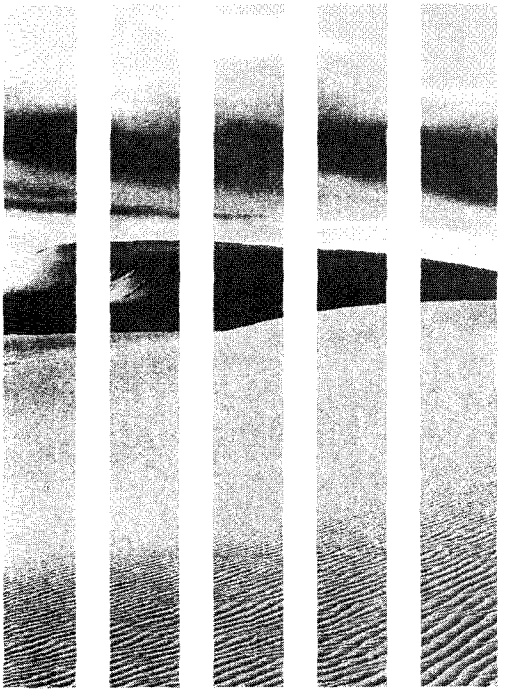


7

FURTHER TOPICS

23

Database Security and Authorization

This chapter discusses the techniques used for protecting the database against persons who are not authorized to access either certain parts of a database or the whole database. Section 23.1 provides an introduction to security issues and the threats to databases and an overview of the countermeasures that are covered in the rest of this chapter. Section 23.2 discusses the mechanisms used to grant and revoke privileges in relational database systems and in SQL, mechanisms that are often referred to as **discretionary access control.** Section 23.3 offers an overview of the mechanisms for enforcing multiple levels of security—a more recent concern in database system security that is known as **mandatory access control.** It also introduces the more recently developed strategy of **role-based access control.** Section 23.4 briefly discusses the security problem in statistical databases. Section 23.5 introduces flow control and mentions problems associated with covert channels. Section 23.6 is a brief summary of encryption and public key infrastructure schemes. Section 23.7 summarizes the chapter. Readers who are interested only in basic database security mechanisms will find it sufficient to cover the material in Sections 23.1 and 23.2.

23.1 Introduction to Database Security Issues

23.1.1 Types of Security

Database security is a very broad area that addresses many issues, including the following:

- Legal and ethical issues regarding the right to access certain information. Some information may be deemed to be private and cannot be accessed legally by unauthorized persons. In the United States, there are numerous laws governing privacy of information.
- Policy issues at the governmental, institutional, or corporate level as to what kinds of information should not be made publicly available—for example, credit ratings and personal medical records.
- System-related issues such as the *system levels* at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level, or the DBMS level.
- The need in some organizations to identify multiple *security levels* and to categorize the data and users based on these classifications—for example, top secret, secret, confidential, and unclassified. The security policy of the organization with respect to permitting access to various classifications of data must be enforced.

Threats to Databases. Threats to databases result in the loss or degradation of some or all of the following security goals: integrity, availability, and confidentiality.

- *Loss of integrity:* Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creation, insertion, modification, changing the status of data, and deletion. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity is not corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
- *Loss of availability:* Database availability refers to making objects available to a human user or a program to which they have a legitimate right.
- *Loss of confidentiality:* Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.

To protect databases against these types of threats four kinds of countermeasures can be implemented: access control, inference control, flow control, and encryption. We discuss each of these in this chapter.

In a multiuser database system, the DBMS must provide techniques to enable certain users or user groups to access selected portions of a database without gaining access to the rest of the database. This is particularly important when a large integrated database is to be used by many different users within the same organization. For example, sensitive

information such as employee salaries or performance reviews should be kept confidential from most of the database system's users. A DBMS typically includes a **database security and authorization subsystem** that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:

- *Discretionary security mechanisms:* These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
- *Mandatory security mechanisms:* These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification level to see only the data items classified at the user's own (or lower) classification level. An extension of this is *role-based security*, which enforces policies and privileges based on the concept of roles.

We discuss discretionary security in Section 23.2 and mandatory and role-based security in Section 23.3.

A second security problem common to all computer systems is that of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole. This function is called **access control** and is handled by creating user accounts and passwords to control the login process by the DBMS. We discuss access control techniques in Section 23.1.3.

A third security problem associated with databases is that of controlling the access to a **statistical database,** which is used to provide statistical information or summaries of values based on various criteria. For example, a database for population statistics may provide statistics based on age groups, income levels, size of household, education levels, and other criteria. Statistical database users such as government statisticians or market research firms are allowed to access the database to retrieve statistical information about a population but not to access the detailed confidential information on specific individuals. Security for statistical databases must ensure that information on individuals cannot be accessed. It is sometimes possible to deduce or infer certain facts concerning individuals from queries that involve only summary statistics on groups; consequently, this must not be permitted either. This problem, called **statistical database security,** is discussed briefly in Section 23.4. The corresponding countermeasures are called **inference control** measures.

Another security issue is that of **flow control,** which prevents information from flowing in such a way that it reaches unauthorized users. It is discused in Section 23.5. Channels that are pathways for information to flow implicitly in ways that violate the security policy of an organization are called **covert channels.** We briefly discuss some issues related to covert channels in Section 23.5.1.

A final security issue is **data encryption,** which is used to protect sensitive data (such as credit card numbers) that is being transmitted via some type of communications network. Encryption can be used to provide additional protection for sensitive portions of a database as well. The data is **encoded** using some coding algorithm. An unauthorized user who accesses encoded data will have difficulty deciphering it, but authorized users are given decoding or

decrypting algorithms (or keys) to decipher the data. Encrypting techniques that are very difficult to decode without a key have been developed for military applications. Section 23.6 briefly discusses encryption techniques, including popular techniques such as public key encryption, which is heavily used to support Web-based transactions against databases, and digital signatures, which are used in personal communications.

A complete discussion of security in computer systems and databases is outside the scope of this textbook. We give only a brief overview of database security techniques here. The interested reader can refer to several of the references discussed in the selected bibliography at the end of this chapter for a more comprehensive discussion.

23.1.2 Database Security and the DBA

As we discussed in Chapter 1, the database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization. The DBA has a **DBA account** in the DBMS, sometimes called a **system** or **superuser account,** which provides powerful capabilities that are not made available to regular database accounts and users.[1] DBA-privileged commands include commands for granting and revoking privileges to individual accounts, users, or user groups and for performing the following types of actions:

1. *Account creation:* This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. *Privilege granting:* This action permits the DBA to grant certain privileges to certain accounts.
3. *Privilege revocation:* This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. *Security level assignment:* This action consists of assigning user accounts to the appropriate security classification level.

The DBA is responsible for the overall security of the database system. Action 1 in the preceding list is used to control access to the DBMS as a whole, whereas actions 2 and 3 are used to control *discretionary* database authorization, and action 4 is used to control *mandatory* authorization.

23.1.3 Access Protection, User Accounts, and Database Audits

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account**

1. This account is similar to the *root* or *superuser* accounts that are given to computer system administrators, allowing access to restricted operating system commands.

number and **password** for the user if there is a legitimate need to access the database. The user must **log in** to the DBMS by entering the account number and password whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database. Application programs can also be considered as users and can be required to supply passwords.

It is straightforward to keep track of database users and their accounts and passwords by creating an encrypted table or file with the two fields AccountNumber and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session,** which consists of the sequence of database interactions that a user performs from the time of logging in to the time of logging off. When a user logs in, the DBMS can record the user's account number and associate it with the terminal from which the user logged in. All operations applied from that terminal are attributed to the user's account until the user logs off. It is particularly important to keep track of update operations that are applied to the database so that, if the database is tampered with, the DBA can find out which user did the tampering.

To keep a record of all updates applied to the database and of the particular user who applied each update, we can modify the *system log*. Recall from Chapters 17 and 19 that the **system log** includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash. We can expand the log entries so that they also include the account number of the user and the online terminal ID that applied each operation recorded in the log. If any tampering with the database is suspected, a **database audit** is performed, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period. When an illegal or unauthorized operation is found, the DBA can determine the account number used to perform this operation. Database audits are particularly important for sensitive databases that are updated by many transactions and users, such as a banking database that is updated by many bank tellers. A database log that is used mainly for security purposes is sometimes called an **audit trail.**

23.2 Discretionary Access Control Based on Granting and Revoking Privileges

The typical method of enforcing **discretionary access control** in a database system is based on the granting and revoking of **privileges.** Let us consider privileges in the context of a relational DBMS. In particular, we will discuss a system of privileges somewhat similar to the one originally developed for the SQL language (see Chapter 8). Many current relational DBMSs use some variation of this technique. The main idea is to include statements in the query language that allow the DBA and selected users to grant and revoke privileges.

23.2.1 Types of Discretionary Privileges

In SQL2, the concept of an **authorization identifier** is used to refer, roughly speaking, to a user account (or group of user accounts). For simplicity, we will use the words *user* or *account* interchangeably in place of *authorization identifier*. The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus, having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to use the database system:

- *The account level:* At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- *The relation (or table) level:* At this level, the DBA can control the privilege to access each individual relation or view in the database.

The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query. Notice that these account privileges apply to the account in general. If a certain account does not have the CREATE TABLE privilege, no relations can be created from that account. Account-level privileges *are not* defined as part of SQL2; they are left to the DBMS implementers to define. In earlier versions of SQL, a CREATETAB privilege existed to give an account the privilege to create tables (relations).

The second level of privileges applies to the **relation level,** whether they are base relations or virtual (view) relations. These privileges *are* defined for SQL2. In the following discussion, the term *relation* may refer either to a base relation or to a view, unless we explicitly specify one or the other. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the *relation and attribute level only*. Although this is quite general, it makes it difficult to create accounts with limited privileges. The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the **access matrix model,** where the rows of a matrix M represent *subjects* (users, accounts, programs) and the columns represent *objects* (relations, records, columns, views, operations). Each position $M(i, j)$ in the matrix represents the types of privileges (read, write, update) that subject i holds on object j.

To control the granting and revoking of relation privileges, each relation R in a database is assigned an **owner account,** which is typically the account that was used when the relation was created in the first place. The owner of a relation is given *all* privileges on that relation. In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command (see Section 8.1.1). The owner account holder can pass privileges on any of the owned relations to other users by **granting** privileges to their

accounts. In SQL the following types of privileges can be granted on each individual relation R:

- SELECT (retrieval or read) privilege on R: Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from R.
- MODIFY privileges on R: This gives the account the capability to modify tuples of R. In SQL this privilege is further divided into UPDATE, DELETE, and INSERT privileges to apply the corresponding SQL command to R. In addition, both the INSERT and UPDATE privileges can specify that only certain attributes of R can be updated by the account.
- REFERENCES privilege on R: This gives the account the capability to reference relation R when specifying integrity constraints. This privilege can also be restricted to specific attributes of R.

Notice that to create a view, the account must have SELECT privilege on *all relations* involved in the view definition.

23.2.2 Specifying Privileges Using Views

The mechanism of **views** is an important discretionary authorization mechanism in its own right. For example, if the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only those attributes and then grant SELECT on V to B. The same applies to limiting B to retrieving only certain tuples of R; a view V' can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access. We shall illustrate this discussion with the example given in Section 23.2.5.

23.2.3 Revoking Privileges

In some cases it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL a REVOKE command is included for the purpose of canceling privileges. We will see how the REVOKE command is used in the example in Section 23.2.5.

23.2.4 Propagation of Privileges Using the GRANT OPTION

Whenever the owner A of a relation R grants a privilege on R to another account B, the privilege can be given to B *with* or *without* the **GRANT OPTION**. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts. Suppose that B is given the GRANT OPTION by A and that B then grants the privilege on R to a

third account C, also with GRANT OPTION. In this way, privileges on *R* can **propagate** to other accounts without the knowledge of the owner of *R*. If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.

It is possible for a user to receive a certain privilege from two or more sources. For example, A4 may receive a certain UPDATE *R* privilege from *both* A2 and A3. In such a case, if A2 revokes this privilege from A4, A4 will still continue to have the privilege by virtue of having been granted it from A3. If A3 later revokes the privilege from A4, A4 totally loses the privilege. Hence, a DBMS that allows propagation of privileges must keep track of how all the privileges were granted so that revoking of privileges can be done correctly and completely.

23.2.5 An Example

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations; then the DBA must issue the following GRANT command in SQL:

```
GRANT CREATETAB TO A1;
```

The CREATETAB (create table) privilege gives account A1 the capability to create new database tables (base relations) and is hence an *account privilege*. This privilege was part of earlier versions of SQL but is now left to each individual system implementation to define.

In SQL2, the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command, as follows:

```
CREATE SCHEMA EXAMPLE AUTHORIZATION A1;
```

Now user account A1 can create tables under the schema called EXAMPLE. To continue our example, suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT shown in Figure 23.1; A1 is then the **owner** of these two relations and hence has *all the relation privileges* on each of them.

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. A1 can issue the following command:

```
GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;
```

EMPLOYEE

NAME	SSN	BDATE	ADDRESS	SEX	SALARY	DNO

DEPARTMENT

DNUMBER	DNAME	MGRSSN

FIGURE 23.1 Schemas for the two relations EMPLOYEE and DEPARTMENT.

Notice that the owner account A1 of a relation automatically has the GRANT OPTION, allowing it to grant privileges on the relation to other accounts. However, account A2 cannot grant INSERT and DELETE privileges on the EMPLOYEE and DEPARTMENT tables, because A2 was not given the GRANT OPTION in the preceding command.

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the following command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION;
```

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

Notice that A4 cannot propagate the SELECT privilege to other accounts because the GRANT OPTION was not given to A4.

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

```
REVOKE SELECT ON EMPLOYEE FROM A3;
```

The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege any more.

Next, suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO = 5. A1 then can create the following view:

```
CREATE VIEW A3EMPLOYEE AS
SELECT NAME, BDATE, ADDRESS
FROM EMPLOYEE
WHERE DNO = 5;
```

After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

Finally, suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE; A1 can then issue the following command:

```
GRANT UPDATE ON EMPLOYEE (SALARY) TO A4;
```

The UPDATE or INSERT privilege can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute specific, because this specificity can easily be controlled by creating the appropriate views that include only the desired attributes and granting the corresponding privileges on the views. However, because updating views is not always possible (see Chapter 9), the

UPDATE and INSERT privileges are given the option to specify particular attributes of a base relation that may be updated.

23.2.6 Specifying Limits on Propagation of Privileges

Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and are not a part of SQL. Limiting **horizontal propagation** to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts. **Vertical propagation** is more complicated; it limits the depth of the granting of privileges. Granting a privilege with a vertical propagation of zero is equivalent to granting the privilege with *no* GRANT OPTION. If account A grants a privilege to account B with the vertical propagation set to an integer number $j > 0$, this means that the account B has the GRANT OPTION on that privilege, but B can grant the privilege to other accounts only with a vertical propagation *less than* j. In effect, vertical propagation limits the sequence of GRANT OPTIONs that can be given from one account to the next based on a single original grant of the privilege.

We now briefly illustrate horizontal and vertical propagation limits—which are *not available* currently in SQL or other relational systems—with an example. Suppose that A1 grants SELECT to A2 on the EMPLOYEE relation with horizontal propagation equal to 1 and vertical propagation equal to 2. A2 can then grant SELECT to at most one account because the horizontal propagation limitation is set to 1. In addition, A2 cannot grant the privilege to another account except with vertical propagation set to 0 (no GRANT OPTION) or 1; this is because A2 must reduce the vertical propagation by at least 1 when passing the privilege to others. As this example shows, horizontal and vertical propagation techniques are designed to limit the propagation of privileges.

23.3 Mandatory Access Control and Role-Based Access Control for Multilevel Security[2]

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: A user either has or does not have a certain privilege. In many applications, an *additional security policy* is needed that classifies data and users based on security classes. This approach, known as **mandatory access control,** would typically be *combined* with the discretionary access control mechanisms described in Section 23.2. It is important to note that most commercial DBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in

2. The contribution of Fariborz Farahmand to this and subsequent sections is appreciated.

government, military, and intelligence applications, as well as in many industrial and corporate applications.

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where TS ≥ S ≥ C ≥ U, to illustrate our discussion. The commonly used model for multilevel security, known as the Bell-LaPadula model, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the **clearance** (classification) of a subject S as **class(S)** and to the **classification** of an object O as **class(O).** Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless class(S) ≥ class(O). This is known as the **simple security property.**
2. A subject S is not allowed to write an object O unless class(S) ≤ class(O). This is known as the **star property** (or *-property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a **classification attribute** C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. Hence, a **multilevel relation** schema R with n attributes would be represented as

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

where each C_i represents the *classification attribute* associated with attribute A_i.

The value of the TC attribute in each tuple t—which is the *highest* of all attribute classification values within t—provides a general classification for the tuple itself, whereas each C_i provides a finer security classification for each attribute value within the tuple. The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering.** In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the *apparent key*. This leads to the concept of

polyinstantiation,[3] where several tuples can have the same apparent key value but have different attribute values for users at different classification levels.

We illustrate these concepts with the simple example of a multilevel relation shown in Figure 23.2a, where we display the classification attribute values next to each attribute's value. Assume that the `Name` attribute is the apparent key, and consider the query SELECT * FROM employee. A user with security clearance S would see the same relation shown in Figure 23.2a, since all tuple classifications are less than or equal to S. However, a user with security clearance C would not be allowed to see values for `Salary` of `Brown` and `JobPerformance` of `Smith`, since they have higher classification. The tuples would be *filtered* to appear as shown in Figure 23.2b, with `Salary` and `JobPerformance`

(a) EMPLOYEE

Name		Salary		JobPerformance		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

(b) EMPLOYEE

Name		Salary		JobPerformance		TC
Smith	U	40000	C	null	C	C
Brown	C	null	C	Good	C	C

(c) EMPLOYEE

Name		Salary		JobPerformance		TC
Smith	U	null	U	null	U	U

(d) EMPLOYEE

Name		Salary		JobPerformance		TC
Smith	U	40000	C	Fair	S	S
Smith	U	40000	C	Excellent	C	C
Brown	C	80000	S	Good	C	S

FIGURE 23.2 A multilevel relation to illustrate multilevel security. (a) The original EMPLOYEE tuples. (b) Appearance of EMPLOYEE after filtering for classification C users. (c) Appearance of EMPLOYEE after filtering for classification U users. (d) Polyinstantiation of the Smith tuple.

3. This is similar to the notion of having multiple versions in the database that represent the same real-world object.

appearing as null. For a user with security clearance U, the filtering allows only the `Name` attribute of Smith to appear, with all the other attributes appearing as null (Figure 23.2c). Thus, filtering introduces null values for attribute values whose security classification is higher than the user's security clearance.

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the *same* security classification within each individual tuple. In addition, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user can see the key if the user is permitted to see any part of the tuple at all. Other integrity rules, called **null integrity** and **interinstance integrity,** informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

To illustrate polyinstantiation further, suppose that a user with security clearance C tries to update the value of `JobPerformance` of `Smith` in Figure 23.2 to `'Excellent'`; this corresponds to the following SQL update being issued:

```
UPDATE EMPLOYEE
SET JobPerformance = 'Excellent'
WHERE Name = 'Smith';
```

Since the view provided to users with security clearance C (see Figure 23.2b) permits such an update, the system should not reject it; otherwise, the user could infer that some nonnull value exists for the `JobPerformance` attribute of `Smith` rather than the null value that appears. This is an example of inferring information through what is known as a **covert channel,** which should not be permitted in highly secure systems (see Section 23.5.1). However, the user should not be allowed to overwrite the existing value of `JobPerformance` at the higher classification level. The solution is to create a **polyinstantiation** for the `Smith` tuple at the lower classification level C, as shown in Figure 23.2d. This is necessary since the new tuple cannot be filtered from the existing tuple at classification S.

The basic update operations of the relational model (insert, delete, update) must be modified to handle this and similar situations, but this aspect of the problem is outside the scope of our presentation. We refer the interested reader to the end-of-chapter bibliography for further details.

23.3.1 Comparing Discretionary Access Control and Mandatory Access Control

Discretionary Access Control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains. The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs. The reason is that discretionary authorization models do not impose any control on how information is propagated and used once it has been accessed by users authorized to do so. By contrast, mandatory policies ensure a high

degree of protection—in a way, they prevent any illegal flow of information. They are therefore suitable for military types of applications, which require a high degree of protection. However, mandatory policies have the drawback of being too rigid in that they require a strict classification of subjects and objects into security levels, and therefore they are applicable to very few environments. In many practical situations, discretionary policies are preferred because they offer a better trade-off between security and applicability.

23.3.2 Role-Based Access Control

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprisewide systems. Its basic notion is that permissions are associated with roles, and users are assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands discussed under DAC can then be used to assign and revoke privileges from roles.

RBAC appears to be a viable alternative to traditional discretionary and mandatory access controls; it ensures that only authorized users are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to many roles, but it maps to only one user or a single subject. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.

Role hierarchy in RBAC is a natural way of organizing roles to reflect the organization's lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and antisymmetric.

Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as the time and duration of role activations, and timed triggering of a role by an activation of another role. Using an RBAC model is a highly desirable goal for addressing the key security requirements of Web-based applications. Roles can be assigned to workflow tasks so that a user with any of the roles related to a task may be authorized to execute it and may play a certain role for a certain duration only.

RBAC models have several desirable features, such as flexibility, policy neutrality, better support for security management and administration, and other aspects that make them attractive candidates for developing secure Web-based applications. In contrast, DAC and mandatory access control (MAC) models lack capabilities needed to support the security requirements of emerging enterprises and Web-based applications. In addition, RBAC models can represent traditional DAC and MAC policies as well as user-defined or organization-specific policies. Thus, RBAC becomes a superset model that can in turn mimic the behavior of DAC and MAC systems. Furthermore, an RBAC model provides a natural mechanism for addressing the security issues related to the execution of tasks and workflows. Easier deployment over the Internet has been another reason for the success of RBAC models.

23.3.3 Access Control Policies for E-Commerce and the Web

Electronic commerce (**E-commerce**) environments are characterized by any transactions that are done electronically. They require elaborate access control policies that go beyond traditional DBMSs. In conventional database environments, access control is usually performed using a set of authorizations stated by security officers or users according to some security policies. Such a simple paradigm is not well suited for a dynamic environment like e-commerce. Furthermore, in an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience. Such peculiarities call for more flexibility in specifying access control policies. The access control mechanism must be flexible enough to support a wide spectrum of heterogeneous protection objects.

A second related requirement is the support for content-based access control. Content-based access control allows one to express access control policies that take the protection object content into account. In order to support content-based access control, access control policies must allow inclusion of conditions based on the object content.

A third requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications rather than on very specific and individual characteristics (e.g., user IDs). A possible solution, to better take into account user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age, position within an organization). For instance, by using credentials, one can simply formulate policies such as "Only permanent staff with 5 or more years of service can access documents related to the internals of the system."

It is believed that the XML language can play a key role in access control for e-commerce applications.[4] The reason is that XML is becoming the common representation language for document interchange over the Web, and is also becoming the language for e-commerce. Thus, on the one hand there is the need to make XML representations secure, by providing access control mechanisms specifically tailored to the protection of XML documents. On the other hand, access control information (that is, access control policies and user credentials) can be expressed using XML itself. The Directory Service Markup Language provides a foundation for this: a standard for communicating with the directory services that will be responsible for providing and authenticating user credentials. The uniform presentation of both protection objects and access control policies can be applied to policies and credentials themselves. For instance, some credential properties (such as the user name) may be accessible to everyone, whereas other properties may be visible only to a restricted class of users. Additionally, the use of an XML-based language for specifying credentials and access control policies facilitates secure credential submission and export of access control policies.

4. See Thuraisingham et al. (2001).

23.4 Introduction to Statistical Database Security

Statistical databases are used mainly to produce statistics on various populations. The database may contain confidential data on individuals, which should be protected from user access. However, users are permitted to retrieve statistical information on the populations, such as averages, sums, counts, maximums, minimums, and standard deviations. The techniques that have been developed to protect the privacy of individual information are outside the scope of this book. We will only illustrate the problem with a very simple example, which refers to the relation shown in Figure 23.3. This is a PERSON relation with the attributes NAME, SSN, INCOME, ADDRESS, CITY, STATE, ZIP, SEX, and LAST_DEGREE.

A **population** is a set of tuples of a relation (table) that satisfy some selection condition. Hence each selection condition on the PERSON relation will specify a particular population of PERSON tuples. For example, the condition SEX = 'M' specifies the male population; the condition ((SEX = 'F') AND (LAST_DEGREE = 'M.S.' OR LAST_DEGREE = 'PH.D.')) specifies the female population that has an M.S. or PH.D. degree as their highest degree; and the condition CITY = 'Houston' specifies the population that lives in Houston.

Statistical queries involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. **Statistical database security** techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called **statistical queries.**

It is the responsibility of a database management system to ensure the confidentiality of information about individuals, while still providing useful statistical summaries of data about those individuals to users. Provision of **privacy protection** of users in a statistical database is paramount; its violation is illustrated in the following example.

In some cases it is possible to **infer** the values of individual tuples from a sequence of statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples. As an illustration, consider the following two statistical queries:

```
Q1: SELECT COUNT (*) FROM PERSON
    WHERE <CONDITION>;
Q2: SELECT AVG (INCOME) FROM PERSON
    WHERE <CONDITION>;
```

PERSON

NAME	SSN	INCOME	ADDRESS	CITY	STATE	ZIP	SEX	LAST_DEGREE

FIGURE 23.3 The PERSON relation schema for illustrating statistical database security.

Now suppose that we are interested in finding the SALARY of 'Jane Smith', and we know that she has a PH.D. degree and that she lives in the city of Bellaire, Texas. We issue the statistical query Q1 with the following condition:

```
(LAST_DEGREE='PH.D.' AND SEX='F' AND CITY='Bellaire' AND
STATE='Texas')
```

If we get a result of 1 for this query, we can issue Q2 with the same condition and find the income of Jane Smith. Even if the result of Q1 on the preceding condition is not 1 but is a small number—say, 2 or 3—we can issue statistical queries using the functions MAX, MIN, and AVERAGE to identify the possible range of values for the INCOME of Jane Smith.

The possibility of inferring individual information from statistical queries is reduced if no statistical queries are permitted whenever the number of tuples in the population specified by the selection condition falls below some threshold. Another technique for prohibiting retrieval of individual information is to prohibit sequences of queries that refer repeatedly to the same population of tuples. It is also possible to introduce slight inaccuracies or "noise" into the results of statistical queries deliberately, to make it difficult to deduce individual information from the results. Another technique is partitioning of the database. Partitioning implies that records are stored in groups of some minimum size; queries can refer to any complete group or set of groups, but never to subsets of records within a group. The interested reader is referred to the bibliography for a discussion of these techniques.

23.5 Introduction to Flow Control

Flow control regulates the distribution or flow of information among accessible objects. A flow between object X and object Y occurs when a program reads values from X and writes values into Y. **Flow controls** check that information contained in some objects does not flow explicitly or implicitly into less protected objects. Thus, a user cannot get indirectly in Y what he or she cannot get directly from X. Active flow control began in the early 1970s. Most flow controls employ some concept of security class; the transfer of information from a sender to a receiver is allowed only if the receiver's security class is at least as privileged as the sender's. Examples of a flow control include preventing a service program from leaking a customer's confidential data, and blocking the transmission of secret military data to an unknown classified user.

A **flow policy** specifies the channels along which information is allowed to move. The simplest flow policy specifies just two classes of information: confidential (C) and nonconfidential (N), and allows all flows except those from class C to class N. This policy can solve the confinement problem that arises when a service program handles data such as customer information, some of which may be confidential. For example, an income-tax computing service might be allowed to retain the customer's address and the bill for services rendered, but not the customer's income or deductions.

Access control mechanisms are responsible for checking users' authorizations for resource access: Only granted operations are executed. Flow controls can be enforced by

an extended access control mechanism, which involves assigning a security class (usually called the *clearance*) to each running program. The program is allowed to read a particular memory segment only if its security class is as high as that of the segment. It is allowed to write in a segment only if its class is as low as that of the segment. This automatically ensures that no information transmitted by the person can move from a higher to a lower class. For example, a military program with a secret clearance can read only from objects that are unclassified and confidential and it can only write into objects that are secret or top secret.

Two types of flow can be distinguished: *explicit flows*, occurring as a consequence of assignment instructions, such as $Y := f(X_1, X_n)$; and *implicit flows* generated by conditional instructions, such as if $f(X_{m+1}, \ldots, X_n)$ then $y := f(X_1, X_m)$.

Flow control mechanisms must verify that only authorized flows, both explicit and implicit, are executed. A set of rules must be satisfied to ensure secure information flows. Rules can be expressed using flow relations among classes and assigned to information, stating the authorized flows within a system. (An information flow from A to B occurs when information associated with A affects the value of information associated with B. The flow results from operations that cause information transfer from one object to another.) These relations can define, for a class, the set of classes where information (classified in that class) can flow, or can state the specific relations to be verified between two classes to allow information flow from one to the other. In general, flow control mechanisms implement the controls by assigning a label to each object and by specifying the security class of the object. Labels are then used to verify the flow relations defined in the model.

23.5.1 Covert Channels

A covert channel allows a transfer of information that violates the security or the policy. Specifically, a **covert channel** allows information to pass from a higher classification level to a lower classification level through improper means. Covert channels can be classified into two broad categories: storage and timing channels. The distinguishing feature between the two is that in a **timing channel** the information is conveyed by the timing of events or processes, whereas **storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.

In a simple example of a covert channel, consider a distributed database system in which two nodes have user security levels of secret (S) and unclassified (U). In order for a transaction to commit, both nodes must agree to commit. They mutually can only do operations that are consistent with the *-property, which states that in any transaction, the S site cannot write or pass information to the U site. However, if these two sites collude to set up a covert channel between them, a transaction involving secret data may be committed unconditionally by the U site, but the S site may do so in some predefined agreed-upon way so that certain information may be passed on from the S site to the U site, violating the *-property. This may be achieved where the transaction runs repeatedly, but the actions taken by the S site implicitly convey information to the U site. Measures such as

locking that we discussed in Chapters 17 and 18 prevent concurrent writing of the information by users with different security levels into the same objects, preventing the storage-type covert channels. Operating systems and distributed databases provide control over the multiprogramming of operations that allow a sharing of resources without the possibility of encroachment of one program or process into another's memory or other resources in the system, thus preventing timing-oriented covert channels. In general, covert channels are not a major problem in well-implemented robust database implementations. However, certain schemes may be contrived by clever users that implicitly transfer information.

Some security experts believe that one way to avoid covert channels is for programmers to not actually gain access to sensitive data that a program is supposed to process after the program has been put into operation. For example, a programmer for a bank has no need to access the names or balances in depositors' accounts. Programmers for brokerage firms do not need to know what buy and sell orders exist for clients. During program testing, access to a form of real data or some sample test data may be justifiable, but not after the program has been accepted for regular use.

23.6 Encryption and Public Key Infrastructures

The previous methods of access and flow control, despite being strong countermeasures, may not be able to protect databases from some threats. Suppose we communicate data, but our data falls into the hands of some nonlegitimate user. In this situation, by using encryption we can disguise the message so that even if the transmission is diverted, the message will not be revealed. Encryption is a means of maintaining secure data in an insecure environment. Encryption consists of applying an **encryption algorithm** to data using some prespecified **encryption key**. The resulting data has to be **decrypted** using a **decryption key** to recover the original data.

23.6.1 The Data and Advanced Encryption Standards

The Data Encryption Standard (DES) is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad. DES can provide end-to-end encryption on the channel between the sender A and receiver B. The DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition). The algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles. Plaintext (the original form of the message) is encrypted as blocks of 64 bits. Although the key is 64 bits long, in effect the key can be any 56-bit number. After questioning the adequacy of DES, the National Institute of Standards (NIST) introduced the Advanced Encryption Standards (AES). This algorithm has a block size of 128 bits, compared with DES's 56-block size, and can use keys of

128, 192, or 256 bits, compared with DES's 56-bit key. AES introduces more possible keys, compared with DES, and thus takes a much longer time to crack.

23.6.2 Public Key Encryption

In 1976 Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption.** Public key algorithms are based on mathematical functions rather than operations on bit patterns. They also involve the use of two separate keys, in contrast to conventional encryption, which uses only one key. The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication. The two keys used for public key encryption are referred to as the **public key** and the **private key.** Invariably, the private key is kept secret, but it is referred to as a *private key* rather than a *secret key* (the key used in conventional encryption) to avoid confusion with conventional encryption.

A public key encryption scheme, or infrastructure, has six ingredients:

1. *Plaintext:* This is the data or readable message that is fed into the algorithm as input.
2. *Encryption algorithm:* The encryption algorithm performs various transformations on the plaintext.

3 and 4. *Public and private keys:* These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input.

5. *Ciphertext:* This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
6. *Decryption algorithm:* This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

As the name suggests, the public key of the pair is made public for others to use, whereas the private key is known only to its owner. A general-purpose public key cryptographic algorithm relies on one key for encryption and a different but related one for decryption. The essential steps are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private.
3. If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.
4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.

The RSA Public Key Encryption Algorithm. One of the first public key schemes was introduced in 1978 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and is named after them as the RSA scheme. The RSA scheme has since then reigned supreme as the most widely accepted and implemented approach to public key encryption. The RSA encryption algorithm incorporates results from number theory, combined with the difficulty of determining the prime factors of a target. The RSA algorithm also operates with modular arithmetic—mod n.

Two keys, d and e, are used for decryption and encryption. An important property is that they can be interchanged. n is chosen as a large integer that is a product of two large distinct prime numbers, a and b. The encryption key e is a randomly chosen number between 1 and n that is relatively prime to $(a - 1) \times (b - 1)$. The plaintext block P is encrypted as P^e mod n. Because the exponentiation is performed mod n, factoring P^e to uncover the encrypted plaintext is difficult. However, the decrypting key d is carefully chosen so that $(P^e)^d$ mod $n = P$. The decryption key d can be computed from the condition that $d \times e = 1 \bmod ((a - 1) \times (b - 1))$. Thus, the legitimate receiver who knows d simply computes $(P^e)^d$ mod $n = P$ and recovers P without having to factor P^e.

23.6.3 Digital Signatures

A digital signature is an example of using encryption techniques to provide authentication services in electronic commerce applications. Like a handwritten signature, a **digital signature** is a means of associating a mark unique to an individual with a body of text. The mark should be unforgettable, meaning that others should be able to check that the signature does come from the originator.

A digital signature consists of a string of symbols. If a person's digital signature were always the same for each message, then one could easily counterfeit it by simply copying the string of symbols. Thus, signatures must be different for each use. This can be achieved by making each digital signature a function of the message that it is signing, together with a time stamp. To be unique to each signer and counterfeitproof, each digital signature must also depend on some secret number that is unique to the signer. Thus, in general, a counterfeitproof digital signature must depend on the message and a unique secret number of the signer. The verifier of the signature, however, should not need to know any secret number. Public key techniques are the best means of creating digital signatures with these properties.

23.7 SUMMARY

This chapter discussed several techniques for enforcing security in database systems. It presented the different threats to databases in terms of loss of integrity, availability, and confidentiality. The four types of countermeasures to deal with these problems are access control, inference control, flow control, and encryption. We discussed all of these measures in this chapter.

Security enforcement deals with controlling access to the database system as a whole and controlling authorization to access specific portions of a database. The former is usually done by assigning accounts with passwords to users. The latter can be accomplished by using a system of granting and revoking privileges to individual accounts for accessing specific parts of the database. This approach is generally referred to as discretionary access control. We presented some SQL commands for granting and revoking privileges, and we illustrated their use with examples. Then we gave an overview of mandatory access control mechanisms that enforce multilevel security. These require the classifications of users and data values into security classes and enforce the rules that prohibit flow of information from higher to lower security levels. Some of the key concepts underlying the multilevel relational model, including filtering and polyinstantiation, were presented. Role-based access control was introduced, which assigns privileges based on roles that users play. We briefly discussed the problem of controlling access to statistical databases to protect the privacy of individual information while concurrently providing statistical access to populations of records. The issues related to flow control and the problems associated with covert channels were discussed next. Finally, we covered the area of encryption of data, including the public key infrastructure and digital signatures.

Review Questions

23.1. Discuss what is meant by each of the following terms: *database authorization, access control, data encryption, privileged (system) account, database audit, audit trail.*
   a. Discuss the types of privileges at the account level and those at the relation level.

23.2. Which account is designated as the owner of a relation? What privileges does the owner of a relation have?

23.3. How is the view mechanism used as an authorization mechanism?

23.4. What is meant by granting a privilege?

23.5. What is meant by revoking a privilege?

23.6. Discuss the system of propagation of privileges and the restraints imposed by horizontal and vertical propagation limits.

23.7. List the types of privileges available in SQL.

23.8. What is the difference between *discretionary* and *mandatory* access control?

23.9. What are the typical security classifications? Discuss the simple security property and the *-property, and explain the justification behind these rules for enforcing multilevel security.

23.10. Describe the multilevel relational data model. Define the following terms: *apparent key, polyinstantiation, filtering.*

23.11. What are the relative merits of using DAC or MAC?

23.12. What is role-based access control? In what ways is it superior to DAC and MAC?

23.13. What is a statistical database? Discuss the problem of statistical database security.

23.14. How is privacy related to statistical database security? What meaures can be taken to ensure some degree of privacy in statistical databases?

23.15. What is flow control as a security measure? What types of flow control exist?

23.16. What are covert channels? Give an example of a covert channel.
23.17. What is the goal of encryption? What process is involved in encrypting data and then recovering it at the other end?
23.18. Give an example of an encryption algorithm and explain how it works.
23.19. Repeat the previous question for the popular RSA algorithm.
23.20. What is the public key infrastructure scheme? How does it provide security?
23.21. What are digital signatures? How do they work?

Exercises

23.22. Consider the relational database schema of Figure 5.5. Suppose that all the relations were created by (and hence are owned by) user X, who wants to grant the following privileges to user accounts A, B, C, D, and E:
   a. Account A can retrieve or modify any relation except `dependent` and can grant any of these privileges to other users.
   b. Account B can retrieve all the attributes of `employee` and `department` except for `salary`, `mgrssn`, and `mgrstartdate`.
   c. Account C can retrieve or modify WORKS_ON but can only retrieve the FNAME, MINIT, LNAME, and SSN attributes of EMPLOYEE and the PNAME and PNUMBER attributes of PROJECT.
   d. Account D can retrieve any attribute of EMPLOYEE or `dependent` and can modify DEPENDENT.
   e. Account E can retrieve any attribute of EMPLOYEE but only for EMPLOYEE tuples that have DNO = 3.
   f. Write SQL statements to grant these privileges. Use views where appropriate.

23.23. Suppose that privilege (a) of Exercise 23.1 is to be given with GRANT OPTION but only so that account A can grant it to at most five accounts, and each of these accounts can propagate the privilege to other accounts but *without* the GRANT OPTION privilege. What would the horizontal and vertical propagation limits be in this case?

23.24. Consider the relation shown in Figure 23.2d. How would it appear to a user with classification U? Suppose a classification U user tries to update the salary of '`Smith`' to $50,000; what would be the result of this action?

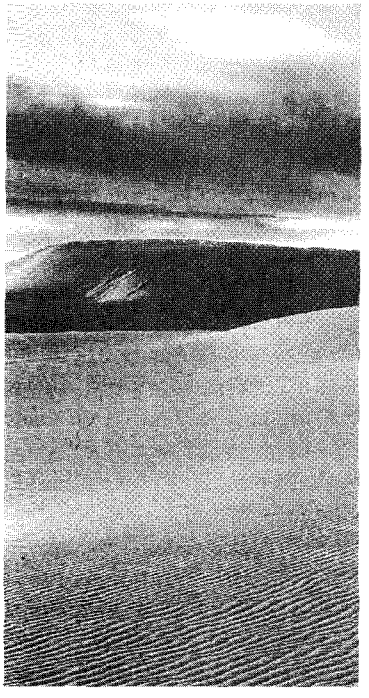
Selected Bibliography

Authorization based on granting and revoking privileges was proposed for the SYSTEM R experimental DBMS and is presented in Griffiths and Wade (1976). Several books discuss security in databases and computer systems in general, including the books by Leiss (1982a) and Fernandez et al. (1981). Denning and Denning (1979) is a tutorial paper on data security.

Many papers discuss different techniques for the design and protection of statistical databases. These include McLeish (1989), Chin and Ozsoyoglu (1981), Leiss (1982), Wong (1984), and Denning (1980). Ghosh (1984) discusses the use of statistical databases for

quality control. There are also many papers discussing cryptography and data encryption, including Diffie and Hellman (1979), Rivest et al. (1978), Akl (1983), Pfleeger (1997), Omura et al. (1990), and Stalling (2000).

Multilevel security is discussed in Jajodia and Sandhu (1991), Denning et al. (1987), Smith and Winslett (1992), Stachour and Thuraisingham (1990), Lunt et al. (1990), and Bertino et al. (2001). Overviews of research issues in database security are given by Lunt and Fernandez (1990), Jajodia and Sandhu (1991), Bertino et al. (1998), Castano et al. (1995), and Thuraisingham et al. (2001). The effects of multilevel security on concurrency control are discussed in Atluri et al. (1997). Security in next-generation, semantic, and object-oriented databases is discussed in Rabbiti et al. (1991), Jajodia and Kogan (1990), and Smith (1990). Oh (1999) presents a model for both discretionary and mandatory security. Security models for Web-based applications and role-based access control are discussed in Joshi et al. (2001). Security issues for managers in the context of e-commerce applications and the need for risk assessment models for selection of appropriate security countermeasures are discussed in Farahmand et al. (2002).



24

Enhanced Data Models for Advanced Applications

As the use of database systems has grown, users have demanded additional functionality from these software packages, with the purpose of making it easier to implement more advanced and complex user applications. Object-oriented databases and object-relational systems do provide features that allow users to extend their systems by specifying additional abstract data types for each application. However, it is quite useful to identify certain common features for some of these advanced applications and to create models that can represent these common features. In addition, specialized storage structures and indexing methods can be implemented to improve the performance of these common features. These features can then be implemented as abstract data type or class libraries and separately purchased with the basic DBMS software package. The term **datablade** has been used in Informix and **cartridge** in Oracle (see Chapter 22) to refer to such optional submodules that can be included in a DBMS package. Users can utilize these features directly if they are suitable for their applications, without having to reinvent, reimplement, and reprogram such common features.

This chapter introduces database concepts for some of the common features that are needed by advanced applications and that are starting to have widespread use. The features we will cover are *active rules* that are used in active database applications, *temporal concepts* that are used in temporal database applications, and briefly some of the issues involving *multimedia databases*. We will also discuss *deductive databases*. It is important to note that each of these topics is very broad, and we can give only a brief introduction to each area. In fact, each of these areas can serve as the sole topic for a complete book.

In Section 24.1, we will introduce the topic of active databases, which provide additional functionality for specifying **active rules.** These rules can be automatically triggered by events that occur, such as a database update or a certain time being reached, and can initiate certain actions that have been specified in the rule declaration if certain conditions are met. Many commercial packages already have some of the functionality provided by active databases in the form of **triggers.** Triggers are now part of the SQL-99 standard.

In Section 24.2, we will introduce the concepts of **temporal databases,** which permit the database system to store a history of changes, and allow users to query both current and past states of the database. Some temporal database models also allow users to store future expected information, such as planned schedules. It is important to note that many database applications are already temporal, but are often implemented without having much temporal support from the DBMS package—that is, the temporal concepts were implemented in the application programs that access the database.

Section 24.3 will give a brief overview of spatial and multimedia databases. **Spatial databases** provide concepts for databases that keep track of objects in a multidimensional space. For example, cartographic databases that store maps include two-dimensional spatial positions of their objects, which include countries, states, rivers, cities, roads, seas, and so on. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points. **Multimedia databases** provide features that allow users to store and query different types of multimedia information, which includes **images** (such as pictures or drawings), **video clips** (such as movies, news reels, or home videos), **audio clips** (such as songs, phone messages, or speeches), and **documents** (such as books or articles).

In Section 24.4, we discuss deductive databases,[1] an area that is at the intersection of databases, logic, and artificial intelligence or knowledge bases. A **deductive database system** is a database system that includes capabilities to define **(deductive) rules,** which can deduce or infer additional information from the facts that are stored in a database. Because part of the theoretical foundation for some deductive database systems is mathematical logic, such rules are often referred to as **logic databases.** Other types of systems, referred to as **expert database systems** or **knowledge-based systems,** also incorporate reasoning and inferencing capabilities; such systems use techniques that were developed in the field of artificial intelligence, including semantic networks, frames, production systems, or rules for capturing domain-specific knowledge.

Readers may choose to peruse the particular topics they are interested in, as the sections in this chapter are practically independent of one another.

1. Section 24.4 is a summary of Chapter 25 from the third edition. The full chapter will be available on the book Web site.

24.1 Active Database Concepts and Triggers

Rules that specify actions that are automatically triggered by certain events have been considered as important enhancements to a database system for quite some time. In fact, the concept of **triggers**—a technique for specifying certain types of active rules—has existed in early versions of the SQL specification for relational databases and triggers are now part of the SQL-99 standard. Commercial relational DBMSs—such as Oracle, DB2, and SYBASE—have had various versions of triggers available. However, much research into what a general model for active databases should look like has been done since the early models of triggers were proposed. In Section 24.1.1, we will present the general concepts that have been proposed for specifying rules for active databases. We will use the syntax of the Oracle commercial relational DBMS to illustrate these concepts with specific examples, since Oracle triggers are close to the way rules are specified in the SQL standard. Section 24.1.2 will discuss some general design and implementation issues for active databases. We then give examples of how active databases are implemented in the STARBURST experimental DBMS in Section 24.1.3, since STARBURST provides for many of the concepts of generalized active databases within its framework. Section 24.1.4 discusses possible applications of active databases. Finally, Section 24.1.5 describes how triggers are declared in the SQL-99 standard.

24.1.1 Generalized Model for Active Databases and Oracle Triggers

The model that has been used for specifying active database rules is referred to as the **Event-Condition-Action,** or **ECA** model. A rule in the ECA model has three components:

1. The **event** (or events) that triggers the rule: These events are usually database update operations that are explicitly applied to the database. However, in the general model, they could also be temporal events[2] or other kinds of external events.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Let us consider some examples to illustrate these concepts. The examples are based on a much simplified variation of the COMPANY database application from Figure 5.7, which

2. An example would be a temporal event specified as a periodic time, such as: Trigger this rule every day at 5:30 A.M.

is shown in Figure 24.1, with each employee having a name (NAME), social security number (SSN), salary (SALARY), department to which they are currently assigned (DNO, a foreign key to DEPARTMENT), and a direct supervisor (SUPERVISOR_SSN, a (recursive) foreign key to EMPLOYEE). For this example, we assume that null is allowed for DNO, indicating that an employee may be temporarily unassigned to any department. Each department has a name (DNAME), number (DNO), the total salary of all employees assigned to the department (TOTAL_SAL), and a manager (MANAGER_SSN, a foreign key to EMPLOYEE).

Notice that the TOTAL_SAL attribute is really a derived attribute, whose value should be the sum of the salaries of all employees who are assigned to the particular department. Maintaining the correct value of such a derived attribute can be done via an active rule. We first have to determine the **events** that *may cause* a change in the value of TOTAL_SAL, which are as follows:

1. Inserting (one or more) new employee tuples.
2. Changing the salary of (one or more) existing employees.
3. Changing the assignment of existing employees from one department to another.
4. Deleting (one or more) employee tuples.

In the case of event 1, we only need to recompute TOTAL_SAL if the new employee is immediately assigned to a department—that is, if the value of the DNO attribute for the new employee tuple is not null (assuming null is allowed for DNO). Hence, this would be the **condition** to be checked. A similar condition could be checked for event 2 (and 4) to determine whether the employee whose salary is changed (or who is being deleted) is currently assigned to a department. For event 3, we will always execute an action to maintain the value of TOTAL_SAL correctly, so no condition is needed (the action is always executed).

The **action** for events 1, 2, and 4 is to automatically update the value of TOTAL_SAL for the employee's department to reflect the newly inserted, updated, or deleted employee's salary. In the case of event 3, a twofold action is needed; one to update the TOTAL_SAL of the employee's old department and the other to update the TOTAL_SAL of the employee's new department.

The four active rules (or triggers) R1, R2, R3, and R4—corresponding to the above situation—can be specified in the notation of the Oracle DBMS as shown in Figure 24.2a. Let us consider rule R1 to illustrate the syntax of creating triggers in Oracle. The CREATE

EMPLOYEE

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN

DEPARTMENT

DNAME	DNO	TOTAL_SAL	MANAGER_SSN

FIGURE 24.1 A simplified COMPANY database used for active rule examples.

(a)

```
R1:  CREATE TRIGGER TOTALSAL1
     AFTER INSERT ON EMPLOYEE
     FOR EACH ROW
     WHEN (NEW.DNO IS NOT NULL)
         UPDATE DEPARTMENT
         SET TOTAL_SAL=TOTAL_SAL + NEW.SALARY
         WHERE DNO=NEW.DNO;

R2:  CREATE TRIGGER TOTALSAL2
     AFTER UPDATE OF SALARY ON EMPLOYEE
     FOR EACH ROW
     WHEN (NEW.DNO IS NOT NULL)
         UPDATE DEPARTMENT
         SET TOTAL_SAL=TOTAL_SAL + NEW.SALARY – OLD.SALARY
         WHERE DNO=NEW.DNO;

R3:  CREATE TRIGGER TOTALSAL3
     AFTER UPDATE OF DNO ON EMPLOYEE
     FOR EACH ROW
         BEGIN
         UPDATE DEPARTMENT
         SET TOTAL_SAL=TOTAL_SAL + NEW.SALARY
         WHERE DNO=NEW.DNO;
         UPDATE DEPARTMENT
         SET TOTAL_SAL=TOTAL_SAL– OLD.SALARY
         WHERE DNO=OLD.DNO;
         END;

R4:  CREATE TRIGGER TOTALSAL4
     AFTER DELETE ON EMPLOYEE
     FOR EACH ROW
     WHEN (OLD.DNO IS NOT NULL)
         UPDATE DEPARTMENT
         SET TOTAL_SAL=TOTAL_SAL – OLD.SALARY
         WHERE DNO=OLD.DNO;
```

(b)

```
R5:  CREATE TRIGGER INFORM_SUPERVISOR1
     BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN ON EMPLOYEE
     FOR EACH ROW
     WHEN
     (NEW.SALARY > (SELECT SALARY FROM EMPLOYEE
                    WHERE SSN=NEW.SUPERVISOR_SSN))
         INFORM_SUPERVISOR(NEW. SUPERVISOR_SSN, NEW.SSN);
```

FIGURE 24.2 Specifying active rules as triggers in Oracle notation. (a) Triggers for automatically maintaining the consistency of TOTAL_SAL of DEPARTMENT. (b) Trigger for comparing an employee's salary with that of his or her supervisor.

TRIGGER statement specifies a trigger (or active rule) name—TOTALSAL1 for R1. The AFTER-clause specifies that the rule will be triggered *after* the events that trigger the rule occur. The triggering events—an insert of a new employee in this example—are specified following the AFTER keyword.[3] The ON-clause specifies the relation on which the rule is specified—EMPLOYEE for R1. The *optional* keywords FOR EACH ROW specify that the rule will be triggered *once for each row* that is affected by the triggering event.[4] The *optional* WHEN-clause is used to specify any conditions that need to be checked after the rule is triggered but before the action is executed. Finally, the action(s) to be taken are specified as a PL/SQL block, which typically contains one or more SQL statements or calls to execute external procedures.

The four triggers (active rules) R1, R2, R3, and R4 illustrate a number of features of active rules. First, the basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. These are specified by the keywords **INSERT, DELETE,** and **UPDATE** in Oracle notation. In the case of **UPDATE** one may specify the attributes to be updated—for example, by writing **UPDATE OF** SALARY, DNO. Second, the rule designer needs to have a way to refer to the tuples that have been inserted, deleted, or modified by the triggering event. The keywords **NEW** and **OLD** are used in Oracle notation; NEW is used to refer to a newly inserted or newly updated tuple, whereas OLD is used to refer to a deleted tuple or to a tuple before it was updated.

Thus rule R1 is triggered after an INSERT operation is applied to the EMPLOYEE relation. In R1, the condition (**NEW**.DNO **IS NOT NULL**) is checked, and if it evaluates to true, meaning that the newly inserted employee tuple is related to a department, then the action is executed. The action updates the DEPARTMENT tuple(s) related to the newly inserted employee by adding their salary (**NEW**.SALARY) to the TOTAL_SAL attribute of their related department.

Rule R2 is similar to R1, but it is triggered by an UPDATE operation that updates the SALARY of an employee rather than by an INSERT. Rule R3 is triggered by an update to the DNO attribute of EMPLOYEE, which signifies changing an employee's assignment from one department to another. There is no condition to check in R3, so the action is executed whenever the triggering event occurs. The action updates both the old department and new department of the reassigned employees by adding their salary to TOTAL_SAL of their *new* department and subtracting their salary from TOTAL_SAL of their *old* department. Note that this should work even if the value of DNO was null, because in this case no department will be selected for the rule action.[5]

It is important to note the effect of the optional FOR EACH ROW clause, which signifies that the rule is triggered separately *for each tuple*. This is known as a **row-level trigger.** If this clause was left out, the trigger would be known as a **statement-level trigger**

3. As we shall see later, it is also possible to specify BEFORE instead of AFTER, which indicates that the rule is triggered *before the triggering event is executed.*

4. Again, we shall see later that an alternative is to trigger the rule *only once* even if multiple rows (tuples) are affected by the triggering event.

5. R1, R2, and R4 can also be written without a condition. However, they may be more efficient to execute with the condition since the action is not invoked unless it is required.

and would be triggered once for each triggering statement. To see the difference, consider the following update operation, which gives a 10 percent raise to all employees assigned to department 5. This operation would be an event that triggers rule R2:

```
UPDATE      EMPLOYEE
SET         SALARY = 1.1 * SALARY
WHERE       DNO = 5;
```

Because the above statement could update multiple records, a rule using row-level semantics, such as R2 in Figure 24.2, would be triggered *once for each row*, whereas a rule using statement-level semantics is triggered *only once*. The Oracle system allows the user to choose which of the above two options is to be used for each rule. Including the optional FOR EACH ROW clause creates a row-level trigger, and leaving it out creates a statement-level trigger. Note that the keywords NEW and OLD can only be used with row-level triggers.

As a second example, suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor. Several events can trigger this rule: inserting a new employee, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external procedure INFORM_SUPERVISOR,[6] which will notify the supervisor. The rule could then be written as in R5 (see Figure 24.2b).

Figure 24.3 shows the syntax for specifying some of the main options available in Oracle triggers. We will describe the syntax for triggers in the SQL-99 standard in Section 24.1.5.

24.1.2 Design and Implementation Issues for Active Databases

The previous section gave an overview of some of the main concepts for specifying active rules. In this section, we discuss some additional issues concerning how rules are designed and implemented. The first issue concerns activation, deactivation, and grouping of rules.

```
<trigger>  ::= CREATE TRIGGER <trigger name>
               (AFTER | BEFORE ) <triggering events> ON <table name>
               [ FOR EACH ROW ]
               [ WHEN <condition> ]
               <trigger actions> ;
<triggering events> ::=<trigger event> {OR <trigger event> }
<trigger event>::=INSERT | DELETE | UPDATE [ OF <column name> {, <column name>} ]
<trigger action> ::=<PL/SQL block>
```

FIGURE 24.3 A syntax summary for specifying triggers in the Oracle system (main options only).

6. Assuming that an appropriate external procedure has been declared. This is a feature that is now available in SQL.

In addition to creating rules, an active database system should allow users to *activate*, *deactivate*, and *drop* rules by referring to their rule names. A **deactivated rule** will not be triggered by the triggering event. This feature allows users to selectively deactivate rules for certain periods of time when they are not needed. The **activate command** will make the rule active again. The **drop command** deletes the rule from the system. Another option is to group rules into named **rule sets,** so the whole set of rules could be activated, deactivated, or dropped. It is also useful to have a command that can trigger a rule or rule set via an explicit PROCESS RULES command issued by the user.

The second issue concerns whether the triggered action should be executed *before*, *after*, or *concurrently with* the triggering event. A related issue is whether the action being executed should be considered as a *separate transaction* or whether it should be part of the same transaction that triggered the rule. We will first try to categorize the various options. It is important to note that not all options may be available for a particular active database system. In fact, most commercial systems are *limited to one or two of the options* that we will now discuss.

Let us assume that the triggering event occurs as part of a transaction execution. We should first consider the various options for how the triggering event is related to the evaluation of the rule's condition. The rule *condition evaluation* is also known as **rule consideration,** since the action is to be executed only after considering whether the condition evaluates to true or false. There are three main possibilities for rule consideration:

1. *Immediate consideration:* The condition is evaluated as part of the same transaction as the triggering event, and is evaluated *immediately*. This case can be further categorized into three options:
   - Evaluate the condition *before* executing the triggering event.
   - Evaluate the condition *after* executing the triggering event.
   - Evaluate the condition *instead of* executing the triggering event.
2. *Deferred consideration:* The condition is evaluated at the end of the transaction that included the triggering event. In this case, there could be many triggered rules waiting to have their conditions evaluated.
3. *Detached consideration:* The condition is evaluated as a separate transaction, spawned from the triggering transaction.

The next set of options concerns the relationship between evaluating the rule condition and *executing* the rule action. Here, again, three options are possible: **immediate, deferred,** and **detached** execution. However, most active systems use the first option. That is, as soon as the condition is evaluated, if it returns true, the action is *immediately* executed.

The Oracle system (see Section 24.1.1) uses the *immediate consideration* model, but it allows the user to specify for each rule whether the *before* or *after* option is to be used with immediate condition evaluation. It also uses the *immediate execution* model. The STARBURST system (see Section 24.1.3) uses the *deferred consideration* option, meaning that all rules triggered by a transaction wait until the triggering transaction reaches its end and issues its COMMIT WORK command before the rule conditions are evaluated.[7]

7. STARBURST also allows the user to explicitly start rule consideration via a PROCESS RULES command.

Another issue concerning active database rules is the distinction between *row-level rules* versus *statement-level rules*. Because SQL update statements (which act as triggering events) can specify a set of tuples, one has to distinguish between whether the rule should be considered once for the *whole statement* or whether it should be considered separately *for each row* (that is, tuple) affected by the statement. The SQL-99 standard (see Section 24.1.5) and the Oracle system (see Section 24.1.1) allow the user to choose which of the above two options is to be used for each rule, whereas STARBURST uses statement-level semantics only. We will give examples of how statement-level triggers can be specified in Section 24.1.3.

One of the difficulties that may have limited the widespread use of active rules, in spite of their potential to simplify database and software development, is that there are no easy-to-use techniques for designing, writing, and verifying rules. For example, it is quite difficult to verify that a set of rules is **consistent,** meaning that two or more rules in the set do not contradict one another. It is also difficult to guarantee **termination** of a set of rules under all circumstances. To briefly illustrate the termination problem, consider the rules in Figure 24.4. Here, rule R1 is triggered by an INSERT event on TABLE1 and its action includes an update event on ATTRIBUTE1 of TABLE2. However, rule R2's triggering event is an UPDATE event on ATTRIBUTE1 of TABLE2, and its action includes an INSERT event on TABLE1. It is easy to see in this example that these two rules can trigger one another indefinitely, leading to nontermination. However, if dozens of rules are written, it is very difficult to determine whether termination is guaranteed or not.

If active rules are to reach their potential, it is necessary to develop tools for the design, debugging, and monitoring of active rules that can help users in designing and debugging their rules.

24.1.3 Examples of Statement-Level Active Rules in STARBURST

We now give some examples to illustrate how rules can be specified in the STARBURST experimental DBMS. This will allow us to demonstrate how statement-level rules can be written, since these are the only types of rules allowed in STARBURST.

```
R1:  CREATE TRIGGER T1
     AFTER INSERT ON TABLE1
     FOR EACH ROW
               UPDATE TABLE2
               SET ATTRIBUTE1=...;

R2:  CREATE TRIGGER T2
     AFTER UPDATE OF ATTRIBUTE1 ON TABLE2
     FOR EACH ROW
               INSERT INTO TABLE1 VALUES (...);
```

FIGURE 24.4 An example to illustrate the termination problem for active rules.

The three active rules R1S, R2S, and R3S in Figure 24.5 correspond to the first three rules in Figure 24.2, but use STARBURST notation and statement-level semantics. We can explain the rule structure using rule R1S. The CREATE RULE statement specifies a rule name—TOTALSAL1 for R1S. The ON-clause specifies the relation on which the rule is specified—EMPLOYEE for R1S. The WHEN-clause is used to specify the **events** that trigger the rule.[8] The *optional* IF-clause is used to specify any **conditions** that need to be checked.

```
R1S:  CREATE RULE TOTALSAL1 ON EMPLOYEE
      WHEN INSERTED
      IF    EXISTS(SELECT * FROM INSERTED WHERE DNO IS NOT NULL)
      THEN  UPDATE  DEPARTMENT AS D
            SET     D.TOTAL_SAL=D.TOTAL_SAL +
                    (SELECT SUM(I.SALARY) FROM INSERTED AS I WHERE D.DNO = I.DNO)
            WHERE   D.DNO IN (SELECT DNO FROM INSERTED);

R2S:  CREATE RULE TOTALSAL2 ON EMPLOYEE
      WHEN  UPDATED (SALARY)
      IF    EXISTS(SELECT * FROM NEW-UPDATED WHERE DNO IS NOT NULL)
            OR EXISTS(SELECT * FROM OLD-UPDATED WHERE DNO IS NOT NULL)
      THEN  UPDATE DEPARTMENT AS D
            SET     D.TOTAL_SAL=D.TOTAL_SAL +
                    (SELECT SUM(N.SALARY) FROM NEW-UPDATED AS N WHERE
                       D.DNO =N.DNO) -
                    (SELECT SUM(O.SALARY) FROM OLD-UPDATED AS O WHERE
                       D.DNO=O.DNO)
            WHERE   D.DNO IN (SELECT DNO FROM NEW-UPDATED) OR
                    D.DNO IN (SELECT DNO FROM OLD-UPDATED);

R3S:  CREATE RULE TOTALSAL3 ON EMPLOYEE
      WHEN  UPDATED(DNO)
      THEN  UPDATE DEPARTMENT AS D
            SET     D.TOTAL_SAL=D.TOTAL_SAL +
                    (SELECT SUM(N.SALARY) FROM NEW-UPDATED AS N WHERE
                       D.DNO=N.DNO)
            WHERE   D.DNO IN (SELECT DNO FROM NEW-UPDATED);

            UPDATE DEPARTMENT AS D
            SET     D.TOTAL_SAL=D.TOTAL_SAL -
                    (SELECT SUM(O.SALARY) FROM OLD-UPDATED AS O WHERE
                       D.DNO=O.DNO)
            WHERE   D.DNO IN (SELECT DNO FROM OLD-UPDATED);
```

FIGURE 24.5 Active rules using statement-level semantics in STARBURST notation.

8. Note that the WHEN keyword specifies *events* in STARBURST but is used to specify the rule *condition* in SQL and Oracle triggers.

Finally, the THEN-clause is used to specify the **action** (or actions) to be taken, which are typically one or more SQL statements.

In STARBURST, the basic events that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. These are specified by the keywords **INSERTED, DELETED,** and **UPDATED** in STARBURST notation. Second, the rule designer needs to have a way to refer to the tuples that have been modified. The keywords **INSERTED, DELETED, NEW-UPDATED,** and **OLD-UPDATED** are used in STARBURST notation to refer to four **transition tables** (relations) that include the newly inserted tuples, the deleted tuples, the updated tuples *before* they were updated, and the updated tuples *after* they were updated, respectively. Obviously, depending on the triggering events, only some of these transition tables may be available. The rule writer can refer to these tables when writing the condition and action parts of the rule. Transition tables contain tuples of the same type as those in the relation specified in the ON-clause of the rule—for R1S, R2S, and R3S, this is the EMPLOYEE relation.

In statement-level semantics, the rule designer can only refer to the transition tables as a whole and the rule is triggered only once, so the rules must be written differently than for row-level semantics. Because multiple employee tuples may be inserted in a single insert statement, we have to check if *at least one* of the newly inserted employee tuples is related to a department. In R1S, the condition

```
EXISTS(SELECT * FROM INSERTED WHERE DNO IS NOT NULL)
```

is checked, and if it evaluates to true, then the action is executed. The action updates in a single statement the DEPARTMENT tuple(s) related to the newly inserted employee(s) by adding their salaries to the TOTAL_SAL attribute of each related department. Because more than one newly inserted employee may belong to the same department, we use the SUM aggregate function to ensure that all their salaries are added.

Rule R2S is similar to R1S, but is triggered by an UPDATE operation that updates the salary of one or more employees rather than by an INSERT. Rule R3S is triggered by an update to the DNO attribute of EMPLOYEE, which signifies changing one or more employees' assignment from one department to another. There is no condition in R3S, so the action is executed whenever the triggering event occurs.[9] The action updates both the old department(s) and new department(s) of the reassigned employees by adding their salary to TOTAL_SAL of each *new* department and subtracting their salary from TOTAL_SAL of each *old* department.

In our example, it is more complex to write the statement-level rules than the row-level rules, as can be illustrated by comparing Figures 24.2 and 24.5. However, this is not a general rule, and other types of active rules may be easier to specify using statement-level notation than when using row-level notation.

The execution model for active rules in STARBURST uses **deferred consideration.** That is, all the rules that are triggered within a transaction are placed in a set—called the **conflict**

9. As in the Oracle examples, rules R1S and R2S can be written without a condition. However, they may be more efficient to execute with the condition since the action is not invoked unless it is required.

set—which is not considered for evaluation of conditions and execution until the transaction ends (by issuing its COMMIT WORK command). STARBURST also allows the user to explicitly start rule consideration in the middle of a transaction via an explicit PROCESS RULES command. Because multiple rules must be evaluated, it is necessary to specify an order among the rules. The syntax for rule declaration in STARBURST allows the specification of *ordering* among the rules to instruct the system about the order in which a set of rules should be considered.[10] In addition, the transition tables—INSERTED, DELETED, NEW-UPDATED, and OLD-UPDATED—contain the *net effect* of all the operations within the transaction that affected each table, since multiple operations may have been applied to each table during the transaction.

24.1.4 Potential Applications for Active Databases

We now briefly discuss some of the potential applications of active rules. Obviously, one important application is to allow **notification** of certain conditions that occur. For example, an active database may be used to monitor, say, the temperature of an industrial furnace. The application can periodically insert in the database the temperature reading records directly from temperature sensors, and active rules can be written that are triggered whenever a temperature record is inserted, with a condition that checks if the temperature exceeds the danger level, and the action to raise an alarm.

Active rules can also be used to **enforce integrity constraints** by specifying the types of events that may cause the constraints to be violated and then evaluating appropriate conditions that check whether the constraints are actually violated by the event or not. Hence, complex application constraints, often known as **business rules** may be enforced that way. For example, in the UNIVERSITY database application, one rule may monitor the grade point average of students whenever a new grade is entered, and it may alert the advisor if the GPA of a student falls below a certain threshold; another rule may check that course prerequisites are satisfied before allowing a student to enroll in a course; and so on.

Other applications include the automatic **maintenance of derived data,** such as the examples of rules R1 through R4 that maintain the derived attribute TOTAL_SAL whenever individual employee tuples are changed. A similar application is to use active rules to maintain the consistency of **materialized views** (see Chapter 9) whenever the base relations are modified. This application is also relevant to the new data warehousing technologies (see Chapter 28). A related application is to maintain **replicated tables** consistent by specifying rules that modify the replicas whenever the master table is modified.

24.1.5 Triggers in SQL-99

Triggers in the SQL-99 standard are quite similar to the examples we discussed in Section 24.1.1, with some minor syntactic differences. The basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE.

10. If no order is specified between a pair of rules, the system default order is based on placing the rule declared first ahead of the other rule.

In the case of UPDATE one may specify the attributes to be updated. Both row-level and statement-level triggers are allowed, indicated in the trigger by the clauses FOR EACH ROW and FOR EACH STATEMENT, respectively. One syntactic difference is that the trigger may specify particular tuple variable names for the old and new tuples instead of using the keywords NEW and OLD as in Figure 24.1. Trigger T1 in Figure 24.6 shows how the row-level trigger R2 from Figure 24.1(a) may be specified in SQL-99. Inside the REFERENCING clause, we named tuple variables (aliases) O and N to refer to the OLD tuple (before modification) and NEW tuple (after modification), respectively. Trigger T2 in Figure 24.6 shows how the statement-level trigger R2S from Figure 24.5 may be specified in SQL-99. For a statement-level trigger, the REFERENCING clause is used to refer to the table of all new tuples (newly inserted or newly updated) as N, whereas the table of all old tuples (deleted tuples or tuples before they were updated) is referred to as O.

24.2 Temporal Database Concepts

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been developed since the early days of database usage. However, in creating these applications, it was mainly left to

```
T1:   CREATE TRIGGER TOTALSAL1
      AFTER UPDATE OF SALARY ON EMPLOYEE
      REFERENCING OLD ROW AS O, NEW ROW AS N
      FOR EACH ROW
      WHEN (N.DNO IS NOT NULL)
            UPDATE DEPARTMENT
            SET TOTAL_SAL = TOTAL_SAL + N.SALARY - O.SALARY
            WHERE DNO = N.DNO;
T2:   CREATE TRIGGER TOTALSAL2
      AFTER UPDATE OF SALARY ON EMPLOYEE
      REFERENCING OLD TABLE AS O, NEW TABLE AS N
      FOR EACH STATEMENT
      WHEN EXISTS(SELECT * FROM N WHERE N.DNO IS NOT NULL) OR
           EXISTS(SELECT * FROM O WHERE O.DNO IS NOT NULL)
            UPDATE DEPARTMENT AS D
            SET D.TOTAL_SAL = D.TOTAL_SAL
            + (SELECT SUM(N.SALARY) FROM N WHERE D.DNO=N.DNO)
            - (SELECT SUM(O.SALARY) FROM O WHERE D.DNO=O.DNO)
            WHERE DNO IN ((SELECT DNO FROM N) UNION (SELECT DNO FROM O));
```

FIGURE 24.6 Trigger T1 illustrating the syntax for defining triggers in SQL-99.

the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include *healthcare*, where patient histories need to be maintained; *insurance*, where claims and accident histories are required as well as information on the times when insurance policies are in effect; *reservation systems* in general (hotel, airline, car rental, train, etc.), where information on the dates and times when reservations are in effect are required; *scientific databases*, where data collected from experiments includes the time when each data is measured; an so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee. In the UNIVERSITY database, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE; the grade history of a STUDENT; and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. Users often attempted to simplify or ignore temporal aspects because of the complexity that they add to their applications.

In this section, we will introduce some of the concepts that have been developed to deal with the complexity of temporal database applications. Section 24.2.1 gives an overview of how time is represented in databases, the different types of temporal information, and some of the different dimensions of time that may be needed. Section 24.2.2 discusses how time can be incorporated into relational databases. Section 24.2.3 gives some additional options for representing time that are possible in database models that allow complex-structured objects, such as object databases. Section 24.2.4 introduces operations for querying temporal databases, and gives a brief overview of the TSQL2 language, which extends SQL with temporal concepts. Section 24.2.5 focuses on time series data, which is a type of temporal data that is very important in practice.

24.2.1 Time Representation, Calendars, and Time Dimensions

For temporal databases, time is considered to be an *ordered sequence* of **points** in some **granularity** that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second in time using this granularity. In reality, each second is a (short) *time duration*, not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchers have used the term **chronon** instead of point to describe this minimal granularity for a particular application. The main consequence of choosing a minimum granularity—say, one second—is that events occurring within the same second will be considered to be *simultaneous events*, even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (Western), Chinese, Islamic, Hindu, Jewish, Coptic, etc.) with different reference points. A **calendar** organizes time into different time units for convenience. Most

calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth's rotation around its axis), and 7 days into a week. Further grouping of days into months and months into years either follow solar or lunar natural phenomena, and are generally irregular. In the Gregorian calendar, which is used in most Western countries, days are grouped into months that are either 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.

In SQL2, the temporal data types (see Chapter 8) include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including sub-second divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an *anchored* time duration with a fixed starting point, such as the 10-day period from January 1, 1999, to January 10, 1999, inclusive).[11]

Event Information Versus Duration (or State) Information. A temporal database will store information concerning when certain events occur, or when certain facts are considered to be true. There are several different types of temporal information. **Point events** or **facts** are typically associated in the database with a **single time point** in some granularity. For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 1999). Note that even though such events or facts may have different granularities, each is still associated with a *single time value* in the database. This type of information is often represented as **time series data** as we shall discuss in Section 24.2.5. **Duration events** or **facts,** on the other hand, are associated with a specific **time period** in the database.[12] For example, an employee may have worked in a company from August 15, 1993, till November 20, 1998.

A **time period** is represented by its **start** and **end time points** `[START-TIME, END-TIME]`. For example, the above period is represented as `[1993-08-15, 1998-11-20]`. Such a time period is often interpreted to mean the *set of all time points* from start-time to end-time, inclusive, in the specified granularity. Hence, assuming day granularity, the period `[1993-08-15, 1998-11-20]` represents the set of all days from August 15, 1993, until November 20, 1998, inclusive.[13]

11. Unfortunately, the terminology has not been used consistently. For example, the term *interval* is often used to denote an anchored duration. For consistency, we shall use the SQL terminology.

12. This is the same as an anchored duration. It has also been frequently called a **time interval,** but to avoid confusion we will use **period** to be consistent with SQL terminology.

13. The representation `[1993-08-15, 1998-11-20]` is called a *closed interval* representation. One can also use an *open interval,* denoted `[1993-08-15, 1998-11-21)`, where the set of points *does not include* the end point. Although the latter representation is sometimes more convenient, we shall use closed intervals throughout to avoid confusion.

Valid Time and Transaction Time Dimensions. Given a particular event or fact that is associated with a particular time point or time period in the database, the association may be interpreted to mean different things. The most natural interpretation is that the associated time is the time that the event occurred, or the period during which the fact was considered to be true *in the real world.* If this interpretation is used, the associated time is often referred to as the **valid time.** A temporal database using this interpretation is called a **valid time database.**

However, a different interpretation can be used, where the associated time refers to the time when the information was actually stored in the database; that is, it is the value of the system time clock when the information is valid *in the system.*[14] In this case, the associated time is called the **transaction time.** A temporal database using this interpretation is called a **transaction time database.**

Other interpretations can also be intended, but these two are considered to be the most common ones, and they are referred to as **time dimensions.** In some applications, only one of the dimensions is needed and in other cases both time dimensions are required, in which case the temporal database is called a **bitemporal database.** If other interpretations are intended for time, the user can define the semantics and program the applications appropriately, and it is called a **user-defined time.**

The next section shows with examples how these concepts can be incorporated into relational databases, and Section 24.2.3 shows an approach to incorporate temporal concepts into object databases.

24.2.2 Incorporating Time in Relational Databases Using Tuple Versioning

Valid Time Relations. Let us now see how the different types of temporal databases may be represented in the relational model. First, suppose that we would like to include the history of changes as they occur in the real world. Consider again the database in Figure 24.1, and let us assume that, for this application, the granularity is day. Then, we could convert the two relations EMPLOYEE and DEPARTMENT into **valid time relations** by adding the attributes VST (Valid Start Time) and VET (Valid End Time), whose data type is DATE in order to provide day granularity. This is shown in Figure 24.7a, where the relations have been renamed EMP_VT and DEPT_VT, respectively.

Consider how the EMP_VT relation differs from the nontemporal EMPLOYEE relation (Figure 24.1).[15] In EMP_VT, each tuple V represents a **version** of an employee's information that is valid (in the real world) only during the time period [V.VST, V.VET], whereas in EMPLOYEE each tuple represents only the current state or current version of each employee. In EMP_VT, the **current version** of each employee typically has a special value, *now,* as its

14. The explanation is more involved, as we shall see in Section 24.2.3.

15. A nontemporal relation is also called a **snapshot relation** as it shows only the *current snapshot* or *current state* of the database.

(a) EMP_VT

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN	VST	VET

DEPT_VT

DNAME	DNO	TOTAL_SAL	MANAGER_SSN	VST	VET

(b) EMP_TT

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN	TST	TET

DEPT_TT

DNAME	DNO	TOTAL_SAL	MANAGER_SSN	TST	TET

(c) EMP_BT

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN	VST	VET	TST	TET

DEPT_BT

DNAME	DNO	TOTAL_SAL	MANAGER_SSN	VST	VET	TST	TET

FIGURE 24.7 Different types of temporal relational databases. (a) Valid time database schema. (b) Transaction time database schema. (c) Bitemporal database schema.

valid end time. This special value, ***now,*** is a **temporal variable** that implicitly represents the current time as time progresses. The nontemporal EMPLOYEE relation would only include those tuples from the EMP_VT relation whose VET is *now*.

Figure 24.8 shows a few tuple versions in the valid-time relations EMP_VT and DEPT_VT. There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan. We can now see how a valid time relation should behave when information is changed. Whenever one or more attributes of an employee are **updated,** rather than actually overwriting the old values, as would happen in a nontemporal relation, the system should create a new version and **close** the current version by changing its VET to the end time. Hence, when the user issued the command to update the salary of Smith effective on June 1, 2003, to $30000, the second version of Smith was created (see Figure 24.8). At the time of this update, the first version of Smith was the current version, with *now* as its VET, but after the update *now* was changed to May 31, 2003 (one less than June 1, 2003, in day granularity), to indicate that the version has become a **closed** or **history version** and that the new (second) version of Smith is now the current one.

EMP_VT

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN	VST	VET
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31
Smith	123456789	30000	5	333445555	2003-06-01	now
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31
Wong	333445555	40000	5	888665555	2002-04-01	now
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10
Narayan	666884444	38000	5	333445555	2003-08-01	now

. . .

DEPT_VT

DNAME	DNO	MANAGER_SSN	VST	VET
Research	5	888665555	2001-09-20	2002-03-31
Research	5	333445555	2002-04-01	now

. . .

FIGURE 24.8 Some tuple versions in the valid time relations EMP_VT and DEPT_VT.

It is important to note that in a valid time relation, the user must generally provide the valid time of an update. For example, the salary update of Smith may have been entered in the database on May 15, 2003, at 8:52:12 A.M., say, even though the salary change in the real world is effective on June 1, 2003. This is called a **proactive update,** since it is applied to the database *before* it becomes effective in the real world. If the update was applied to the database *after* it became effective in the real world, it is called a **retroactive update.** An update that is applied at the same time when it becomes effective is called a **simultaneous update.**

The action that corresponds to **deleting** an employee in a nontemporal database would typically be applied to a valid time database by *closing the current version* of the employee being deleted. For example, if Smith leaves the company effective January 19, 2004, then this would be applied by changing VET of the current version of Smith from *now* to 2004-01-19. In Figure 24.8, there is no current version for Brown, because he presumably left the company on 2002-08-10 and was *logically deleted.* However, because the database is temporal, the old information on Brown is still there.

The operation to **insert** a new employee would correspond to *creating the first tuple version* for that employee, and making it the current version, with the VST being the effective (real world) time when the employee starts work. In Figure 24.7, the tuple on Narayan illustrates this, since the first version has not been updated yet.

Notice that in a valid time relation, the *nontemporal key,* such as SSN in EMPLOYEE, is no longer unique in each tuple (version). The new relation key for EMP_VT is a combination of the nontemporal key and the valid start time attribute VST,[16] so we use (SSN, VST) as

16. A combination of the nontemporal key and the valid end time attribute VET could also be used.

primary key. This is because, at any point in time, there should be *at most one valid version* of each entity. Hence, the constraint that any two tuple versions representing the same entity should have *nonintersecting valid time periods* should hold on valid time relations. Notice that if the nontemporal primary key value may change over time, it is important to have a unique **surrogate key attribute,** whose value never changes for each real world entity, in order to relate together all versions of the same real world entity.

Valid time relations basically keep track of the history of changes as they become effective in the *real world*. Hence, if all real-world changes are applied, the database keeps a history of the *real-world states* that are represented. However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual *database state* at any point in time. If the actual database states are more important to an application, then one should use *transaction time relations*.

Transaction Time Relations. In a transaction time database, whenever a change is applied to the database, the actual **timestamp** of the transaction that applied the change (insert, delete, or update) is recorded. Such a database is most useful when changes are applied *simultaneously* in the majority of cases—for example, real-time stock trading or banking transactions. If we convert the nontemporal database of Figure 24.1 into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into **transaction time relations** by adding the attributes TST (Transaction Start Time) and TET (Transaction End Time), whose data type is typically TIMESTAMP. This is shown in Figure 24.7b, where the relations have been renamed EMP_TT and DEPT_TT, respectively.

In EMP_TT, each tuple V represents a *version* of an employee's information that was created at actual time V.TST and was (logically) removed at actual time V.TET (because the information was no longer correct). In EMP_TT, the *current version* of each employee typically has a special value, ***uc* (Until Changed),** as its transaction end time, which indicates that the tuple represents correct information *until it is changed* by some other transaction.[17] A transaction time database has also been called a **rollback database,**[18] because a user can logically roll back to the actual database state at any past point in time T by retrieving all tuple versions V whose transaction time period [V.TST, V.TET] includes time point T.

Bitemporal Relations. Some applications require both valid time and transaction time, leading to **bitemporal relations.** In our example, Figure 24.7c shows how the EMPLOYEE and DEPARTMENT non-temporal relations in Figure 24.1 would appear as bitemporal relations EMP_BT and DEPT_BT, respectively. Figure 24.9 shows a few tuples in these relations. In these tables, tuples whose transaction end time TET is *uc* are the ones representing currently valid information, whereas tuples whose TET is an absolute timestamp are tuples that were valid until (just before) that timestamp. Hence, the tuples with *uc* in Figure 24.9 correspond to the valid time tuples in Figure 24.7. The transaction start time attribute TST in each tuple is the timestamp of the transaction that created that tuple.

17. The *uc* variable in transaction time relations corresponds to the *now* variable in valid time relations. The semantics are slightly different though.

18. The term rollback here does not have the same meaning as *transaction rollback* (see Chapter 19) during recovery, where the transaction updates are *physically undone*. Rather, here the updates can be *logically undone*, allowing the user to examine the database as it appeared at a previous time point.

EMP_BT

NAME	SSN	SALARY	DNO	SUPERVISOR_SSN	VST	VET	TST	TET
Smith	123456789	25000	5	333445555	2002-06-15	now	2002-06-08,13:05:58	2003-06-04,08:56:12
Smith	123456789	25000	5	333445555	2002-06-15	1998-05-31	2003-06-04,08:56:12	uc
Smith	123456789	30000	5	333445555	2003-06-01	now	2003-06-04,08:56:12	uc
Wong	333445555	25000	4	999887777	1999-08-20	now	1999-08-20,11:18:23	2001-01-07,14:33:02
Wong	333445555	25000	4	999887777	1999-08-20	1996-01-31	2001-01-07,14:33:02	uc
Wong	333445555	30000	5	999887777	2001-02-01	now	2001-01-07,14:33:02	2002-03-28,09:23:57
Wong	333445555	30000	5	999887777	2001-02-01	1997-03-31	2002-03-28,09:23:57	uc
Wong	333445555	40000	5	888665555	2002-04-01	now	2002-03-28,09:23:57	uc
Brown	222447777	28000	4	999887777	2001-05-01	now	2001-04-27,16:22:05	2002-08-12,10:11:07
Brown	222447777	28000	4	999887777	2001-05-01	1997-08-10	2002-08-12,10:11:07	uc
Narayan	666884444	38000	5	333445555	2003-08-01	now	2003-07-28,09:25:37	uc
...								

DEPT_VT

DNAME	DNO	MANAGER_SSN	VST	VET	TST	TET
Research	5	888665555	2001-09-20	now	2001-09-15,14:52:12	2001-03-28,09:23:57
Research	5	888665555	2001-09-20	1997-03-31	2002-03-28,09:23:57	uc
Research	5	333445555	2002-04-01	now	2002-03-28,09:23:57	uc

FIGURE 24.9 Some tuple versions in the bitemporal relations EMP_BT and DEPT_BT.

Now consider how an **update operation** would be implemented on a bitemporal relation. In this model of bitemporal databases,[19] *no attributes are physically changed* in any tuple except for the transaction end time attribute TET with a value of *uc*.[20] To illustrate how tuples are created, consider the EMP_BT relation. The *current version* V of an employee has *uc* in its TET attribute and *now* in its VET attribute. If some attribute—say, SALARY—is updated, then the transaction T that performs the update should have two parameters: the new value of SALARY and the valid time VT when the new salary becomes effective (in the real world). Assume that VT− is the time point before VT in the given valid time granularity and that transaction T has a timestamp TS(T). Then, the following physical changes would be applied to the EMP_BT table:

1. Make a copy V2 of the current version V; set V2.VET to VT−, V2.TST to TS(T), V2.TET to *uc*, and insert V2 in EMP_BT; V2 is a copy of the previous current version V *after it is closed* at valid time VT−.
2. Make a copy V3 of the current version V; set V3.VST to VT, V3.VET to *now*, V3.SALARY to the new salary value, V3.TST to TS(T), V3.TET to *uc*, and insert V3 in EMP_BT; V3 represents the new current version.

19. There have been many proposed temporal database models. We are describing specific models here as examples to illustrate the concepts.

20. Some bitemporal models allow the VET attribute to be changed also, but the interpretations of the tuples are different in those models.

3. Set V.TET to TS(T) since the current version is no longer representing correct information.

As an illustration, consider the first three tuples V1, V2, and V3 in EMP_BT in Figure 24.9. Before the update of Smith's salary from 25000 to 30000, only V1 was in EMP_BT and it was the current version and its TET was *uc*. Then, a transaction T whose timestamp TS(T) is 2003-06-04,08:56:12 updates the salary to 30000 with the effective valid time of 2003-06-01. The tuple V2 is created, which is a copy of V1 except that its VET is set to 2003-05-31, one day less than the new valid time and its TST is the timestamp of the updating transaction. The tuple V3 is also created, which has the new salary, its VST is set to 2003-06-01, and its TST is also the timestamp of the updating transaction. Finally, the TET of V1 is set to the timestamp of the updating transaction, 2003-06-04,08:56:12. Note that this is a *retroactive update*, since the updating transaction ran on June 4, 2003, but the salary change is effective on June 1, 2003.

Similarly, when Wong's salary and department are updated (at the same time) to 30000 and 5, the updating transaction's timestamp is 2001-01-07,14:33:02 and the effective valid time for the update is 2001-02-01. Hence, this is a *proactive update* because the transaction ran on January 7, 2001, but the effective date was February 1, 2001. In this case, tuple V4 is logically replaced by V5 and V6.

Next, let us illustrate how a **delete operation** would be implemented on a bitemporal relation by considering the tuples V9 and V10 in the EMP_BT relation of Figure 24.9. Here, employee Brown left the company effective August 10, 2002, and the logical delete is carried out by a transaction T with TS(T) = 2002-08-12,10:11:07. Before this, V9 was the current version of Brown, and its TET was *uc*. The logical delete is implemented by setting V9.TET to 2002-08-12,10:11:07 to invalidate it, and creating the *final version* V10 for Brown, with its VET = 2002-08-10 (see Figure 24.9). Finally, an **insert operation** is implemented by creating the *first version* as illustrated by V11 in the EMP_BT table.

Implementation Considerations. There are various options for storing the tuples in a temporal relation. One is to store all the tuples in the same table, as in Figures 23.8 and 23.9. Another option is to create two tables: one for the currently valid information and the other for the rest of the tuples. For example, in the bitemporal EMP_BT relation, tuples with *uc* for their TET and *now* for their VET would be in one relation, the *current table*, since they are the ones currently valid (that is, represent the current snapshot), and all other tuples would be in another relation. This allows the database administrator to have different access paths, such as indexes for each relation, and keeps the size of the current table reasonable. Another possibility is to create a third table for corrected tuples whose TET is not *uc*.

Another option that is available is to *vertically partition* the attributes of the temporal relation into separate relations. The reason for this is that, if a relation has many attributes, a whole new tuple version is created whenever any one of the attributes is updated. If the attributes are updated asynchronously, each new version may differ in only one of the attributes, thus needlessly repeating the other attribute values. If a separate relation is created to contain only the attributes that *always change synchronously*, with the primary key replicated in each relation, the database is said to be in **temporal normal**

form. However, to combine the information, a variation of join known as **temporal intersection join** would be needed, which is generally expensive to implement.

It is important to note that bitemporal databases allow a complete record of changes. Even a record of corrections is possible. For example, it is possible that two tuple versions of the same employee may have the same valid time but different attribute values as long as their transaction times are disjoint. In this case, the tuple with the later transaction time is a **correction** of the other tuple version. Even incorrectly entered valid times may be corrected this way. The incorrect state of the database will still be available as a previous database state for querying purposes. A database that keeps such a complete record of changes and corrections has been called an **append only database.**

24.2.3 Incorporating Time in Object-Oriented Databases Using Attribute Versioning

The previous section discussed the **tuple versioning approach** to implementing temporal databases. In this approach, whenever one attribute value is changed, a whole new tuple version is created, even though all the other attribute values will be identical to the previous tuple version. An alternative approach can be used in database systems that support **complex structured objects,** such as object databases (see Chapters 20 and 21) or object-relational systems (see Chapter 22). This approach is called **attribute versioning.**[21]

In attribute versioning, a single complex object is used to store all the temporal changes of the object. Each attribute that changes over time is called a **time-varying attribute,** and it has its values versioned over time by adding temporal periods to the attribute. The temporal periods may represent valid time, transaction time, or bitemporal, depending on the application requirements. Attributes that do not change are called **non-time-varying** and are not associated with the temporal periods. To illustrate this, consider the example in Figure 24.10, which is an attribute versioned valid time representation of EMPLOYEE using the ODL notation for object databases (see Chapter 21). Here, we assumed that name and social security number are non-time-varying attributes (they do not change over time), whereas salary, department, and supervisor are time-varying attributes (they may change over time). Each time-varying attribute is represented as a list of tuples <VALID_START_TIME, VALID_END_TIME, VALUE>, ordered by valid start time.

Whenever an attribute is changed in this model, the current attribute version is *closed* and a **new attribute version** for this attribute only is appended to the list. This allows attributes to change asynchronously. The current value for each attribute has *now* for its VALID_END_TIME. When using attribute versioning, it is useful to include a **lifespan temporal attribute** associated with the whole object whose value is one or more valid time periods that indicate the valid time of existence for the whole object. Logical deletion of the object is implemented by closing the lifespan. The constraint that any time period of an attribute within an object should be a subset of the object's lifespan should be enforced.

21. Attribute versioning can also be used in the nested relational model (see Chapter 22).

```
class Temporal_Salary
{
    attribute   Date            valid_start_time;
    attribute   Date            valid_end_time;
    attribute   float           salary;
};

class Temporal_Dept
{
    attribute   Date            valid_start_time;
    attribute   Date            valid_end_time;
    attribute   Department_VT   dept;
};

class Temporal_Supervisor
{
    attribute   Date            valid_start_time;
    attribute   Date            valid_end_time;
    attribute   Employee_VT     supervisor;
};

class Temporal_Lifespan
{
    attribute   Date            valid_ start_time;
    attribute   Date            valid_end_time;
};

class Employee_VT
(   extent   employees)
{
    attribute   list<Temporal_Lifespan>     lifespan;
    attribute   string                      name;
    attribute   string                      ssn;
    attribute   list<Temporal_Salary>       sal_history;
    attribute   list<Temporal_Dept>         dept_history;
    attribute   list<Temporal_Supervisor>   supervisor_history;
};
```

FIGURE 24.10 Possible ODL schema for a temporal valid time `Employee_VT` object class using attribute versioning.

For bitemporal databases, each attribute version would have a tuple with five components:

<valid_start_time, valid_end_time, trans_start_time, trans_end_time, value>

The object lifespan would also include both valid and transaction time dimensions. The full capabilities of bitemporal databases can hence be available with attribute versioning. Mechanisms similar to those discussed earlier for updating tuple versions can be applied to updating attribute versions.

24.2.4 Temporal Querying Constructs and the TSQL2 Language

So far, we have discussed how data models may be extended with temporal constructs. We now give a brief overview of how query operations need to be extended for temporal querying. Then we briefly discuss the TSQL2 language, which extends SQL for querying valid time, transaction time, and bitemporal relational databases.

In nontemporal relational databases, the typical selection conditions involve attribute conditions, and tuples that satisfy these conditions are selected from the set of *current tuples*. Following that, the attributes of interest to the query are specified by a *projection operation* (see Chapter 5). For example, in the query to retrieve the names of all employees working in department 5 whose salary is greater than 30000, the selection condition would be:

```
((SALARY > 30000) AND (DNO = 5))
```

The projected attribute would be NAME. In a temporal database, the conditions may involve time in addition to attributes. A **pure time condition** involves only time—for example, to select all employee tuple versions that were valid on a certain *time point* T or that were valid *during a certain time period* [T1, T2]. In this case, the specified time period is compared with the valid time period of each tuple version [T.VST, T.VET], and only those tuples that satisfy the condition are selected. In these operations, a period is considered to be equivalent to the set of time points from T1 to T2 inclusive, so the standard set comparison operations can be used. Additional operations, such as whether one time period ends *before* another starts are also needed.[22] Some of the more common operations used in queries are as follows:

[t.VST, t.VET] **INCLUDES** [t1, t2]	Equivalent to t1 ≥ t.VST AND t2 ≤ t.VET
[t.VST, t.VET] **INCLUDED_IN** [t1, t2]	Equivalent to t1 ≤ t.VST AND t2 ≥ t.VET
[t.VST, t.VET] **OVERLAPS** [t1, t2]	Equivalent to (t1 ≤ t.VET AND t2 ≥ t.VST)[23]
[t.VST, t.VET] **BEFORE** [t1, t2]	Equivalent to t1 ≥ t.VET
[t.VST, t.VET] **AFTER** [t1, t2]	Equivalent to t2 ≤ t.VST
[t.VST, t.VET] **MEETS_BEFORE** [t1, t2]	Equivalent to t1 = t.VET + 1[24]
[t.VST, t.VET] **MEETS_AFTER** [t1, t2]	Equivalent to t2 + 1 = t.VST

In addition, operations are needed to manipulate time periods, such as computing the union or intersection of two time periods. The results of these operations may not themselves be periods, but rather **temporal elements**—a collection of one or more *disjoint* time periods such that no two time periods in a temporal element are directly adjacent.

22. A complete set of operations, known as **Allen's algebra,** has been defined for comparing time periods.

23. This operation returns true if the *intersection* of the two periods is not empty; it has also been called INTERSECTS_WITH.

24. Here, 1 (one) refers to one time point in the specified granularity. The MEETS operations basically specify if one period starts immediately after the other period ends.

That is, for any two time periods [T1, T2] and [T3, T4] in a temporal element, the following three conditions must hold:

- [T1, T2] **intersection** [T3, T4] is empty.
- T3 is not the time point following T2 in the given granularity.
- T1 is not the time point following T4 in the given granularity.

The latter conditions are necessary to ensure unique representations of temporal elements. If two time periods [T1, T2] and [T3, T4] are adjacent, they are combined into a single time period [T1, T4]. This is called **coalescing** of time periods. Coalescing also combines intersecting time periods.

To illustrate how pure time conditions can be used, suppose a user wants to select all employee versions that were valid at any point during 2002. The appropriate selection condition applied to the relation in Figure 24.8 would be

[T.VST, T.VET] **OVERLAPS** [2002-01-01, 2002-12-31]

Typically, most temporal selections are applied to the valid time dimension. For a bitemporal database, one usually applies the conditions to the currently correct tuples with *uc* as their transaction end times. However, if the query needs to be applied to a previous database state, an AS_OF T clause is appended to the query, which means that the query is applied to the valid time tuples that were correct in the database at time T.

In addition to pure time conditions, other selections involve **attribute and time conditions.** For example, suppose we wish to retrieve all EMP_VT tuple versions T for employees who worked in department 5 at any time during 2002. In this case, the condition is

([T.VST, T.VET] **OVERLAPS** [2002-01-01, 2002-12-31]) AND (T.DNO = 5)

Finally, we give a brief overview of the TSQL2 query language, which extends SQL with constructs for temporal databases. The main idea behind TSQL2 is to allow users to specify whether a relation is nontemporal (that is, a standard SQL relation) or temporal. The CREATE TABLE statement is extended with an *optional* AS-clause to allow users to declare different temporal options. The following options are available:

- AS VALID STATE <GRANULARITY> (valid time relation with valid time period)
- AS VALID EVENT <GRANULARITY> (valid time relation with valid time point)
- AS TRANSACTION (transaction time relation with transaction time period)
- AS VALID STATE <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time period)
- AS VALID EVENT <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time point)

The keywords STATE and EVENT are used to specify whether a time *period* or time *point* is associated with the valid time dimension. In TSQL2, rather than have the user actually see how the temporal tables are implemented (as we discussed in the previous sections), the TSQL2 language adds query language constructs to specify various types of temporal selections, temporal projections, temporal aggregations, transformation among granularities, and many other concepts. The book by Snodgrass et al. (1995) describes the language.

24.2.5 Time Series Data

Time series data is used very often in financial, sales, and economics applications. They involve data values that are recorded according to a specific predefined sequence of time points. They are hence a special type of **valid event data,** where the event time points are predetermined according to a fixed calendar. Consider the example of closing daily stock prices of a particular company on the New York Stock Exchange. The granularity here is day, but the days that the stock market is open are known (nonholiday weekdays). Hence, it has been common to specify a computational procedure that calculates the particular **calendar** associated with a time series. Typical queries on time series involve **temporal aggregation** over higher granularity intervals—for example, finding the average or maximum *weekly* closing stock price or the maximum and minimum *monthly* closing stock price from the *daily* information.

As another example, consider the daily sales dollar amount at each store of a chain of stores owned by a particular company. Again, typical temporal aggregates would be retrieving the weekly, monthly, or yearly sales from the daily sales information (using the sum aggregate function), or comparing same store monthly sales with previous monthly sales, and so on.

Because of the specialized nature of time series data, and the lack of support in older DBMSs, it has been common to use specialized **time series management systems** rather than general purpose DBMSs for managing such information. In such systems, it has been common to store time series values in sequential order in a file, and apply specialized time series procedures to analyze the information. The problem with this approach is that the full power of high-level querying in languages such as SQL will not be available in such systems.

More recently, some commercial DBMS packages are offering time series extensions, such as the time series datablade of Informix Universal Server (see Chapter 22). In addition, the TSQL2 language provides some support for time series in the form of event tables.

24.3 Multimedia Databases

Because the two topics discussed in this section are very broad, we can give only a very brief introduction to these fields. Section 24.3.1 introduces spatial databases, and Section 24.3.2 briefly discusses multimedia databases.

24.3.1 Introduction to Spatial Database Concepts

Spatial databases provide concepts for databases that keep track of objects in a multidimensional space. For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects—from countries and states to rivers, cities, roads, seas, and so on. These applications are also known as Geographical Information Systems (GIS), and are used in areas such as environmental, emergency, and battle management. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are

related to three-dimensional spatial points. In general, a spatial database stores objects that have spatial characteristics that describe them. The spatial relationships among the objects are important, and they are often needed when querying the database. Although a spatial database can in general refer to an n-dimensional space for any n, we will limit our discussion to two dimensions as an illustration.

The main extensions that are needed for spatial databases are models that can interpret spatial characteristics. In addition, special indexing and storage structures are often needed to improve performance. Let us first discuss some of the model extensions for two-dimensional spatial databases. The basic extensions needed are to include two-dimensional geometric concepts, such as points, lines and line segments, circles, polygons, and arcs, in order to specify the spatial characteristics of objects. In addition, spatial operations are needed to operate on the objects' spatial characteristics—for example, to compute the distance between two objects—as well as spatial Boolean conditions—for example, to check whether two objects spatially overlap. To illustrate, consider a database that is used for emergency management applications. A description of the spatial positions of many types of objects would be needed. Some of these objects generally have static spatial characteristics, such as streets and highways, water pumps (for fire control), police stations, fire stations, and hospitals. Other objects have dynamic spatial characteristics that change over time, such as police vehicles, ambulances, or fire trucks.

The following categories illustrate three typical types of spatial queries:

- *Range query:* Finds the objects of a particular type that are within a given spatial area or within a particular distance from a given location. (For example, finds all hospitals within the Dallas city area, or finds all ambulances within five miles of an accident location.)
- *Nearest neighbor query:* Finds an object of a particular type that is closest to a given location. (For example, finds the police car that is closest to a particular location.)
- *Spatial joins or overlays:* Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another. (For example, finds all cities that fall on a major highway or finds all homes that are within two miles of a lake.)

For these and other types of spatial queries to be answered efficiently, special techniques for spatial indexing are needed. One of the best known techniques is the use of **R-trees** and their variations. R-trees group together objects that are in close spatial physical proximity on the same leaf nodes of a tree-structured index. Since a leaf node can point to only a certain number of objects, algorithms for dividing the space into rectangular subspaces that include the objects are needed. Typical criteria for dividing the space include minimizing the rectangle areas, since this would lead to a quicker narrowing of the search space. Problems such as having objects with overlapping spatial areas are handled in different ways by the many different variations of R-trees. The internal nodes of R-trees are associated with rectangles whose area covers all the rectangles in its subtree. Hence, R-trees can easily answer queries, such as find all objects in a given area by limiting the tree search to those subtrees whose rectangles intersect with the area given in the query.

Other spatial storage structures include quadtrees and their variations. **Quadtrees** generally divide each space or subspace into equally sized areas, and proceed with the subdivisions of each subspace to identify the positions of various objects. Recently, many newer spatial access structures have been proposed, and this area is still an active research area.

24.3.2 Introduction to Multimedia Database Concepts

Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes *images* (such as photos or drawings), *video clips* (such as movies, newsreels, or home videos), *audio clips* (such as songs, phone messages, or speeches), and *documents* (such as books or articles). The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. For example, one may want to locate all video clips in a video database that include a certain person in them, say Bill Clinton. One may also want to retrieve video clips based on certain activities included in them, such as a video clips where a goal is scored in a soccer game by a certain player or team.

The above types of queries are referred to as **content-based retrieval,** because the multimedia source is being retrieved based on its containing certain objects or activities. Hence, a multimedia database must use some model to organize and index the multimedia sources based on their contents. *Identifying the contents* of multimedia sources is a difficult and time-consuming task. There are two main approaches. The first is based on **automatic analysis** of the multimedia sources to identify certain mathematical characteristics of their contents. This approach uses different techniques depending on the type of multimedia source (image, text, video, or audio). The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all the different multimedia sources, but it requires a manual preprocessing phase where a person has to scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index these sources.

In the remainder of this section, we will very briefly discuss some of the characteristics of each type of multimedia source—images, video, audio, and text sources, in that order.

An **image** is typically stored either in raw form as a set of pixel or cell values, or in compressed form to save space. The image *shape descriptor* describes the geometric shape of the raw image, which is typically a rectangle of **cells** of a certain width and height. Hence, each image can be represented by an m by n grid of cells. Each cell contains a pixel value that describes the cell content. In black/white images, pixels can be one bit. In gray scale or color images, a pixel is multiple bits. Because images may require large amounts of space, they are often stored in compressed form. Compression standards, such as GIF or JPEG, use various mathematical transformations to reduce the number of cells stored but still maintain the main image characteristics. The mathematical transforms

that can be used include Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and wavelet transforms.

To identify objects of interest in an image, the image is typically divided into homogeneous segments using a *homogeneity predicate*. For example, in a color image, cells that are adjacent to one another and whose pixel values are close are grouped into a segment. The homogeneity predicate defines the conditions for how to automatically group those cells. Segmentation and compression can hence identify the main characteristics of an image.

A typical image database query would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern. There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group together stored images that are close in the distance metric so as to limit the search space. The second approach, called the **transformation approach,** measures image similarity by having a small number of transformations that can transform one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the latter approach is more general, it is also more time consuming and difficult.

A **video source** is typically represented as a sequence of frames, where each frame is a still image. However, rather than identifying the objects and activities in every individual frame, the video is divided into **video segments,** where each segment is made up of a sequence of contiguous frames that includes the same objects/activities. Each segment is identified by its starting and ending frames. The objects and activities identified in each video segment can be used to index the segments. An indexing technique called *frame segment trees* has been proposed for video indexing. The index includes both objects, such as persons, houses, cars, and activities, such as a person *delivering* a speech or two people *talking*. Videos are also often compressed using standards such as MPEG.

A **text/document source** is basically the full text of some article, book, or magazine. These sources are typically indexed by identifying the keywords that appear in the text and their relative frequencies. However, filler words are eliminated from that process. Because there could be too many keywords when attempting to index a collection of documents, techniques have been developed to reduce the number of keywords to those that are most relevant to the collection. A technique called *singular value decompositions* (SVD), which is based on matrix transformations, can be used for this purpose. An indexing technique called *telescoping vector trees*, or TV-trees, can then be used to group similar documents together.

Audio sources include stored recorded messages, such as speeches, class presentations, or even surveillance recording of phone messages or conversations by law enforcement. Here, discrete transforms can be used to identify the main characteristics of a certain person's voice in order to have similarity based indexing and retrieval. Audio characteristic features include loudness, intensity, pitch, and clarity.

24.4 Introduction to Deductive Databases

24.4.1 Overview of Deductive Databases

In a deductive database system, we typically specify rules through a **declarative language**—a language in which we specify what to achieve rather than how to achieve it. An **inference engine** (or **deduction mechanism**) within the system can deduce new facts from the database by interpreting these rules. The model used for deductive databases is closely related to the relational data model, and particularly to the domain relational calculus formalism (see Section 6.6). It is also related to the field of **logic programming** and the **Prolog** language. The deductive database work based on logic has used Prolog as a starting point. A variation of Prolog called **Datalog** is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language. Although the language structure of Datalog resembles that of Prolog, its operational semantics—that is, how a Datalog program is to be executed—is still different.

A deductive database uses two main types of specifications: facts and rules. **Facts** are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names. Recall that a tuple in a relation describes some real-world fact whose meaning is partly determined by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is determined solely by its *position* within the tuple. **Rules** are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications. The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of basic relational views.

The evaluation of Prolog programs is based on a technique called *backward chaining*, which involves a top-down evaluation of goals. In the deductive databases that use Datalog, attention has been devoted to handling large volumes of data stored in a relational database. Hence, evaluation techniques have been devised that resemble those for a bottom-up evaluation. Prolog suffers from the limitation that the order of specification of facts and rules is significant in evaluation; moreover, the order of literals (defined later in Section 24.4.3) within a rule is significant. The execution techniques for Datalog programs attempt to circumvent these problems.

24.4.2 Prolog/Datalog Notation

The notation used in Prolog/Datalog is based on providing predicates with unique names. A **predicate** has an implicit meaning, which is suggested by the predicate name, and a fixed number of **arguments.** If the arguments are all constant values, the predicate simply states that a certain fact is true. If, on the other hand, the predicate has variables as arguments, it is either considered as a query or as part of a rule or constraint. Throughout this chapter, we adopt the Prolog convention that all **constant values** in a predicate are either *numeric* or character strings; they are represented as identifiers (or names) starting with *lowercase letters* only, whereas **variable names** always start with an *uppercase letter.*

Consider the example shown in Figure 24.11, which is based on the relational database of Figure 5.6, but in a much simplified form. There are three predicate names: *supervise*, *superior*, and *subordinate*. The `supervise` predicate is defined via a set of facts, each of which has two arguments: a supervisor name, followed by the name of a *direct* supervisee (subordinate) of that supervisor. These facts correspond to the actual data that is stored in the database, and they can be considered as constituting a set of tuples in a relation SUPERVISE with two attributes whose schema is

```
SUPERVISE(Supervisor,Supervisee)
```

Thus, `supervise(X,Y)` states the fact that "X supervises Y." Notice the omission of the attribute names in the Prolog notation. Attribute names are only represented by virtue of the position of each argument in a predicate: the first argument represents the supervisor, and the second argument represents a direct subordinate.

The other two predicate names are defined by rules. The main contribution of deductive databases is the ability to specify recursive rules, and to provide a framework for inferring new information based on the specified rules. A rule is of the form **head :- body,** where :- is read as "if and only if." A rule usually has a **single predicate** to the left of the :- symbol—called the **head** or **left-hand side** (LHS) or **conclusion** of the rule—and **one or more predicates** to the right of the :- symbol—called the **body** or **right-hand side** (RHS) or **premise(s)** of the rule. A predicate with constants as arguments is said to be **ground;** we also refer to it as an **instantiated predicate.** The arguments of the predicates that appear in a rule typically include a number of variable symbols, although predicates can also contain constants as arguments. A rule specifies that, if a particular assignment or **binding** of constant values to the variables in the body (RHS predicates) makes *all* the RHS predicates **true,** it also makes the head (LHS predicate) true by using the same assignment of constant values to variables. Hence, a rule provides us with a way of generating new facts that are instantiations of the head of the rule. These new facts are based on facts that

(a)

```
        Facts
supervise(franklin,john).
supervise(franklin,ramesh).
supervise(franklin,joyce).
supervise(jennifer,alicia).
supervise(jennifer,ahmad).
supervise(james,franklin).
supervise(james,jennifer).
     ...

        Rules
superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).
subordinate(X,Y) :- superior(Y,X).

        Queries
superior(james,Y)?
superior(james,joyce)?
```

(b)

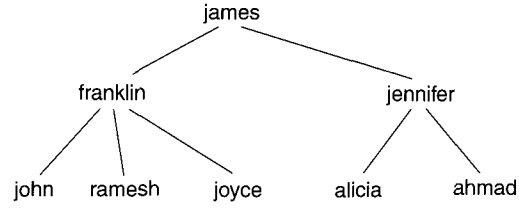


FIGURE 24.11 (a) Prolog notation. (b) The supervisory tree.

already exist, corresponding to the instantiations (or bindings) of predicates in the body of the rule. Notice that by listing multiple predicates in the body of a rule we implicitly apply the **logical and** operator to these predicates. Hence, the commas between the RHS predicates may be read as meaning "and."

Consider the definition of the predicate `superior` in Figure 24.11, whose first argument is an employee name and whose second argument is an employee who is either a *direct* or an *indirect* subordinate of the first employee. By *indirect subordinate*, we mean the subordinate of some subordinate down to any number of levels. Thus `superior(X,Y)` stands for the fact that "X is a superior of Y" through direct or indirect supervision. We can write two rules that together specify the meaning of the new predicate. The first rule under Rules in the figure states that, for every value of X and Y, if `supervise(X,Y)`—the rule body—is true, then `superior(X,Y)`—the rule head—is also true, since Y would be a direct subordinate of X (at one level down). This rule can be used to generate all direct superior/subordinate relationships from the facts that define the supervise predicate. The second recursive rule states that, if `supervise(X, Z)` *and* `superior(Z, Y)` are *both* true, then `superior(X,Y)` is also true. This is an example of a **recursive rule,** where one of the rule body predicates in the RHS is the same as the rule head predicate in the LHS. In general, the rule body defines a number of premises such that, if they are all true, we can deduce that the conclusion in the rule head is also true. Notice that, if we have two (or more) rules with the same head (LHS predicate), it is equivalent to saying that the predicate is true (that is, that it can be instantiated) if *either one* of the bodies is true; hence, it is equivalent to a **logical or** operation. For example, if we have two rules `X :- Y` and `X :- Z`, they are equivalent to a rule `X :- Y` **`or`** `Z`. The latter form is not used in deductive systems, however, because it is not in the standard form of rule, called a Horn clause, as we discuss in Section 24.4.4.

A Prolog system contains a number of **built-in** predicates that the system can interpret directly. These typically include the equality comparison operator `=(X,Y)`, which returns true if X and Y are identical and can also be written as `X=Y` by using the standard infix notation.[25] Other comparison operators for numbers, such as <, <=, >, and >=, can be treated as binary predicates. Arithmetic functions such as +, −, *, and / can be used as arguments in predicates in Prolog. In contrast, Datalog (in its basic form) does *not* allow functions such as arithmetic operations as arguments; indeed, this is one of the main differences between Prolog and Datalog. However, later extensions to Datalog have been proposed to include functions.

A **query** typically involves a predicate symbol with some variable arguments, and its meaning (or "answer") is to deduce all the different constant combinations that, when **bound** (assigned) to the variables, can make the predicate true. For example, the first query in Figure 24.11 requests the names of all subordinates of "james" at any level. A different type of query, which has only constant symbols as arguments, returns either a true or a false result, depending on whether the arguments provided can be deduced from

25. A Prolog system typically has a number of different equality predicates that have different interpretations.

the facts and rules. For example, the second query in Figure 24.11 returns true, since `superior(james, joyce)` can be deduced.

24.4.3 Datalog Notation

In Datalog, as in other logic-based languages, a program is built from basic objects called **atomic formulas.** It is customary to define the syntax of logic-based languages by describing the syntax of atomic formulas and identifying how they can be combined to form a program. In Datalog, atomic formulas are **literals** of the form $p(a_1, a_2, \ldots, a_n)$, where p is the predicate name and n is the number of arguments for predicate p. Different predicate symbols can have different numbers of arguments, and the number of arguments n of predicate p is sometimes called the **arity** or **degree** of p. The arguments can be either constant values or variable names. As mentioned earlier, we use the convention that constant values either are numeric or start with a *lowercase* character, whereas variable names always start with an *uppercase* character.

A number of **built-in predicates** are included in Datalog, which can also be used to construct atomic formulas. The built-in predicates are of two main types: the binary comparison predicates `<(less)`, `<=(less_or_equal)`, `>(greater)`, and `>= (greater_or_equal)` over ordered domains; and the comparison predicates `= (equal)` and `/= (not_equal)` over ordered or unordered domains. These can be used as binary predicates with the same functional syntax as other predicates—for example by writing `less(X, 3)`—or they can be specified by using the customary infix notation `X<3`. Notice that, because the domains of these predicates are potentially infinite, they should be used with care in rule definitions. For example, the predicate `greater(X, 3)`, if used alone, generates an infinite set of values for X that satisfy the predicate (all integer numbers greater than 3).

A **literal** is either an atomic formula as defined earlier—called a **positive literal**—or an atomic formula preceded by **not.** The latter is a negated atomic formula, called a **negative literal.** Datalog programs can be considered to be a *subset* of the **predicate calculus** formulas, which are somewhat similar to the formulas of the domain relational calculus (see Section 6.7). In Datalog, however, these formulas are first converted into what is known as **clausal form** before they are expressed in Datalog; and only formulas given in a restricted clausal form, called Horn clauses,[26] can be used in Datalog.

24.4.4 Clausal Form and Horn Clauses

Recall from Section 6.6 that a formula in the relational calculus is a condition that includes predicates called *atoms* (based on relation names). In addition, a formula can have quantifiers—namely, the *universal quantifier* (for all) and the *existential quantifier*

26. Named after the mathematician Alfred Horn.

(there exists). In clausal form, a formula must be transformed into another formula with the following characteristics:

- All variables in the formula are universally quantified. Hence, it is not necessary to include the universal quantifiers (for all) explicitly; the quantifiers are removed, and all variables in the formula are *implicitly* quantified by the universal quantifier.
- In clausal form, the formula is made up of a number of clauses, where each **clause** is composed of a number of *literals* connected by OR logical connectives only. Hence, each clause is a *disjunction* of literals.
- The *clauses themselves* are connected by AND logical connectives only, to form a formula. Hence, the *clausal form of a formula* is a *conjunction* of clauses.

It can be shown that *any formula can be converted into clausal form*. For our purposes, we are mainly interested in the form of the individual clauses, each of which is a disjunction of literals. Recall that literals can be positive literals or negative literals. Consider a clause of the form:

$$\text{not}(P_1) \text{ OR } \text{not}(P_2) \text{ OR } \ldots \text{ OR } \text{not}(P_n) \text{ OR } Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (1)$$

This clause has n negative literals and m positive literals. Such a clause can be transformed into the following equivalent logical formula:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n => Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (2)$$

where => is the **implies** symbol. The formulas (1) and (2) are equivalent, meaning that their truth values are always the same. This is the case because, if all the P_i literals ($i = 1, 2, \ldots, n$) are true, the formula (2) is true only if at least one of the Q_i's is true, which is the meaning of the => (implies) symbol. For formula (1), if all the P_i literals ($i = 1, 2, \ldots, n$) are true, their negations are all false; so in this case formula (1) is true only if at least one of the Q_i's is true. In Datalog, rules are expressed as a restricted form of clauses called **Horn clauses,** in which a clause can contain *at most one* positive literal. Hence, a Horn clause is either of the form

$$\text{not}(P_1) \text{ OR } \text{not}(P_2) \text{ OR } \ldots \text{ OR } \text{not}(P_n) \text{ OR } Q \quad (3)$$

or of the form

$$\text{not}(P_1) \text{ OR } \text{not}(P_2) \text{ OR } \ldots \text{ OR } \text{not}(P_n) \quad (4)$$

The Horn clause in (3) can be transformed into the clause

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n => Q \quad (5)$$

which is written in Datalog as the following rule

$$Q :- P_1, P_2, \ldots, P_n. \quad (6)$$

The Horn clause in (4) can be transformed into

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n => \quad (7)$$

which is written in Datalog as follows:

$$P_1, P_2, \ldots, P_n. \quad (8)$$

A Datalog rule, as in (6), is hence a Horn clause, and its meaning, based on formula (5), is that if the predicates P_1 and P_2 and . . . and P_n are all true for a particular binding to their variable arguments, then Q is also true and can hence be inferred. The Datalog expression (8) can be considered as an integrity constraint, where all the predicates must be true to satisfy the query.

In general, a query in Datalog consists of two components:

- A Datalog program, which is a finite set of rules.
- A literal $P(X_1, X_2, \ldots, X_n)$, where each X_i is a variable or a constant.

A Prolog or Datalog system has an internal **inference engine** that can be used to process and compute the results of such queries. Prolog inference engines typically return one result to the query (that is, one set of values for the variables in the query) at a time and must be prompted to return additional results. On the contrary, Datalog returns results set-at-a-time.

24.4.5 Interpretations of Rules

There are two main alternatives for interpreting the theoretical meaning of rules: *proof-theoretic* and *model-theoretic*. In practical systems, the inference mechanism within a system defines the exact interpretation, which may not coincide with either of the two theoretical interpretations. The inference mechanism is a computational procedure and hence provides a computational interpretation of the meaning of rules. In this section, we first discuss the two theoretical interpretations. Inference mechanisms are then discussed briefly as a way of defining the meaning of rules.

In the **proof-theoretic** interpretation of rules, we consider the facts and rules to be true statements, or **axioms. Ground axioms** contain no variables. The facts are ground axioms that are given to be true. Rules are called **deductive axioms,** since they can be used to deduce new facts. The deductive axioms can be used to construct proofs that derive new facts from existing facts. For example, Figure 24.12 shows how to prove the fact `superior(james, ahmad)` from the rules and facts given in Figure 24.11. The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query. The process of proving whether a certain fact (theorem) holds is known as *theorem proving*.

```
1. superior(X,Y) :- supervise(X,Y).                    (rule 1)
2. superior(X,Y) :- supervise(X,Z), superior(Z,Y).     (rule 2)

3. supervise(jennifer,ahmad).                          (ground axiom, given)
4. supervise(james,jennifer).                          (ground axiom, given)
5. superior(jennifer,ahmad).                           (apply rule 1 on 3)
6. superior(james,ahmad).                              (apply rule 2 on 4 and 5)
```

FIGURE 24.12 Proving a new fact.

The second type of interpretation is called the **model-theoretic** interpretation. Here, given a finite or an infinite domain of constant values,[27] we assign to a predicate every possible combination of values as arguments. We must then determine whether the predicate is true or false. In general, it is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an **interpretation** of the set of predicates. For example, consider the interpretation shown in Figure 24.13 for the predicates `supervise` and `superior`. This interpretation assigns a truth value (true or false) to every possible combination of argument values (from a finite domain) for the two predicates.

An interpretation is called a **model** for a *specific set of rules* if those rules are *always true* under that interpretation; that is, for any values assigned to the variables in the rules, the head of the rules is true when we substitute the truth values assigned to the predicates

Rules

superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).

Interpretation

Known Facts:

supervise(franklin,john) is **true**.
supervise(franklin,ramesh) is **true**.
supervise(franklin,joyce) is **true**.
supervise(jennifer,alicia) is **true**.
supervise(jennifer,ahmad) is **true**.
supervise(james,franklin) is **true**.
supervise(james,jennifer) is **true**.
supervise(X,Y) is **false** for all other possible (X,Y) combinations.

Derived Facts:

superior(franklin,john) is **true**.
superior(franklin,ramesh) is **true**.
superior(franklin,joyce) is **true**.
superior(jennifer,alicia) is **true**.
superior(jennifer,ahmad) is **true**.
superior(james,franklin) is **true**.
superior(james,jennifer) is **true**.
superior(james,john) is **true**.
superior(james,ramesh) is **true**.
superior(james,joyce) is **true**.
superior(james,alicia) is **true**.
superior(james,ahmad) is **true**.
superior(X,Y) is **false** for all other possible (X,Y) combinations.

FIGURE 24.13 An interpretation that is a minimal model.

27. The most commonly chosen domain is finite and is called the *Herbrand Universe*.

in the body of the rule by that interpretation. Hence, whenever a particular substitution (binding) to the variables in the rules is applied, if all the predicates in the body of a rule are true under the interpretation, the predicate in the head of the rule must also be true. The interpretation shown in Figure 24.13 is a model for the two rules shown, since it can never cause the rules to be violated. Notice that a rule is violated if a particular binding of constants to the variables makes all the predicates in the rule body true but makes the predicate in the rule head false. For example, if `supervise(a,b)` and `superior(b,c)` are both true under some interpretation, but `superior(a,c)` is not true, the interpretation cannot be a model for the recursive rule:

superior(X,Y) :- supervise(X,Z), superior(Z,Y)

In the model-theoretic approach, the meaning of the rules is established by providing a model for these rules. A model is called a **minimal model** for a set of rules if we cannot change any fact from true to false and still get a model for these rules. For example, consider the interpretation in Figure 24.13, and assume that the supervise predicate is defined by a set of known facts, whereas the superior predicate is defined as an interpretation (model) for the rules. Suppose that we add the predicate `superior(james, bob)` to the true predicates. This remains a model for the rules shown, but it is not a minimal model, since changing the truth value of `superior(james, bob)` from true to false still provides us with a model for the rules. The model shown in Figure 24.13 is the minimal model for the set of facts that are defined by the supervise predicate.

In general, the minimal model that corresponds to a given set of facts in the model-theoretic interpretation should be the same as the facts generated by the proof-theoretic interpretation for the same original set of ground and deductive axioms. However, this is generally true only for rules with a simple structure. Once we allow negation in the specification of rules, the correspondence between interpretations *does not* hold. In fact, with negation, numerous minimal models are possible for a given set of facts.

A third approach to interpreting the meaning of rules involves defining an inference mechanism that is used by the system to deduce facts from the rules. This inference mechanism would define a **computational interpretation** to the meaning of the rules. The Prolog logic programming language uses its inference mechanism to define the meaning of the rules and facts in a Prolog program. Not all Prolog programs correspond to the proof-theoretic or model-theoretic interpretations; it depends on the type of rules in the program. However, for many simple Prolog programs, the Prolog inference mechanism infers the facts that correspond either to the proof-theoretic interpretation or to a minimal model under the model-theoretic interpretation.

24.4.6 Datalog Programs and Their Safety

There are two main methods of defining the truth values of predicates in actual Datalog programs. **Fact-defined predicates** (or **relations**) are defined by listing all the combinations of values (the tuples) that make the predicate true. These correspond to base relations whose contents are stored in a database system. Figure 24.14 shows the fact-defined predicates `employee`, `male`, `female`, `department`, `supervise`, `project`, and `workson`,

```
employee(john).
employee(franklin).
employee(alicia).
employee(jennifer).
employee(ramesh).
employee(joyce).
employee(ahmad).
employee(james).

salary(john,30000).
salary(franklin,40000).
salary(alicia,25000).
salary(jennifer,43000).
salary(ramesh,38000).
salary(joyce,25000).
salary(ahmad,25000).
salary(james,55000).

department(john,research).
department(franklin,research).
department(alicia,administration).
department(jennifer,administration).
department(ramesh,research).
department(joyce,research).
department(ahmad,administration).
department(james,headquarters).

supervise(franklin,john).
supervise(franklin,ramesh).
supervise(franklin,joyce).
supervise(jennifer,alicia).
supervise(jennifer,ahmad).
supervise(james,franklin).
supervise(james,jennifer).

male(john).
male(franklin).
male(ramesh).
male(ahmad).
male(james).

female(alicia).
female(jennifer).
female(joyce).

project(productx).
project(producty).
project(productz).
project(computerization).
project(reorganization).
project(newbenefits).

workson(john,productx,32).
workson(john,producty,8).
workson(ramesh,productz,40).
workson(joyce,productx,20).
workson(joyce,producty,20).
workson(franklin,producty,10).
workson(franklin,productz,10).
workson(franklin,computerization,10).
workson(franklin,reorganization,10).
workson(alicia,newbenefits,30).
workson(alicia,computerization,10).
workson(ahmad,computerization,35).
workson(ahmad,newbenefits,5).
workson(jennifer,newbenefits,20).
workson(jennifer,reorganization,15).
workson(james,reorganization,10).
```

FIGURE 24.14 Fact predicates for part of the database from Figure 5.6.

which correspond to part of the relational database shown in Figure 5.6. **Rule-defined predicates** (or **views**) are defined by being the head (LHS) of one or more Datalog rules; they correspond to *virtual relations* whose contents can be inferred by the inference engine. Figure 24.15 shows a number of rule-defined predicates.

A program or a rule is said to be **safe** if it generates a *finite* set of facts. The general theoretical problem of determining whether a set of rules is safe is undecidable. However, one can determine the safety of restricted forms of rules. For example, the rules shown in Figure 24.16 are safe. One situation where we get unsafe rules that can generate an infinite number of facts arises when one of the variables in the rule can range over an infinite domain of values, and that variable is not limited to ranging over a finite relation. For example, consider the rule

```
big_salary(Y) :- Y>60000
```

Here, we can get an infinite result if Y ranges over all possible integers. But suppose that we change the rule as follows:

```
big_salary(Y) :- employee(X), salary(X,Y), Y>60000
```

```
superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).

subordinate(X,Y) :- superior(Y,X).

supervisor(X) :- employee(X), supervise(X,Y).

over_40K_emp(X) :- employee(X), salary(X,Y), Y>=40000.
under_40K_supervisor(X) :- supervisor(X), not(over_40_K_emp(X)).
main_productx_emp(X) :- employee(X), workson(X,productx,Y), Y>=20.
president(X) :- employee(X), not(supervise(Y,X)).
```

FIGURE 24.15 Rule-defined predicates.

In the second rule, the result is not infinite, since the values that Y can be bound to are now restricted to values that are the salary of some employee in the database—presumably, a finite set of values. We can also rewrite the rule as follows:

big_salary(Y) :- Y>60000, employee(X), salary(X,Y)

In this case, the rule is still theoretically safe. However, in Prolog or any other system that uses a top-down, depth-first inference mechanism, the rule creates an infinite loop, since we first search for a value for Y and then check whether it is a salary of an employee. The result is generation of an infinite number of Y values, even though these, after a certain point, cannot lead to a set of true RHS predicates. One definition of Datalog considers both rules to be safe, since it does not depend on a particular inference mechanism. Nonetheless, it is generally advisable to write such a rule in the safest form, with the predicates that restrict possible bindings of variables placed first. As another example of an unsafe rule, consider the following rule:

has_something(X,Y) :- employee(X)

Here, an infinite number of Y values can again be generated, since the variable Y appears only in the head of the rule and hence is not limited to a finite set of values. To define safe rules more formally, we use the concept of a limited variable. A variable X is **limited** in a rule if (1) it appears in a regular (not built-in) predicate in the body of the rule; (2) it appears in a predicate of the form X=c or c=X or (c1<=X and X<=c2) in the rule body, where c, c1, and c2 are constant values; or (3) it appears in a predicate of the form X=Y or Y=X in the rule body, where Y is a limited variable. A rule is said to be **safe** if all its variables are limited.

24.4.7 Use of Relational Operations

It is straightforward to specify many operations of the relational algebra in the form of Datalog rules that define the result of applying these operations on the database relations (fact predicates). This means that relational queries and views can easily be specified in Datalog. The additional power that Datalog provides is in the specification of recursive

queries, and views based on recursive queries. In this section, we show how some of the standard relational operations can be specified as Datalog rules. Our examples will use the base relations (fact-defined predicates) `rel_one`, `rel_two`, and `rel_three`, whose schemas are shown in Figure 24.16. In Datalog, we do not need to specify the attribute names as in Figure 24.16; rather, the arity (degree) of each predicate is the important aspect. In a practical system, the domain (data type) of each attribute is also important for operations such as UNION, INTERSECTION, and JOIN, and we assume that the attribute types are compatible for the various operations, as discussed in Chapter 5.

Figure 24.16 illustrates a number of basic relational operations. Notice that, if the Datalog model is based on the relational model and hence assumes that predicates (fact relations and query results) specify sets of tuples, duplicate tuples in the same predicate are automatically eliminated. This may or may not be true, depending on the Datalog inference engine. However, it is definitely *not* the case in Prolog, so any of the rules in Figure 24.16 that involve duplicate elimination are not correct for Prolog. For example, if we want to specify Prolog rules for the UNION operation with duplicate elimination, we must rewrite them as follows:

union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z), not(rel_one(X,Y,Z)).

However, the rules shown in Figure 24.16 should work for Datalog, if duplicates are automatically eliminated. Similarly, the rules for the PROJECT operation shown in Figure

```
rel_one(A,B,C).
rel_two(D,E,F).
rel_three(G,H,I,J).

select_one_A_eq_c(X,Y,Z) :- rel_one(c,Y,Z).
select_one_B_less_5(X,Y,Z) :- rel_one(X,Y,Z), Y<5.
select_one_A_eq_c_and_B_less_5(X,Y,Z) :- rel_one(c,Y,Z), Y<5.

select_one_A_eq_c_or_B_less_5(X,Y,Z) :- rel_one(c,Y,Z).
select_one_A_eq_c_or_B_less_5(X,Y,Z) :- rel_one(X,Y,Z), Y<5.

project_three_on_G_H(W,X) :- rel_three(W,X,Y,Z).

union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z).

intersect_one_two(X,Y,Z) :- rel_one(X,Y,Z), rel_two(X,Y,Z).

difference_two_one(X,Y,Z) :- rel_two(X,Y,Z), not(rel_one(X,Y,Z)).

cart_prod_one_three(T,U,V,W,X,Y,Z) :-
                        rel_one(T,U,V), rel_three(W,X,Y,Z).

natural_join_one_three_C_eq_G(U,V,W,X,Y,Z) :-
                        rel_one(U,V,W), rel_three(W,X,Y,Z).
```

FIGURE 24.16 Predicates for illustrating relational operations.

24.16 should work for Datalog in this case, but they are not correct for Prolog, since duplicates would appear in the latter case.

24.4.8 Evaluation of Nonrecursive Datalog Queries

In order to use Datalog as a deductive database system, it is appropriate to define an inference mechanism based on relational database query processing concepts. The inherent strategy involves a bottom-up evaluation, starting with base relations; the order of operations is kept flexible and subject to query optimization. In this section, we discuss an inference mechanism based on relational operations that can be applied to **nonrecursive** Datalog queries. We use the fact and rule base shown in Figures 24.14 and 24.15 to illustrate our discussion.

If a query involves only fact-defined predicates, the inference becomes one of searching among the facts for the query result. For example, a query such as

department(X,research)?

is a selection of all employee names X who work for the `research` department. In relational algebra, it is the query:

$\pi_{\$1}\ (\sigma_{\$2\ =\ \text{"Research"}}\ (\text{department}))$

which can be answered by searching through the fact-defined predicate `department(X,Y)`. The query involves relational SELECT and PROJECT operations on a base relation, and it can be handled by the database query processing and optimization techniques discussed in Chapter 15.

When a query involves rule-defined predicates, the inference mechanism must compute the result based on the rule definitions. If a query is nonrecursive and involves a predicate p that appears as the head of a rule $p :- p_1, p_2, \ldots, p_n$, the strategy is first to compute the relations corresponding to $p_1, p_2, \ldots, p_n$ and then to compute the relation corresponding to p. It is useful to keep track of the dependency among the predicates of a deductive database in a **predicate dependency graph.** Figure 24.17 shows the graph for the fact and rule predicates shown in Figures 24.14 and 24.15. The dependency graph contains a **node** for each predicate. Whenever a predicate A is specified in the body (RHS) of a rule, and the head (LHS) of that rule is the predicate B, we say that B **depends on** A, and we draw a directed edge from A to B. This indicates that, in order to compute the facts for the predicate B (the rule head), we must first compute the facts for all the predicates A in the rule body. If the dependency graph has no cycles, we call the rule set **nonrecursive.** If there is at least one cycle, the rule set is called **recursive.** In Figure 24.17, there is one recursively defined predicate—namely, superior—which has a recursive edge pointing back to itself. In addition, because the predicate subordinate depends on superior, it also requires recursion in computing its result.

A query that includes only nonrecursive predicates is called a **nonrecursive query.** In this section, we discuss only inference mechanisms for nonrecursive queries. In Figure 24.17, any query that does not involve the predicates subordinate or superior is nonrecursive. In the predicate dependency graph, the nodes corresponding to fact-defined

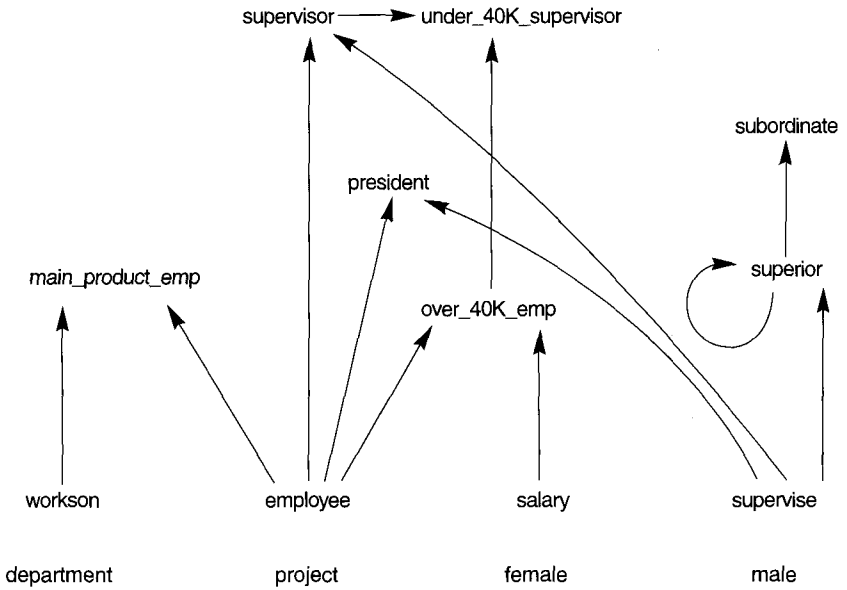


FIGURE 24.17 Predicate dependency graph for Figures 24.14 and 24.15.

predicates do not have any incoming edges, since all fact-defined predicates have their facts stored in a database relation. The contents of a fact-defined predicate can be computed by directly retrieving the tuples in the corresponding database relation.

The main function of an inference mechanism is to compute the facts that correspond to query predicates. This can be accomplished by generating a **relational expression** involving relational operators as SELECT, PROJECT, JOIN, UNION, and SET DIFFERENCE (with appropriate provision for dealing with safety issues) that, when executed, provides the query result. The query can then be executed by utilizing the internal query processing and optimization operations of a relational database management system. Whenever the inference mechanism needs to compute the fact set corresponding to a nonrecursive rule-defined predicate p, it first locates all the rules that have p as their head. The idea is to compute the fact set for each such rule and then to apply the UNION operation to the results, since UNION corresponds to a logical OR operation. The dependency graph indicates all predicates q on which each p depends, and since we assume that the predicate is nonrecursive, we can always determine a partial order among such predicates q. Before computing the fact set for p, we first compute the fact sets for all predicates q on which p depends, based on their partial order. For example, if a query involves the predicate `under_40K_supervisor`, we must first compute both supervisor and `over_40K_emp`. Since the latter two depend only on the fact-defined predicates `employee`, `salary`, and `supervise`, they can be computed directly from the stored database relations.

This concludes our introduction to deductive databases. Additional material may be found at the book Web site, where the complete Chapter 25 from the third edition is available. This includes a discussion on algorithms for recursive query processing.

24.5 SUMMARY

In this chapter, we introduced database concepts for some of the common features that are needed by advanced applications: active databases, temporal databases, and spatial and multimedia databases. It is important to note that each of these topics is very broad and warrants a complete textbook.

We first introduced the topic of active databases, which provide additional functionality for specifying active rules. We introduced the event-condition-action or ECA model for active databases. The rules can be automatically triggered by events that occur—such as a database update—and they can initiate certain actions that have been specified in the rule declaration if certain conditions are true. Many commercial packages already have some of the functionality provided by active databases in the form of triggers. We discussed the different options for specifying rules, such as row-level versus statement-level, before versus after, and immediate versus deferred. We gave examples of row-level triggers in the Oracle commercial system, and statement-level rules in the STARBURST experimental system. The syntax for triggers in the SQL-99 standard was also discussed. We briefly discussed some design issues and some possible applications for active databases.

We then introduced some of the concepts of temporal databases, which permit the database system to store a history of changes and allow users to query both current and past states of the database. We discussed how time is represented and distinguished between the valid time and transaction time dimensions. We then discussed how valid time, transaction time, and bitemporal relations can be implemented using tuple versioning in the relational model, with examples to illustrate how updates, inserts, and deletes are implemented. We also showed how complex objects can be used to implement temporal databases using attribute versioning. We then looked at some of the querying operations for temporal relational databases and gave a very brief introduction to the TSQL2 language.

We then turned to spatial and multimedia databases. Spatial databases provide concepts for databases that keep track of objects that have spatial characteristics, and they require models for representing these spatial characteristics and operators for comparing and manipulating them. Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures or drawings), video clips (such as movies, news reels, or home videos), audio clips (such as songs, phone messages, or speeches), and documents (such as books or articles). We gave a very brief overview of the various types of media sources and how multimedia sources may be indexed.

We concluded the chapter with an introduction to deductive databases and Datalog.

Review Questions

24.1. What are the differences between row-level and statement-level active rules?

24.2. What are the differences among immediate, deferred, and detached *consideration* of active rule conditions?

24.3. What are the differences among immediate, deferred, and detached *execution* of active rule actions?

24.4. Briefly discuss the consistency and termination problems when designing a set of active rules.
24.5. Discuss some applications of active databases.
24.6. Discuss how time is represented in temporal databases and compare the different time dimensions.
24.7. What are the differences between valid time, transaction time, and bitemporal relations?
24.8. Describe how the insert, delete, and update commands should be implemented on a valid time relation.
24.9. Describe how the insert, delete, and update commands should be implemented on a bitemporal relation.
24.10. Describe how the insert, delete, and update commands should be implemented on a transaction time relation.
24.11. What are the main differences between tuple versioning and attribute versioning?
24.12. How do spatial databases differ from regular databases?
24.13. What are the different types of multimedia sources?
24.14. How are multimedia sources indexed for content-based retrieval?

Exercises

24.15. Consider the COMPANY database described in Figure 5.6. Using the syntax of Oracle triggers, write active rules to do the following:
   a. Whenever an employee's project assignments are changed, check if the total hours per week spent on the employee's projects are less than 30 or greater than 40; if so, notify the employee's direct supervisor.
   b. Whenever an EMPLOYEE is deleted, delete the PROJECT tuples and DEPENDENT tuples related to that employee, and if the employee is managing a department or supervising any employees, set the MGRSSN for that department to null and set the SUPERSSN for those employees to null.
24.16. Repeat 24.15 but use the syntax of STARBURST active rules.
24.17. Consider the relational schema shown in Figure 24.18. Write active rules for keeping the SUM_COMMISSIONS attribute of SALES_PERSON equal to the sum of the COMMISSION attribute in SALES for each sales person. Your rules should also check if the

SALES

S_ID	V_ID	COMMISSION

SALES_PERSON

SALESPERSON_ID	NAME	TITLE	PHONE	SUM_COMMISSIONS

FIGURE 24.18 Database schema for sales and salesperson commissions in Exercise 24.17.

SUM_COMMISSIONS exceeds 100000; if it does, call a procedure NOTIFY_MANAGER(S_ID). Write both statement-level rules in STARBURST notation and row-level rules in Oracle.

24.18. Consider the UNIVERSITY EER schema of Figure 4.10. Write some rules (in English) that could be implemented via active rules to enforce some common integrity constraints that you think are relevant to this application.

24.19. Discuss which of the updates that created each of the tuples shown in Figure 24.9 were applied retroactively and which were applied proactively.

24.20. Show how the following updates, if applied in sequence, would change the contents of the bitemporal EMP_BT relation in Figure 24.9. For each update, state whether it is a retroactive or proactive update.

a. On 2004-03-10,17:30:00, the salary of NARAYAN is updated to 40000, effective on 2004-03-01.
b. On 2003-07-30,08:31:00, the salary of SMITH was corrected to show that it should have been entered as 31000 (instead of 30000 as shown), effective on 2003-06-01.
c. On 2004-03-18,08:31:00, the database was changed to indicate that NARAYAN was leaving the company (i.e., logically deleted) effective 2004-03-31.
d. On 2004-04-20,14:07:33, the database was changed to indicate the hiring of a new employee called JOHNSON, with the tuple <'JOHNSON', '334455667', 1, NULL> effective on 2004-04-20.
e. On 2004-04-28,12:54:02, the database was changed to indicate that WONG was leaving the company (i.e., logically deleted) effective 2004-06-01.
f. On 2004-05-05,13:07:33, the database was changed to indicate the rehiring of BROWN, with the same department and supervisor but with salary 35000 effective on 2004-05-01.

24.21. Show how the updates given in Exercise 24.20, if applied in sequence, would change the contents of the valid time EMP_VT relation in Figure 24.8.

24.22. Add the following facts to the example database in Figure 24.3:

```
supervise (ahmad,bob), supervise (franklin,gwen).
```

First modify the supervisory tree in Figure 24.1b to reflect this change. Then modify the diagram in Figure 24.4 showing the top-down evaluation of the query `superior(james,Y)`.

24.23. Consider the following set of facts for the relation `parent(X,Y)`, where `Y` is the parent of `X`:

```
parent(a,aa), parent(a,ab), parent(aa,aaa), parent(aa,aab),
parent(aaa,aaaa), parent(aaa,aaab).
```

Consider the rules

```
r1: ancestor(X,Y) :- parent(X,Y)
r2: ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)
```

which define ancestor Y of X as above.

a. Show how to solve the Datalog query

```
ancestor(aa,X)?
```

using the naive strategy. Show your work at each step.

b. Show the same query by computing only the changes in the ancestor relation and using that in rule 2 each time.

[This question is derived from Bancilhon and Ramakrishnan (1986).]

24.24. Consider a deductive database with the following rules:

```
ancestor(X,Y) :- father(X,Y)
ancestor(X,Y) :- father(X,Z), ancestor(Z,Y)
```

Notice that "`father(X,Y)`" means that `Y` is the father of `X`; "`ancestor(X,Y)`" means that `Y` is the ancestor of `X`. Consider the fact base

```
father(Harry,Issac), father(Issac,John), father(John,Kurt).
```

a. Construct a model theoretic interpretation of the above rules using the given facts.

b. Consider that a database contains the above relations `father(X,Y)`, another relation `brother(X,Y)`, and a third relation `birth(X,B)`, where `B` is the birthdate of person `X`. State a rule that computes the first cousins of the following variety: their fathers must be brothers.

c. Show a complete Datalog program with fact-based and rule-based literals that computes the following relation: list of pairs of cousins, where the first person is born after 1960 and the second after 1970. You may use "greater than" as a built-in predicate. (*Note:* Sample facts for brother, birth, and person must also be shown.)

24.25. Consider the following rules:

```
reachable(X,Y) :- flight(X,Y)
reachable(X,Y) :- flight(X,Z), reachable(Z,Y)
```

where `reachable(X,Y)` means that city `Y` can be reached from city `X`, and `flight(X,Y)` means that there is a flight to city `Y` from city `X`.

a. Construct fact predicates that describe the following:
   i. Los Angeles, New York, Chicago, Atlanta, Frankfurt, Paris, Singapore, Sydney are cities.
   ii. The following flights exist: LA to NY, NY to Atlanta, Atlanta to Frankfurt, Frankfurt to Atlanta, Frankfurt to Singapore, and Singapore to Sydney. (*Note:* No flight in reverse direction can be automatically assumed.)

b. Is the given data cyclic? If so, in what sense?

c. Construct a model theoretic interpretation (that is, an interpretation similar to the one shown in Figure 25.3) of the above facts and rules.

d. Consider the query

```
reachable(Atlanta,Sydney)?
```

How will this query be executed using naive and seminaive evaluation? List the series of steps it will go through.

e. Consider the following rule-defined predicates:

```
round-trip-reachable(X,Y) :- reachable(X,Y), reachable(Y,X)
duration(X,Y,Z)
```

Draw a predicate dependency graph for the above predicates. (*Note:* `duration(X,Y,Z)` means that you can take a flight from `X` to `Y` in `Z` hours.)

f. Consider the following query: What cities are reachable in 12 hours from Atlanta? Show how to express it in Datalog. Assume built-in predicates like `greater-than(X,Y)`. Can this be converted into a relational algebra statement in a straightforward way? Why or why not?

g. Consider the predicate `population(X,Y)` where `Y` is the population of city `X`. Consider the following query: List all possible bindings of the predicate `pair (X,Y)`, where `Y` is a city that can be reached in two flights from city `X`, which has over 1 million people. Show this query in Datalog. Draw a corresponding query tree in relational algebraic terms.

Selected Bibliography

The book by Zaniolo et al. (1997) consists of several parts, each describing an advanced database concept such as active, temporal, and spatial/text/multimedia databases. Widom and Ceri (1996) and Ceri and Fraternali (1997) focus on active database concepts and systems. Snodgrass et al. (1995) describe the TSQL2 language and data model. Khoshafian and Baker (1996), Faloutsos (1996), and Subrahmanian (1998) describe multimedia database concepts. Tansel et al. (1992) is a collection of chapters on temporal databases.

STARBURST rules are described in Widom and Finkelstein (1990). Early work on active databases includes the HiPAC project, discussed in Chakravarthy et al. (1989) and Chakravarthy (1990). A glossary for temporal databases is given in Jensen et al. (1994). Snodgrass (1987) focuses on TQuel, an early temporal query language.

Temporal normalization is defined in Navathe and Ahmed (1989). Paton (1999) and Paton and Diaz (1999) survey active databases. Chakravarthy et al. (1994) describe SENTINEL, and object-based active systems. Lee et al. (1998) discuss time series management.

The early developments of the logic and database approach are surveyed by Gallaire et al. (1984). Reiter (1984) provides a reconstruction of relational database theory, while Levesque (1984) provides a discussion of incomplete knowledge in light of logic. Gallaire and Minker (1978) provide an early book on this topic. A detailed treatment of logic and databases appears in Ullman (1989, vol. 2), and there is a related chapter in Volume 1 (1988). Ceri, Gottlob, and Tanca (1990) present a comprehensive yet concise treatment of logic and databases. Das (1992) is a comprehensive book on deductive databases and logic programming. The early history of Datalog is covered in Maier and Warren (1988). Clocksin and Mellish (1994) is an excellent reference on Prolog language.

Aho and Ullman (1979) provide an early algorithm for dealing with recursive queries, using the least fixed-point operator. Bancilhon and Ramakrishnan (1986) give an excellent and detailed description of the approaches to recursive query processing, with detailed examples of the naive and seminaive approaches. Excellent survey articles on

deductive databases and recursive query processing include Warren (1992) and Ramakrishnan and Ullman (1993). A complete description of the seminaive approach based on relational algebra is given in Bancilhon (1985). Other approaches to recursive query processing include the recursive query/subquery strategy of Vieille (1986), which is a top-down interpreted strategy, and the Henschen-Naqvi (1984) top-down compiled iterative strategy. Balbin and Rao (1987) discuss an extension of the seminaive differential approach for multiple predicates.

The original paper on magic sets is by Bancilhon et al. (1986). Beeri and Ramakrishnan (1987) extend it. Mumick et al. (1990) show the applicability of magic sets to nonrecursive nested SQL queries. Other approaches to optimizing rules without rewriting them appear in Vieille (1986, 1987). Kifer and Lozinskii (1986) propose a different technique. Bry (1990) discusses how the top-down and bottom-up approaches can be reconciled. Whang and Navathe (1992) describe an extended disjunctive normal form technique to deal with recursion in relational algebra expressions for providing an expert system interface over a relational DBMS.

Chang (1981) describes an early system for combining deductive rules with relational databases. The LDL system prototype is described in Chimenti et al. (1990). Krishnamurthy and Naqvi (1989) introduce the "choice" notion in LDL. Zaniolo (1988) discusses the language issues for the LDL system. A language overview of CORAL is provided in Ramakrishnan et al. (1992), and the implementation is described in Ramakrishnan et al. (1993). An extension to support object-oriented features, called CORAL++, is described in Srivastava et al. (1993). Ullman (1985) provides the basis for the NAIL! system, which is described in Morris et al. (1987). Phipps et al. (1991) describe the GLUE-NAIL! deductive database system.

Zaniolo (1990) reviews the theoretical background and the practical importance of deductive databases. Nicolas (1997) gives an excellent history of the developments leading up to DOODs. Falcone et al. (1997) survey the DOOD landscape. References on the VALIDITY system include Friesen et al. (1995), Vieille (1997), and Dietrich et al. (1999).

25

Distributed Databases and Client–Server Architectures

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. 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 technologies for accessing distributed heterogeneous data sources.

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 and networking (see the Selected Bibliography at the end of this chapter).

Section 25.1 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 25.2. Section 25.3 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 25.4 and 25.5 introduce distributed database query and transaction processing techniques, respectively. Section 25.6 discusses how the client-server architectural concepts are related to distributed databases. Section 25.7 elaborates on future issues in client-server architectures. Section 25.8 discusses distributed database features of the Oracle RDBMS.

For a short introduction to the topic, only sections 25.1, 25.3, and 25.6 may be covered.

25.1 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 interrelationships 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 2.5.

2. This definition and some of the discussion in this section are 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 Chapter 26. None of those qualifies as DDB by the definition given earlier.

25.1.1 Parallel Versus Distributed Technology

Turning our attention to parallel 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 25.1 contrasts these different architectures.

25.1.2 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 5.5 that we have been discussing throughout the

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

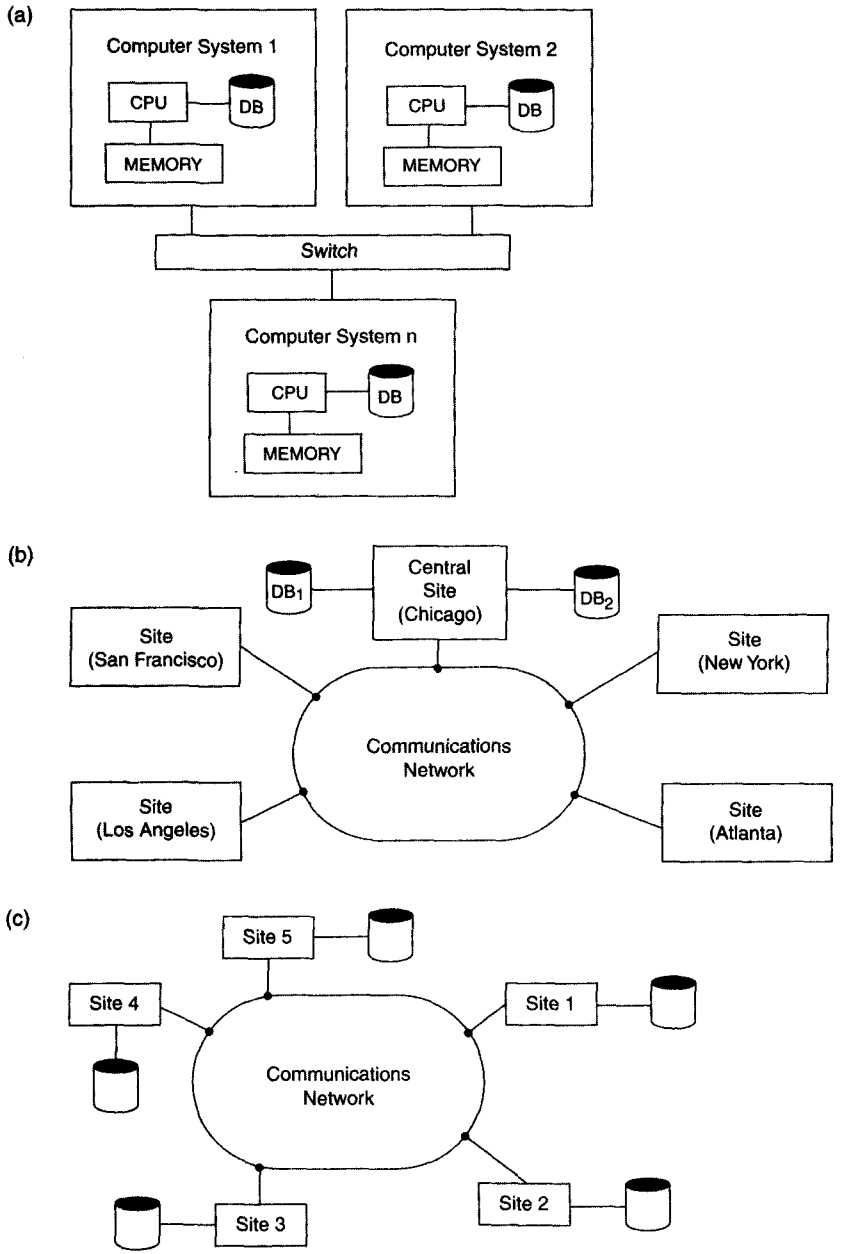


FIGURE 25.1 *Some different database system architectures. (a) Shared nothing* architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.

book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows, as we shall discuss in Section 25.2) and stored with possible replication as shown in Figure 25.2. 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 25.2, 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.

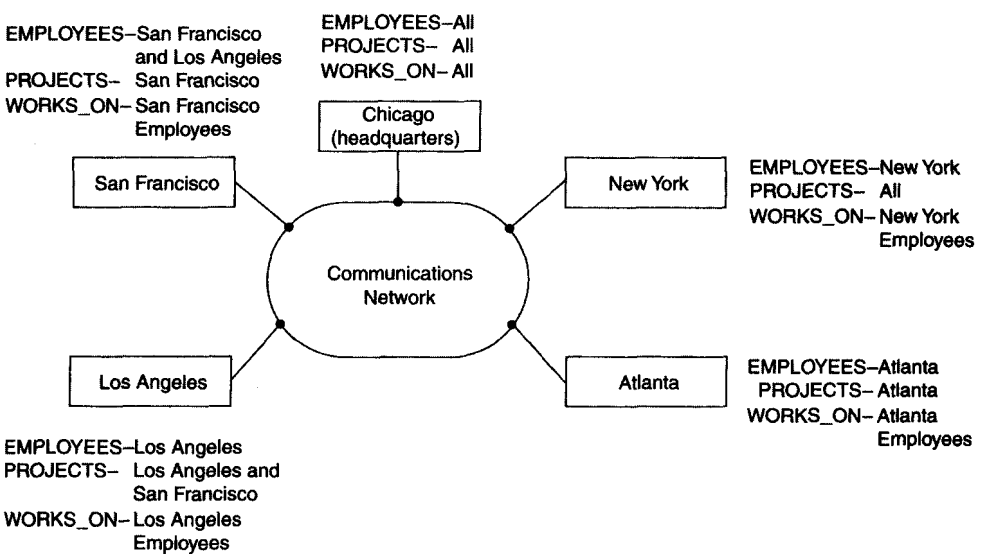


FIGURE 25.2 Data distribution and replication among distributed databases

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 **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.

25.1.3 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 25.1c.

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.

25.2 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.

25.2.1 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 5.5.

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 5.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 5.6, 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 5.6 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),

4. Of course, in an actual situation, there will be many more tuples in the relations than those shown in Figure 5.6.

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** 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 ≠ 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 ∪ L2 ∪ . . . ∪ Ln = ATTRS(R).
- Li ∩ Lj = PK(R) for any i ≠ 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 5.5 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 = TRUE (that is, all tuples are selected) and L ≠ ATTRS(R), we get a vertical fragment, and if C ≠ TRUE and L = ATTRS(R), we get a horizontal fragment. Finally, if C ≠ TRUE and L ≠ ATTRS(R), we get a mixed fragment. Notice that a relation can itself be considered a fragment with C = TRUE and L = 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.

25.2.2 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 25.5.

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.

25.2.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database of Figures 5.5 and 5.6. 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 5.6 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 5.5. We can then vertically fragment the resulting EMPLOYEE fragments to include only the attributes {NAME, SSN, SALARY, SUPERSSN, DNO}. Figure 25.3 shows the mixed fragments EMPD5 and EMPD4, which include the EMPLOYEE tuples satisfying the conditions DNO = 5 and DNO = 4,

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 25.3 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.

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 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 5.6. 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 25.4.

In Figure 25.4, 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 25.4. The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution of Figure 25.3, 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.

25.3 Types of Distributed Database Systems

The term distributed database management system can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous;** otherwise, it is called **heterogeneous.** Another factor related to the degree of homogeneity is the **degree of local autonomy.** If there is no provision for the local site to function as a stand-alone DBMS, then the system has **no local autonomy.** On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

At one extreme of the autonomy spectrum, we have a DDBMS that "looks like" a centralized DBMS to the user. A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS—which means that no local

autonomy exists. At the other extreme we encounter a type of DDBMS called a *federated DDBMS* (or a *multidatabase system*). In such a system, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA and hence has a very high degree of *local autonomy*. The term **federated database system (FDBS)** is used when there is some global view or schema of the federation of databases that is shared by the applications. On the other hand, a **multidatabase system** does not have a global schema and interactively constructs one as needed by the application. Both systems are hybrids between distributed and centralized systems and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense.

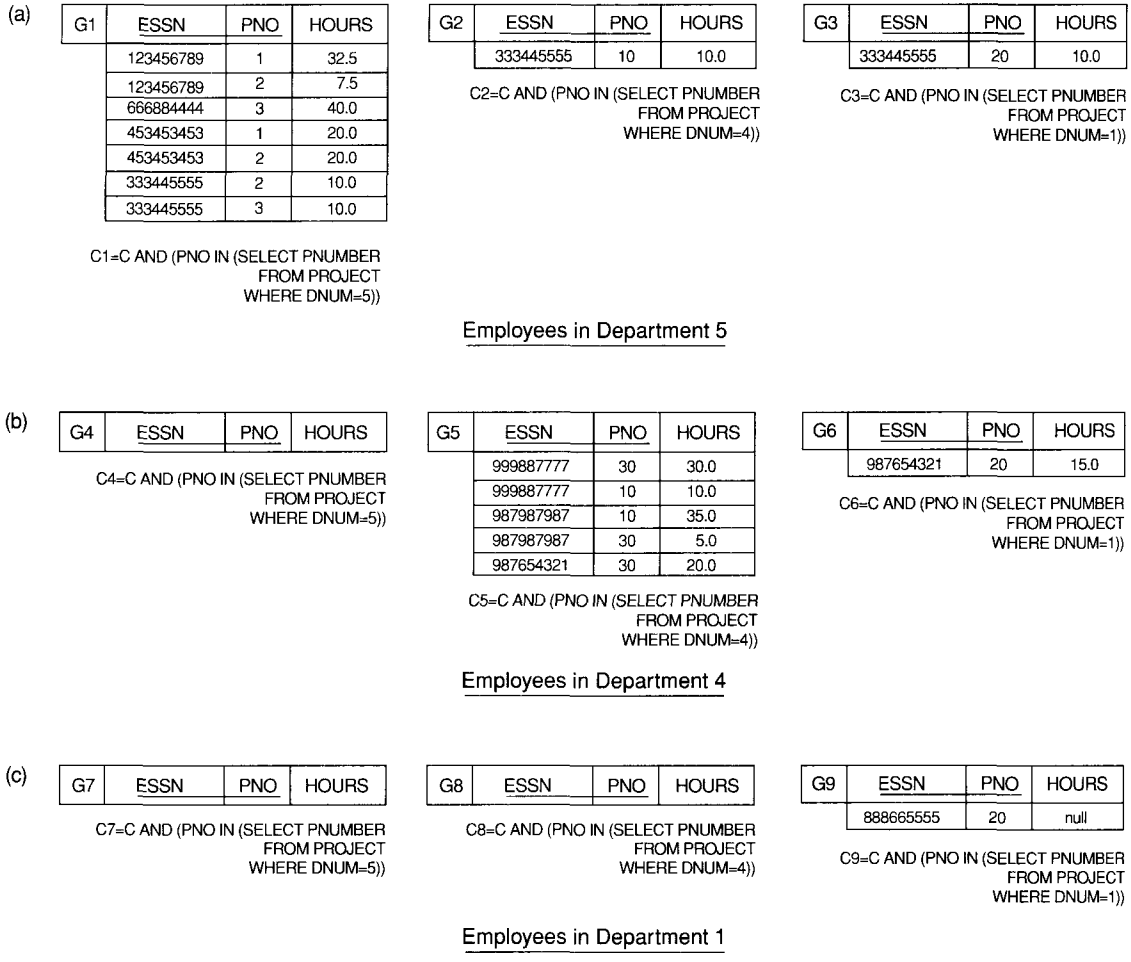


FIGURE 25.4 Complete and disjoint fragments of the WORKS_ON relation. (a) Fragments of WORKS_ON for employees working in department 5 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=5)]). (b) Fragments of WORKS_ON for employees working in department 4 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=4)]). (c) Fragments of WORKS_ON for employees working in department 1 (C=[ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=1)]).

In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS, and a third an object or hierarchical DBMS; in such a case it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server. We briefly discuss the issues affecting the design of FDBSs below.

Federated Database Management Systems Issues. The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FDBS.

- *Differences in data models:* Databases in an organization come from a variety of data models including the so-called legacy models (network and hierarchical, see Appendixes E and F), the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.
- *Differences in constraints:* Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.
- *Differences in query languages:* Even with the same data model, the languages and their versions vary. For example, SQL has multiple versions like SQL-89, SQL-92, and SQL-99, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

Semantic Heterogeneity. Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The **design autonomy** of component DBSs refers to their freedom of choosing the following design parameters, which in turn affect the eventual complexity of the FDBS:

- *The universe of discourse from which the data is drawn:* For example, two customer accounts, databases in the federation may be from United States and Japan with entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations would also present a problem. Hence, relations in these two databases which have identical names—CUSTOMER or ACCOUNT—may have some common and some entirely distinct information.
- *Representation and naming:* The representation and naming of data elements and the structure of the data model may be prespecified for each local database.

- *The understanding, meaning, and subjective interpretation of data.* This is a chief contributor to semantic heterogeneity.
- *Transaction and policy constraints:* These deal with serializability criteria, compensating transactions, and other transaction policies.
- *Derivation of summaries:* Aggregation, summarization, and other data-processing features and operations supported by the system.

Communication autonomy of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy** refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

A typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 25.5. In this architecture, the **local schema** is the conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generation of mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the schema for a user group or an application, as in the three-level schema architecture.[6]

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

25.4 Query Processing in Distributed Databases

We now give an overview of how a DDBMS processes and optimizes a query. We first discuss the communication costs of processing a distributed query; we then discuss a special operation, called a *semijoin*, that is used in optimizing some types of queries in a DDBMS.

6. For a detailed discussion of the autonomies and the five-level architecture of FDBMSs, see Sheth and Larson (1990).

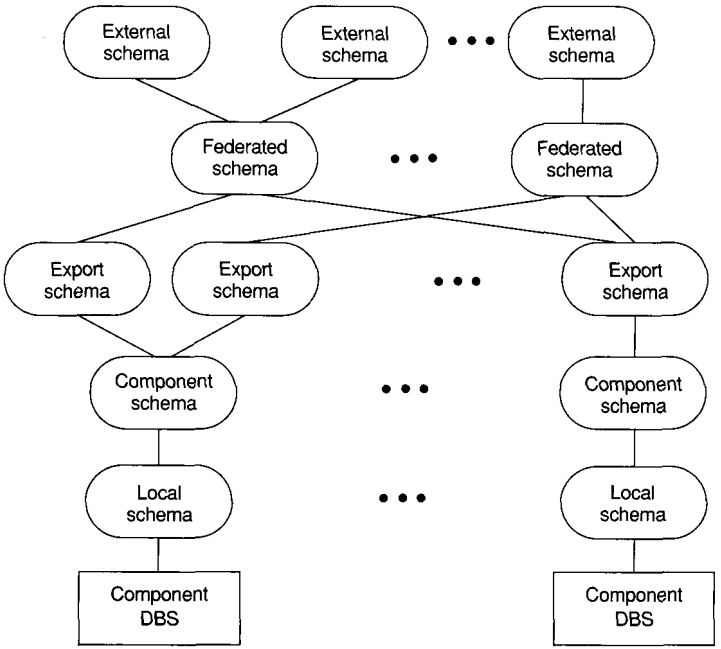


FIGURE 25.5 The five-level schema architecture in a federated database system (FDBS). *Source:* Adapted from Sheth and Larson, *Federated Database Systems for Managing Distributed Heterogeneous Autonomous Databases. ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

25.4.1 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 15. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become quite significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple example queries. Suppose that the EMPLOYEE and DEPARTMENT relations of Figure 5.5 are distributed as shown in Figure 25.6. We will assume in this example that neither relation is fragmented. According to Figure 25.6, the size of the EMPLOYEE relation is $100 * 10{,}000 = 10^6$ bytes, and the size of the DEPARTMENT relation is $35 * 100 = 3500$ bytes. Consider the query Q: "For each employee, retrieve the employee

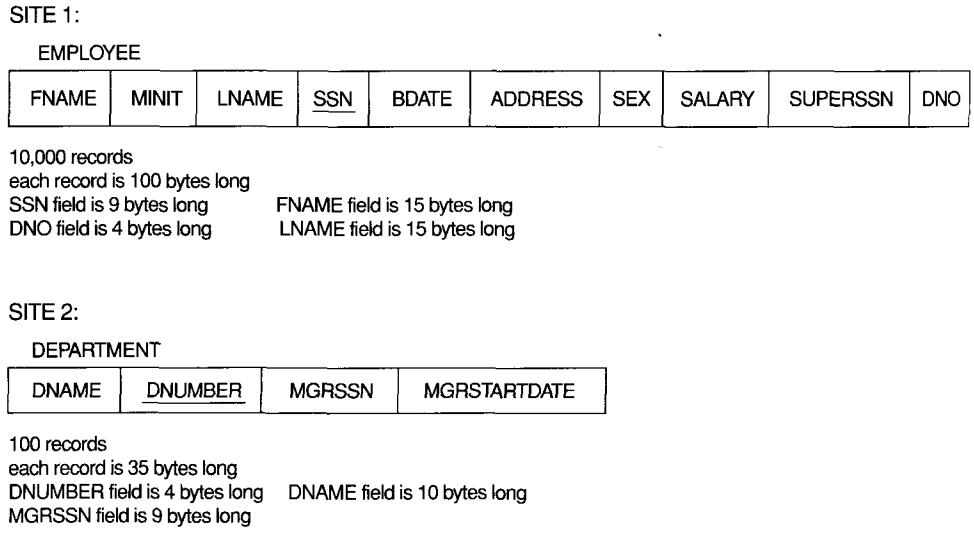


FIGURE 25.6 Example to illustrate volume of data transferred.

name and the name of the department for which the employee works." This can be stated as follows in the relational algebra:

$$Q: \pi_{FNAME, LNAME, DNAME}(EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long*. The query is submitted at a distinct site 3, which is called the **result site** because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case a total of 1,000,000 + 3500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case 400,000 + 3500 = 403,500 bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query Q′: "For each department, retrieve the department name and the name of the department manager." This can be stated as follows in the relational algebra:

$$Q': \pi_{FNAME, LNAME, DNAME}(DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE)$$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query Q apply to Q′, except that the result of Q′ includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case a total of 1,000,000 + 3500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 100 = 4000 bytes, so 4000 + 1,000,000 = 1,004,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case 4000 + 3500 = 7500 bytes must be transferred.

Again, we would choose strategy 3—in this case by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q′.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3500 = 403,500 bytes must be transferred for Q and 4000 + 3500 = 7500 bytes for Q′.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin.** We introduce this operation and discuss distributed execution using semijoins next.

25.4.2 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation R to the site where the other relation S is located; this column is then joined with S. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R. Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be quite an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q′:

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q, we transfer F = π_{DNUMBER}(DEPARTMENT), whose size is 4 * 100 = 400 bytes, whereas, for Q′, we transfer F′ = π_{MGRSSN}(DEPARTMENT), whose size is 9 * 100 = 900 bytes.

2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q, we transfer R = $\pi_{DNO,\ FNAME,\ LNAME}$(F $\bowtie_{DNUMBER=DNOEMPLOYEE}$), whose size is 34 * 10,000 = 340,000 bytes, whereas, for Q′, we transfer R′ = $\pi_{MGRSSN,\ FNAME,\ LNAME}$(F′ $\bowtie_{MGRSSN=SSN}$ EMPLOYEE), whose size is 39 * 100 = 3900 bytes.
3. Execute the query by joining the transferred file R or R′ with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4800 bytes for Q′. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q′ only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin operation** R $\ltimes_{A=B}$S, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression π_R(R$\bowtie_{A=B}$S). In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring F = π_B(S) to the site where R resides and then joining F with R, thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

R $\ltimes$ S $\neq$ S $\ltimes$ R

25.4.3 Query and Update Decomposition

In a DDBMS with *no distribution transparency*, the user phrases a query directly in terms of specific fragments. For example, consider another query Q: "Retrieve the names and hours per week for each employee who works on some project controlled by department 5," which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 25.3, and those at site 1 are shown in Figure 5.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS5 and WORKS_ON5 relations at site 2 (Figure 25.3) or the PROJECT and WORKS_ON relations at site 1 (Figure 5.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency*.

On the other hand, a DDBMS that supports *full distribution, fragmentation,* and *replication transparency* allows the user to specify a query or update request on the schema of Figure 5.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms to be discussed in Section 25.5. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. In addition, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS

catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard,** is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 5.6) are TRUE (all tuples), and the attribute lists are * (all attributes). For the fragments shown in Figure 25.3, we have the guard conditions and attribute lists shown in Figure 25.7. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple <'Alex', 'B', 'Coleman', '345671239', '22-APR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> would be decomposed by the DDBMS into two insert requests: the first inserts the preceding tuple in the EMPLOYEE fragment

(a) EMPD5
attribute list: FNAME,MINIT,LNAME,SSN,SALARY,SUPERSSN, DNO
guard condition: DNO=5
DEP5
attribute list: * (all attributes DNAME,DNUMBER,MGRSSN,MGRSTARTDATE)
guard condition: DNUMBER=5
DEP5_LOCS
attribute list: * (all attributes DNUMBER,LOCATION)
guard condition: DNUMBER=5
PROJS5
attribute list: * (all attributes PNAME,PNUMBER,PLOCATION,DNUM)
guard condition: DNUM=5
WORKS_ON5
attribute list: * (all attributes ESSN,PNO,HOURS)
guard condition: ESSN IN (π_{SSN} (EMPD5)) OR PNO IN ($\pi_{PNUMBER}$ (PROJS5))

EMPD4
(b) attribute list: FNAME,MINIT,LNAME,SSN,SALARY,SUPERSSN, DNO
guard condition: DNO=4
DEP4
attribute list: * (all attributes DNAME,DNUMBER,MGRSSN,MGRSTARTDATE)
guard condition: DNUMBER=4
DEP4_LOCS
attribute list: * (all attributes DNUMBER,LOCATION)
guard condition: DNUMBER=4
PROJS4
attribute list: * (all attributes PNAME,PNUMBER,PLOCATION,DNUM)
guard condition: DNUM=4
WORKS_ON4
attribute list: * (all attributes ESSN,PNO,HOURS)
guard condition: ESSN IN (π_{SSN} (EMPD4))
OR PNO IN ($\pi_{PNUMBER}$ (PROJS4))

FIGURE 25.7 Guard conditions and attributes lists for fragments. (a) Site 2 fragments. (b) Site 3 fragments.

at site 1, and the second inserts the projected tuple <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4> in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard conditions. For example, consider the query Q: "Retrieve the names and hours per week for each employee who works on some project controlled by department 5"; this can be specified in SQL on the schema of Figure 5.5 as follows:

```
Q:  SELECT  FNAME, LNAME, HOURS
    FROM    EMPLOYEE, PROJECT, WORKS_ON
    WHERE   DNUM=5 AND PNUMBER=PNO AND ESSN=SSN;
```

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (DNUM = 5 AND PNUMBER = PNO) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T1 \leftarrow \pi_{ESSN}(PROJS5 \bowtie_{PNUMBER=PNO} WORKS\_ON5)$$
$$T2 \leftarrow \pi_{ESSN,\ FNAME,\ LNAME}(T1 \bowtie_{ESSN=SSN} EMPLOYEE)$$
$$RESULT \leftarrow \pi_{FNAME,\ LNAME,\ HOURS}(T2 * WORKS\_ON5)$$

This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying (DNUM = 5) and that WORKS_ON5 contains all tuples to be joined with PROJS5; hence, subquery T1 can be executed at site 2, and the projected column ESSN can be sent to site 1. Subquery T2 can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

25.5 Overview of Concurrency Control and Recovery in Distributed Databases

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These include the following:

- *Dealing with* **multiple copies** *of the data items:* The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.
- *Failure of individual sites:* The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up to date with the rest of the sites before it rejoins the system.

- *Failure of communication links:* The system must be able to deal with failure of one or more of the communication links that connect the sites. An extreme case of this problem is that **network partitioning** may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.
- *Distributed commit:* Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The **two-phase commit protocol** (see Chapter 19) is often used to deal with this problem.
- *Distributed deadlock:* Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed concurrency control and recovery techniques must deal with these and other problems. In the following subsections, we review some of the techniques that have been suggested to deal with recovery and concurrency control in DDBMSs.

25.5.1 Distributed Concurrency Control Based on a Distinguished Copy of a Data Item

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques for centralized databases. We discuss these techniques in the context of extending centralized *locking*. Similar extensions apply to other concurrency control techniques. The idea is to designate *a particular copy* of each data item as a **distinguished copy.** The locks for this data item are associated *with the distinguished copy*, and all locking and unlocking requests are sent to the site that contains that copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site technique,** all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site.** Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item. We discuss these techniques next.

Primary Site Technique. In this method a single **primary site** is designated to be the **coordinator site** *for all database items*. Hence, all locks are kept at that site, and all requests for locking or unlocking are sent there. This method is thus an extension of the centralized locking approach. For example, if all transactions follow the two-phase locking protocol, serializability is guaranteed. The advantage of this approach is that it is a simple extension of the centralized approach and hence is not overly complex. However, it has certain inherent disadvantages. One is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. A second disadvantage is that failure of the primary site paralyzes the system, since all locking information is kept at that site. This can limit system reliability and availability.

Although all locks are accessed at the primary site, the items themselves can be accessed at any site at which they reside. For example, once a transaction obtains a READ_LOCK on a data item from the primary site, it can access any copy of that data item. However, once a transaction obtains a WRITE_LOCK and updates a data item, the DDBMS is responsible for updating *all copies* of the data item before releasing the lock.

Primary Site with Backup Site. This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site.** All locking information is maintained at both the primary and the backup sites. In case of primary site failure, the backup site takes over as primary site, and a new backup site is chosen. This simplifies the process of recovery from failure of the primary site, since the backup site takes over and processing can resume after a new backup site is chosen and the lock status information is copied to that site. It slows down the process of acquiring locks, however, because all lock requests and granting of locks must be recorded at *both the primary and the backup sites* before a response is sent to the requesting transaction. The problem of the primary and backup sites becoming overloaded with requests and slowing down the system remains undiminished.

Primary Copy Technique. This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items *stored at different sites*. Failure of one site affects any transactions that are accessing locks on items whose primary copies reside at that site, but other transactions are not affected. This method can also use backup sites to enhance reliability and availability.

Choosing a New Coordinator Site in Case of Failure. Whenever a coordinator site fails in any of the preceding techniques, the sites that are still running must choose a new coordinator. In the case of the primary site approach with *no* backup site, all executing transactions must be aborted and restarted in a tedious recovery process. Part of the recovery process involves choosing a new primary site and creating a lock manager process and a record of all lock information at that site. For methods that use backup sites, transaction processing is suspended while the backup site is designated as the new primary site and a new backup site is chosen and is sent copies of all the locking information from the new primary site.

If a backup site X is about to become the new primary site, X can choose the new backup site from among the system's running sites. However, if no backup site existed, or if both the primary and the backup sites are down, a process called **election** can be used to choose the new coordinator site. In this process, any site Y that attempts to communicate with the coordinator site repeatedly and fails to do so can assume that the coordinator is down and can start the election process by sending a message to all running sites proposing that Y become the new coordinator. As soon as Y receives a majority of yes votes, Y can declare that it is the new coordinator. The election algorithm itself is quite complex, but this is the main idea behind the election method. The algorithm also resolves any attempt by two or more sites to become coordinator at the same time. The references in the Selected Bibliography at the end of this chapter discuss the process in detail.

25.5.2 Distributed Concurrency Control Based on Voting

The concurrency control methods for replicated items discussed earlier all use the idea of a distinguished copy that maintains the locks for that item. In the **voting method,** there is no distinguished copy; rather, a lock request is sent to all sites that includes a copy of the data item. Each copy maintains its own lock and can grant or deny the request for it. If a transaction that requests a lock is granted that lock by *a majority* of the copies, it holds the lock and informs *all copies* that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain *time-out period,* it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved. Simulation studies have shown that voting has higher message traffic among sites than do the distinguished copy methods. If the algorithm takes into account possible site failures during the voting process, it becomes extremely complex.

25.5.3 Distributed Recovery

The recovery process in distributed databases is quite involved. We give only a very brief idea of some of the issues here. In some cases it is quite difficult even to determine whether a site is down without exchanging numerous messages with other sites. For example, suppose that site X sends a message to site Y and expects a response from Y but does not receive it. There are several possible explanations:

- The message was not delivered to Y because of communication failure.
- Site Y is down and could not respond.
- Site Y is running and sent a response, but the response was not delivered.

Without additional information or the sending of additional messages, it is difficult to determine what actually happened.

Another problem with distributed recovery is distributed commit. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on *every* site cannot be lost. This means that every site must first have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol, discussed in Section 19.6, is often used to ensure the correctness of distributed commit.

25.6 An Overview of 3-Tier Client-Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we discussed so far. Instead, distributed database applications are being developed in the context of the client-server architec-

tures. We already introduced the two-tier client-server architecture in Section 2.5. It is now more common to use a three-tier architecture, particular in Web applications. This architecture is illustrated in Figure 25.8.

In the three-tier client-server architecture, the following three layers exist:

1. Presentation layer (client): This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages used include HTML, JAVA, JavaScript, PERL, Visual Basic, and so on. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. Application layer (business logic): This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI or other database access techniques.
3. Database server: This layer handles query and update requests from the application layer, processes the requests, and send the results. Usually SQL is used to access the database if it is relational or object-relational and stored database pro-

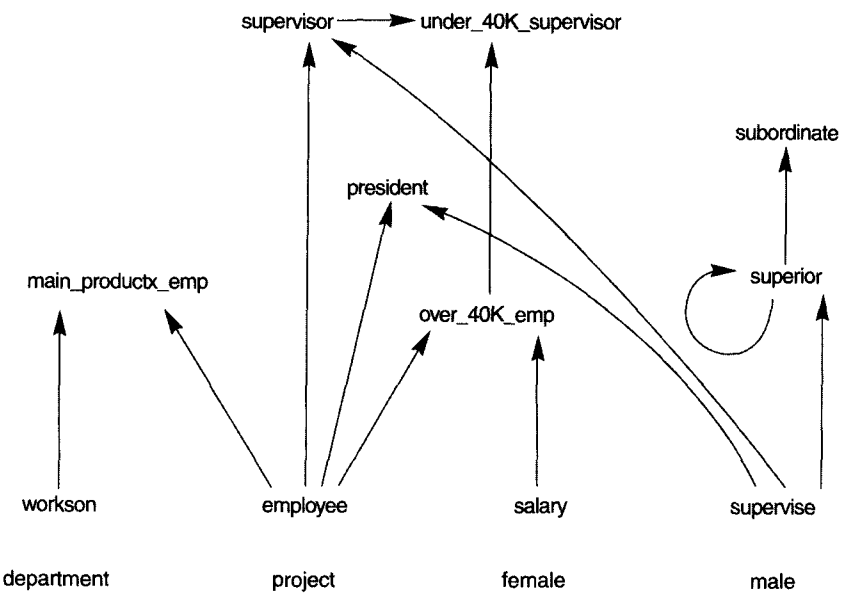


FIGURE 25.8 The three-tier client-server architecture.

cedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 26) when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality between client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an SQL server is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, SQL/CLI (see Chapter 9).

In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 26) so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail. We discussed distributed recovery and concurrency control in Section 25.5.

If the DDBMS has the capabilty to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency.** Some DDBMSs do not provide distribution transparency, instead requiring that applications be aware of the details of data distribution.

25.7 DISTRIBUTED DATABASES IN ORACLE

In the client-server architecture, the Oracle database system is divided into two parts: (1) a front-end as the client portion, and (2) a back-end as the server portion. The client portion is the front-end database application that interacts with the user. The client has no data access responsibility and merely handles the requesting, processing, and presentation of data managed by the server. The server portion runs Oracle and handles the functions related to concurrent shared access. It accepts SQL and PL/SQL statements originating from client applications, processes them, and sends the results back to the client. Oracle client-server applications provide location transparency by making location of data transparent to users; several features like views, synonyms, and procedures contribute to this. Global naming is achieved by using `<TABLENAME.@,DATABASENAME>` to refer to tables uniquely.

Oracle uses a two-phase commit protocol to deal with concurrent distributed transactions. The COMMIT statement triggers the two-phase commit mechanism. The RECO (recoverer) background process automatically resolves the outcome of those distributed transactions in which the commit was interrupted. The RECO of each local Oracle Server automatically commits or rolls back any "in-doubt" distributed transactions consistently on all involved nodes. For long-term failures, Oracle allows each local DBA to manually commit or roll back any in-doubt transactions and free up resources. Global consistency can be maintained by restoring the database at each site to a predetermined fixed point in the past.

Oracle's distributed database architecture is shown in Figure 25.9. A node in a distributed database system can act as a client, as a server, or both, depending on the situation. The figure shows two sites where databases called HQ (headquarters) and Sales are kept. For example, in the application shown running at the headquarters, for an SQL statement issued against local data (for example, `DELETE FROM DEPT ...`), the HQ computer acts as a server, whereas for a statement against remote data (for example, `INSERT INTO EMP@SALES`), the HQ computer acts as a client.

All Oracle databases in a distributed database system (DDBS) use Oracle's networking software Net8 for interdatabase communication. Net8 allows databases to communicate across networks to support remote and distributed transactions. It packages SQL statements into one of the many communication protocols to facilitate client to server communication and then packages the results back similarly to the client. Each database has a unique global name provided by a hierarchical arrangement of network domain names that is prefixed to the database name to make it unique.

Oracle supports database links that define a one-way communication path from one Oracle database to another. For example,

```
CREATE DATABASE LINK sales.us.americas;
```

establishes a connection to the sales database in Figure 25.9 under the network domain `us` that comes under domain `americas`.

Data in an Oracle DDBS can be replicated using snapshots or replicated master tables. Replication is provided at the following levels:

- *Basic replication:* Replicas of tables are managed for read-only access. For updates, data must be accessed at a single primary site.

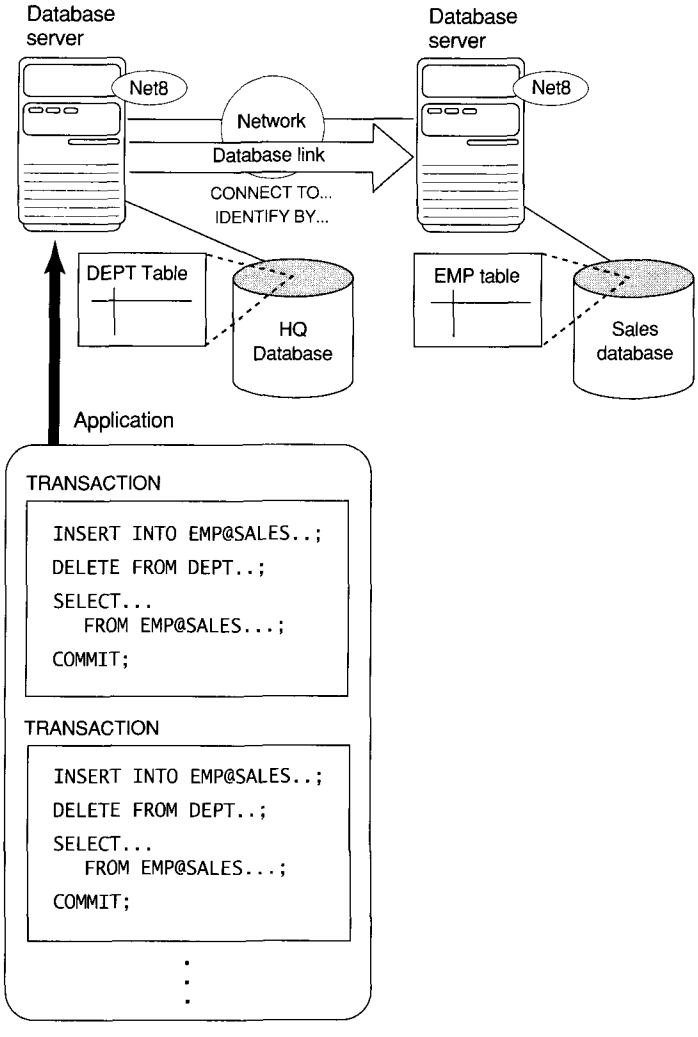


FIGURE 25.9 Oracle distributed database systems. *Source:* From Oracle (1997a). Copyright © Oracle Corporation 1997. All rights reserved.

- *Advanced (symmetric) replication:* This extends beyond basic replication by allowing applications to update table replicas throughout a replicated DDBS. Data can be read and updated at any site. This requires additional software called Oracle's advanced replication option. A **snapshot** generates a copy of a part of the table by means of a query called the *snapshot defining query*. A simple snapshot definition looks like this:

```
CREATE SNAPSHOT sales.orders AS
SELECT * FROM sales.orders@hq.us.americas;
```

Oracle groups snapshots into refresh groups. By specifying a refresh interval, the snapshot is automatically refreshed periodically at that interval by up to ten **Snapshot Refresh Processes (SNPs).** If the defining query of a snapshot contains a distinct or aggregate function, a GROUP BY or CONNECT BY clause, or join or set operations, the snapshot is termed a **complex snapshot** and requires additional processing. Oracle (up to version 7.3) also supports ROWID snapshots that are based on physical row identifiers of rows in the master table.

Heterogeneous Databases in Oracle. In a heterogeneous DDBS, at least one database is a non-Oracle system. **Oracle Open Gateways** provides access to a non-Oracle database from an Oracle server, which uses a database link to access data or to execute remote procedures in the non-Oracle system. The Open Gateways feature includes the following:

- *Distributed transactions:* Under the two-phase commit mechanism, transactions may span Oracle and non-Oracle systems.
- *Transparent SQL access:* SQL statements issued by an application are transparently transformed into SQL statements understood by the non-Oracle system.
- *Pass-through SQL and stored procedures:* An application can directly access a non-Oracle system using that system's version of SQL. Stored procedures in a non-Oracle SQL-based system are treated as if they were PL/SQL remote procedures.
- *Global query optimization:* Cardinality information, indexes, etc., at the non-Oracle system are accounted for by the Oracle Server query optimizer to perform global query optimization.
- *Procedural access:* Procedural systems like messaging or queuing systems are accessed by the Oracle server using PL/SQL remote procedure calls.

In addition to the above, data dictionary references are translated to make the non-Oracle data dictionary appear as a part of the Oracle Server's dictionary. Character set translations are done between national language character sets to connect multilingual databases.

25.8 SUMMARY

In this chapter we provided an introduction to distributed databases. This is a very broad topic, and we discussed only some of the basic techniques used with distributed databases. We first discussed the reasons for distribution and the potential advantages of distributed databases over centralized systems. We also defined the concept of distribution transparency and the related concepts of fragmentation transparency and replication transparency. We discussed the design issues related to data fragmentation, replication, and distribution, and we distinguished between horizontal and vertical fragments of relations. We discussed the use of data replication to improve system reliability and availability. We categorized DDBMSs by using criteria such as degree of homogeneity of software modules and degree of local autonomy. We dis-

cussed the issues of federated database management in some detail focusing on the needs of supporting various types of autonomies and dealing with semantic heterogeneity.

We illustrated some of the techniques used in distributed query processing, and discussed the cost of communication among sites, which is considered a major factor in distributed query optimization. We compared different techniques for executing joins and presented the semijoin technique for joining relations that reside on different sites. We briefly discussed the concurrency control and recovery techniques used in DDBMSs. We reviewed some of the additional problems that must be dealt with in a distributed environment that do not appear in a centralized environment.

We then discussed the client-server architecture concepts and related them to distributed databases, and we described some of the facilities in Oracle to support distributed databases.

Review Questions

25.1. What are the main reasons for and potential advantages of distributed databases?

25.2. What additional functions does a DDBMS have over a centralized DBMS?

25.3. What are the main software modules of a DDBMS? Discuss the main functions of each of these modules in the context of the client–server architecture.

25.4. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?

25.5. Why is data replication useful in DDBMSs? What typical units of data are replicated?

25.6. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?

25.7. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?

25.8. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?

25.9. Discuss what is meant by the following terms: *degree of homogeneity of a* DDBMS, *degree of local autonomy of a* DDBMS, *federated* DBMS, *distribution transparency*, *fragmentation transparency*, *replication transparency*, *multidatabase system*.

25.10. Discuss the naming problem in distributed databases.

25.11. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?

25.12. Discuss the semijoin method for executing an equijoin of two files located at different sites. Under what conditions is an equijoin strategy efficient?

25.13. Discuss the factors that affect query decomposition. How are guard conditions and attribute lists of fragments used during the query decomposition process?

25.14. How is the decomposition of an update request different from the decomposition of a query? How are guard conditions and attribute lists of fragments used during the decomposition of an update request?

25.15. Discuss the factors that do not appear in centralized systems that affect concurrency control and recovery in distributed systems.

25.16. Compare the primary site method with the primary copy method for distributed concurrency control. How does the use of backup sites affect each?
25.17. When are voting and elections used in distributed databases?
25.18. What are the software components in a client-server DDBMS? Compare the two-tier and three-tier client-server architectures.

Exercises

25.19. Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 25.3 and the fragments at site 1 are as shown in Figure 5.6. For each of the following queries, show at least two strategies of decomposing and executing the query. Under what conditions would each of your strategies work well?
   a. For each employee in department 5, retrieve the employee name and the names of the employee's dependents.
   b. Print the names of all employees who work in department 5 but who work on some project *not* controlled by department 5.

25.20. Consider the following relations:

```
BOOKS (Book#, Primary_author, Topic, Total_stock, $price)
BOOKSTORE (Store#, City, State, Zip, Inventory_value)
STOCK (Store#, Book#, Qty)
```

TOTAL_STOCK is the total number of books in stock, and INVENTORY_VALUE is the total inventory value for the store in dollars.

   a. Give an example of two simple predicates that would be meaningful for the BOOKSTORE relation for horizontal partitioning.
   b. How would a derived horizontal partitioning of STOCK be defined based on the partitioning of BOOKSTORE?
   c. Show predicates by which BOOKS may be horizontally partitioned by topic.
   d. Show how the STOCK may be further partitioned from the partitions in (b) by adding the predicates in (c).

25.21. Consider a distributed database for a bookstore chain called National Books with 3 sites called `EAST`, `MIDDLE`, and `WEST`. The relation schemas are given in question 24.20. Consider that BOOKS are fragmented by $PRICE amounts into:

B_1: `BOOK1`: up to $20.
B_2: `BOOK2`: from $20.01 to $50.
B_3: `BOOK3`: from $50.01 to $100.
B_4: `BOOK4`: $100.01 and above.

Similarly, BOOKSTORES are divided by `Zipcodes` into:

S_1: `EAST: Zipcodes` up to 35000.
S_2: `MIDDLE: Zipcodes` 35001 to 70000.
S_3: `WEST: Zipcodes` 70001 to 99999.

Assume that STOCK is a derived fragment based on BOOKSTORE only.

a. Consider the query:

```
SELECT Book#, Total_stock
FROM Books
WHERE $price > 15 and $price < 55;
```

Assume that fragments of BOOKSTORE are non-replicated and assigned based on region. Assume further that BOOKS are allocated as:

EAST: B_1, B_4
MIDDLE: B_1, B_2
WEST: B_1, B_2, B_3, B_4

Assuming the query was submitted in EAST, what remote subqueries does it generate? (write in SQL).

b. If the bookprice of BOOK#= 1234 is updated from $45 to $55 at site MIDDLE, what updates does that generate? Write in English and then in SQL.
c. Given an example query issued at WEST that will generate a subquery for MIDDLE.
d. Write a query involving selection and projection on the above relations and show two possible query trees that denote different ways of execution.

25.22. Consider that you have been asked to propose a database architecture in a large organization, General Motors, as an example, to consolidate all data including legacy databases (from Hierarchical and Network models, which are explained in Appendices C and D; no specific knowledge of these models is needed) as well as relational databases, which are geographically distributed so that global applications can be supported. Assume that alternative one is to keep all databases as they are, while alternative two is to first convert them to relational and then support the applications over a distributed integrated database.

a. Draw two schematic diagrams for the above alternatives showing the linkages among appropriate schemas. For alternative one, choose the approach of providing export schemas for each database and constructing unified schemas for each application.
b. List the steps one has to go through under each alternative from the present situation until global applications are viable.
c. Compare these from the issues of: (i) design time considerations, and (ii) runtime considerations.

Selected Bibliography

The textbooks by Ceri and Pelagatti (1984a) and Ozsu and Valduriez (1999) are devoted to distributed databases. Halsaal (1996), Tannenbaum (1996), and Stallings (1997) are textbooks on data communications and computer networks. Comer (1997) discusses networks and internets. Dewire (1993) is a textbook on client-server computing. Ozsu et al. (1994) has a collection of papers on distributed object management.

Distributed database design has been addressed in terms of horizontal and vertical fragmentation, allocation, and replication. Ceri et al. (1982) defined the concept of minterm horizontal fragments. Ceri et al. (1983) developed an integer programming based optimization model for horizontal fragmentation and allocation. Navathe et al. (1984) developed algorithms for vertical fragmentation based on attribute affinity and showed a variety of contexts for vertical fragment allocation. Wilson and Navathe (1986) present an analytical model for optimal allocation of fragments. Elmasri et al. (1987) discuss fragmentation for the ECR model; Karlapalem et al. (1994) discuss issues for distributed design of object databases. Navathe et al. (1996) discuss mixed fragmentation by combining horizontal and vertical fragmentation; Karlapalem et al. (1996) present a model for redesign of distributed databases.

Distributed query processing, optimization, and decomposition are discussed in Hevner and Yao (1979), Kerschberg et al. (1982), Apers et al. (1983), Ceri and Pelagatti (1984), and Bodorick et al. (1992). Bernstein and Goodman (1981) discuss the theory behind semijoin processing. Wong (1983) discusses the use of relationships in relation fragmentation. Concurrency control and recovery schemes are discussed in Bernstein and Goodman (1981a). Kumar and Hsu (1998) have some articles related to recovery in distributed databases. Elections in distributed systems are discussed in Garcia-Molina (1982). Lamport (1978) discusses problems with generating unique timestamps in a distributed system.

A concurrency control technique for replicated data that is based on voting is presented by Thomas (1979). Gifford (1979) proposes the use of weighted voting, and Paris (1986) describes a method called voting with witnesses. Jajodia and Mutchler (1990) discuss dynamic voting. A technique called *available copy* is proposed by Bernstein and Goodman (1984), and one that uses the idea of a group is presented in ElAbbadi and Toueg (1988). Other recent work that discusses replicated data includes Gladney (1989), Agrawal and ElAbbadi (1990), ElAbbadi and Toueg (1990), Kumar and Segev (1993), Mukkamala (1989), and Wolfson and Milo (1991). Bassiouni (1988) discusses optimistic protocols for DDB concurrency control. Garcia-Molina (1983) and Kumar and Stonebraker (1987) discuss techniques that use the semantics of the transactions. Distributed concurrency control techniques based on locking and distinguished copies are presented by Menasce et al. (1980) and Minoura and Wiederhold (1982). Obermark (1982) presents algorithms for distributed deadlock detection.

A survey of recovery techniques in distributed systems is given by Kohler (1981). Reed (1983) discusses atomic actions on distributed data. A book edited by Bhargava (1987) presents various approaches and techniques for concurrency and reliability in distributed systems.

Federated database systems were first defined in McLeod and Heimbigner (1985). Techniques for schema integration in federated databases are presented by Elmasri et al. (1986), Batini et al. (1986), Hayne and Ram (1990), and Motro (1987). Elmagarmid and Helal (1988) and Gamal-Eldin et al. (1988) discuss the update problem in heterogeneous DDBSs. Heterogeneous distributed database issues are discussed in Hsiao and Kamel (1989). Sheth and Larson (1990) present an exhaustive survey of federated database management.

Recently, multidatabase systems and interoperability have become important topics. Techniques for dealing with semantic incompatibilities among multiple databases are examined in DeMichiel (1989), Siegel and Madnick (1991), Krishnamurthy et al. (1991), and Wang and Madnick (1989). Castano et al. (1998) present an excellent survey of techniques for analysis of schemas. Pitoura et al. (1995) discuss object orientation in multidatabase systems.

Transaction processing in multidatabases is discussed in Mehrotra et al. (1992), Georgakopoulos et al. (1991), Elmagarmid et al. (1990), and Brietbart et al. (1990), among others. Elmagarmid et al. (1992) discuss transaction processing for advanced applications, including engineering applications discussed in Heiler et al. (1992).

The workflow systems, which are becoming popular to manage information in complex organizations, use multilevel and nested transactions in conjunction with distributed databases. Weikum (1991) discusses multilevel transaction management. Alonso et al. (1997) discuss limitations of current workflow systems.

A number of experimental distributed DBMSs have been implemented. These include distributed INGRES (Epstein et al., 1978), DDTS (Devor and Weeldreyer, 1980), SDD-1 (Rothnie et al., 1980), System R* (Lindsay et al., 1984), SIRIUS-DELTA (Ferrier and Stangret, 1982), and MULTIBASE (Smith et al., 1981). The OMNIBASE system (Rusinkiewicz et al., 1988) and the Federated Information Base developed using the Candide data model (Navathe et al., 1994) are examples of federated DDBMS. Pitoura et al. (1995) present a comparative survey of the federated database system prototypes. Most commercial DBMS vendors have products using the client-server approach and offer distributed versions of their systems. Some system issues concerning client-server DBMS architectures are discussed in Carey et al. (1991), DeWitt et al. (1990), and Wang and Rowe (1991). Khoshafian et al. (1992) discuss design issues for relational DBMSs in the client-server environment. Client-server management issues are discussed in many books, such as Zantinge and Adriaans (1996).