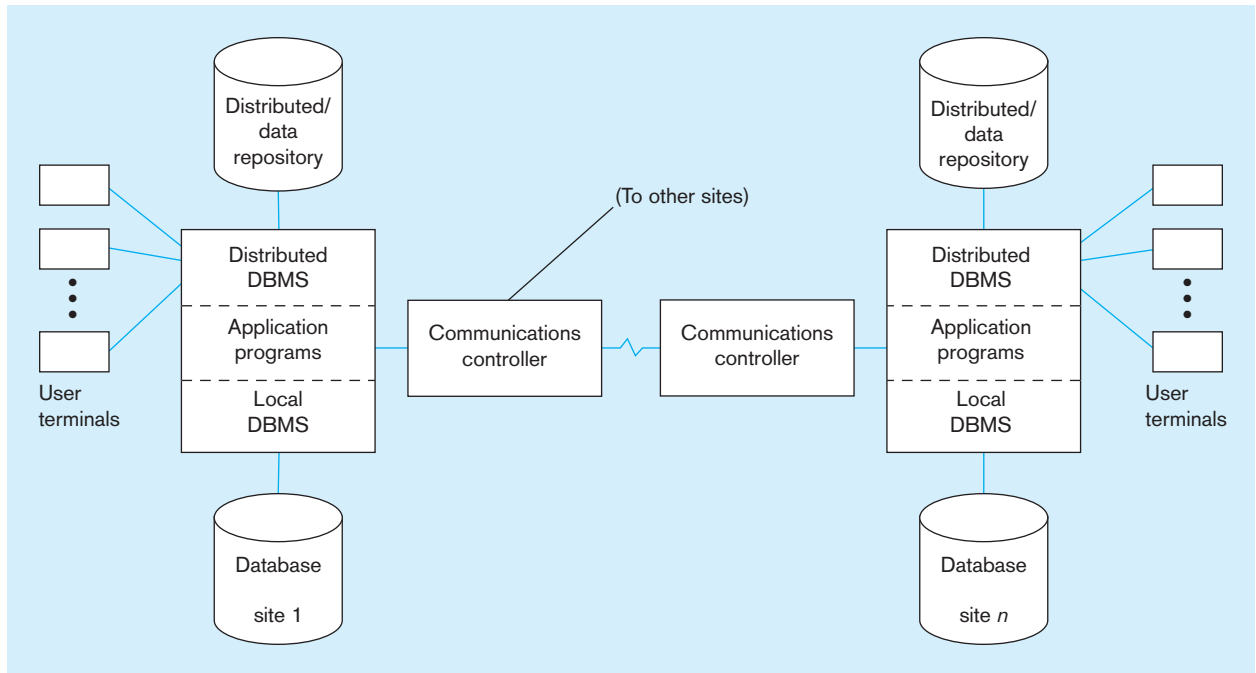


Figure 13-10
Distributed DBMS architecture

Conceptually, there could be different DBMSs running at each local site, with one master DBMS controlling the interaction across database parts. Such an environment is called a *heterogeneous distributed database*, as defined earlier in the chapter. Although ideal, complete heterogeneity is not practical today; limited capabilities exist with some products when each DBMS follows the same data architecture (e.g., relational).

Figure 13-10 shows one popular architecture for a computer system with a distributed DBMS capability. Each site has a local DBMS that manages the database stored at that site. Also, each site has a copy of the distributed DBMS and the associated distributed data dictionary/directory (DD/D). The distributed DD/D contains the location of all data in the network, as well as data definitions. Requests for data by users or application programs are first processed by the distributed DBMS, which determines whether the transaction is local or global. A **local transaction** is one in which the required data are stored entirely at the local site. A **global transaction** requires reference to data at one or more nonlocal sites to satisfy the request. For local transactions, the distributed DBMS passes the request to the local DBMS; for global transactions, the distributed DBMS routes the request to other sites as necessary. The distributed DBMSs at the participating sites exchange messages as needed to coordinate the processing of the transaction until it is completed (or aborted, if necessary). This process may be quite complex, as we will see.

The DBMS (and its data model) at one site may be different from that at another site; for example, site A may have a relational DBMS, whereas site B has a network DBMS. In this case, the distributed DBMS must translate the request so that it can be processed by the local DBMS. The capability for handling mixed DBMSs and data models is a state-of-the-art development that is beginning to appear in some commercial DBMS products.

In our discussion of an architecture for a distributed system (Figure 13-10), we assumed that copies of the distributed DBMS and DD/D exist at each site. (Thus, the DD/D is itself an example of data replication.) An alternative is to locate the distributed DBMS and DD/D at a central site, and other strategies are also

Local transaction: In a distributed database, a transaction that requires reference only to data that are stored at the site where the transaction originates.

Global transaction: In a distributed database, a transaction that requires reference to data at one or more nonlocal sites to satisfy the request.

possible. However, the centralized solution is vulnerable to failure and therefore is less desirable.

A distributed DBMS should isolate users as much as possible from the complexities of distributed database management. Stated differently, the distributed DBMS should make transparent the location of data in the network as well as other features of a distributed database. Four key objectives of a distributed DBMS, when met, ease the construction of programs and the retrieval of data in a distributed system. These objectives, which are described below, are the following: location transparency, replication transparency, failure transparency, and concurrency transparency. To fully understand failure and concurrency transparency, we also discuss the concept of a commit protocol. Finally, we describe query optimization, which is an important function of a distributed DBMS.

Location Transparency

Although data are geographically distributed and may move from place to place, with *location transparency* users (including programmers) can act as if all the data were located at a single node. To illustrate location transparency, consider the distributed database in Figure 13-9. This company maintains warehouses and associated purchasing functions in San Mateo, California; Tulsa, Oklahoma; and New York City. The company's engineering offices are in San Mateo, and its sales offices are in New York City. Suppose that a marketing manager in San Mateo, California, wanted a list of all company customers whose total purchases exceed \$100,000. From a terminal in San Mateo, with location transparency, the manager could enter the following request:

```
SELECT *
  FROM CUSTOMER
 WHERE TOTAL_SALES < 100,000;
```

Notice that this SQL request does not require the user to know where the data are physically stored. The distributed DBMS at the local site (San Mateo) will consult the distributed DD/D and determine that this request must be routed to New York. When the selected data are transmitted and displayed in San Mateo, it appears to the user at that site that the data were retrieved locally (unless there is a lengthy communications delay!).

Now consider a more complex request that requires retrieval of data from more than one site. For example, consider the Parts logical file in Figure 13-9, which is geographically partitioned into physically distributed database files stored on computers near their respective warehouse location: San Mateo parts, Tulsa parts, and New York parts. Suppose that an inventory manager in Tulsa wishes to construct a list of orange-colored parts (regardless of location). This manager could use the following query to assemble this information from the three sites:

```
SELECT DISTINCT PART_NUMBER, PART_NAME
  FROM PART
 WHERE COLOR = 'Orange'
 ORDER BY PART_NO;
```

In forming this query, the user need not be aware that the parts data exist at various sites (assuming location transparency), and that therefore this is a global transaction. Without location transparency, the user would have to reference the parts data at each site separately and then assemble the data (possibly using a UNION operation) to produce the desired results.

If the DBMS does not directly support location transparency, a database administrator can accomplish virtual location transparency for users by creating views. (See

Chapter 8 for a discussion of views in SQL). For the distributed database of Figure 13-9, the following view virtually consolidates part records into one table:

```
CREATE VIEW ALL_PART AS
  (SELECT PART_NUMBER, PART_NAME FROM SAN_MATEO_PART
  UNION
  SELECT PART_NUMBER, PART_NAME FROM TULSA_PART
  UNION
  SELECT PART_NUMBER, PART_NAME FROM NEW_YORK_PART);
```

. . . where the three part table names are synonyms for the tables at three remote sites.

The preceding examples concern read-only transactions. Can a local user also update data at a remote site (or sites)? With today's distributed DBMS products, a user can certainly update data stored at one remote site, such as the Customer data in this example. Thus, a user in Tulsa could update bill-of-material data stored in San Mateo. A more complex problem arises in updating data stored at multiple sites, such as the Vendor file. We discuss this problem in the next section.

To achieve location transparency, the distributed DBMS must have access to an accurate and current data dictionary/directory that indicates the location (or locations) of all data in the network. When the directories are distributed (as in the architecture shown in Figure 13-9), they must be synchronized so that each copy of the directory reflects the same information concerning the location of data. Although much progress has been made, true location transparency is not yet available in most systems today.

Replication Transparency

Although the same data item may be replicated at several nodes in a network, with **replication transparency** (sometimes called *fragmentation transparency*) the programmer (or other user) may treat the item as if it were a single item at a single node.

To illustrate replication transparency, see the Standard Price List file (Figure 13-9). An identical copy of this file is maintained at all three nodes (full replication). First, consider the problem of reading part (or all) of this file at any node. The distributed DBMS will consult the data directory and determine that this is a local transaction (i.e., it can be completed using data at the local site only). Thus, the user need not be aware that the same data are stored at other sites.

Now suppose that the data are replicated at some (but not all) sites (partial replication). If a read request originates at a site that does not contain the requested data, that request will have to be routed to another site. In this case, the distributed DBMS should select the remote site that will provide the fastest response. The choice of site will probably depend on current conditions in the network (such as availability of communications lines). Thus, the distributed DBMS (acting in concert with other network facilities) should dynamically select an optimum route. Again, with replication transparency, the requesting user need not be aware that this is a global (rather than local) transaction.

A more complex problem arises when one or more users attempt to update replicated data. For example, suppose that a manager in New York wants to change the price of one of the parts. This change must be accomplished accurately and concurrently at all three sites, or the data will not be consistent. With replication transparency, the New York manager can enter the data as if this were a local transaction and be unaware that the same update is accomplished at all three sites. However, to guarantee that data integrity is maintained, the system must also provide concurrency transparency and failure transparency, which we discuss next.

Replication transparency: A design goal for a distributed database, which says that although a given data item may be replicated at several nodes in a network, a programmer or user may treat the data item as if it were a single item at a single node. Also called fragmentation transparency.

Failure Transparency

Each site (or node) in a distributed system is subject to the same types of failure as in a centralized system (erroneous data, disk head crash, and so on). However, there is the additional risk of failure of a communications link (or loss of messages). For a system to be robust, it must be able to *detect* a failure, *reconfigure* the system so that computation may continue, and *recover* when a processor or link is repaired.

Error detection and system reconfiguration are probably the functions of the communications controller or processor, rather than the DBMS. However, the distributed DBMS is responsible for database recovery when a failure has occurred. The distributed DBMS at each site has a component called the **transaction manager** that performs the following functions:

Transaction manager: In a distributed database, a software module that maintains a log of all transactions and an appropriate concurrency control scheme.

1. Maintains a log of transactions and before and after database images
2. Maintains an appropriate concurrency control scheme to ensure data integrity during parallel execution of transactions at that site

For global transactions, the transaction managers at each participating site cooperate to ensure that all update operations are synchronized. Without such cooperation, data integrity can be lost when a failure occurs. To illustrate how this might happen, suppose (as we did earlier) that a manager in New York wants to change the price of a part in the Standard Price List file (Figure 13-9). This transaction is global: every copy of the record for that part (three sites) must be updated. Suppose that the price list records in New York and Tulsa are successfully updated; however, due to transmission failure, the price list record in San Mateo is not updated. Now the data records for this part are in disagreement, and an employee may access an inaccurate price for that part.

Failure transparency: A design goal for a distributed database, which guarantees that either all the actions of each transaction are committed or else none of them is committed.

With **failure transparency**, either all the actions of a transaction are committed or none of them are committed. Once a transaction occurs, its effects survive hardware and software failures. In the vendor example, when the transaction failed at one site, the effect of that transaction was not committed at the other sites. Thus, the old vendor rating remains in effect at all sites until the transaction can be successfully completed.

Commit Protocol

Commit protocol: An algorithm to ensure that a transaction is successfully completed or else it is aborted.

Two-phase commit: An algorithm for coordinating updates in a distributed database.

To ensure data integrity for real-time, distributed update operations, the cooperating transaction managers execute a **commit protocol**, which is a well-defined procedure (involving an exchange of messages) to ensure that a global transaction is either successfully completed at each site or else aborted. The most widely used protocol is called a **two-phase commit**. A two-phase commit protocol ensures that concurrent transactions at multiple sites are processed as though they were executed in the same, serial order at all sites. A two-phase commit works something like arranging a meeting between many people. First, the site originating the global transaction or an overall coordinating site (like the person trying to schedule a meeting) sends a request to each of the sites that will process some portion of the transaction. In the case of scheduling a meeting, the message might be “Are you available at a given date and time?” Each site processes the subtransaction (if possible), but does not immediately commit (or store) the result to the local database. Instead, the result is stored in a temporary file. In our meeting analogy, each person writes the meeting on his or her calendar in pencil. Each site does, however, lock (prohibit other updating) its portion of the database being updated (as each person would prohibit other appointments at the same tentative meeting time). Each site notifies the originating site when it has completed its subtransaction. When all sites have responded, the originating site now initiates the two-phase commit protocol:

1. A message is broadcast to every participating site, asking whether that site is willing to commit its portion of the transaction at that site. Each site returns an “OK” or “not OK” message. This would be like a message that each person can or cannot attend the meeting. This is often called the prepare phase. An “OK” says that the remote site promises to allow the initiating request to govern the transaction at the remote database.
2. The originating site collects the messages from all sites. If all are “OK,” it broadcasts a message to all sites to commit the portion of the transaction handled at each site. If one or more responses are “not OK,” it broadcasts a message to all sites to abort the transaction. This is often called the commit phase. Again, our hypothetical meeting arranger would confirm or abort plans for the meeting depending on the response from each person. It is possible for a transaction to fail during the commit phase (that is, between commits among the remote sites), even though it passed the prepare phase; in this case, the transaction is said to be in limbo. A limbo transaction can be identified by a timeout or polling. With a timeout (no confirmation of commit for a specified time period), it is not possible to distinguish between a busy or failed site. Polling can be expensive in terms of network load and processing time.

This description of a two-phase commit protocol is highly simplified. For a more detailed discussion of this and other protocols, see Date (1995).

With a two-phase commit strategy for synchronizing distributed data, committing a transaction is slower than if the originating location were able to work alone. Recent improvements in this traditional approach to two-phase commit are aimed at reducing the delays caused by the extensive coordination inherent in this approach. Three improvement strategies have been developed (McGovern, 1993):

1. *Read-only commit optimization* This approach identifies read-only portions of a transaction and eliminates the need for confirmation messages on these portions. For example, a transaction might include checking an inventory balance before entering a new order. The reading of the inventory balance within the transaction boundaries can occur without the callback confirmation.
2. *Lazy commit optimization* This approach allows those sites that can update to proceed to update and other sites that cannot immediately update are allowed to “catch up” later.
3. *Linear commit optimization* This approach permits each part of a transaction to be committed in sequence, rather than holding up a whole transaction when subtransaction parts are delayed from being processed.

Concurrency Transparency

The problem of concurrency control for a single (centralized) database is discussed in depth in Chapter 12. When multiple users access and update a database, data integrity may be lost unless locking mechanisms are used to protect the data from the effects of concurrent updates. The problem of concurrency control is more complex in a distributed database, because the multiple users are spread out among multiple sites and the data are often replicated at several sites, as well.

The objective of concurrency management is easy to define but often difficult to implement in practice. Although the distributed system runs many transactions concurrently, **concurrency transparency** allows each transaction to appear as if it were the only activity in the system. Thus, when several transactions are processed concurrently, the results must be the same as if each transaction were processed in serial order.

Concurrency transparency: A design goal for a distributed database, with the property that although a distributed system runs many transactions, it appears that a given transaction is the only activity in the system. Thus, when several transactions are processed concurrently, the results must be the same as if each transaction were processed in serial order.

The transaction managers at each site must cooperate to provide concurrency control in a distributed database. Three basic approaches may be used: locking and versioning, which are explained in Chapter 12 as concurrency control methods in any database environment, and time-stamping. A few special aspects of locking in a distributed database are discussed in Date (1995). The next section reviews the time-stamping approach.

Time-stamping: In distributed databases, a concurrency control mechanism that assigns a globally unique time stamp to each transaction. Time-stamping is an alternative to the use of locks in distributed databases.

Time-stamping With this approach, every transaction is given a globally unique time stamp, which generally consists of the clock time when the transaction occurred and the site ID. **Time-stamping** ensures that even if two events occur simultaneously at different sites, each will have a unique time stamp.

The purpose of time-stamping is to ensure that transactions are processed in serial order, thereby avoiding the use of locks (and the possibility of deadlocks). Every record in the database carries the time stamp of the transaction that last updated it. If a new transaction attempts to update that record and its time stamp is *earlier* than that carried in the record, the transaction is assigned a new time stamp and restarted. Thus, a transaction cannot process a record until its time stamp is *later* than that carried in the record, and therefore it cannot interfere with another transaction.

To illustrate time-stamping, suppose that a database record carries the time stamp 168, which indicates that a transaction with time stamp 168 was the most recent transaction to update that record successfully. A new transaction with time stamp 170 attempts to update the same record. This update is permitted, because the transaction's time stamp is later than the record's current time stamp. When the update is committed, the record time stamp will be reset to 170. Now, suppose instead that a record with time stamp 165 attempts to update the record. This update will not be allowed, because the time stamp is earlier than that carried in the record. Instead, the transaction time stamp will be reset to that of the record (168), and the transaction will be restarted.

The major advantage of time-stamping is that locking and deadlock detection (and the associated overhead) are avoided. The major disadvantage is that the approach is conservative, in that transactions are sometimes restarted even when there is no conflict with other transactions.

Query Optimization

With distributed databases, the response to a query may require the DBMS to assemble data from several different sites (although with location transparency, the user is unaware of this need). A major decision for the DBMS is how to process a query, which is affected by both the way a user formulates a query and the intelligence of the distributed DBMS to develop a sensible plan for processing. Date (1983) provides an excellent yet simple example of this problem. Consider the following situation adapted from Date. A simplified procurement (relational) database has the following three relations:

SUPPLIER (SUPPLIER_NUMBER,CITY)	10,000 records, stored in Detroit
PART (PART_NUMBER, COLOR)	100,000 records, stored in Chicago
SHIPMENT(SUPPLIER_NUMBER, PART_NUMBER)	1,000,000 records, stored in Detroit

A query is made (in SQL) to list the supplier numbers for Cleveland suppliers of red parts:

```
SELECT SUPPLIER.SUPPLIER_NUMBER
FROM SUPPLIER, SHIPMENT, PART
WHERE SUPPLIER.CITY = 'Cleveland'
      AND SHIPMENT.PART_NUMBER = PART.PART_NUMBER
      AND PART.COLOR = 'Red';
```


Table 13-2 Query-Processing Strategies in a Distributed Database Environment (Adapted from Date, 1983)

<i>Method</i>	<i>Time</i>
Move PART relation to Detroit, and process whole query at Detroit computer.	18.7 minutes
Move SUPPLIER and SHIPMENT relations to Chicago, and process whole query at Chicago computer.	28 hours
JOIN SUPPLIER and SHIPMENT at the Detroit computer, PROJECT these down to only tuples for Cleveland suppliers, and then for each of these, check at the Chicago computer to determine if associated PART is red.	2.3 days
PROJECT PART at the Chicago computer down to just the red items, and for each, check at the Detroit computer to see if there is some SHIPMENT involving that PART and a Cleveland SUPPLIER.	20 seconds
JOIN SUPPLIER and SHIPMENT at the Detroit computer, PROJECT just SUPPLIER_NUMBER and PART_NUMBER for only Cleveland SUPPLIERS, and move this qualified projection to Chicago for matching with red PARTs.	16.7 minutes
Select just red PARTs at the Chicago computer and move the result to Detroit for matching with Cleveland SUPPLIERS.	1 second

Each record in each relation is 100 characters long, there are ten red parts, a history of 100,000 shipments from Cleveland, and a negligible query computation time compared with communication time. Also, there is a communication system with a data transmission rate of 10,000 characters per second and one second access delay to send a message from one node to another.

Date identifies six plausible query-processing strategies for this situation and develops the associated communication times; these strategies and times are summarized in Table 13-2. Depending on the choice of strategy, the time required to satisfy the query ranges from one second to 2.3 days! Although the last strategy is best, the fourth strategy is also acceptable.

In general, this example indicates that it is often advisable to break a query in a distributed database environment into components that are isolated at different sites, determine which site has the potential to yield the fewest qualified records, and then move this result to another site where additional work is performed. Obviously, more than two sites require even more complex analyses and more complicated heuristics to guide query processing.

A distributed DBMS typically uses the following three steps to develop a query processing plan (Özsu and Valduriez, 1992):

1. *Query decomposition* In this step, the query is simplified and rewritten into a structured, relational algebra form.
2. *Data localization* Here, the query is transformed from a query referencing data across the network as if the database were in one location into one or more fragments that each explicitly reference data at only one site.
3. *Global optimization* In this final step, decisions are made about the order in which to execute query fragments, where to move data between sites, and where parts of the query will be executed.

Certainly, the design of the database interacts with the sophistication of the distributed DBMS to yield the performance for queries. A distributed database will be designed based on the best possible understanding of how and where the data will be used. Given the database design (which allocates data partitions to one or more sites), however, all queries, whether anticipated or not, must be processed as efficiently as possible.

One technique used to make processing a distributed query more efficient is to use what is called a **semijoin** operation (Elmasri and Navathe, 1989). In a semijoin,

Semijoin: A joining operation used with distributed databases in which only the joining attribute from one site is transmitted to the other site, rather than all the selected attributes from every qualified row.

Figure 13-11

Distributed database, one table at each of two sites

Site 1		Site 2	
Customer table		Order table	
Cust_No	10 bytes	Order_No	10 bytes
Cust_Name	50 bytes	Cust_No	10 bytes
Zip_Code	10 bytes	Order_Date	4 bytes
SIC	5 bytes	Order_Amount	6 bytes
10,000 rows		400,000 rows	

only the joining attribute is sent from one site to another, and then only the required rows are returned. If only a small percentage of the rows participate in the join, then the amount of data being transferred is minimal.

For example, consider the distributed database in Figure 13-11. Suppose that a query at site 1 asks to display the Cust_Name, SIC, and Order_Date for all customers in a particular Zip_Code range and an Order_Amount above a specified limit. Assume that 10 percent of the customers fall in the Zip_Code range and 2 percent of the orders are above the amount limit. A semijoin would work as follows:

1. A query is executed at site 1 to create a list of the Cust_No values in the desired Zip_Code range. So 10,000 customers * .1, or 1,000 rows satisfy the Zip_Code qualification. Thus, 1,000 rows of 10 bytes each for the Cust_No attribute (the joining attribute), or 10,000 bytes, will be sent to site 2.
2. A query is executed at site 2 to create a list of the Cust_No and Order_Date values to be sent back to site 1 to compose the final result. If we assume roughly the same number of orders for each customer, then 40,000 rows of the Order table will match with the customer numbers sent from site 1. If we assume any customer order is equally likely to be above the amount limit, then 800 rows (40,000 * .02) of the Order table rows are relevant to this query. For each row, the Cust_No and Order_Date need to be sent to site 1, or 14 bytes * 800 rows, thus 11,200 bytes.

The total data transferred is only 21,200 bytes using the semijoin just described. Compare this total to simply sending the subset of each table needed at one site to the other site:

- To send data from site 1 to site 2 would require sending the Cust_No, Cust_Name, and SIC (65 bytes) for 1,000 rows of the Customer table (65,000 bytes) to site 2.
- To send data from site 2 to site 1 would require sending Cust_No and Order_Date (14 bytes) for 8,000 rows of the Order table (112,000 bytes).

Clearly, the semijoin approach saves network traffic, which can be a major contributing factor to the overall time to respond to a user's query.

A distributed DBMS uses a cost model to predict the execution time (for data processing and transmission) of alternative execution plans. The cost model is performed before the query is executed based on general network conditions; consequently, the actual cost may be more or less, depending on the actual network and node loads, database reorganizations, and other dynamic factors. Thus, the parameters of the cost model should be periodically updated as general conditions change in the network (e.g., as local databases are redesigned, network paths are changed, and DBMSs at local sites are replaced).

Evolution of Distributed DBMS

Distributed database management is still an emerging, rather than established, technology. Current releases of distributed DBMS products do not provide all of the features described in the previous sections. For example, some products provide location transparency for read-only transactions but do not yet support global updates. To illustrate the evolution of distributed DBMS products, we briefly describe three stages in this evolution: remote unit of work, distributed unit of work, and distributed request. Then, in the next section, we summarize the major features of leading distributed DBMSs (those present in these packages at the time of writing this text).

In the following discussion, the term *unit of work* refers to the sequence of instructions required to process a transaction. That is, it consists of the instructions that begin with a “begin transaction” operation and end with either a “commit” or a “rollback” operation.

Remote Unit of Work The first stage allows multiple SQL statements to be originated at one location and executed as a single unit of work on a single remote DBMS. Both the originating and receiving computers must be running the same DBMS. The originating computer does not consult the data directory to locate the site containing the selected tables in the remote unit of work. Instead, the originating application must know where the data reside and connect to the remote DBMS prior to each remote unit of work. Thus, the remote unit of work concept does not support location transparency.

A remote unit of work (also called a remote transaction) allows updates at the single remote computer. All updates within a unit of work are tentative until a commit operation makes them permanent or a rollback undoes them. Thus transaction integrity is maintained for a single remote site; however, an application cannot assure transaction integrity when more than one remote location is involved. Referring to the database in Figure 13-9, an application in San Mateo could update the Part file in Tulsa and transaction integrity would be maintained. However, that application could not simultaneously update the Part file in two or more locations and still be assured of maintaining transaction integrity. Thus the remote unit of work also does not provide failure transparency.

Distributed Unit of Work A distributed unit of work allows various statements within a unit of work to refer to *multiple* remote DBMS locations. This approach supports some location transparency, because the data directory is consulted to locate the DBMS containing the selected table in each statement. However, all tables in a single SQL statement must be at the same location. Thus, a distributed unit of work would not allow the following query, designed to assemble parts information from all three sites in Figure 13-9:

```
SELECT DISTINCT   PART_NUMBER, PART_NAME
FROM              PART
WHERE             COLOR = 'ORANGE'
ORDER BY         PART_NUMBER;
```

Similarly, a distributed unit of work would not allow a single SQL statement that attempts to update data at more than one location. For example, the following SQL statement is intended to update the part file at three locations:

```
UPDATE          PART
SET             UNIT_PRICE = 127.49
WHERE          PART_NUMBER = 12345;
```

This update (if executed) would set the unit price of part number 12345 to \$127.49 at Tulsa, San Mateo, and New York (Figure 13-9). The statement would

not be acceptable as a distributed unit of work, however, because the single SQL statement refers to data at more than one location. The distributed unit of work does support protected updates involving multiple sites, provided that each SQL statement refers to a table (or tables) at one site only. For example, suppose in Figure 13-9 we want to increase the balance of part number 12345 in Tulsa and at the same time decrease the balance of the same part in New York (perhaps to reflect an inventory adjustment). The following SQL statements could be used:

```
UPDATE    PART
SET       BALANCE = BALANCE - 50
WHERE    PART_NUMBER = 12345 AND LOCATION = 'TULSA'
UPDATE    PART
SET       BALANCE = BALANCE + 50
WHERE    PART_NUMBER = 12345 AND LOCATION = 'NEW YORK';
```

Under the distributed unit of work concept, either this update will be committed at both locations or it will be rolled back and (perhaps) attempted again. We conclude from these examples that the distributed unit of work supports some (but not all) of the transparency features described earlier in this section.

Distributed Request The distributed request allows a single SQL statement to refer to tables in more than one remote DBMS, overcoming a major limitation of the distributed unit of work. The distributed request supports true location transparency, because a single SQL statement can refer to tables at multiple sites. However, the distributed request may or may not support replication transparency or failure transparency. It will probably be some time before a true distributed DBMS, one that supports all of the transparency features we described earlier, appears on the market.

DISTRIBUTED DBMS PRODUCTS

Most of the leading vendors of database management systems have a distributed version. In most cases, to utilize all distributed database capabilities, one vendor's DBMS must be running at each node (a homogeneous distributed database environment). Client/server forms of a distributed database are arguably the most common form in existence today. In a client/server environment (see Chapter 9 for an explanation of client/server databases), it is very easy to define a database with tables on several nodes in a local or wide area network. Once a user program establishes a linkage with each remote site, and suitable database middleware is loaded, full location transparency is achieved. So, in a client/server database form, distributed databases are readily available to any information systems developer, and heterogeneity of DBMS is possible.

Although their approaches are constantly changing, it is illustrative to overview how different vendors address distributed database management. Probably the most interesting aspect is the differences across products. These differences (summarized in Table 13-3) suggest how difficult it is to select a distributed DBMS product, because the exact capabilities of the DBMS must be carefully matched with the needs of an organization. Also, with so many options, and with each product handling distributed data differently, it is almost impossible to outline general principles for managing a distributed database. The design of any distributed database requires careful analysis of both the business's needs and the intricacies of the DBMS. Thompson (1997) also recommends that a distributed DBMS product should be used only when you really need a distributed DBMS. Do not use a distributed DBMS to create a backup database for a mission-critical application; easier solutions, such as RAID (see Chapter 6), exist for simpler needs.

Table 13-3 Distributed DBMSs

<i>Vendor</i>	<i>Product</i>	<i>Important Features</i>
IBM	• DB2 DataPropagator	<ul style="list-style-type: none"> • Works with DB2; replicates data to “regional transactional” databases • Primary site and asynchronous updates • Read-only sites subscribe to primary site • Subscription to subset or query result • Heterogeneous databases
	• Distributed Relational Database Architecture (DRDA)	
Sybase	• DataJoiner	• Middleware to access non-IBM databases
	• Replication Server	<ul style="list-style-type: none"> • Primary site and distributed read-only sites • Update to read-only site as one transaction • Hierarchical replication • Data and stored procedures replicated
Oracle	• SQL Anywhere Studio	• Mobile databases
	• Table Snapshot Option	• Periodic snapshots sent to read-only (mobile) sites
	• Symmetric Replication option	<ul style="list-style-type: none"> • Asynchronous and synchronous with multiple updatable copies and replication from any node to any other node (bidirectional) • Differential refresh • DBA controls replication • Two phase commit
Computer Associates	• Advantage Ingres Enterprise Relational Database Replicator Option	<ul style="list-style-type: none"> • All database copies updatable • Hierarchical replication • DBA registers data for replication and other sites subscribe • Master/slave form allows slave to be an Ingres database
Microsoft	• SQL Server 2005	<ul style="list-style-type: none"> • Primary site and distributed read-only sites • Publish and subscribe, with articles and publications • One database can pass copies of publications to other sites • Mobile databases

Summary

This chapter covered various issues and technologies for distributed databases. We saw that a distributed database is a single logical database that is spread across computers in multiple locations connected by a data communications network. A distributed database differs from a decentralized database, in which distributed data are not interconnected. In a distributed database, the network must allow users to share the data as transparently as possible, yet must allow each node to operate autonomously, especially when network linkages are broken or specific nodes fail. Business conditions today encourage the use of distributed databases: dispersion and autonomy of business units (including globalization of organizations), need for data sharing, and the costs and reliability of data communications. A distributed database environment may be homogeneous, involving the same DBMS at each

node, or heterogeneous, with potentially different DBMSs at different nodes. Also, a distributed database environment may keep all copies of data and related data in immediate synchronization or may tolerate planned delays in data updating through asynchronous methods.

There are numerous advantages to distributed databases. The most important of these are the following: increased reliability and availability of data, local control by users over their data, modular (or incremental) growth, reduced communications costs, and faster response to requests for data. There are also several costs and disadvantages of distributed databases: Software is more costly and complex, processing overhead often increases, maintaining data integrity is often more difficult, and if data are not distributed properly, response to requests for data may be very slow.

There are several options for distributing data in a network: data replication, horizontal partitioning, vertical partitioning, and combinations of these approaches. With data replication, a separate copy of the database (or part of the database) is stored at each of two or more sites. Data replication can result in improved reliability and faster response, can be done simply under certain circumstances, allows nodes to operate more independently (yet coordinated) of each other, and reduces network traffic; however, additional storage capacity is required, and immediate updating at each of the sites may be difficult. Replicated data can be updated by taking periodic snapshots of an official record of data and sending the snapshots to replicated sites. These snapshots can involve all data or only the data that have changed since the last snapshot. With horizontal partitioning, some of the rows of a relation are placed at one site, and other rows are placed in a relation at another site (or several sites). On the other hand, vertical partitioning distributes the columns of a relation among different sites. The objectives of data partitioning include improved performance and security. Combinations of data replication and horizontal and vertical partitioning are often used. Organizational factors, frequency and location of queries and transactions, possible growth of data and node, technology, and the need for reliability influence the choice of a data distribution design.

To have a distributed database, there must be a distributed DBMS that coordinates the access to data at the various nodes. Requests for data by users or application programs are first processed by the distributed DBMS, which determines whether the transaction is local (can be processed at the local site), remote (can be processed at some other site), or global (requires access to data at several nonlocal sites). For global transactions, the distributed DBMS consults the data directory and routes parts of the request as necessary, and then consolidates results from the remote sites.

A distributed DBMS should isolate users from the complexities of distributed database management. By location transparency, we mean that although data are geographically distributed, they appear to users as if all of the data were located at a single node. By replication transparency, we mean that although a data item may be stored at several different nodes, the user may treat the item as if it were a single item at a single node. With failure transparency, either all the actions of a transaction are completed at each site, or else none of them are committed. Distributed databases can be designed to allow temporary inconsistencies across the nodes, when immediate synchronization is not necessary. With concurrency transparency, each transaction appears to be the only activity in the system. Failure and concurrency transparency can be managed by commit protocols, which coordinate updates across nodes, locking data, and times-tamping.

A key decision made by a distributed DBMS is how to process a global query. The time to process a global query can vary from a few seconds to many hours depending on how intelligent the DBMS is in producing an efficient query-processing plan. A query-processing plan involves decomposing the query into a structured set of steps, identifying different steps with local data at different nodes in the distributed database, and, finally, choosing a sequence and location for executing each step of the query.

Few (if any) distributed DBMS products provide all forms of transparency, all forms of data replication and partitioning, and the same level of intelligence in distributed query processing. These products are, however, improving rapidly as the business pressures for distributed systems increase. Leading vendors of relational database products have introduced distributed versions with tools to help a database administrator design and manage a distributed database.

CHAPTER REVIEW

Key Terms

Asynchronous distributed database
Commit protocol
Concurrency transparency
Decentralized database
Distributed database
Failure transparency

Global transaction
Local autonomy
Local transaction
Location transparency
Replication transparency

Semijoin
Synchronous distributed database
Timestamping
Transaction manager
Two-phase commit

Review Questions

- Define each of the following terms:
 - distributed database
 - location transparency
 - two-phase commit
 - global transaction
 - local autonomy
 - time-stamping
 - transaction manager
- Match the following terms to the appropriate definition:

___ replication transparency	a. guarantees that all or none of the updates occur in a transaction across a distributed database
___ unit of work	b. the appearance that a given transaction is the only transaction running against a distributed database
___ global transaction	c. treating copies of data as if there were only one copy
___ concurrency transparency	d. references data at more than one location
___ replication	e. sequence of instructions required to process a transaction
___ failure transparency	f. a good database distribution strategy for read-only data
- Contrast the following terms:
 - distributed database; decentralized database
 - homogeneous distributed database; heterogeneous distributed database
 - location transparency; local autonomy
 - asynchronous distributed database; synchronous distributed database
 - horizontal partition; vertical partition
 - full refresh; differential refresh
 - push replication; pull replication
 - local transaction; global transaction
- Briefly describe six business conditions that are encouraging the use of distributed databases.
- Explain two types of homogeneous distributed databases.
- Briefly describe five major characteristics of homogeneous distributed databases.
- Briefly describe four major characteristics of heterogeneous distributed databases.
- Briefly describe five advantages for distributed databases compared with centralized databases.
- Briefly describe four costs and disadvantages of distributed databases.
- Briefly describe five advantages to the data replication form of distributed databases.
- Briefly describe two disadvantages to the data replication form of distributed databases.
- Explain under what circumstances a snapshot replication approach would be best.
- Explain under what circumstances a near real-time replication approach would be best.
- Briefly describe five factors that influence whether data replication is a viable distributed database design strategy for an application.
- Explain the advantages and disadvantages of horizontal partitioning for distributed databases.
- Explain the advantages and disadvantages of vertical partitioning for distributed databases.
- Briefly describe five factors that influence the selection of a distributed database design strategy.
- Briefly describe six unique functions performed by a distributed database management system.
- Briefly explain the effect of location transparency on an author of an ad hoc database query.
- Briefly explain the effect of replication transparency on an author of an ad hoc database query.
- Briefly explain in what way two-phase commit can still fail to create a completely consistent distributed database.
- Briefly describe three improvements to the two-phase commit protocol.
- Briefly describe the three steps in distributed query processing.
- Briefly explain the conditions that suggest the use of a semi-join will result in faster distributed query processing.

Problems and Exercises

Problems and Exercises 1–3 refer to the distributed database shown in Figure 13-9.

1. Name the type of transparency (location, replication, failure, concurrency) that is indicated by each statement.
 - a. End users in New York and Tulsa are updating the Engineering Parts database in San Mateo at the same time. Neither user is aware that the other is accessing the data, and the system protects the data from lost updates due to interference.
 - b. An end user in Tulsa deletes an item from the Standard Price List at the site. Unknown to the user, the distributed DBMS also deletes that item from the Standard Price List in San Mateo and New York.
 - c. A user in San Mateo initiates a transaction to delete a part from San Mateo parts and simultaneously to add that part to New York parts. The transaction is completed in San Mateo but due to transmission failure is not completed in New York. The distributed DBMS automatically reverses the transaction at San Mateo and notifies the user to retry the transaction.
 - d. An end user in New York requests the balance on hand for part number 33445. The user does not know where the record for this part is located. The distributed DBMS consults the directory and routes the request to San Mateo.
2. Consider the Standard Price List in Figure 13-9.
 - a. Write an SQL statement that will increase the Unit_Price of Part_Number 56789 by 10 percent.
 - b. Indicate whether the statement you wrote in part a is acceptable under each of the following protocols:
 - Remote unit of work
 - Distributed unit of work
 - Distributed request
3. Consider the four parts databases in Figure 13-9.
 - a. Write an SQL statement that will increase the Balance in Part_Number 56789 in San Mateo Parts by 10 percent and another SQL statement that will decrease the Balance in Part_Number 12345 in New York Parts by 10 percent.
 - b. Indicate whether the statement you wrote in part a is acceptable under each of the following protocols:
 - Remote unit of work
 - Distributed unit of work
 - Distributed request
4. Speculate on why you think a truly heterogeneous distributed database environment is so difficult to achieve. What specific difficulties exist in this environment?
5. Explain the major factors at work in creating the drastically different results for the six query-processing strategies outlined in Table 13-2.
 6. Do any of the six query-processing strategies in Table 13-2 utilize a semijoin? If so, explain how a semijoin is used. If not, explain how you might use a semijoin to create an efficient query-processing strategy or why the use of a semijoin will not work in this situation.
 7. Consider the SUPPLIER, PART, and SHIPMENT relations and distributed database mentioned in the section on Query Optimization in this chapter.
 - a. Write a global SQL query (submitted in Chicago) to display the Part_Number and Color for every part that is not supplied by a supplier in Columbus.
 - b. Design three alternative query-processing strategies for your answer to part a.
 - c. Develop a table similar to Table 13-2 to compare the processing times for these three strategies.
 - d. Which of your three strategies was best and why?
 - e. Would data replication or horizontal or vertical partitioning of the database allow you to create an even more efficient query-processing strategy? Why or why not?
 8. Consider the following normalized relations for a database in a large retail store chain:

STORE (Store_ID, Region, Manager_ID, Square_Feet)

EMPLOYEE (Employee_ID, Where_Work, Employee_Name, Employee_Address)

DEPARTMENT (Department_ID, Manager_ID, Sales_Goal)

SCHEDULE (Department_ID, Employee_ID, Date)

Assume that a data communications network links a computer at corporate headquarters with a computer in each retail outlet. The chain includes fifty stores with an average of seventy-five employees per store. There are ten departments in each store. A daily schedule is maintained for 5 months (the previous 2 months, the current month, and next 2 months). Further assume that

 - Each store manager updates the employee work schedule for her or his store roughly five times per hour.
 - The corporation generates all payroll checks, employee notices, and other mailings for all employees for all stores.
 - The corporation establishes a new sales goal each month for each department.
 - The corporation hires and fires store managers and controls all information about store managers; store managers hire and fire all store employees and control all information about employees in that store.
 - a. Would you recommend a distributed database, a centralized database, or a set of decentralized databases for this retail store chain?

- b. Assuming that some form of distributed database is justified, what would you recommend as a data distribution strategy for this retail store chain?

Problems and Exercises 9–14 refer to the Fitchwood Insurance Company, a case study introduced in the Problems and Exercises for Chapter 11.

9. Let's assume that the data mart needs to be accessed by Fitchwood's main office as well as its service center in Florida. Keeping in mind that data are updated weekly, would you recommend a distributed database, a centralized database, or set of decentralized databases? State any assumptions.
10. Assuming that a distributed database is justified, what would you recommend for a data distribution strategy?
11. Explain how you would accomplish weekly updates of the data mart given that a distributed database was justified.
12. The sales and marketing organization would also like to enable agents to access the data mart in order to produce commission reports and to follow-up on clients. Assuming that there are 10 different offices, what strategy would you recommend for distributing the data mart?
13. How would your strategy change for Problem and Exercise 12 if management did not want agents to have a copy of any data but their own? Explain how you would accomplish this.
14. How would your overall distribution strategy differ if this were an OLTP system instead of a data mart?
15. Research the Web for relevant articles on Web services and how they may impact distributed databases. Report on your findings.

Field Exercises

1. Visit an organization that has installed a distributed database management system. Explore the following questions:
 - a. Does the organization have a truly distributed database? If so, how are the data distributed: replication, horizontal partitioning, or vertical partitioning?
 - b. What commercial distributed DBMS products are used? What were the reasons the organization selected these products? What problems or limitations has the organization found with these products?
 - c. To what extent does this system provide each of the following:
 - Location transparency
 - Replication transparency
 - Concurrency transparency
 - Failure transparency
 - Query optimization
 - d. What are the organization's plans for future evolution of its distributed databases?
 - e. Talk with a database administrator in the organization to explore how decisions are made concerning the location

Problems and Exercises 16–19 relate to the Pine Valley Furniture Company case study discussed throughout the text.

16. Pine Valley Furniture has opened up another office for receiving and processing orders. This office will deal exclusively with customers west of the Mississippi River. The order processing center located at the manufacturing plant will process orders for customers west of the Mississippi River as well as international customers. All products will still be shipped to customers from the manufacturing facility, thus inventory levels must be accessed and updated from both offices. Would you recommend a distributed database or a centralized database? Explain your answer.
17. Management would like to consider utilizing one centralized database at the manufacturing facility that can be accessed via a wide area network (WAN) from the remote order processing center. Discuss the advantages and disadvantages of this.
18. Assuming that management decides on a distributed database, what data distribution strategy would you recommend?
19. Certain items are available to only international customers and customers on the East Coast. How would this change your distribution strategy?
20. Management has decided to add an additional warehouse for customers west of the Mississippi. Items that are not custom built are shipped from this warehouse. Custom built and specialty items are shipped from the manufacturing facility. What additional tables and changes in distribution strategy, if any, would be needed in order to accommodate this?

of data in the network. What factors are considered in this decision? Are any analytical tools used? If so, is the database administrator satisfied that the tools help to make the processing of queries efficient?

2. Using the World Wide Web, investigate the latest distributed database product offerings from the DBMS vendors mentioned in this chapter. Update the description of the features for one of the distributed DBMS products listed. Search for distributed DBMS products from other vendors and include information about these products in your answer.
3. Visit an organization that has installed a client/server database environment. Explore the following questions:
 - a. What distributed database features do the client/server DBMSs in use offer?
 - b. Is the organization attempting to achieve the same benefits from a client/server environment as are outlined in this chapter for distributed databases? Which of these benefits are they achieving? Which cannot be achieved with client/server technologies?


References

- Bell, D., and J. Grimson. 1992. *Distributed Database Systems*. Reading, MA: Addison-Wesley.
- Buretta, M. 1997. *Data Replication: Tools and Techniques for Managing Distributed Information*. New York: Wiley.
- Date, C. J. 1983. *An Introduction to Database Systems*, Vol. 2. Reading, MA: Addison-Wesley.
- Date, C. J. 1995. *An Introduction to Database Systems*, 6th ed. Reading, MA: Addison-Wesley.
- Edelstein, H. 1993. "Replicating Data." *DBMS* 6,6 (June): 59–64.
- Edelstein, H. 1995a. "The Challenge of Replication, Part I." *DBMS* 8,3 (March): 46–52.
- Elmasri, R., and S. B. Navathe. 1989. *Fundamentals of Database Systems*. Menlo Park, CA: Benjamin/Cummings.
- Froemming, G. 1996. "Design and Replication: Issues with Mobile Applications—Part I." *DBMS* 9,3 (March): 48–56.
- Koop, P. 1995. "Replication at Work." *DBMS* 8,3 (March): 54–60.
- McGovern, D. 1993. "Two-Phased Commit or Replication." *Database Programming & Design* 6,5 (May): 35–44.
- Özsu, M. T., and P. Valduriez. 1992. "Distributed Database Systems: Where Were We?" *Database Programming & Design* 5,4 (April): 49–55.
- Thé, L. 1994. "Distribute Data Without Choking the Net." *Datamation* 40,1 (January 7): 35–38.
- Thompson, C. 1997. "Database Replication: Comparing Three Leading DBMS Vendors' Approaches to Replication." *DBMS* 10,5 (May): 76–84.

Further Reading

- Edelstein, H. 1995. "The Challenge of Replication, Part II." *DBMS* 8,4 (April): 62–70, 103.

Web Resources

 **www.wide.ad.jp** The WIDE project is a research and development project that is concerned with extending the technology of distributed and active databases to provide added value to advanced, application-oriented software products implementing workflow techniques. These problems have been tackled by a consortium involving partners from organizations in Spain, Italy, and the Netherlands.

www.compapp.dcu.ie This site, maintained by Dublin City University, has a variety of material on several important aspects of distributed databases.

databases.about.com This Web site contains a variety of news and reviews about various database technologies, including distributed databases.