CHAPTER 24 Distributed Databases and Client-Server Architecture

In this chapter we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. The DDB technology emerged as a merger of two technologies: (1) database technology, and (2) network and data communication technology. The latter has made tremendous strides in terms of wired and wireless technologies—from satellite and cellular communications and Metropolitan Area Networks (MANs) to the standardization of protocols like Ethernet, TCP/IP, and the Asynchronous Transfer Mode (ATM) as well as the explosion of the Internet, including the newly started Internet-2 development. While early databases moved toward centralization and resulted in monolithic gigantic databases in the seventies and early eighties, the trend reversed toward more decentralization and autonomy of processing in the late eighties. With advances in distributed processing and distributed computing that occurred in the operating systems arena, the database research community did considerable work to address the issues of data distribution, distributed query and transaction processing, distributed database metadata management, and other topics, and developed many research prototypes. However, a full-scale comprehensive DDBMS that implements the functionality and techniques proposed in DDB research never emerged as a commercially viable product. Most major vendors redirected their efforts from developing a "pure" DDBMS product into developing systems based on client-server, or toward developing active heterogeneous DBMSs.

Organizations, however, have been very interested in the *decentralization* of processing (at the system level) while achieving an *integration* of the information resources (at the logical level) within their geographically distributed systems of databases, applications, and users. Coupled with the advances in communications, there is now a general endorsement of the client-server approach to application development, which assumes many of the DDB issues.

In this chapter we discuss both distributed databases and client-server architectures,[1] in the development of database technology that is closely tied to advances in communications and network technology. Details of the latter are outside our scope; the reader is referred to a series of texts on data communications (see the Selected Bibliography at the end of this chapter).

Section 24.1 introduces distributed database management and related concepts. Detailed issues of distributed database design, involving fragmenting of data and distributing it over multiple sites with possible replication, are discussed in Section 24.2. Section 24.3 introduces different types of distributed database systems, including federated and multidatabase systems and highlights the problems of heterogeneity and the needs of autonomy in federated database systems, which will dominate for years to come. Sections 24.4 and 24.5 introduce distributed database query and transaction processing techniques, respectively. Section 24.6 discusses how the client-server architectural concepts are related to distributed databases. Section 24.7 elaborates on future issues in client-server architectures. Section 24.8 discusses distributed database features of the Oracle RDBMS.

For a short introduction to the topic, only sections 24.1, 24.3, and 24.6 may be covered.

24.1 Distributed Database Concepts

Distributed databases bring the advantages of distributed computing to the database management domain. A **distributed computing system** consists of a number of processing elements, not necessarily homogeneous, that are interconnected by a computer network, and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The economic viability of this approach stems from two reasons: (1) more computer power is harnessed to solve a complex task, and (2) each autonomous processing element can be managed independently and develop its own applications.

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.[2] A collection of files stored at different nodes of a network and the maintaining of inter relationships among them via hyperlinks has become a common organization on the Internet, with files of Web pages.

1. The reader should review the introduction to client-server architecture in Section 17.1.
2. This definition and some of the discussion in this section is based on Ozsu and Valduriez (1999).

The common functions of database management, including uniform query processing and transaction processing, *do not* apply to this scenario yet. The technology is, however, moving in a direction such that distributed World Wide Web (WWW) databases will become a reality in the near future. We shall discuss issues of accessing databases on the Web in Section 27.1 and mobile and intermittently connected databases in Section 27.3. None of those qualify as DDB by the definition given earlier.

24.1.1 Parallel Versus Distributed Technology

Turning our attention to system architectures, there are two main types of multiprocessor system architectures that are commonplace:

- *Shared memory (tightly coupled) architecture:* Multiple processors share secondary (disk) storage and also share primary memory.
- *Shared disk (loosely coupled) architecture:* Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.[3] Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMS, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared nothing architecture.** In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for parallel databases. Figure 24.1 contrasts these different architectures.

24.1.2 Advantages of Distributed Databases

Distributed database management has been proposed for various reasons ranging from organizational decentralization and economical processing to greater autonomy. We highlight some of these advantages here.

1. *Management of distributed data with different levels of transparency:* Ideally, a DBMS should be **distribution transparent** in the sense of hiding the details of where each file (table, relation) is physically stored within the system. Consider the company database in Figure 7.5 that we have been discussing throughout the book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally

3. If both primary and secondary memories are shared, the architecture is also known as **shared everything architecture.**

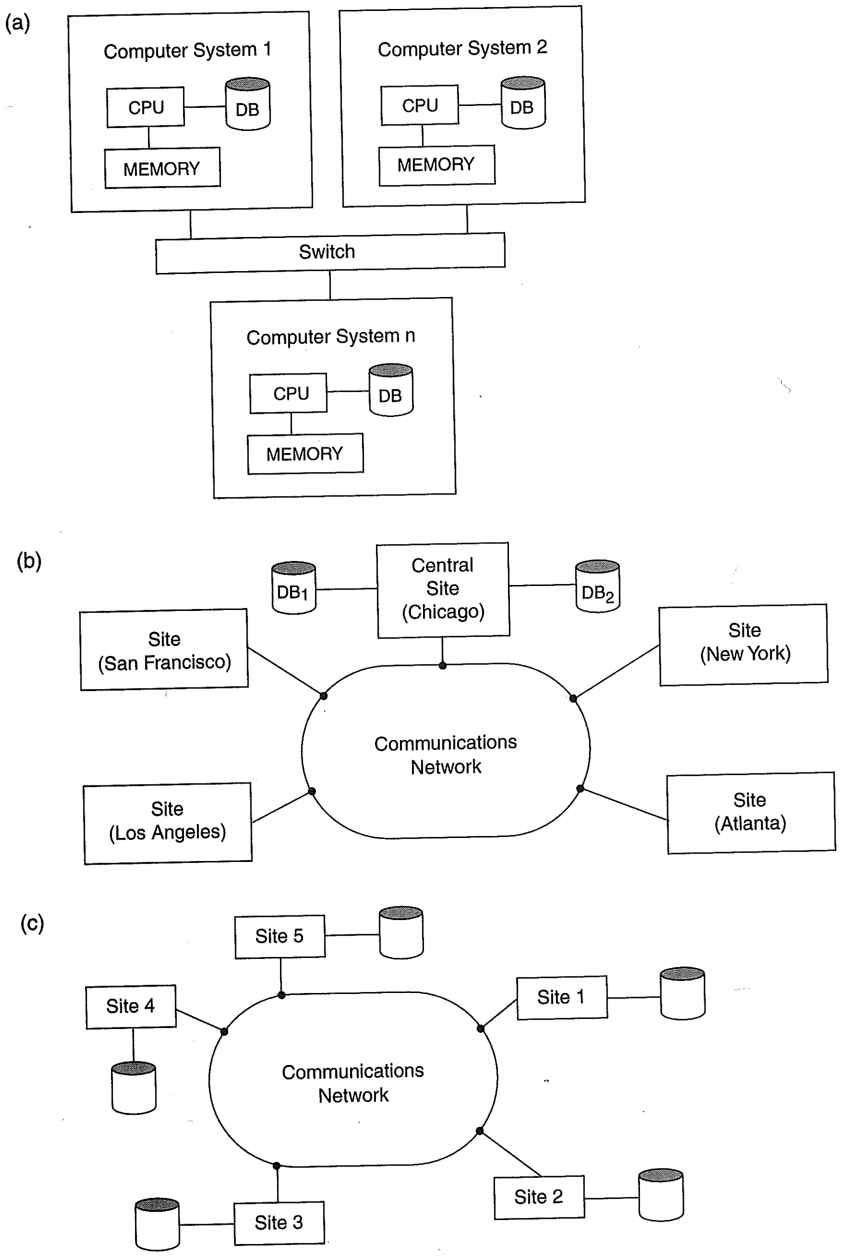

Figure 24.1 Some different database system architectures. (a) Shared nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.

(that is, into sets of rows, as we shall discuss in Section 24.2) and stored with possible replication as shown in Figure 24.2. The following types of transparencies are possible:

- *Distribution or network transparency:* This refers to freedom for the user from the operational details of the network. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of data and the location of the system where the command was issued. **Naming transparency** implies that once a name is specified, the named objects can be accessed unambiguously without additional specification.
- *Replication transparency:* As we show in Figure 24.2, copies of data may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of copies.
- *Fragmentation transparency:* Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation into sets of tuples (rows). **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. A global query by the user must be transformed into several fragment queries. Fragmentation transparency makes the user unaware of the existence of fragments.

2. *Increased reliability and availability:* These are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas

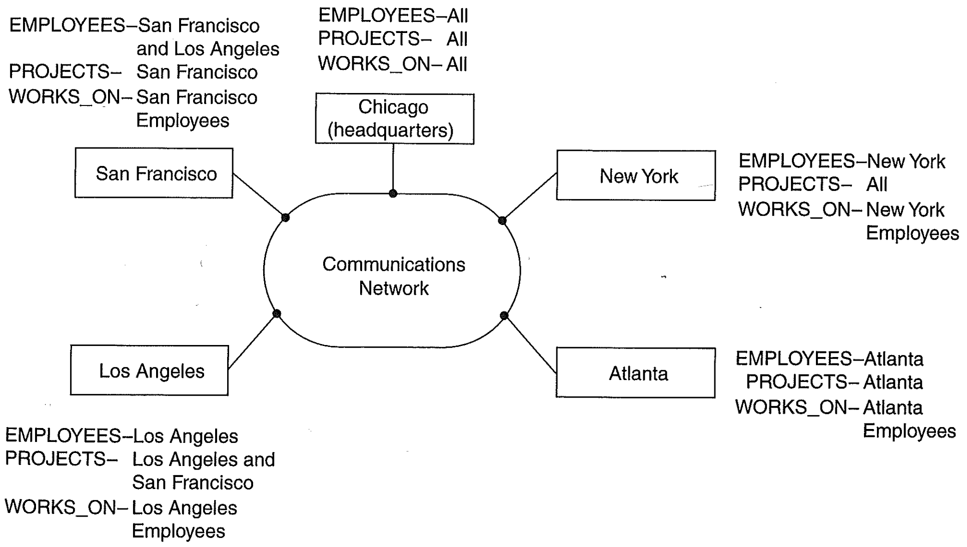

Figure 24.2 Data distribution and replication among distributed databases

availability is the probability that the system is continuously available during a time interval. When the data and DBMS software are distributed over several sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. This improves both reliability and availability. Further improvement is achieved by judiciously *replicating* data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database.

3. *Improved performance:* A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. *Easier expansion:* In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more processors is much easier.

The transparencies we discussed in (1) above lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy,** which gives the users tighter control over their own local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations. In addition, transparency impacts the features that must be provided by the operating system and the DBMS.

24.1.3 Additional Functions of Distributed Databases

Distribution leads to increased complexity in the system design and implementation. To achieve the potential advantages listed previously, the DDBMS software must be able to provide the following functions in addition to those of a centralized DBMS:

- *Keeping track of data:* The ability to keep track of the data distribution, fragmentation, and replication by expanding the DDBMS catalog.
- *Distributed query processing:* The ability to access remote sites and transmit queries and data among the various sites via a communication network.
- *Distributed transaction management:* The ability to devise execution strategies for queries and transactions that access data from more than one site and to synchronize the access to distributed data and maintain integrity of the overall database.

- *Replicated data management:* The ability to decide which copy of a replicated data item to access and to maintain the consistency of copies of a replicated data item.
- *Distributed database recovery:* The ability to recover from individual site crashes and from new types of failures such as the failure of a communication links.
- *Security:* Distributed transactions must be executed with the proper management of the security of the data and the authorization/access privileges of users.
- *Distributed directory (catalog) management:* A directory contains information (metadata) about data in the database. The directory may be global for the entire DDB, or local for each site. The placement and distribution of the directory are design and policy issues.

These functions themselves increase the complexity of a DDBMS over a centralized DBMS. Before we can realize the full potential advantages of distribution, we must find satisfactory solutions to these design issues and problems. Including all this additional functionality is hard to accomplish, and finding optimal solutions is a step beyond that.

At the physical **hardware** level, the following main factors distinguish a DDBMS from a centralized system:

- There are multiple computers, called **sites** or **nodes.**
- These sites must be connected by some type of **communication network** to transmit data and commands among sites, as shown in Figure 24.1(c).

The sites may all be located in physical proximity—say, within the same building or group of adjacent buildings—and connected via a **local area network,** or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network.** Local area networks typically use cables, whereas long-haul networks use telephone lines or satellites. It is also possible to use a combination of the two types of networks.

Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant effect on performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter which type of network is used; it only matters that each site is able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of communication network exists among sites, regardless of the particular topology. We will not address any network specific issues, although it is important to understand that for an efficient operation of a DDBS, network design and performance issues are very critical.

24.2 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

In this section we discuss techniques that are used to break up the database into logical units, called **fragments,** which may be assigned for storage at the various sites. We also discuss the use of **data replication,** which permits certain data to be stored in more than one site, and the process of **allocating** fragments—or replicas of fragments—for storage at the various sites. These techniques are used during the process of **distributed database design.** The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

24.2.1 Data Fragmentation

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is *no replication;* that is, each relation—or portion of a relation—is to be stored at only one site. We discuss replication and its effects later in this section. We also use the terminology of relational databases—similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites. To illustrate our discussion, we use the relational database schema in Figure 7.5.

Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT of Figure 7.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 7.6, and assume there are three computer sites—one for each department in the company.[4] We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

Horizontal Fragmentation. A **horizontal fragment** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved. For example, we may define three horizontal fragments on the EMPLOYEE relation of Figure 7.6 with the following conditions: (DNO = 5), (DNO = 4), and (DNO = 1)—each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (DNUM = 5), (DNUM = 4), and (DNUM = 1)—each fragment contains the PROJECT tuples controlled by a particular department. **Horizontal fragmentation** divides a relation "horizontally" by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation. Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation**

4. Of course, in an actual situation, there will be many more tuples in the relations than those shown in Figure 7.6.

divides a relation "vertically" by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—NAME, BDATE, ADDRESS, and SEX—and the second includes work-related information—SSN, SALARY, SUPERSSN, DNO. This vertical fragmentation is not quite proper because, if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is *no common attribute* between the two fragments. It is necessary to include the primary key or some candidate key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the SSN attribute to the personal information fragment.

Notice that each horizontal fragment on a relation R can be specified by a $\sigma_{Ci}(R)$ operation in the relational algebra. A set of horizontal fragments whose conditions C1, C2, ..., Cn include all the tuples in R—that is, every tuple in R satisfies (C1 OR C2 OR ··· OR Cn)—is called a **complete horizontal fragmentation** of R. In many cases a complete horizontal fragmentation is also **disjoint;** that is, no tuple in R satisfies (Ci AND Cj) for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a π_{Li} (R) operation in the relational algebra. A set of vertical fragments whose projection lists L1, L2, ..., Ln include all the attributes in R but share only the primary key attribute of R is called a **complete vertical fragmentation** of R. In this case the projection lists satisfy the following two conditions:

- $L1 \cup L2 \cup \ldots \cup Ln = ATTRS(R)$.
- $Li \cap Lj = PK(R)$ for any $i \neq j$, where ATTRS(R) is the set of attributes of R and PK(R) is the primary key of R.

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists L1 = {SSN, NAME, BDATE, ADDRESS, SEX} and L2 = {SSN, SALARY, SUPERSSN, DNO} constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation of Figure 7.5 by the conditions (SALARY > 50000) and (DNO = 4); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists L1 = {NAME, ADDRESS} and L2 = {SSN, NAME, SALARY}; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) Fragmentation. We can intermix the two types of fragmentation, yielding a **mixed fragmentation.** For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If $C = \text{TRUE}$ (that is, all tuples are selected) and $L \neq \text{ATTRS}(R)$, we get a vertical fragment, and if $C \neq \text{TRUE}$ and $L = \text{ATTRS}(R)$, we get a horizontal fragment. Finally, if $C \neq \text{TRUE}$ and $L \neq \text{ATTRS}(R)$, we get a mixed fragment. Notice that a relation can itself be considered a fragment with $C = \text{TRUE}$ and $L = \text{ATTRS}(R)$. In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated.** We discuss data replication and allocation next.

24.2.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database.** This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries, because the result of such a query can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there were no replication, as we shall see in Section 24.5.

The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation.**

Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and personal digital assistants and synchronize them periodically with the server database.[5] A description of the replication of fragments is sometimes called a **replication schema.**

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required and transactions can be submitted at any site and if most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

24.2.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database of Figures 7.5 and 7.6. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department*. Further, we assume that these sites mainly access the NAME, SSN, SALARY, and SUPERSSN attributes of EMPLOYEE. Site 1 is used by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database of Figure 7.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, we can first horizontally fragment DEPARTMENT by its key DNUMBER. We then apply derived fragmentation to the relations EMPLOYEE, PROJECT, and DEPT_LOCATIONS relations based on their foreign keys for department number—called DNO, DNUM, and DNUMBER, respectively, in Figure 7.5. We can then vertically fragment the resulting EMPLOYEE fragments to include only the attributes {NAME, SSN, SALARY, SUPERSSN, DNO}. Figure 24.3 shows the mixed fragments EMPD5 and EMPD4, which include the EMPLOYEE tuples satisfying the conditions DNO = 5 and DNO = 4, respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at the headquarters site 1.

We must now fragment the WORKS_ON relation and decide which fragments of WORKS_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS_ON relates an employee e to a project p. We could fragment WORKS_ON based on the department d in which e works *or* based on the department d' that controls p. Fragmentation

5. For a scalable approach to synchronize partially replicated databases, see Mahajan et al. (1998).

(a)

EMPD5	FNAME	MINIT	LNAME	SSN	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	30000	333445555	5
	Franklin	T	Wong	333445555	40000	888665555	5
	Ramesh	K	Narayan	666884444	38000	333445555	5
	Joyce	A	English	453453453	25000	333445555	5

DEP5	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22

DEP5_LOCS	DNUMBER	LOCATION
	5	Bellaire
	5	Sugarland
	5	Houston

WORKS_ON5	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0

PROJS5	PNAME	PNUMBER	PLOCATION	DNUM
	Product X	1	Bellaire	5
	Product Y	2	Sugarland	5
	Product Z	3	Houston	5

Data at Site 2

(b)

EMPD4	FNAME	MINIT	LNAME	SSN	SALARY	SUPERSSN	DNO
	Alicia	J	Zelaya	999887777	25000	987654321	4
	Jennifer	S	Wallace	987654321	43000	888665555	4
	Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP4	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Administration	4	987654321	1995-01-01

DEP4_LOCS	DNUMBER	LOCATION
	4	Stafford

WORKS_ON4	ESSN	PNO	HOURS
	333445555	10	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0

PROJS4	PNAME	PNUMBER	PLOCATION	DNUM
	Computerization	10	Stafford	4
	Newbenefits	30	Stafford	4

Data at Site 3

Figure 24.3 Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

becomes easy if we have a constraint stating that d = d′ for all WORKS_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database of Figure 7.6. For example, the WORKS_ON tuple `<333445555, 10, 10.0>` relates an employee who works for department 5 with a project controlled by department 4. In this case we could fragment WORKS_ON based on the department in which the employee works (which is expressed by the condition C) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure 24.4.

(a)

G1	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0

C1=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))

G2	ESSN	PNO	HOURS
	333445555	10	10.0

C2=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))

G3	ESSN	PNO	HOURS
	333445555	20	10.0

C3=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))

Employees in Department 5

(b)

G4	ESSN	PNO	HOURS

C4=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))

G5	ESSN	PNO	HOURS
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0

C5=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))

G6	ESSN	PNO	HOURS
	987654321	20	15.0

C6=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))

Employees in Department 4

(c)

G7	ESSN	PNO	HOURS

C7=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))

G8	ESSN	PNO	HOURS

C8=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))

G9	ESSN	PNO	HOURS
	888665555	20	null

C9=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))

Employees in Department 1

Figure 24.4 Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=1)]).

In Figure 24.4, the union of fragments G1, G2, and G3 gives all WORKS_ON tuples for employees who work for department 5. Similarly, the union of fragments G4, G5, and G6 gives all WORKS_ON tuples for employees who work for department 4. On the other hand, the union of fragments G1, G4, and G7 gives all WORKS_ON tuples for projects controlled by department 5. The condition for each of the fragments G1 through G9 is shown in Figure 24.4. The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution of Figure 24.3, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G1, G2, G3, G4, and G7 at site 2 and the union of fragments G4, G5, G6, G2, and G8 at site 3. Notice that fragments G2 and G4 are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.