Q.1 a. Explain the role of concurrency control software in DBMS with an example.

Answer:

Concurrency control software in DBMS ensures that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.

b. Define participation constraints with respect to entity in DBMS.

Answer:

The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the minimum number of relationship instances that each entity can participate in.

c. Define Tuple, Attribute, Relation, and Domain in the context of relational model.

Answer:

In a relational model:

- **Tuple:** A row is called a tuple.
- **Attribute:** A column header is called an attribute.
- **Relation:** The table is called a relation.
- **Domains:** The data type describing the types of values that can appear in each column is represented by a domain of possible values.

d. Write a short note on Structured Query Language.

Answer:

Structured query language (SQL) is a database sublanguage that is used in querying, updating, and managing relational databases. It has statements for data definition, query, and update. Hence it is both DDL and DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It has also rules for embedding SQL statements into a general purpose programming language such as Java or COBOL.

e. Explain the three commands used to modify the database.

Answer:

The three commands used to modify the database are:

- **INSERT Command:** INSERT is used to add a single tuple to a relation. We must specify the relation name and the list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.
- **DELETE Command:** The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time.

- **UPDATE Command:** The UPDATE command is used to modify attribute values of one or more selected tuples. A WHERE clause in the UPDATE command selects the tuples to be modified from the single relation.

f. Explain the two levels at which the "goodness" of relation schemas can be measured.

Answer:

The two levels at which the "goodness" of relation schemas can be measured are:

- **Logical (or conceptual) level:** At this level, the quality of the design is measured on the basis of-how users interpret the relation schemas and the meaning of their attributes. Having good relation schema at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly.
- **Implementation (or storage) level:**At this level, the quality of the design is measured on the basis of-how the tuples in a base relation are stored and updated. This level applies only to the schemas of base relations- which will be physically stored as files.

g. Write a short note on Boyce-Codd normal form (BCNF). (7×4)

Answer:

BCNF is a stricter form than 3NF. Every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF. A relation R is in BCNF if whenever a nontrivial functional dependency

$X \rightarrow A$ holds in R, then X is a super key of R.

Q.2 a. Define database management system. What are the functions performed by a typical DBMS? (6)

Answer:

A database management system is a collection of programs that enables users to create and maintain a database. The DBMS is hence a general purpose software system that performs the following functions:

- **Defining a database:** It involves specifying the data types, structures and constraints for the data to be stored in the database.
- **Constructing the database:** It is the process of storing the data itself on some storage medium that is controlled by the DBMS.
- **Manipulating a database:** It includes such functions such as querying the database to retrieve specific data, updating the database to reflect changes in the mini world, and generating reports from the data.
- **Sharing a database:** It allows multiple users and programs to access the database concurrently.
- **Data Security & Integrity:** The DBMS contains functions which handle the security and integrity of data in the application. These can be easily invoked by the application and hence the application programmer need not code these functions in his/her programs.
- **Data Recovery & Concurrency:** Recovery of data after a system failure and concurrent access of records by multiple users are also handled by the DBMS.

b. Explain the three-schema architecture of database systems. (6)

Answer:

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in the database system. The goal of the three-schema architecture is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

- **Internal Level:** The internal level has internal schemas, which describes the physical storage structure of the database. The internal schema uses a physical model and describes the complete details of data storage and access paths for the database.
- **Conceptual Level:** The conceptual level has conceptual schema which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented.
- **External/View Level:** The external or view level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group.

c. Define Data manipulation Language (DML). Explain the different types of DMLs. (6)

Answer:

Once the database schemas are completed and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the data manipulation language (DML) for these purposes. The two main types of DMLs are:

- **High-level or Non-Procedural DML:** It can be used on its own to specify complex database operations in a concise manner. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In a later case, DML statements must be identified within the program so that they can be extracted by a pre-compiler and processed by the DBMS. High-level DMLs such as SQL can specify and retrieve many records in a single DML statement. A query in a high-level DML often specifies which data to retrieve rather than how to retrieve it.
- **Low-level or Procedural DBL:** It must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately.

Q.3 a. Define Entity Types, Entity Sets, and Value Sets. (6)

Answer:

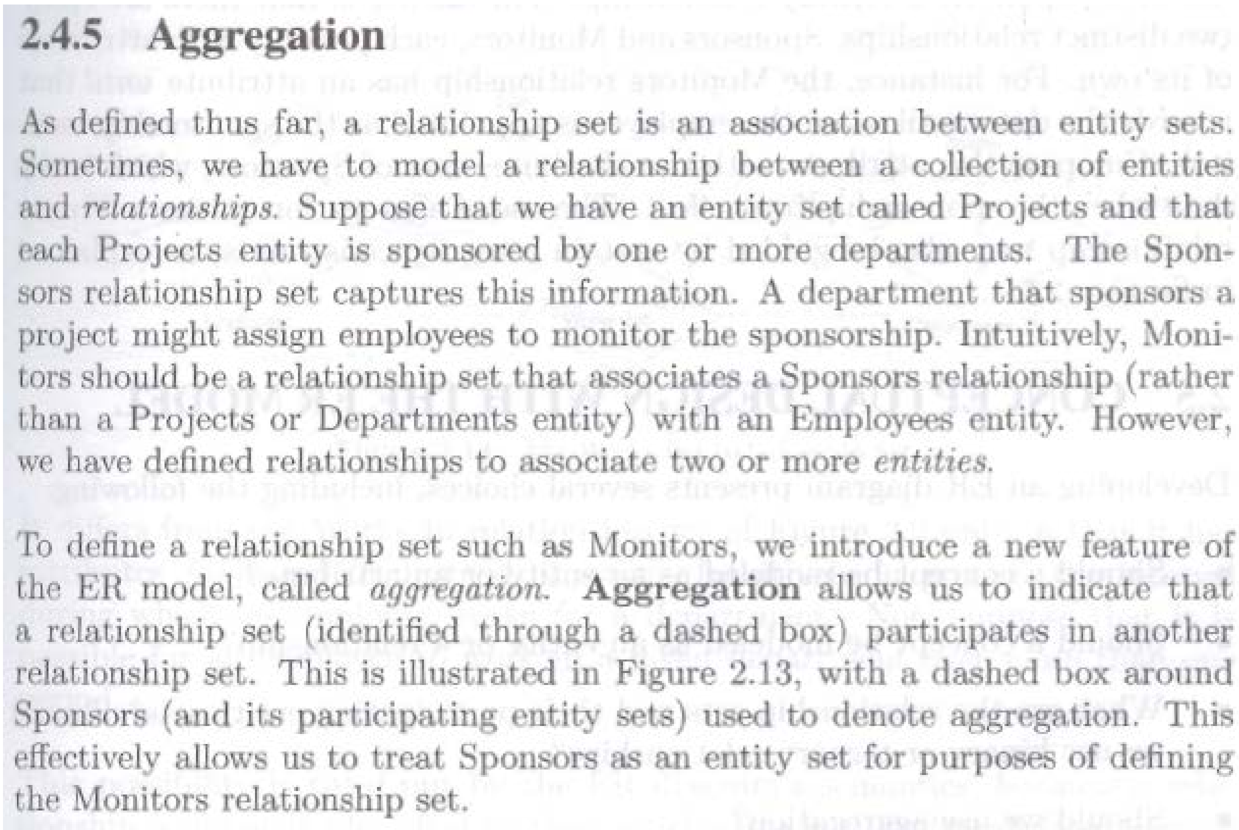
A database usually contains group of entities that are similar. These entities share the same attributes, but each entity has its own value(s) for each attribute.

- **Entity Type:** An entity type defines a collection (or set) of entities that have the same attributes.
- **Entity Set:** The collection of all entities of a particular entity type in the database at any point in time is called an entity set.
- **Value Set:** each simple attribute of an entity type is associated with a value set (or domain of values), which specifies the set of values that may be assigned to that attribute for each individual entity.

b. Explain weak entity and aggregation in ER Model. (6)

Answer:

Entity types that do not have key attributes of their own are called weak entity types. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. A weak entity type always has a total participation constraint (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity. A weak entity type normally has a partial key, which is the set of attributes that can uniquely identify weak entities that are related to the same owner entity.



2.4.5 Aggregation

As defined thus far, a relationship set is an association between entity sets. Sometimes, we have to model a relationship between a collection of entities and *relationships.* Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities.*

To define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation.* **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

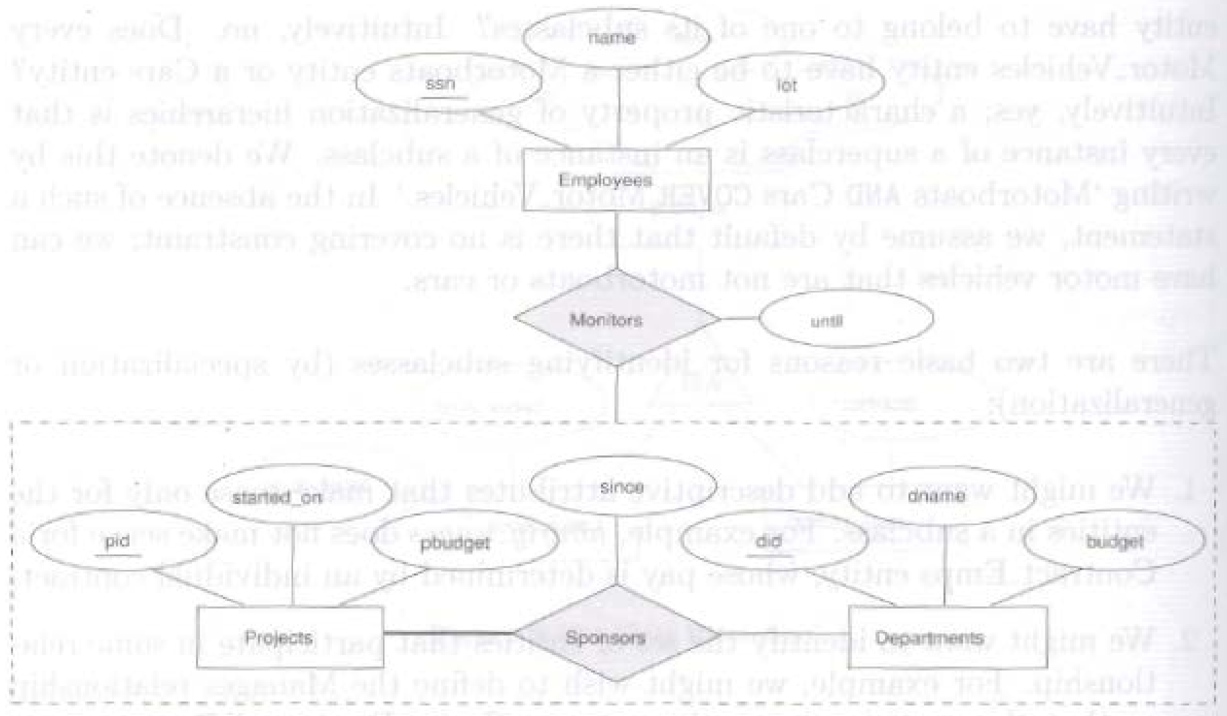


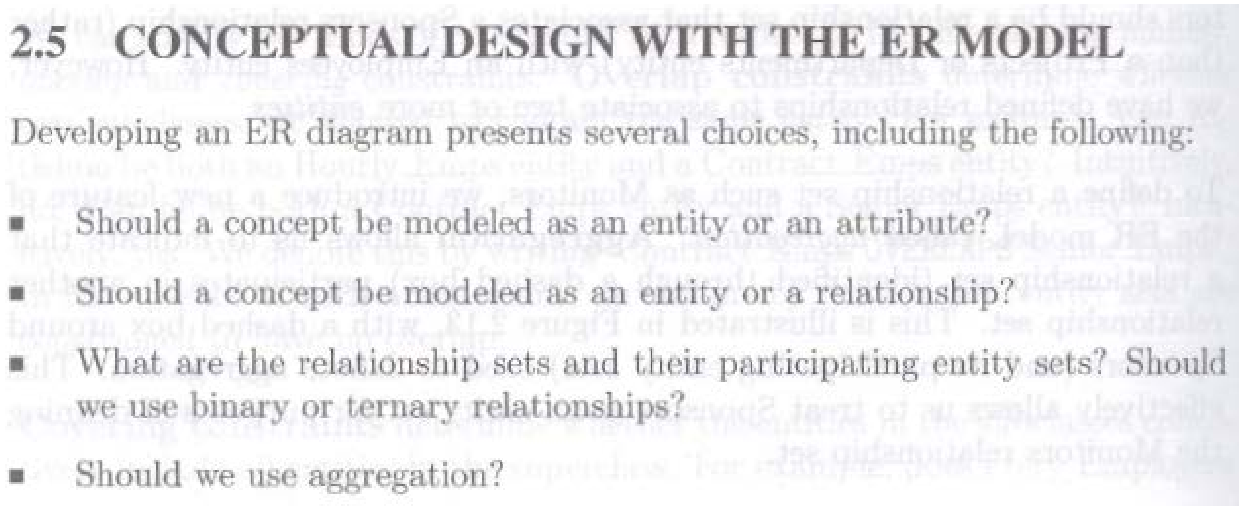
Figure 2.13 Aggregation

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can we not express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the Monitors relationship has an attribute *until* that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

c. Explain conceptual database design. Give an illustration. (6)

Answer:

In the process of database design, after collecting and analyzing all the requirements, the next step is to create a conceptual schema for the database by using a high-level conceptual data model. This step is called conceptual design. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with non-technical users. The high-level conceptual schema can also be used as a reference to ensure that all users' requirements are met and that the requirements do not conflict. This approach enables the database designers to concentrate on specifying the properties of the data without being concerned with the storage details.



2.5 CONCEPTUAL DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than an attribute, consider the relationship set (called Works_In4) shown in Figure 2.14.

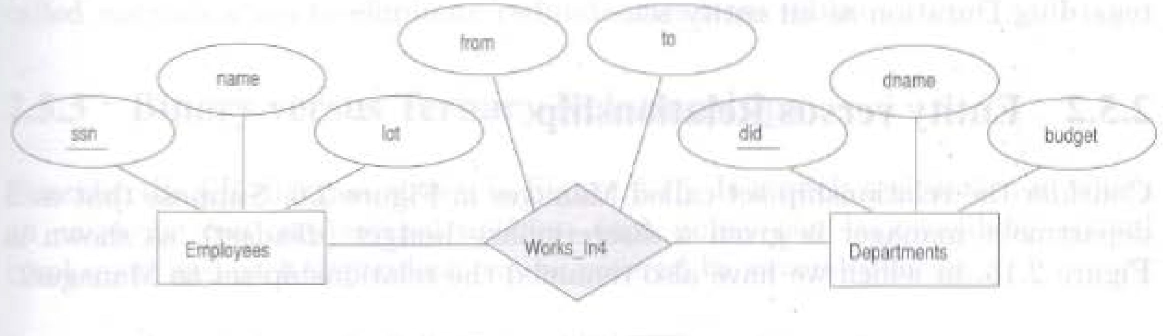


Figure 2.14 The Works_In4 Relationship Set

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics, because a relationship is uniquely identified by the participating entities (recall from Section

2.3). The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

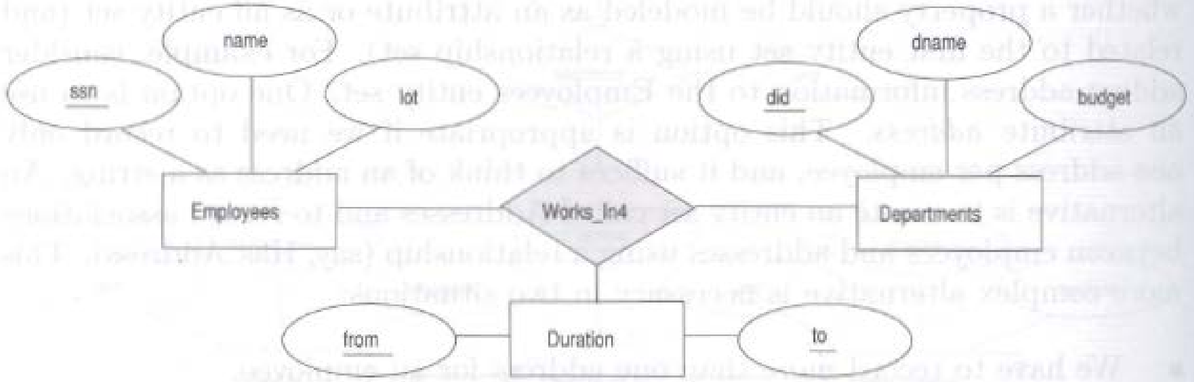


Figure 2.15 The Works_In4 Relationship Set

In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.

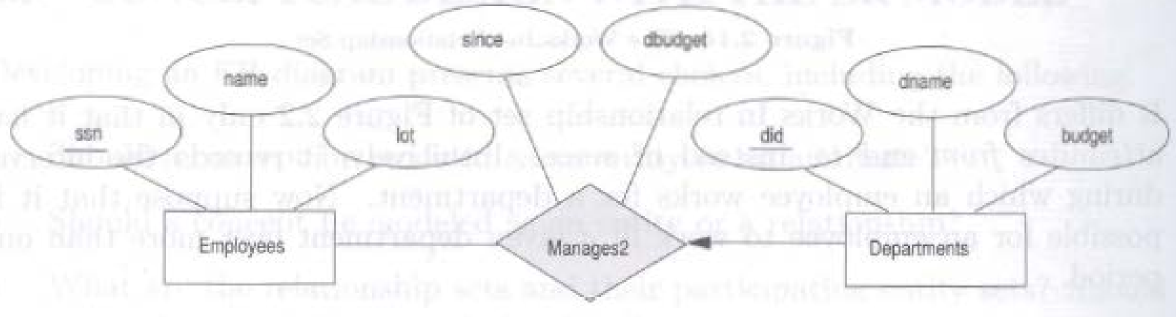


Figure 2.16 Entity versus Relationship

Given a department, we know the manager, as well as the manager's starting date and budget for that department. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case, each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field, leading to redundant storage of the same information. Another problem with this design is that it is misleading; it suggests that the budget is associated with the relationship, when it is actually associated with the manager.

We can address these problems by introducing a new entity set called Managers (which can be placed below Employees in an ISA hierarchy, to show that every manager is also an employee). The attributes *since* and *dbudget* now describe a manager entity, as intended. As a variation, while every manager has a budget, each manager may have a different starting date (as manager) for each department. In this case *dbudget* is an attribute of Managers, but *since* is an attribute of the relationship set between managers and departments.

The imprecise nature of ER modeling can thus make it difficult to recognize underlying entities, and we might associate attributes with relationships rather than the appropriate entities. In general, such mistakes lead to redundant storage of the same information and can cause many problems. We discuss redundancy and its attendant problems in Chapter 19, and present a technique called *normalization* to eliminate redundancies from tables.

2.5.3 Binary versus Ternary Relationships

Consider the ER diagram shown in Figure 2.17. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a

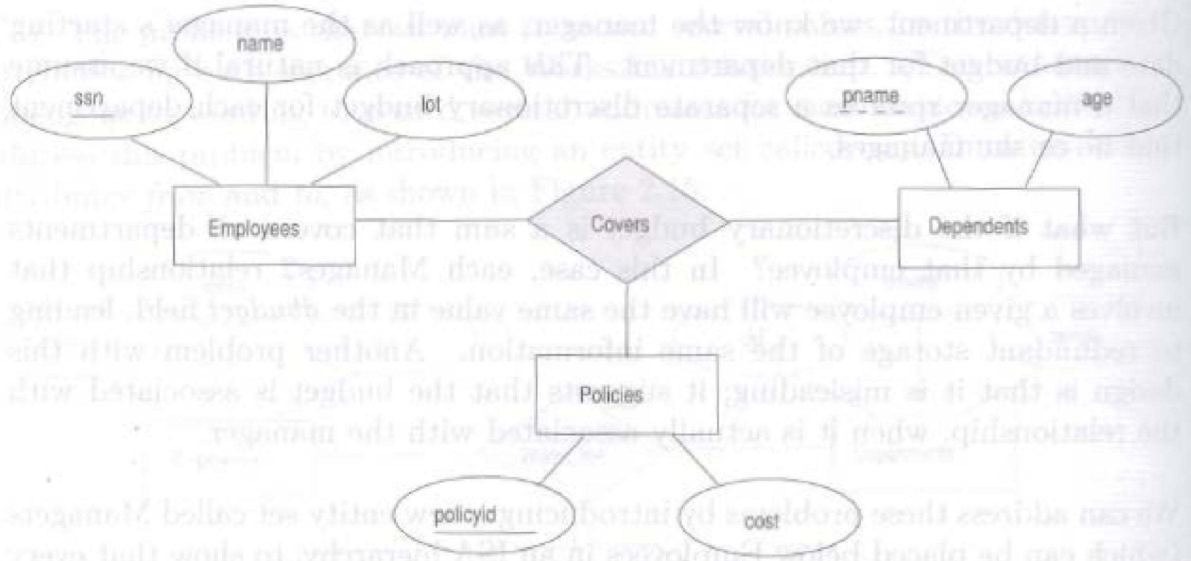


Figure 2.17 Policies as an Entity Set

policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third requirement, the best way to model this situation is to use two binary relationships, as shown in Figure 2.18.

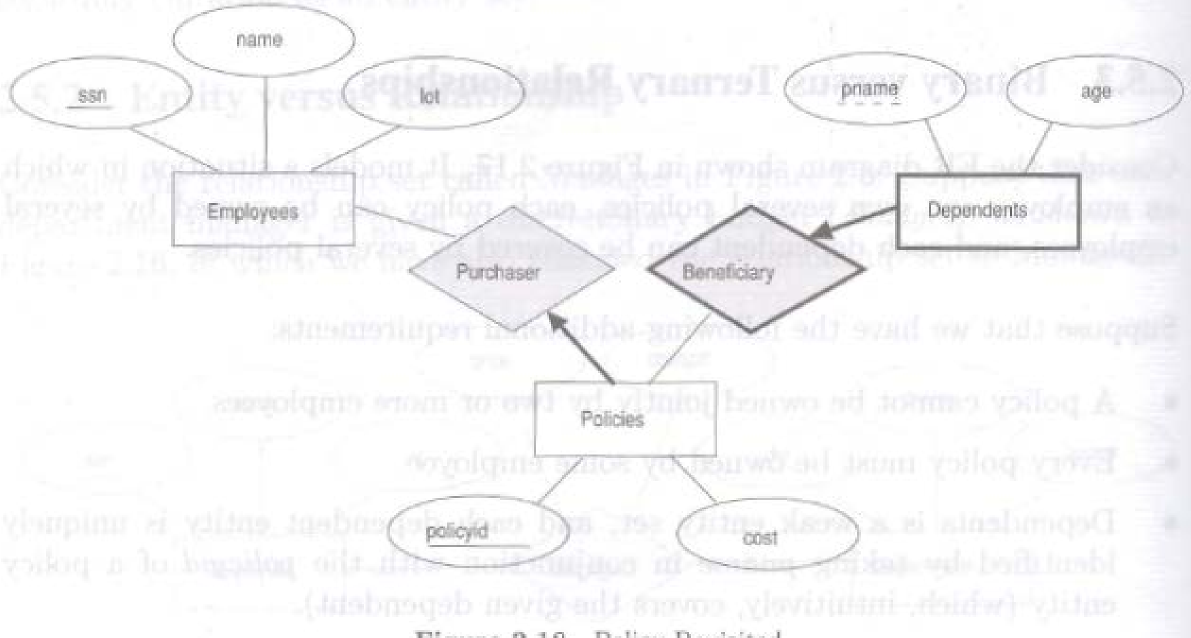
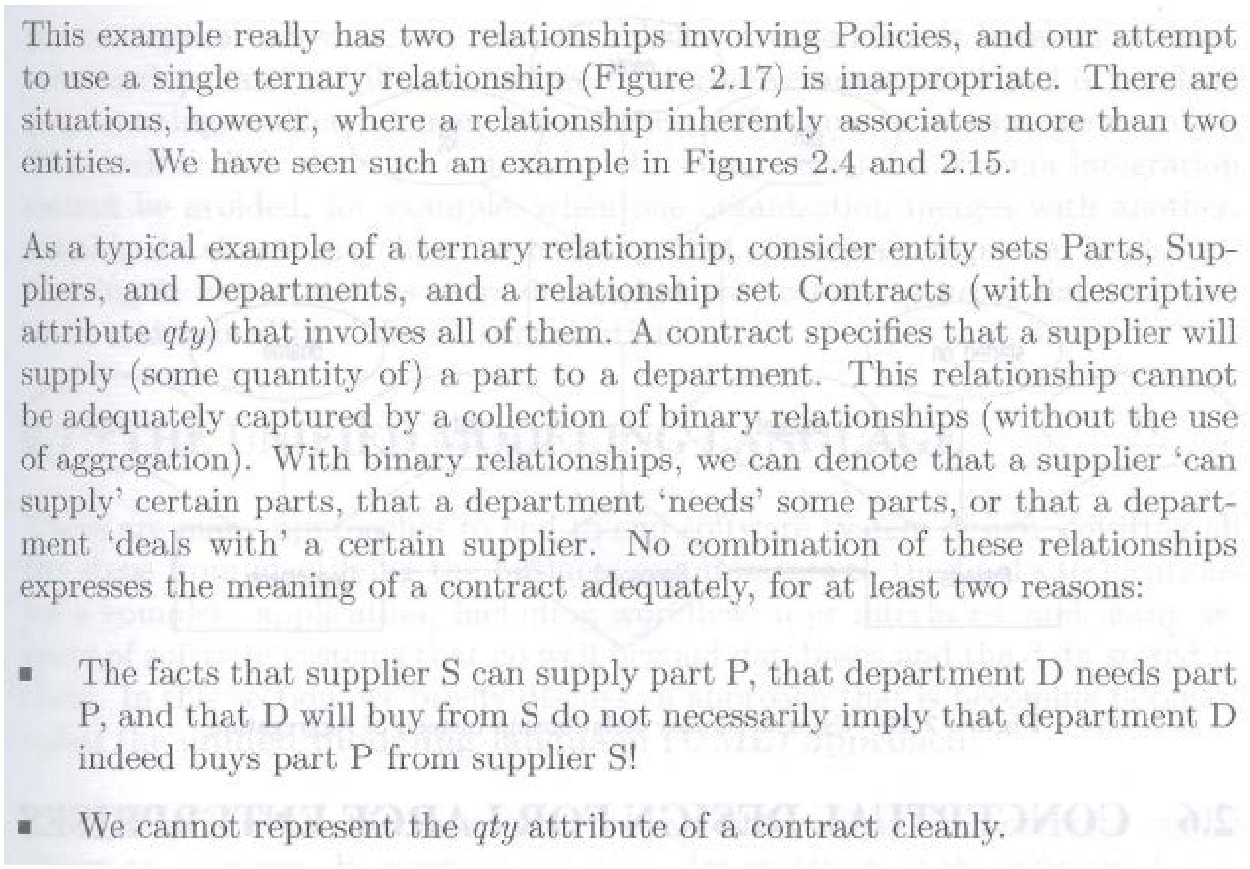


Figure 2.18 Policy Revisited



This example really has two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.17) is inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figures 2.4 and 2.15.

As a typical example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the *qty* attribute of a contract cleanly.

Q.4 a. Define candidate Key and primary Key. (6)

Answer:

- **Candidate Key:** A candidate key is a unique identifier for a tuple (row) within a relation (database table). The candidate key may be either simple (a single attribute) or composite (two or more attributes). By definition, every relation must have at least one candidate key, but it is possible for a relation to have more than one candidate key. If there is only one candidate key, it automatically becomes the primary key for the relation. If there are multiple candidate keys, the designer must designate one as the primary key.
- **Primary Key:** The key field that serves as the unique identifier of a specific tuple (row) in a relation (database table). It is common to designate one of the candidate keys as the primary key of the relation.

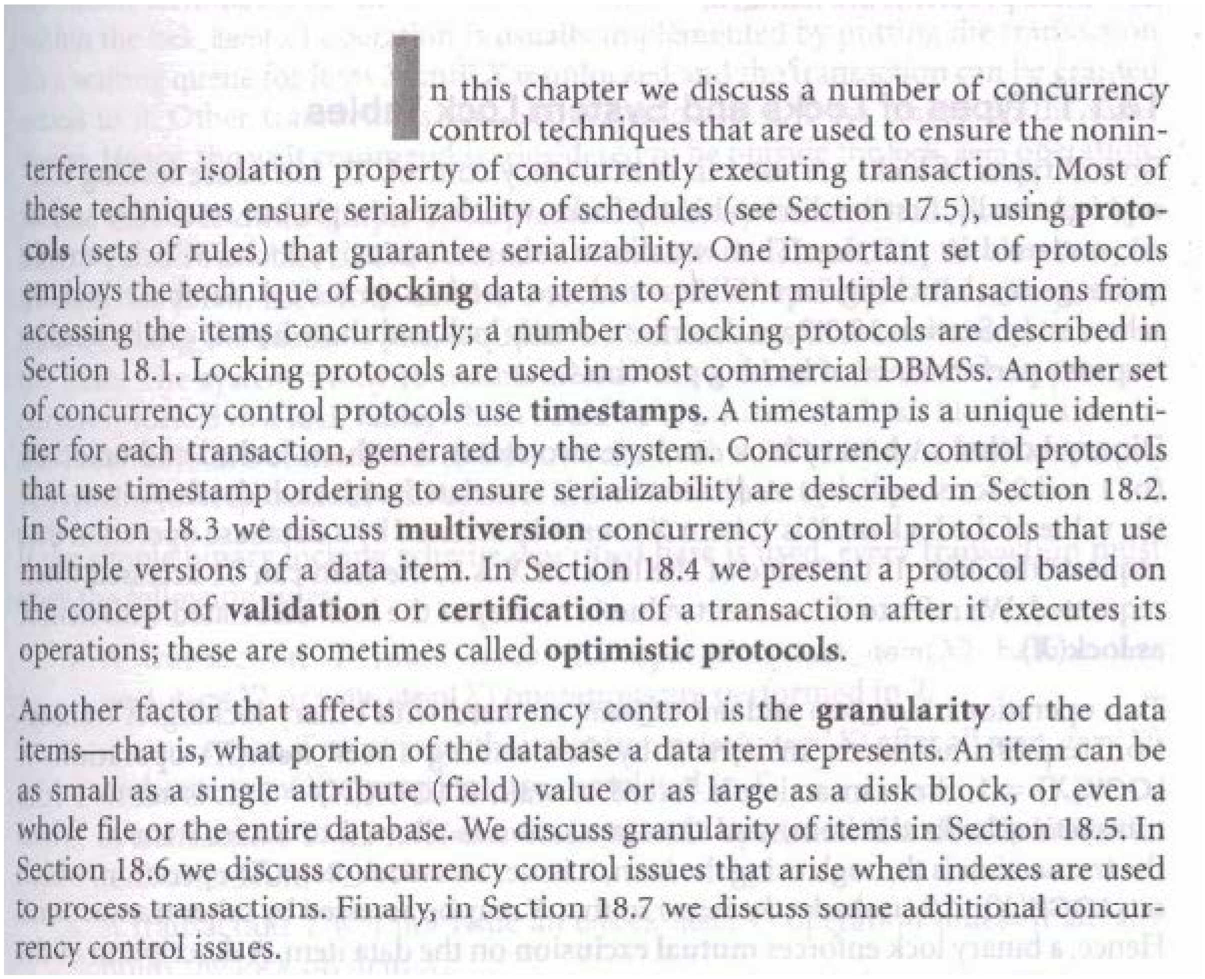
b. Explain Tuple Relational calculus. (6)

Answer:

Relational calculus is a query language for the relational model. In relational calculus, we write one declarative expression to specify a retrieval request, and hence there is no description of how to retrieve it. Therefore, relational calculus is considered to be a non-procedural language. A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated. The tuple relational calculus is based on specifying a number of tuple variables. Each tuple variable ranges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.

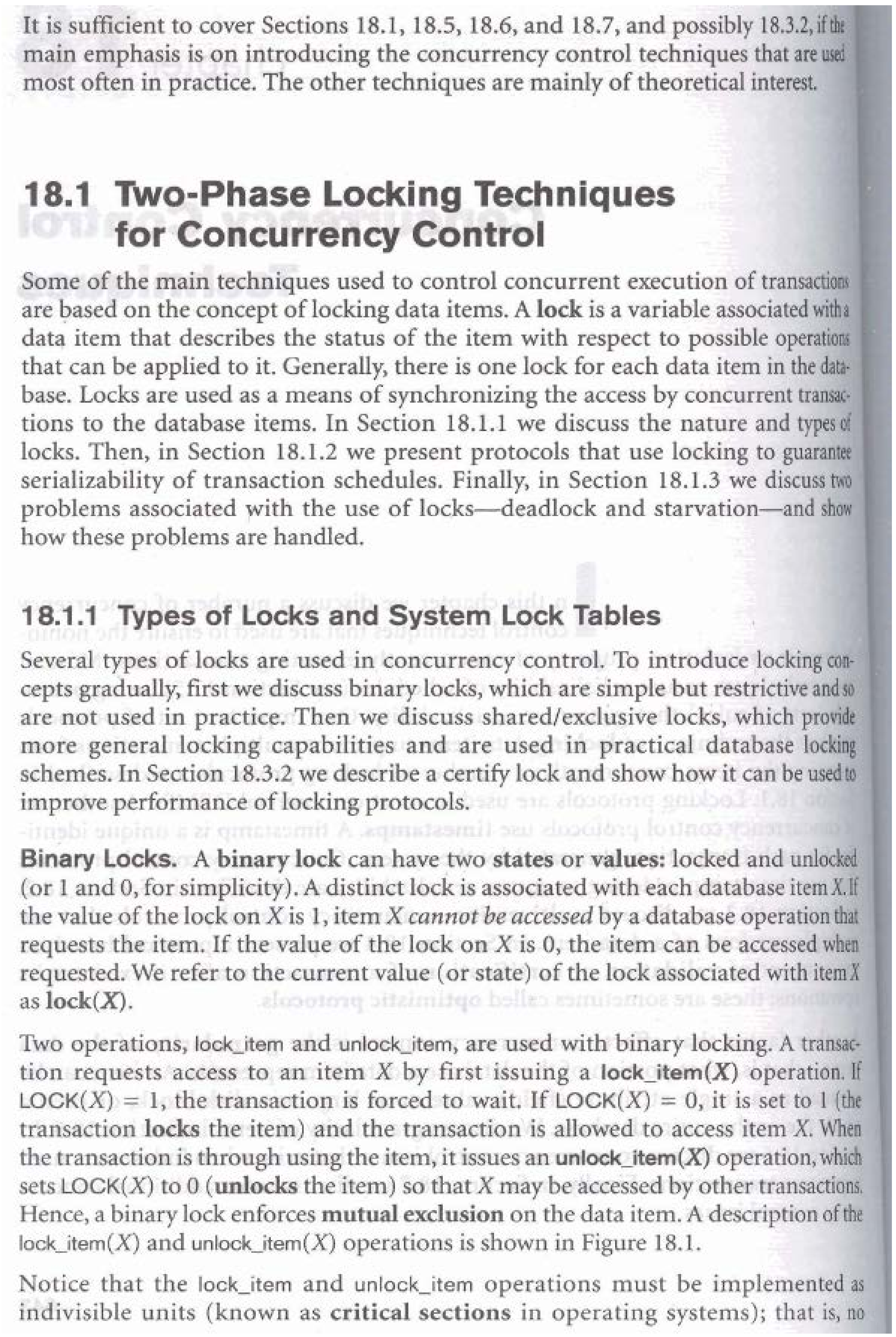
c. Explain concurrency control techniques used in DBMS. (6)

Answer:



In this chapter we discuss a number of concurrency control techniques that are used to ensure the noninterference or isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules (see Section 17.5), using **protocols** (sets of rules) that guarantee serializability. One important set of protocols employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols are described in Section 18.1. Locking protocols are used in most commercial DBMSs. Another set of concurrency control protocols use **timestamps**. A timestamp is a unique identifier for each transaction, generated by the system. Concurrency control protocols that use timestamp ordering to ensure serializability are described in Section 18.2. In Section 18.3 we discuss **multiversion** concurrency control protocols that use multiple versions of a data item. In Section 18.4 we present a protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols.**

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database. We discuss granularity of items in Section 18.5. In Section 18.6 we discuss concurrency control issues that arise when indexes are used to process transactions. Finally, in Section 18.7 we discuss some additional concurrency control issues.



It is sufficient to cover Sections 18.1, 18.5, 18.6, and 18.7, and possibly 18.3.2, if the main emphasis is on introducing the concurrency control techniques that are used most often in practice. The other techniques are mainly of theoretical interest.

18.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 18.1.1 we discuss the nature and types of locks. Then, in Section 18.1.2 we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 18.1.3 we discuss two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled.

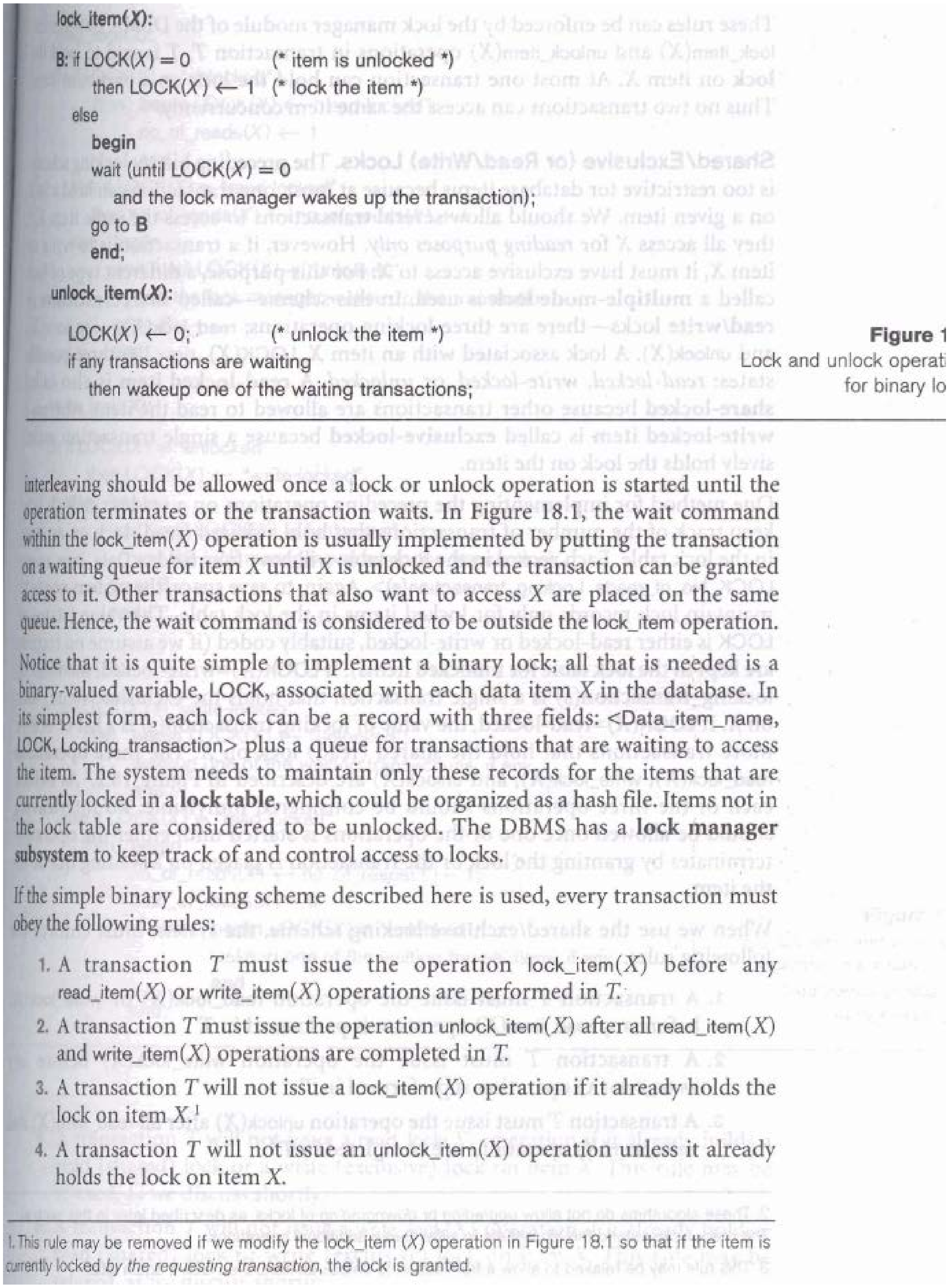
18.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple but restrictive and so are not used in practice. Then we discuss shared/exclusive locks, which provide more general locking capabilities and are used in practical database locking schemes. In Section 18.3.2 we describe a certify lock and show how it can be used to improve performance of locking protocols.

Binary Locks. A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested. We refer to the current value (or state) of the lock associated with item X as **lock(X)**.

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a **lock_item(X)** operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock_item(X)** operation, which sets LOCK(X) to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(X) and unlock_item(X) operations is shown in Figure 18.1.

Notice that the lock_item and unlock_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no



```
lock_item(X):
B: if LOCK(X) = 0          (* item is unlocked *)
       then LOCK(X) ← 1    (* lock the item *)
   else
       begin
       wait (until LOCK(X) = 0
           and the lock manager wakes up the transaction);
       go to B
       end;
unlock_item(X):
   LOCK(X) ← 0;            (* unlock the item *)
   if any transactions are waiting
       then wakeup one of the waiting transactions;
```

Figure 1
Lock and unlock operati
for binary lo

interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 18.1, the wait command within the lock_item(X) operation is usually implemented by putting the transaction on a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed on the same queue. Hence, the wait command is considered to be outside the lock_item operation.

Notice that it is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain only these records for the items that are currently locked in a **lock table**, which could be organized as a hash file. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager** subsystem to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.[1]
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

1. This rule may be removed if we modify the lock_item (X) operation in Figure 18.1 so that if the item is currently locked *by the requesting transaction*, the lock is granted.

These rules can be enforced by the lock manager module of the DBMS. Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to **hold the lock** on item X. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for *reading purposes only*. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

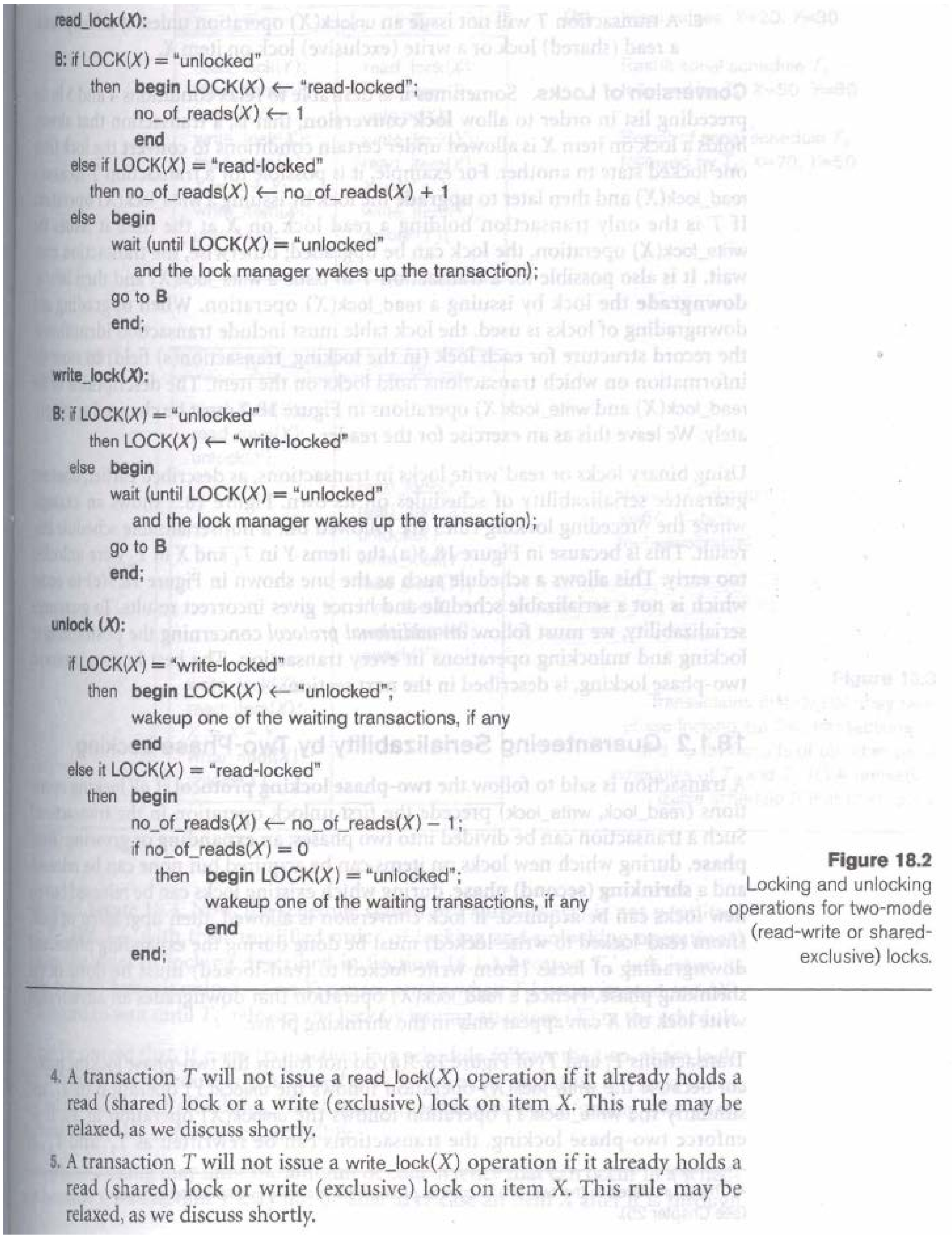
One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have four fields:<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(X)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. The three operations read_lock(X), write_lock(X), and unlock(X) are described in Figure 18.2.[2] As before, each of the three operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed on a waiting queue for the item.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.[3]

2. These algorithms do not allow *upgrading* or *downgrading* of locks, as described later in this section. The reader can extend the algorithms to allow these additional operations.

3. This rule may be relaxed to allow a transaction to unlock an item, then lock it again later.



```
read_lock(X):
B: if LOCK(X) = "unlocked"
      then begin LOCK(X) ← "read-locked";
              no_of_reads(X) ← 1
              end
   else if LOCK(X) = "read-locked"
      then no_of_reads(X) ← no_of_reads(X) + 1
   else begin
         wait (until LOCK(X) = "unlocked"
             and the lock manager wakes up the transaction);
         go to B
         end;

write_lock(X):
B: if LOCK(X) = "unlocked"
      then LOCK(X) ← "write-locked"
   else begin
         wait (until LOCK(X) = "unlocked"
             and the lock manager wakes up the transaction);
         go to B
         end;

unlock (X):
   if LOCK(X) = "write-locked"
      then begin LOCK(X) ← "unlocked";
             wakeup one of the waiting transactions, if any
             end
   else it LOCK(X) = "read-locked"
      then begin
             no_of_reads(X) ← no_of_reads(X) − 1;
             if no_of_reads(X) = 0
                then begin LOCK(X) = "unlocked";
                       wakeup one of the waiting transactions, if any
                       end
             end;
```

Figure 18.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.

6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read_lock(X) and write_lock(X) operations in Figure 18.2 must be changed appropriately. We leave this as an exercise for the reader.

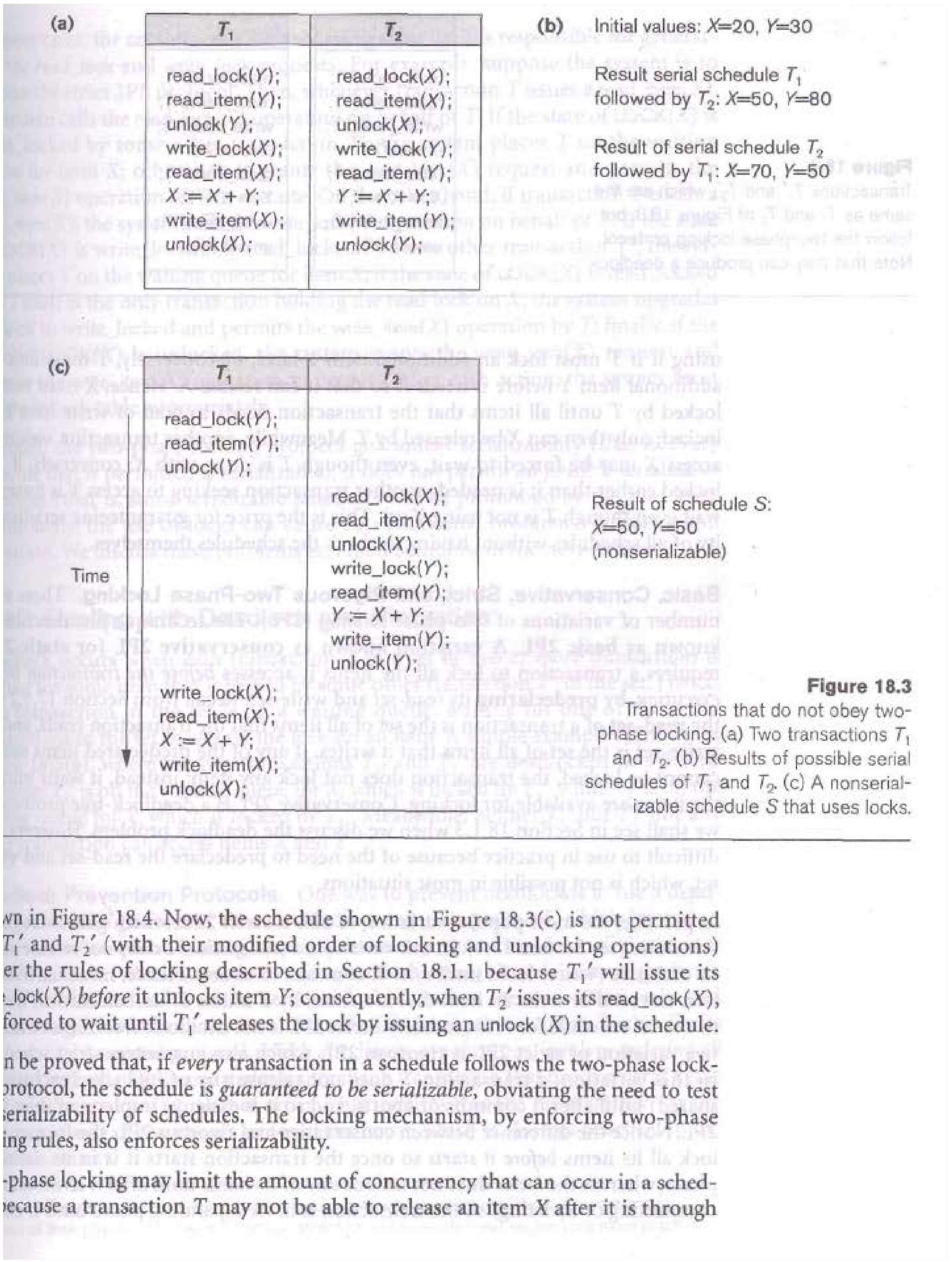
Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 18.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 18.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 18.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best known protocol, two-phase locking, is described in the next section.

18.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction.[4] Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Transactions T_1 and T_2 of Figure 18.3(a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2', as

4. This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 25).



(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Time

Result of schedule S: X=50, Y=50 (nonserializable)

Figure 18.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

wn in Figure 18.4. Now, the schedule shown in Figure 18.3(c) is not permitted T_1' and T_2' (with their modified order of locking and unlocking operations) er the rules of locking described in Section 18.1.1 because T_1' will issue its _lock(X) *before* it unlocks item Y; consequently, when T_2' issues its read_lock(X), forced to wait until T_1' releases the lock by issuing an unlock (X) in the schedule.

n be proved that, if *every* transaction in a schedule follows the two-phase lock-protocol, the schedule is *guaranteed to be serializable*, obviating the need to test serializability of schedules. The locking mechanism, by enforcing two-phase ing rules, also enforces serializability.

-phase locking may limit the amount of concurrency that can occur in a sched-because a transaction T may not be able to release an item X after it is through

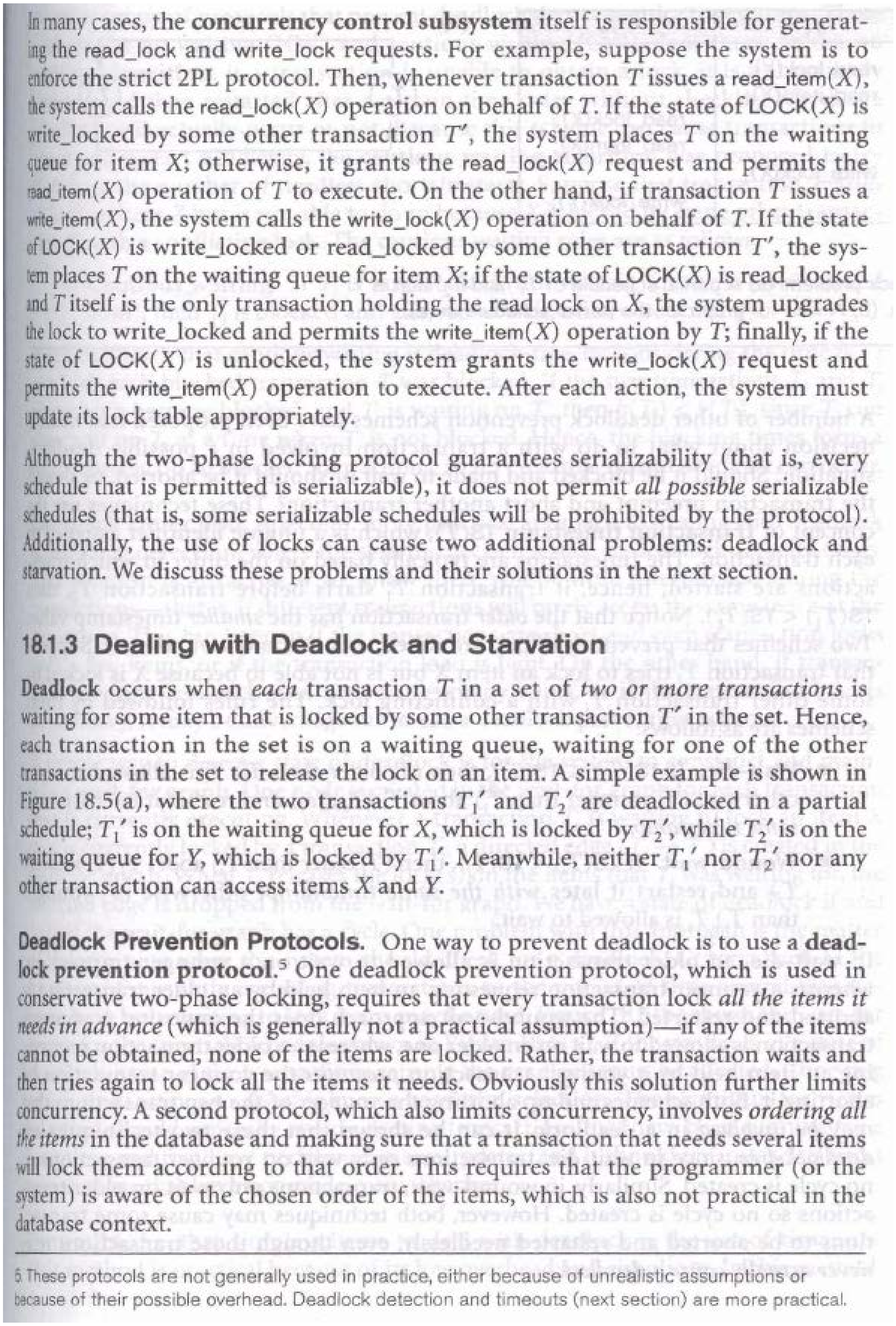
T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Figure 18.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 of Figure 18.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

using it if *T* must lock an additional item *Y* later; or conversely, *T* must lock the additional item *Y* before it needs it so that it can release *X*. Hence, *X* must remain locked by *T* until all items that the transaction needs to read or write have been locked; only then can *X* be released by *T*. Meanwhile, another transaction seeking to access *X* may be forced to wait, even though *T* is done with *X*; conversely, if *Y* is locked earlier than it is needed, another transaction seeking to access *Y* is forced to wait even though *T* is not using *Y* yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. Recall from Section 17.1.2 that the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol, as we shall see in Section 18.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in most situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 17.4). In this variation, a transaction *T* does not release any of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by *T* unless *T* has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL; the former must lock all its items *before it starts* so once the transaction starts it is in its shrinking phase, whereas the latter does not unlock any of its items until *after it terminates* (by committing or aborting) so the transaction is in its expanding phase until it ends.



In many cases, the **concurrency control subsystem** itself is responsible for generating the read_lock and write_lock requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction T issues a read_item(X), the system calls the read_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked by some other transaction T', the system places T on the waiting queue for item X; otherwise, it grants the read_lock(X) request and permits the read_item(X) operation of T to execute. On the other hand, if transaction T issues a write_item(X), the system calls the write_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked or read_locked by some other transaction T', the system places T on the waiting queue for item X; if the state of LOCK(X) is read_locked and T itself is the only transaction holding the read lock on X, the system upgrades the lock to write_locked and permits the write_item(X) operation by T; finally, if the state of LOCK(X) is unlocked, the system grants the write_lock(X) request and permits the write_item(X) operation to execute. After each action, the system must update its lock table appropriately.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol). Additionally, the use of locks can cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

18.1.3 Dealing with Deadlock and Starvation

Deadlock occurs when *each* transaction T in a set of *two or more transactions* is waiting for some item that is locked by some other transaction T' in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. A simple example is shown in Figure 18.5(a), where the two transactions T_1' and T_2' are deadlocked in a partial schedule; T_1' is on the waiting queue for X, which is locked by T_2', while T_2' is on the waiting queue for Y, which is locked by T_1'. Meanwhile, neither T_1' nor T_2' nor any other transaction can access items X and Y.

Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol.**[5] One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

5. These protocols are not generally used in practice, either because of unrealistic assumptions or because of their possible overhead. Deadlock detection and timeouts (next section) are more practical.

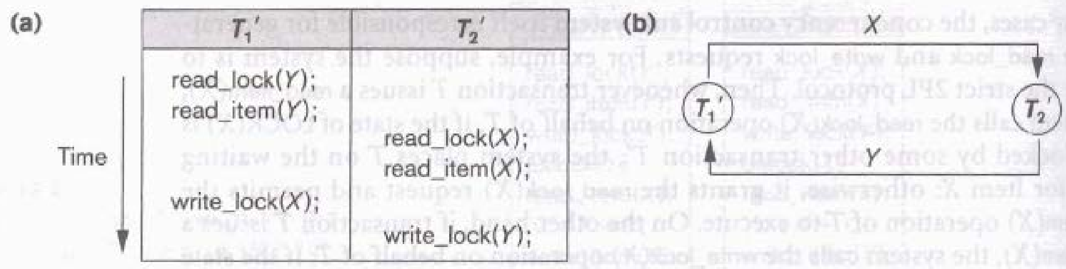
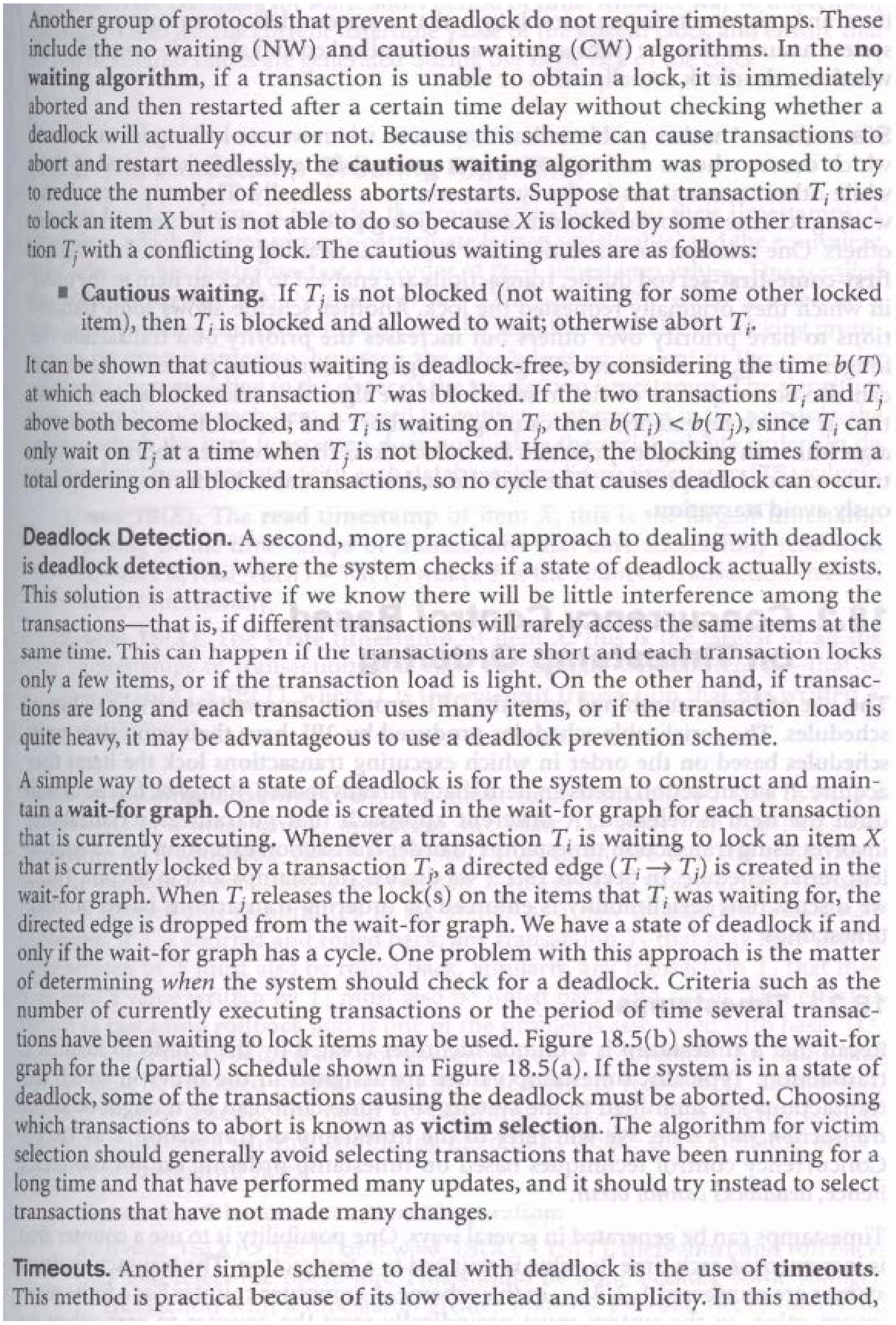


Figure 18.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? These techniques use the concept of **transaction timestamp** TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then TS(T_1) < TS(T_2). Notice that the *older* transaction has the *smaller* timestamp value. Two schemes that prevent deadlock are called wait-die and wound-wait. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are as follows:

- **Wait-die.** If TS(T_i) < TS(T_j), then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp*.
- **Wound-wait.** If TS(T_i) < TS(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

In wait-die, an older transaction is allowed to wait on a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to wait on an older one, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions that *may be involved* in a deadlock. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait on younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait on older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.



Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. Because this scheme can cause transactions to abort and restart needlessly, the **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rules are as follows:

- **Cautious waiting.** If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that cautious waiting is deadlock-free, by considering the time $b(T)$ at which each blocked transaction T was blocked. If the two transactions T_i and T_j above both become blocked, and T_i is waiting on T_j, then $b(T_i) < b(T_j)$, since T_i can only wait on T_j at a time when T_j is not blocked. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection. A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph.** One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge $(T_i \rightarrow T_j)$ is created in the wait-for graph. When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used. Figure 18.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 18.5(a). If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes.

Timeouts. Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

Starvation. Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation.

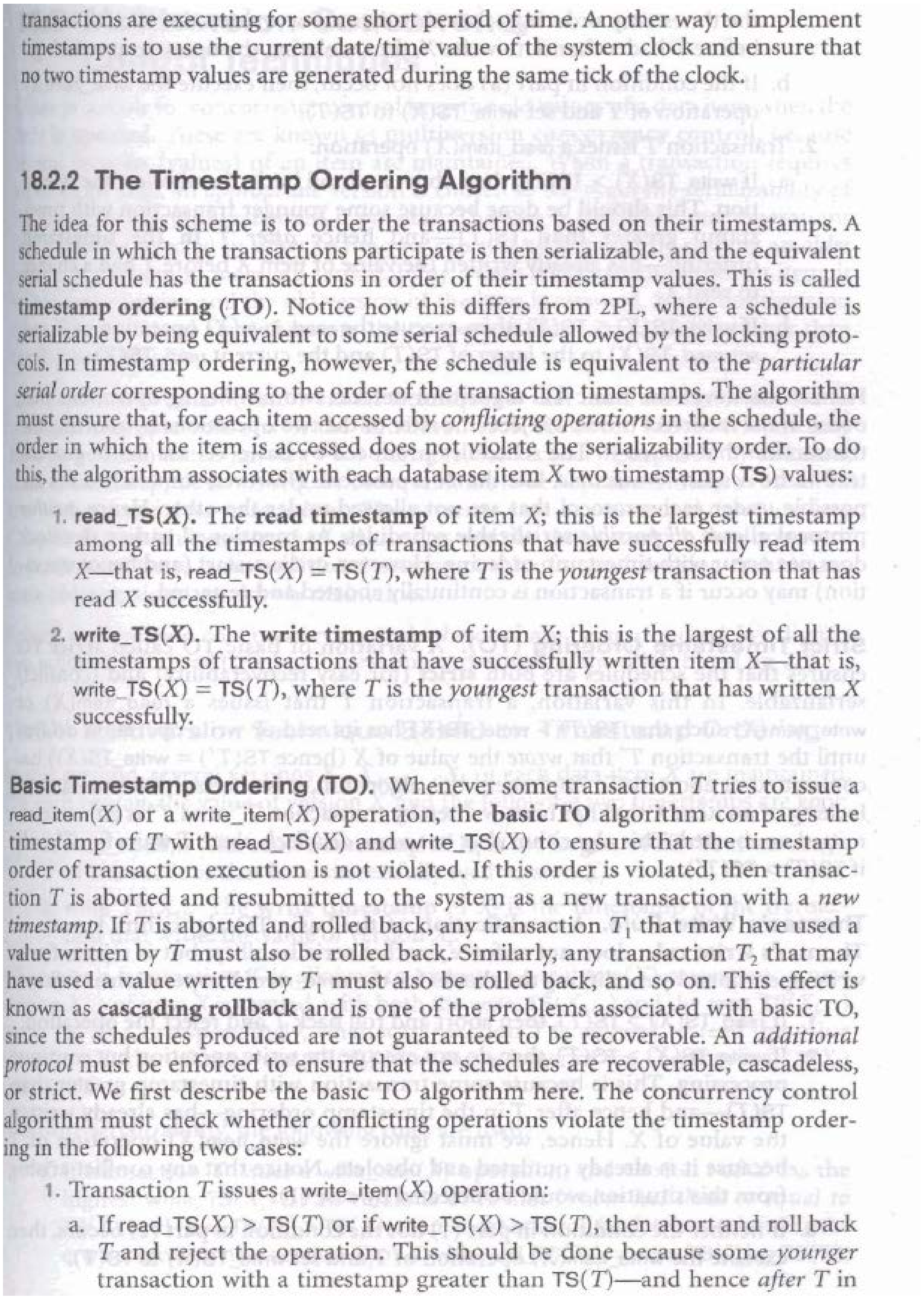
18.2 Concurrency Control Based on Timestamp Ordering

The use of locks, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule. In Section 18.2.1 we discuss timestamps and in Section 18.2.2 we discuss how serializability is enforced by ordering transactions based on their timestamps.

18.2.1 Timestamps

Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction T as $\mathbf{TS}(T)$. Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no



transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

18.2.2 The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called **timestamp ordering** (TO). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (**TS**) values:

1. **read_TS(X).** The **read timestamp** of item X; this is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully.
2. **write_TS(X).** The **write timestamp** of item X; this is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully.

Basic Timestamp Ordering (TO). Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the **basic TO** algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a write_item(X) operation:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(T)—and hence *after* T in

the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Transaction T issues a read_item(X) operation:

   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.

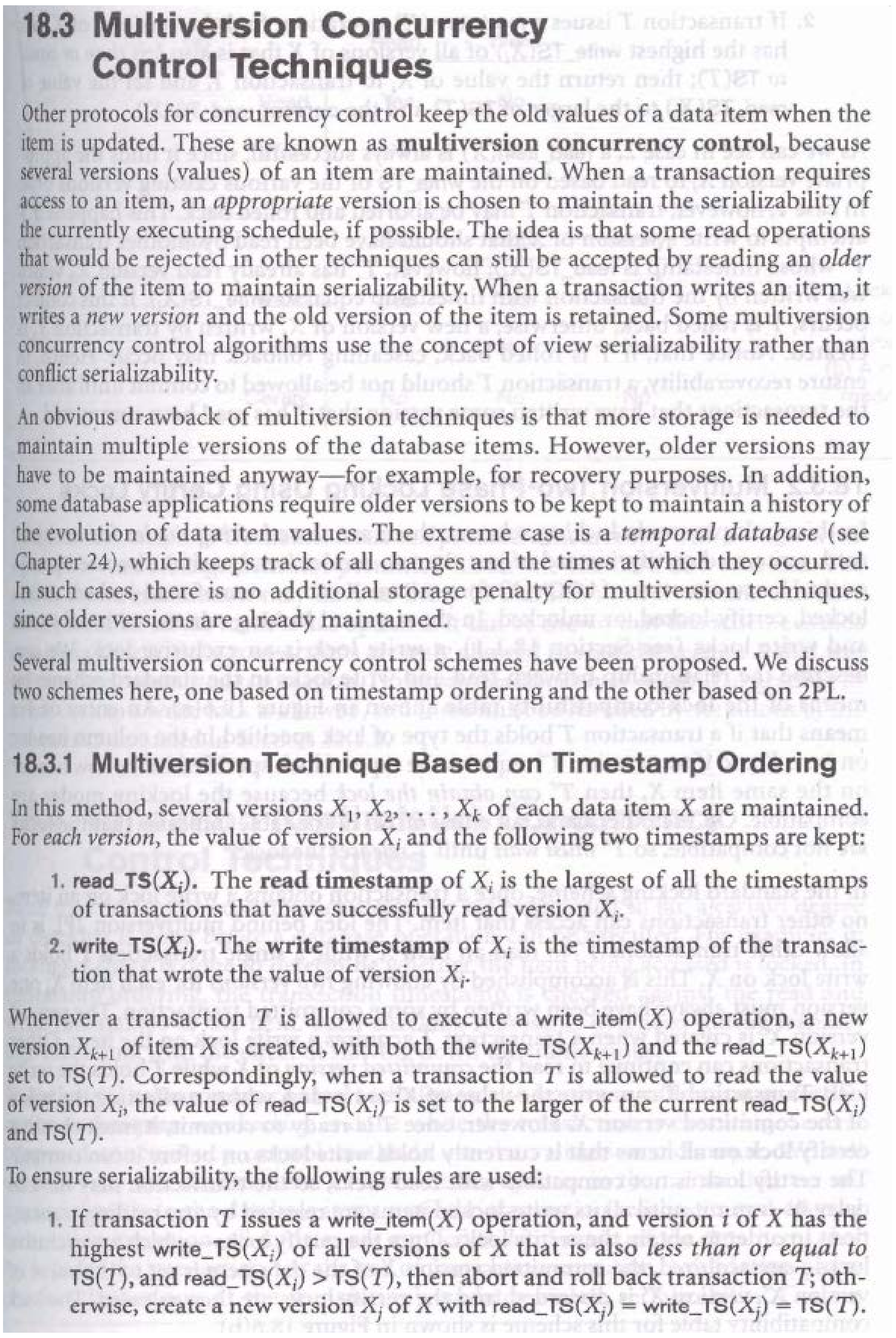
   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

Hence, whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*, like the 2PL protocol. However, some schedules are possible under each protocol that are not allowed under the other. Hence, *neither* protocol allows *all possible* serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO). A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T' that *wrote* the value of X (hence TS(T') = write_TS(X)) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T' until T' is either committed or aborted. This algorithm *does not cause deadlock*, since T waits for T' only if TS(T) > TS(T').

Thomas's Write Rule. A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability; but it rejects fewer write operations, by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Hence, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).



18.3 Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control,** because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version of the item is retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* (see Chapter 24), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on 2PL.

18.3.1 Multiversion Technique Based on Timestamp Ordering

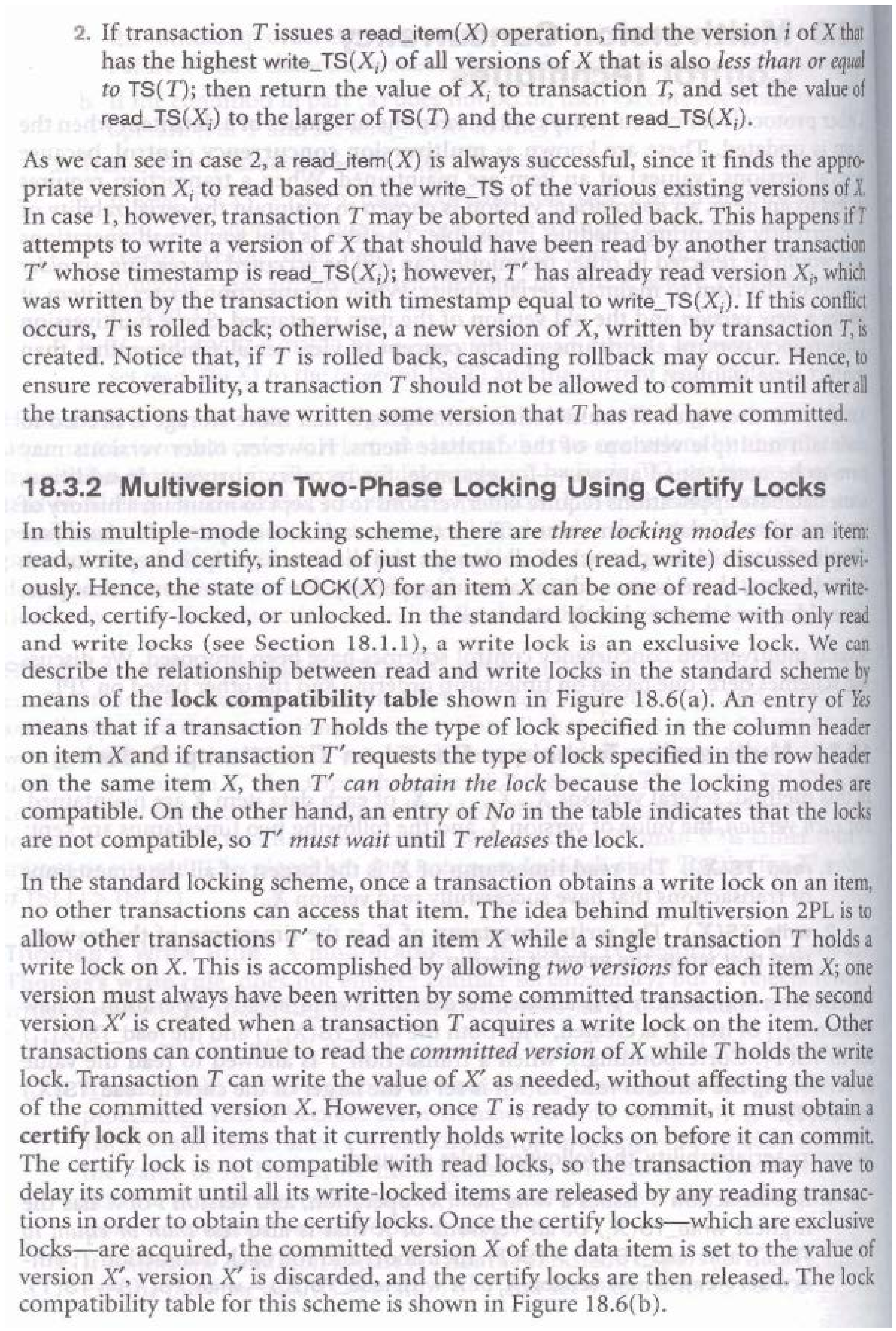
In this method, several versions $X_1, X_2, \ldots, X_k$ of each data item X are maintained. For *each version*, the value of version X_i and the following two timestamps are kept:

1. read_TS(X_i). The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
2. write_TS(X_i). The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following rules are used:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).



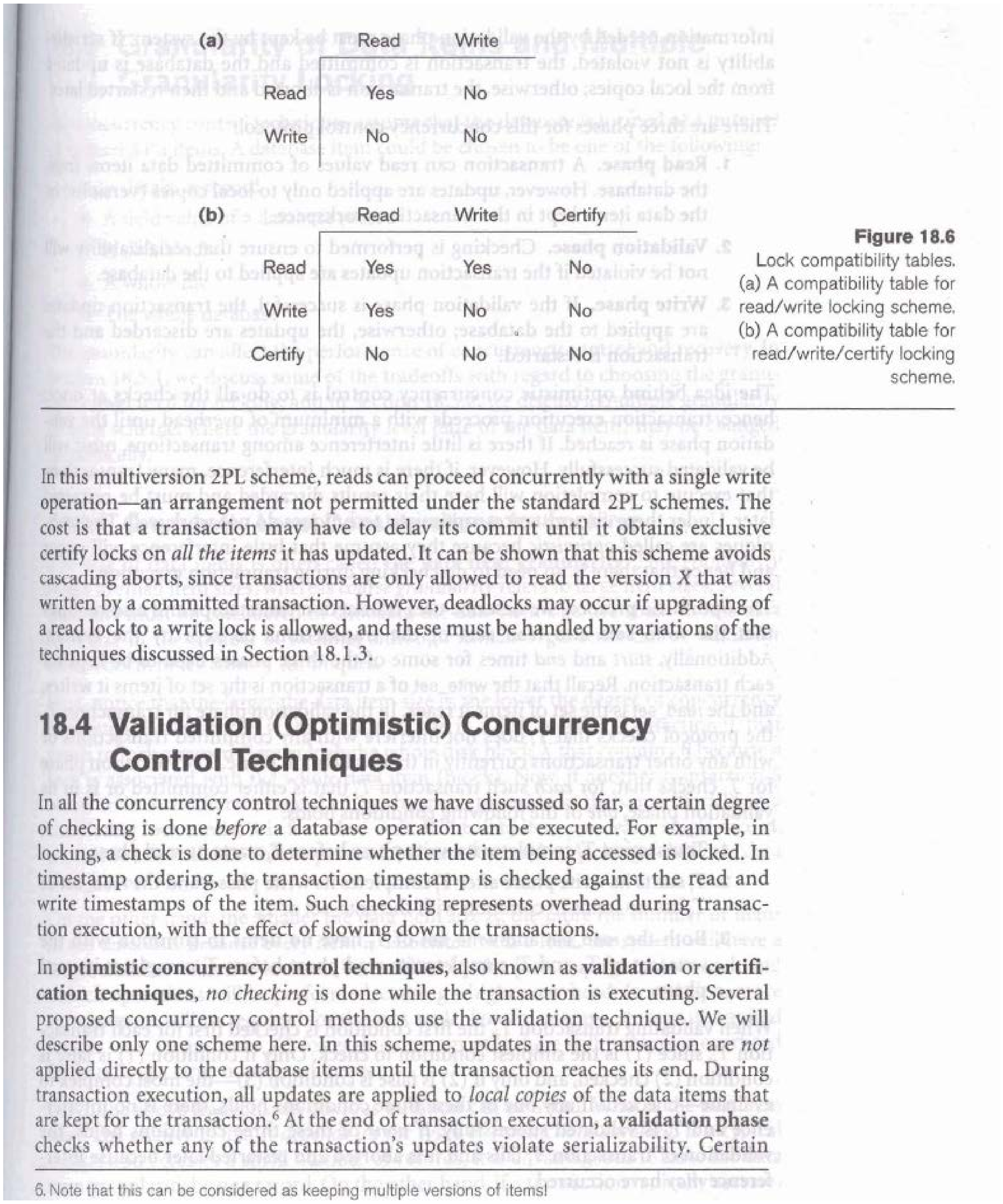
2. If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As we can see in case 2, a read_item(X) is always successful, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T' whose timestamp is read_TS(X_i); however, T' has already read version X_i, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

18.3.2 Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and certify, instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK(X) for an item X can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme with only read and write locks (see Section 18.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 18.6(a). An entry of *Yes* means that if a transaction T holds the type of lock specified in the column header on item X and if transaction T' requests the type of lock specified in the row header on the same item X, then T' *can obtain the lock* because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so T' *must wait* until T *releases* the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T' to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing *two versions* for each item X; one version must always have been written by some committed transaction. The second version X' is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the *committed version* of X while T holds the write lock. Transaction T can write the value of X' as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 18.6(b).



(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 18.6
Lock compatibility tables.
(a) A compatibility table for read/write locking scheme.
(b) A compatibility table for read/write/certify locking scheme.

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed in Section 18.1.3.

18.4 Validation (Optimistic) Concurrency Control Techniques

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several proposed concurrency control methods use the validation technique. We will describe only one scheme here. In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.[6] At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability. Certain

6. Note that this can be considered as keeping multiple versions of items!

information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

The optimistic protocol we describe uses transaction timestamps and also requires that the write_sets and read_sets of the transactions be kept by the system. Additionally, *start* and *end* times for some of the three phases need to be kept for each transaction. Recall that the write_set of a transaction is the set of items it writes, and the read_set is the set of items it reads. In the validation phase for transaction T_i, the protocol checks that T_i does not interfere with any committed transactions or with any other transactions currently in their validation phase. The validation phase for T_i checks that, for *each* such transaction T_j that is either committed or is in its validation phase, *one* of the following conditions holds:

1. Transaction T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.
3. Both the read_set and write_set of T_i have no items in common with the write_set of T_j, and T_j completes its read phase before T_i completes its read phase.

When validating transaction T_i, the first condition is checked first for each transaction T_j, since (1) is the simplest condition to check. Only if condition (1) is false is condition (2) checked, and only if (2) is false is condition (3)—the most complex to evaluate—checked. If any one of these three conditions holds, there is no interference and T_i is validated successfully. If *none* of these three conditions holds, the validation of transaction T_i fails and it is aborted and restarted later because interference *may* have occurred.

Q.5 a. Write a SQL query to find the salaries of all employees of the 'Finance' department, as well as the maximum salary, the minimum salary, and the average salary in this department. (9)

Answer:

```
SELECT   SUM (SALARY), MAX (SALARY), MIN (SALARY), AVG (SALARY)

         FROM     (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)

WHERE    DNAME='FINANCE'.
```

b. How general constraints can be specified by the users in SQL? Explain your answer with an example. (9)

Answer:

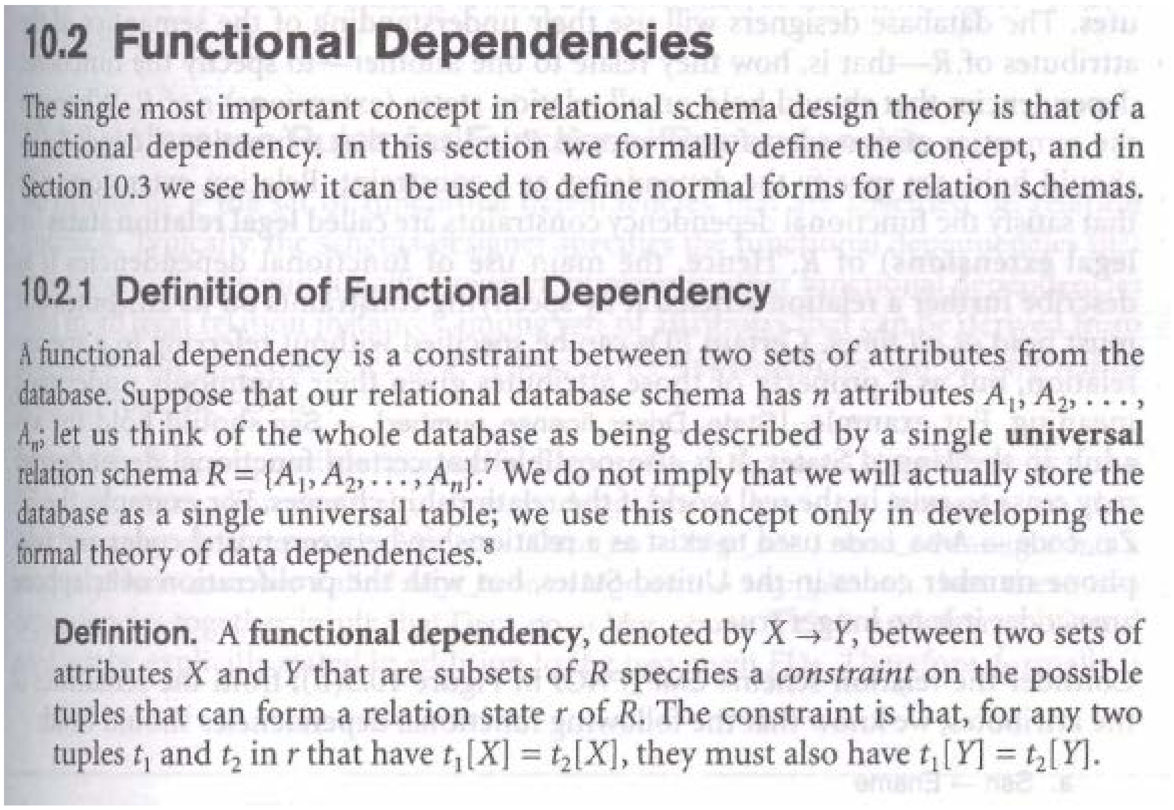
In SQL, users can specify general constrains via declarative assertions, using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL. For example, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL, users may write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT

CHECK (NOT EXISTS

          (SELECT *

FROM  EMPLOYEE E, EMPLOYEE M, DEPARTMENT D

WHERE    E.SALARY> M.SALARY AND

E.DNO=D.NUMBER AND

D.MGRSSN=M.SSN)  );
```

The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition. By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty. Thus, the assertion is violated if the result of the query is not empty.

Q.6 a. Explain functional dependency. Mention any three inference rules used in functional dependencies. (3+3)

Answer:



10.2 Functional Dependencies

The single most important concept in relational schema design theory is that of a functional dependency. In this section we formally define the concept, and in Section 10.3 we see how it can be used to define normal forms for relation schemas.

10.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has *n* attributes $A_1, A_2, \ldots, A_n$; let us think of the whole database as being described by a single **universal** relation schema $R = \{A_1, A_2, \ldots, A_n\}$.[7] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[8]

Definition. A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by,* the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X. The abbreviation for functional dependency is FD or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side.**

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X-value, they must necessarily agree on their Y-value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance $r(R)$—that is, X is a **candidate key** of R—this implies that $X \to Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state $r(R)$ will have the same value of X).
- If $X \to Y$ in R, this does not say whether or not $Y \to X$ in R.

A functional dependency is a property of the **semantics** or **meaning of the attributes.** The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions) r of R. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R. Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} → Ssn should hold for any adult in the United States. It is also possible that certain functional dependencies may cease to exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

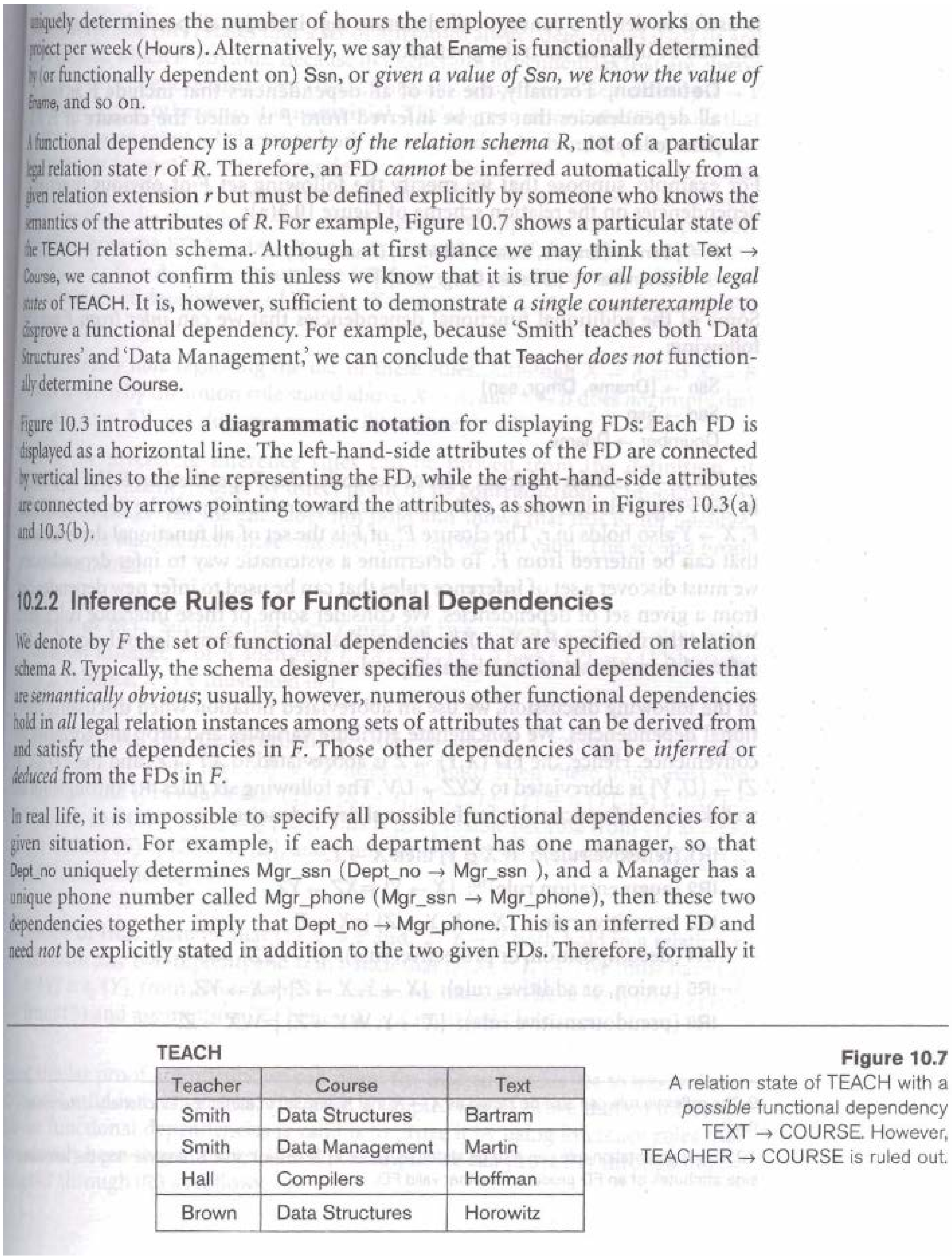
Consider the relation schema EMP_PROJ in Figure 10.3(b); from the semantics of the attributes, we know that the following functional dependencies should hold:

a. Ssn → Ename

b. Pnumber → {Pname, Plocation}

c. {Ssn, Pnumber} → Hours

These functional dependencies specify that (a) the value of an employee's social security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values



uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of* Ename, and so on.

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*. For example, Figure 10.7 shows a particular state of the TEACH relation schema. Although at first glance we may think that Text → Course, we cannot confirm this unless we know that it is true *for all possible legal states* of TEACH. It is, however, sufficient to demonstrate *a single counterexample* to disprove a functional dependency. For example, because 'Smith' teaches both 'Data Structures' and 'Data Management,' we can conclude that Teacher *does not* functionally determine Course.

Figure 10.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by arrows pointing toward the attributes, as shown in Figures 10.3(a) and 10.3(b).

10.2.2 Inference Rules for Functional Dependencies

We denote by *F* the set of functional dependencies that are specified on relation schema *R*. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in *F*. Those other dependencies can be *inferred* or *deduced* from the FDs in *F*.

In real life, it is impossible to specify all possible functional dependencies for a given situation. For example, if each department has one manager, so that Dept_no uniquely determines Mgr_ssn (Dept_no → Mgr_ssn), and a Manager has a unique phone number called Mgr_phone (Mgr_ssn → Mgr_phone), then these two dependencies together imply that Dept_no → Mgr_phone. This is an inferred FD and need *not* be explicitly stated in addition to the two given FDs. Therefore, formally it

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Figure 10.7
A relation state of TEACH with a *possible* functional dependency TEXT → COURSE. However, TEACHER → COURSE is ruled out.

is useful to define a concept called *closure* that includes all possible dependencies that can be inferred from the given set *F*.

Definition. Formally, the set of all dependencies that include *F* as well as all dependencies that can be inferred from *F* is called the **closure** of *F*; it is denoted by F^+.

For example, suppose that we specify the following set *F* of obvious functional dependencies on the relation schema of Figure 10.3(a):

$F = \{\text{Ssn} \rightarrow \{\text{Ename, Bdate, Address, Dnumber}\},$
$\quad \text{Dnumber} \rightarrow \{\text{Dname, Dmgr\_ssn}\}\}$

Some of the additional functional dependencies that we can *infer* from *F* are the following:

$\text{Ssn} \rightarrow \{\text{Dname, Dmgr\_ssn}\}$
$\text{Ssn} \rightarrow \text{Ssn}$
$\text{Dnumber} \rightarrow \text{Dname}$

An FD $X \rightarrow Y$ is **inferred from** a set of dependencies *F* specified on *R* if $X \rightarrow Y$ holds in *every* legal relation state *r* of *R*; that is, whenever *r* satisfies all the dependencies in F, $X \rightarrow Y$ also holds in *r*. The closure F^+ of *F* is the set of all functional dependencies that can be inferred from *F*. To determine a systematic way to infer dependencies, we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies. We consider some of these inference rules next. We use the notation $F \models X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies *F*.

In the following discussion, we use an abbreviated notation when discussing functional dependencies. We concatenate attribute variables and drop the commas for convenience. Hence, the FD $\{X,Y\} \rightarrow Z$ is abbreviated to $XY \rightarrow Z$, and the FD $\{X, Y, Z\} \rightarrow \{U, V\}$ is abbreviated to $XYZ \rightarrow UV$. The following six rules IR1 through IR6 are well-known inference rules for functional dependencies:

IR1 (reflexive rule)[9]: If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule)[10]: $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

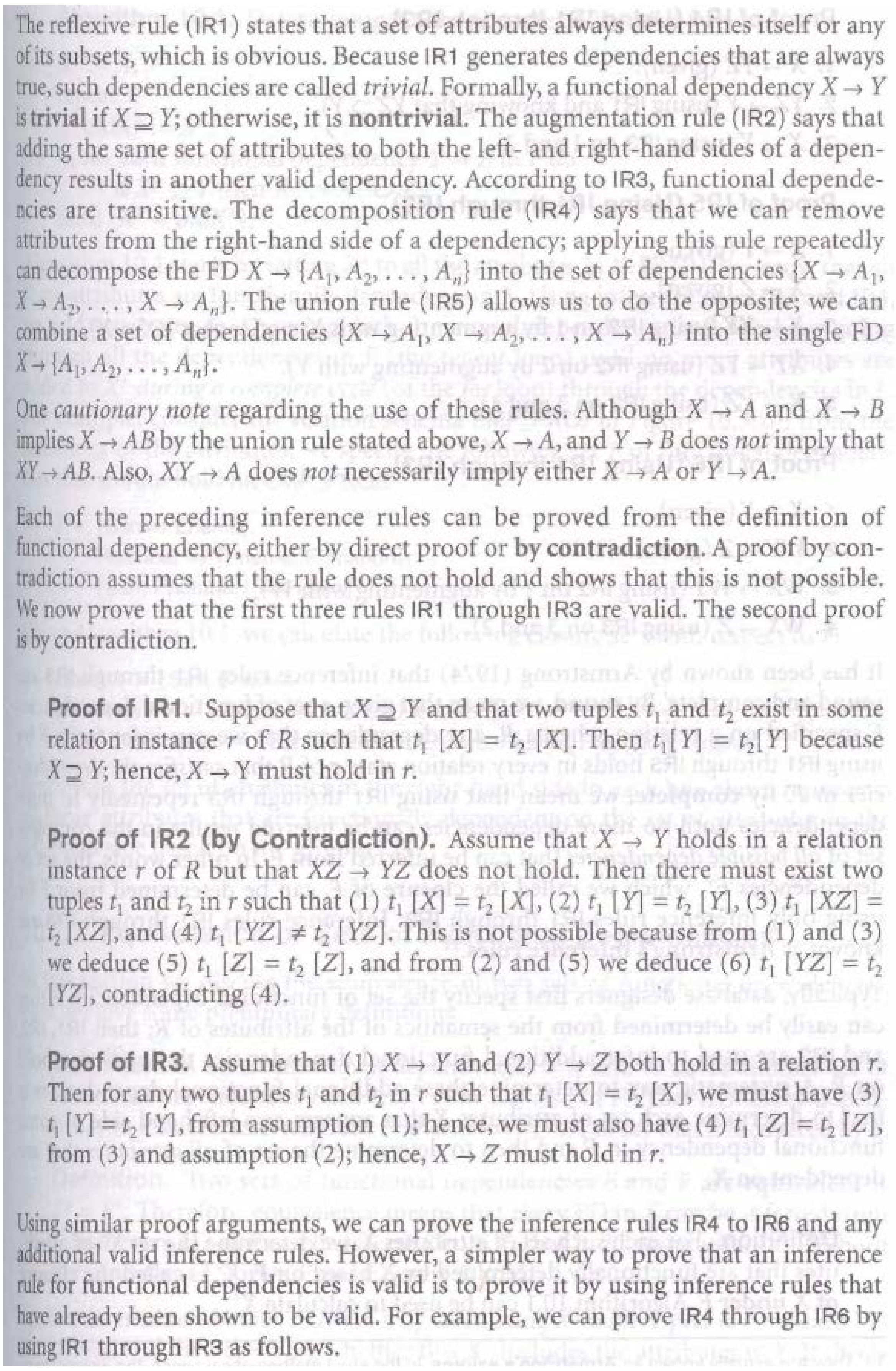
IR4 (decomposition, or projective, rule): $\{X \rightarrow YZ\} \models X \rightarrow Y$.

IR5 (union, or additive, rule): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

IR6 (pseudotransitive rule): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

9. The reflexive rule can also be stated as $X \rightarrow X$; that is, any set of attributes functionally determines itself.

10. The augmentation rule can also be stated as $\{X \rightarrow Y\} \models XZ \rightarrow Y$; that is, augmenting the left-hand side attributes of an FD produces another valid FD.



The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious. Because IR1 generates dependencies that are always true, such dependencies are called *trivial*. Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**. The augmentation rule (IR2) says that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency. According to IR3, functional dependencies are transitive. The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ into the set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$. The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$ into the single FD $X \rightarrow \{A_1, A_2, \ldots, A_n\}$.

One *cautionary note* regarding the use of these rules. Although $X \rightarrow A$ and $X \rightarrow B$ implies $X \rightarrow AB$ by the union rule stated above, $X \rightarrow A$, and $Y \rightarrow B$ does *not* imply that $XY \rightarrow AB$. Also, $XY \rightarrow A$ does *not* necessarily imply either $X \rightarrow A$ or $Y \rightarrow A$.

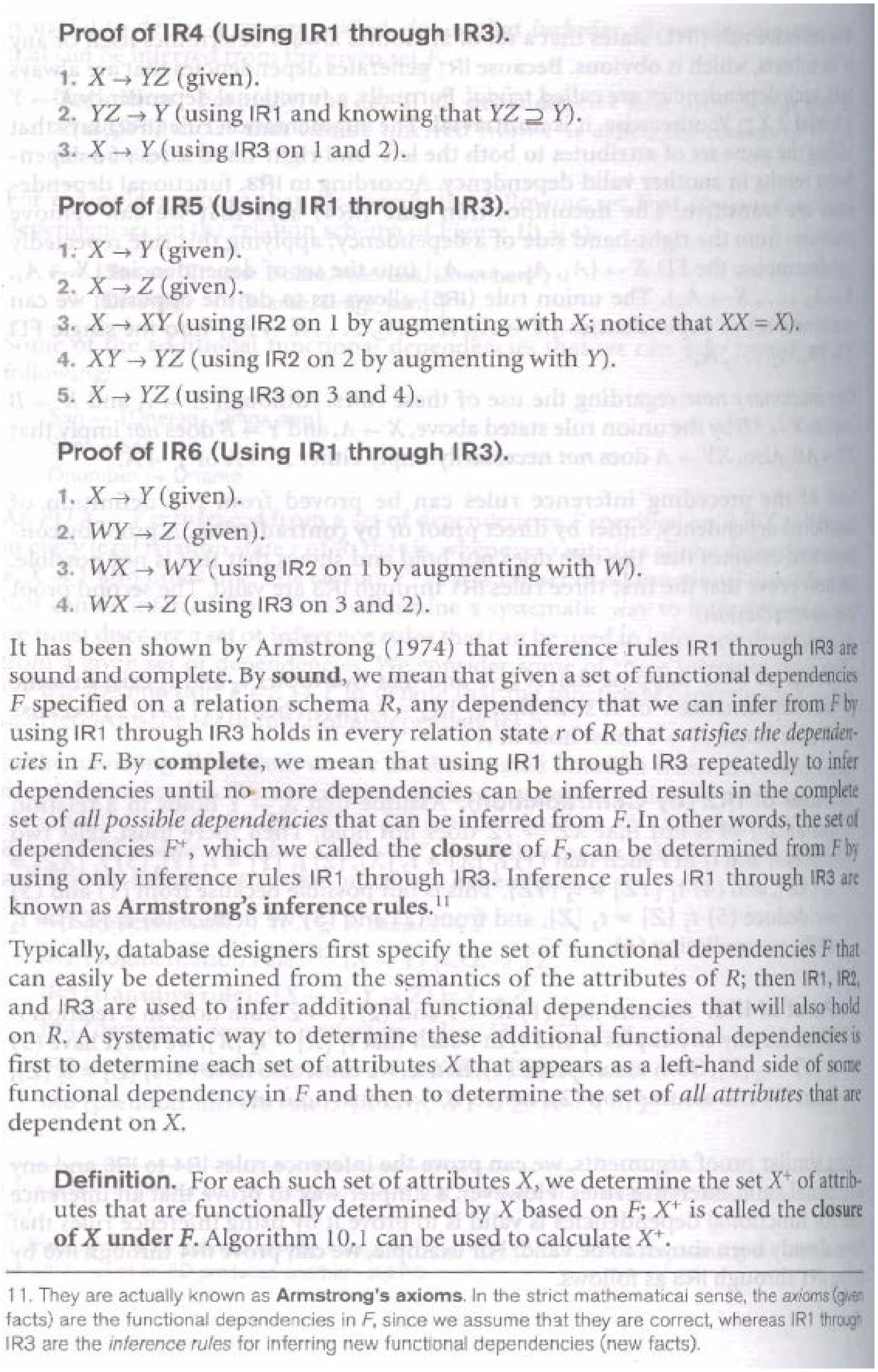
Each of the preceding inference rules can be proved from the definition of functional dependency, either by direct proof or **by contradiction**. A proof by contradiction assumes that the rule does not hold and shows that this is not possible. We now prove that the first three rules IR1 through IR3 are valid. The second proof is by contradiction.

Proof of IR1. Suppose that $X \supseteq Y$ and that two tuples t_1 and t_2 exist in some relation instance r of R such that $t_1[X] = t_2[X]$. Then $t_1[Y] = t_2[Y]$ because $X \supseteq Y$; hence, $X \rightarrow Y$ must hold in r.

Proof of IR2 (by Contradiction). Assume that $X \rightarrow Y$ holds in a relation instance r of R but that $XZ \rightarrow YZ$ does not hold. Then there must exist two tuples t_1 and t_2 in r such that (1) $t_1[X] = t_2[X]$, (2) $t_1[Y] = t_2[Y]$, (3) $t_1[XZ] = t_2[XZ]$, and (4) $t_1[YZ] \neq t_2[YZ]$. This is not possible because from (1) and (3) we deduce (5) $t_1[Z] = t_2[Z]$, and from (2) and (5) we deduce (6) $t_1[YZ] = t_2[YZ]$, contradicting (4).

Proof of IR3. Assume that (1) $X \rightarrow Y$ and (2) $Y \rightarrow Z$ both hold in a relation r. Then for any two tuples t_1 and t_2 in r such that $t_1[X] = t_2[X]$, we must have (3) $t_1[Y] = t_2[Y]$, from assumption (1); hence, we must also have (4) $t_1[Z] = t_2[Z]$, from (3) and assumption (2); hence, $X \rightarrow Z$ must hold in r.

Using similar proof arguments, we can prove the inference rules IR4 to IR6 and any additional valid inference rules. However, a simpler way to prove that an inference rule for functional dependencies is valid is to prove it by using inference rules that have already been shown to be valid. For example, we can prove IR4 through IR6 by using IR1 through IR3 as follows.



Proof of IR4 (Using IR1 through IR3).

1. $X \to YZ$ (given).
2. $YZ \to Y$ (using IR1 and knowing that $YZ \supseteq Y$).
3. $X \to Y$ (using IR3 on 1 and 2).

Proof of IR5 (Using IR1 through IR3).

1. $X \to Y$ (given).
2. $X \to Z$ (given).
3. $X \to XY$ (using IR2 on 1 by augmenting with X; notice that $XX = X$).
4. $XY \to YZ$ (using IR2 on 2 by augmenting with Y).
5. $X \to YZ$ (using IR3 on 3 and 4).

Proof of IR6 (Using IR1 through IR3).

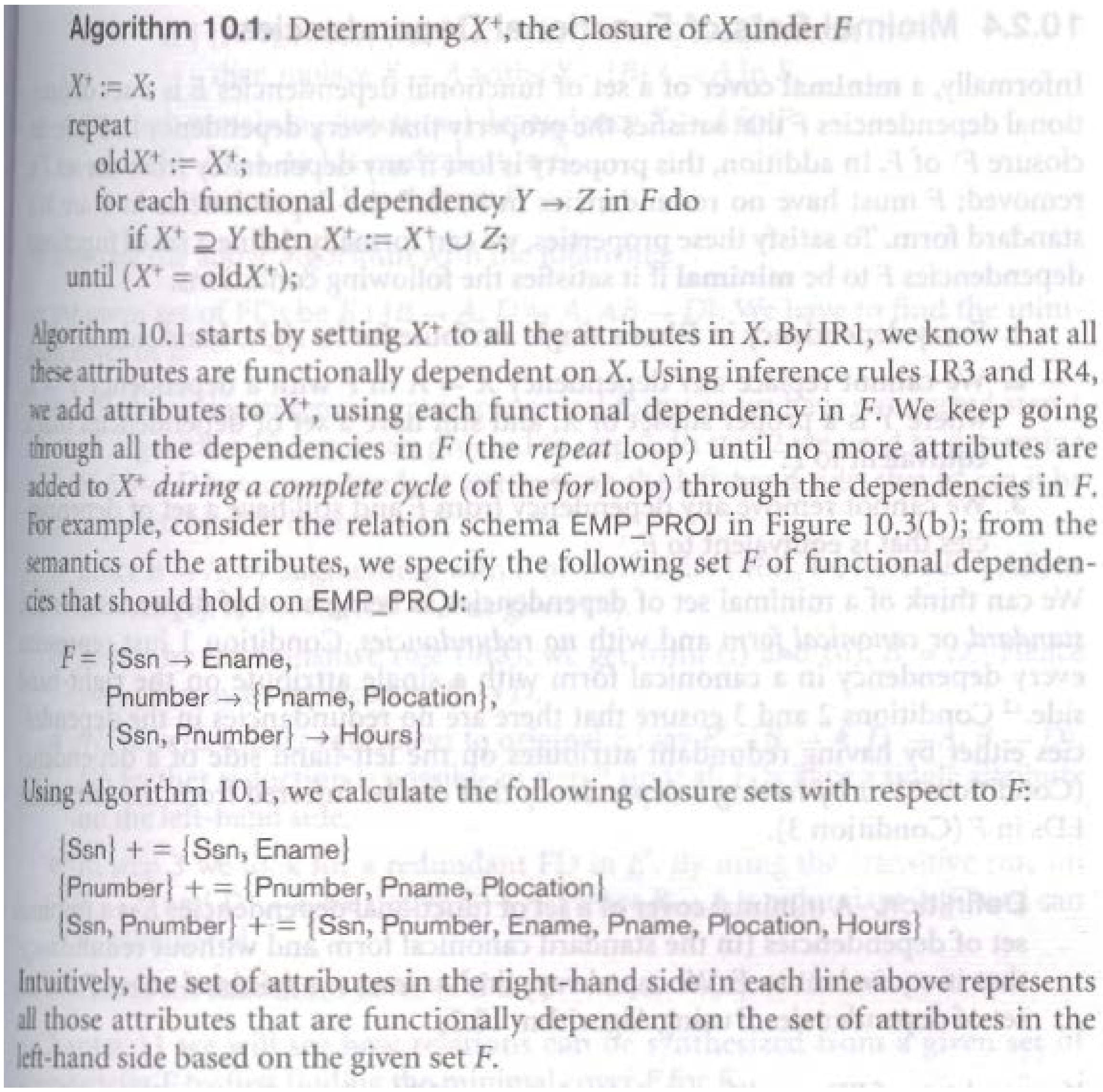
1. $X \to Y$ (given).
2. $WY \to Z$ (given).
3. $WX \to WY$ (using IR2 on 1 by augmenting with W).
4. $WX \to Z$ (using IR3 on 3 and 2).

It has been shown by Armstrong (1974) that inference rules IR1 through IR3 are sound and complete. By **sound**, we mean that given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R that *satisfies the dependencies* in F. By **complete**, we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of *all possible dependencies* that can be inferred from F. In other words, the set of dependencies F^+, which we called the **closure** of F, can be determined from F by using only inference rules IR1 through IR3. Inference rules IR1 through IR3 are known as **Armstrong's inference rules.**[11]

Typically, database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then IR1, IR2, and IR3 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X.

Definition. For each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the **closure of X under F.** Algorithm 10.1 can be used to calculate X^+.

11. They are actually known as **Armstrong's axioms**. In the strict mathematical sense, the *axioms* (given facts) are the functional dependencies in F, since we assume that they are correct, whereas IR1 through IR3 are the *inference rules* for inferring new functional dependencies (new facts).



Algorithm 10.1. Determining X^+, the Closure of X under F

```
$X^+ := X$;
repeat
    old$X^+ := X^+$;
    for each functional dependency $Y \rightarrow Z$ in $F$ do
        if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
until ($X^+ =$ old$X^+$);
```

Algorithm 10.1 starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the *repeat* loop) until no more attributes are added to X^+ *during a complete cycle* (of the *for* loop) through the dependencies in F. For example, consider the relation schema EMP_PROJ in Figure 10.3(b); from the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PROJ:

F = {Ssn → Ename,
Pnumber → {Pname, Plocation},
{Ssn, Pnumber} → Hours}

Using Algorithm 10.1, we calculate the following closure sets with respect to F:

{Ssn} + = {Ssn, Ename}
{Pnumber} + = {Pnumber, Pname, Plocation}
{Ssn, Pnumber} + = {Ssn, Pnumber, Ename, Pname, Plocation, Hours}

Intuitively, the set of attributes in the right-hand side in each line above represents all those attributes that are functionally dependent on the set of attributes in the left-hand side based on the given set F.

b. Explain 2NF, 3NF and 4NF. Give an example for each. (3x2)
Answer:

10.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies discussed in Section 10.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key*. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

10.4.1 General Definition of Second Normal Form

Definition. A relation schema R is in **second normal form (2NF)** if every nonprime attribute A in R is not partially dependent on *any* key of R.[16]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 10.11(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

Based on the two candidate keys Property_id# and {County_name, Lot#}, we know that the functional dependencies FD1 and FD2 of Figure 10.11(a) hold. We choose Property_id# as the primary key, so it is underlined in Figure 10.11(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate
FD4: Area → Price

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3.

16. This definition can be restated as follows: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on *every* key of R.

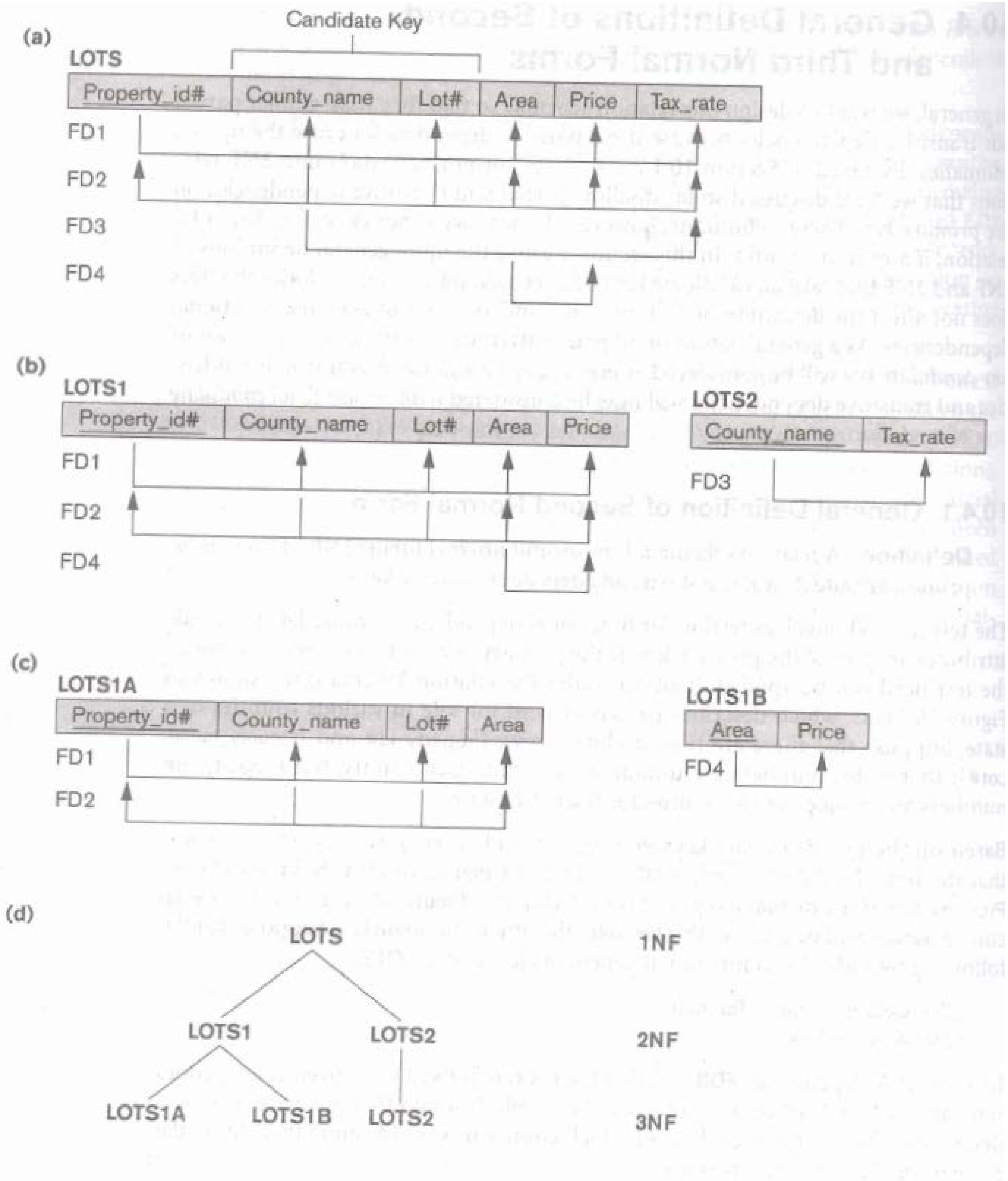
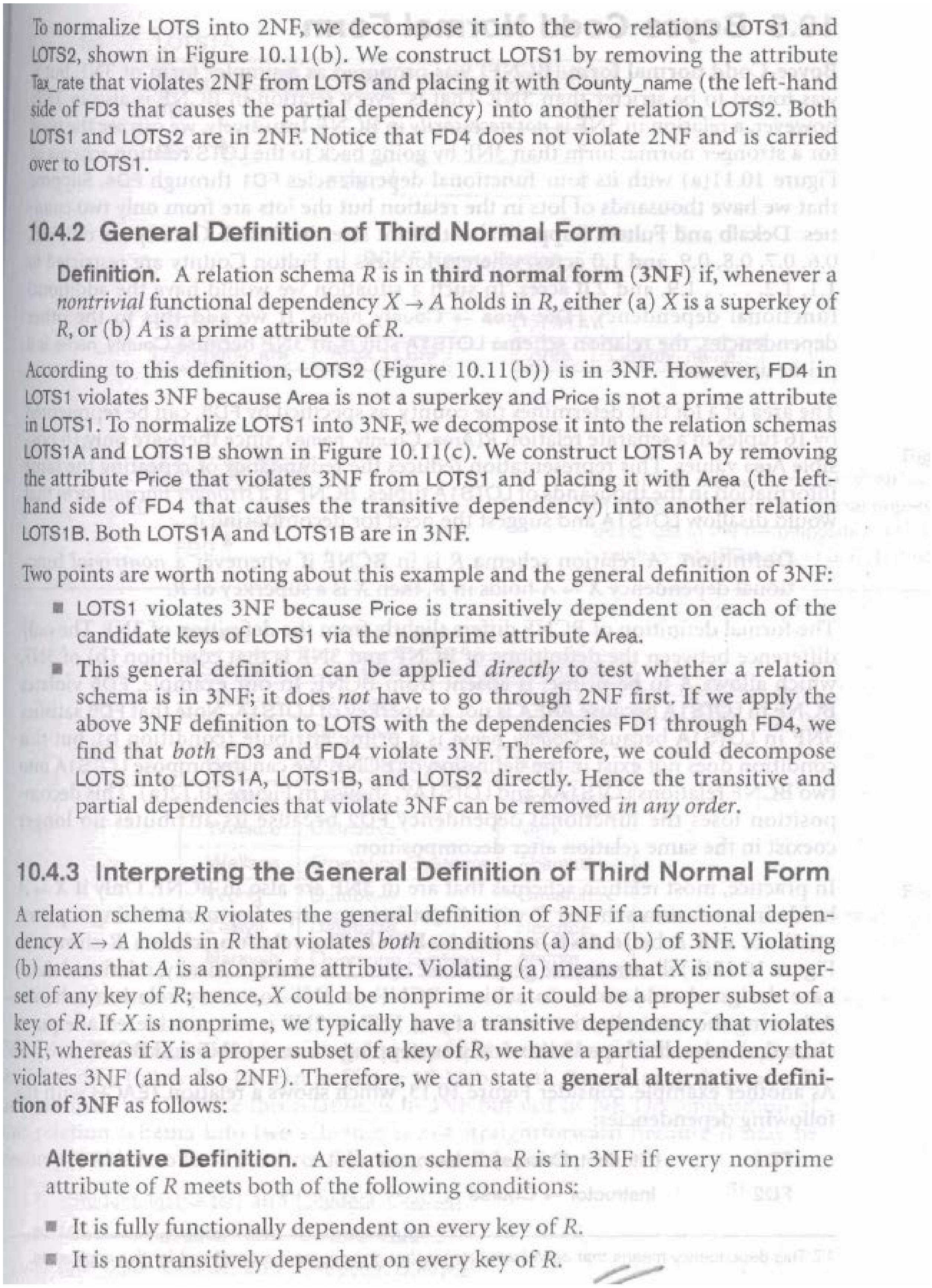


Figure 10.11
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 10.11(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

10.4.2 General Definition of Third Normal Form

Definition. A relation schema R is in **third normal form** (3NF) if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

According to this definition, LOTS2 (Figure 10.11(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 10.11(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence the transitive and partial dependencies that violate 3NF can be removed *in any order*.

10.4.3 Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that violates *both* conditions (a) and (b) of 3NF. Violating (b) means that A is a nonprime attribute. Violating (a) means that X is not a superset of any key of R; hence, X could be nonprime or it could be a proper subset of a key of R. If X is nonprime, we typically have a transitive dependency that violates 3NF, whereas if X is a proper subset of a key of R, we have a partial dependency that violates 3NF (and also 2NF). Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- It is fully functionally dependent on every key of R.
- It is nontransitively dependent on every key of R.

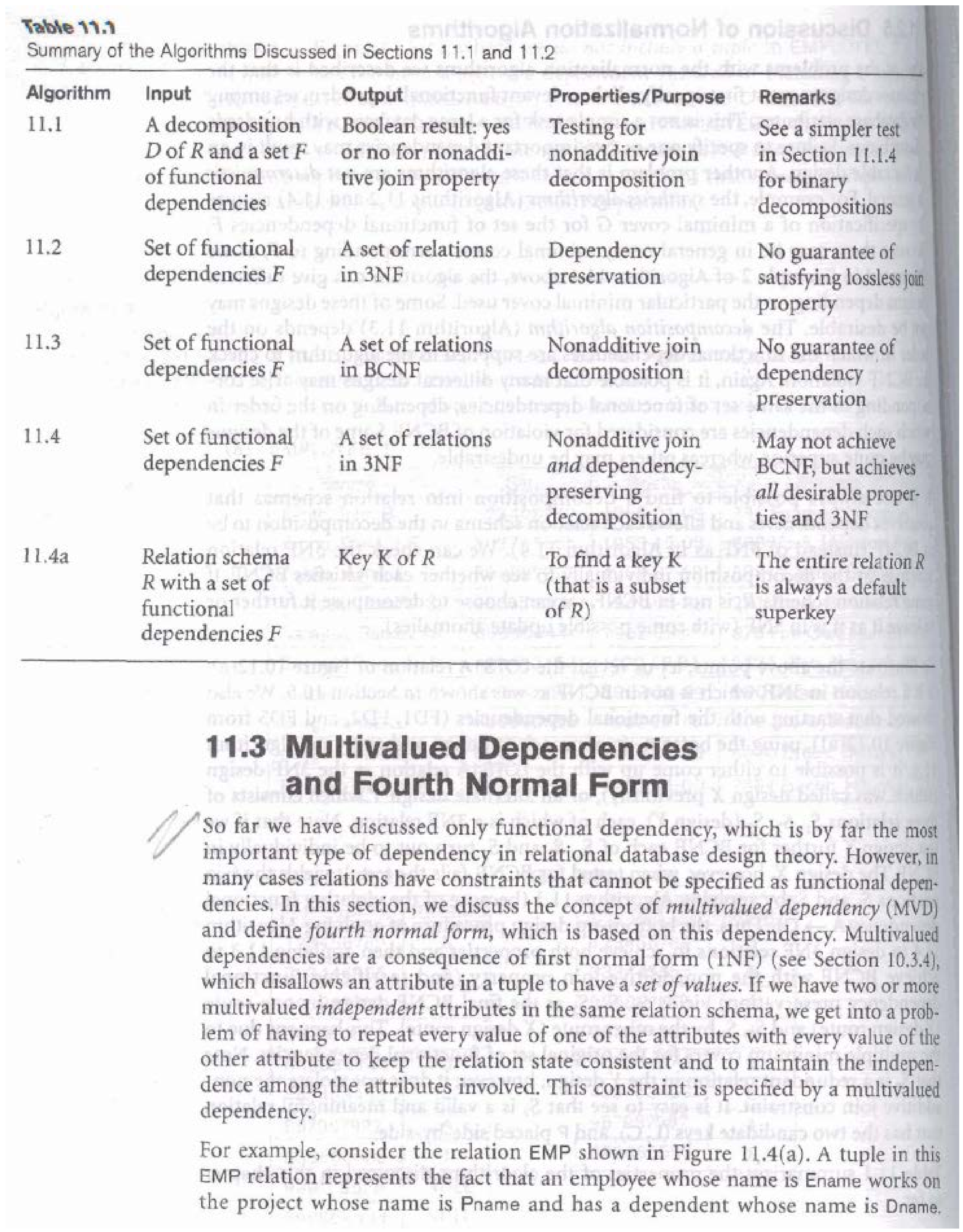


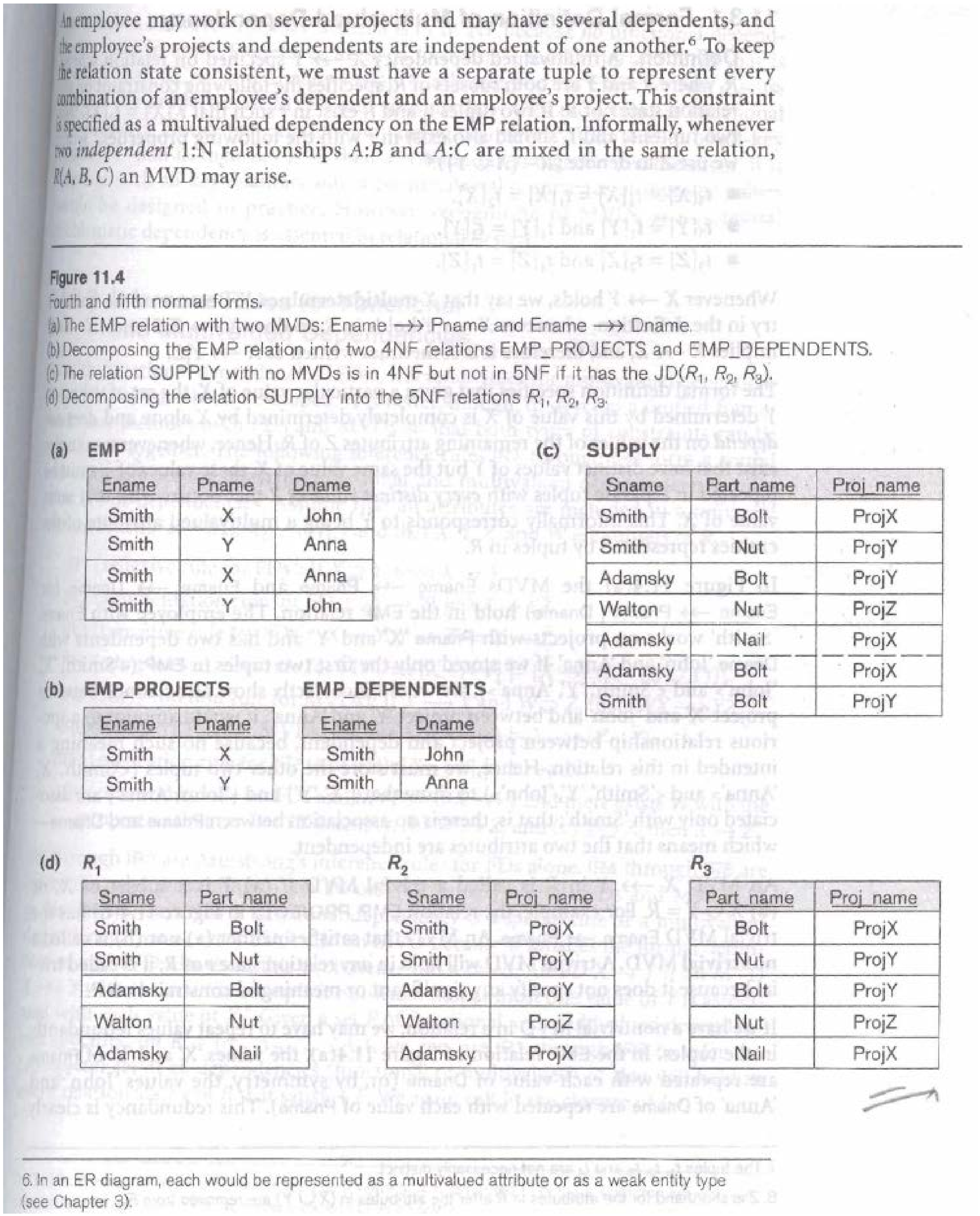
Table 11.1
Summary of the Algorithms Discussed in Sections 11.1 and 11.2

Algorithm	Input	Output	Properties/Purpose	Remarks
11.1	A decomposition *D* of *R* and a set *F* of functional dependencies	Boolean result: yes or no for nonadditive join property	Testing for nonadditive join decomposition	See a simpler test in Section 11.1.4 for binary decompositions
11.2	Set of functional dependencies *F*	A set of relations in 3NF	Dependency preservation	No guarantee of satisfying lossless join property
11.3	Set of functional dependencies *F*	A set of relations in BCNF	Nonadditive join decomposition	No guarantee of dependency preservation
11.4	Set of functional dependencies *F*	A set of relations in 3NF	Nonadditive join *and* dependency-preserving decomposition	May not achieve BCNF, but achieves *all* desirable properties and 3NF
11.4a	Relation schema *R* with a set of functional dependencies *F*	Key K of *R*	To find a key K (that is a subset of *R*)	The entire relation *R* is always a default superkey

11.3 Multivalued Dependencies and Fourth Normal Form

So far we have discussed only functional dependency, which is by far the most important type of dependency in relational database design theory. However, in many cases relations have constraints that cannot be specified as functional dependencies. In this section, we discuss the concept of *multivalued dependency* (MVD) and define *fourth normal form*, which is based on this dependency. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 10.3.4), which disallows an attribute in a tuple to have a *set of values*. If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure 11.4(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname.



An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another.[6] To keep the relation state consistent, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multivalued dependency on the EMP relation. Informally, whenever two *independent* 1:N relationships *A*:*B* and *A*:*C* are mixed in the same relation, *R*(*A*, *B*, *C*) an MVD may arise.

Figure 11.4
Fourth and fifth normal forms.
(a) The EMP relation with two MVDs: Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.
(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3).
(d) Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3.

(a) **EMP**

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) **EMP_PROJECTS**

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

(c) **SUPPLY**

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

6. In an ER diagram, each would be represented as a multivalued attribute or as a weak entity type (see Chapter 3).

c. **Explain the possible reasons for a transaction to fail in the middle of execution in DBMS. (6)**

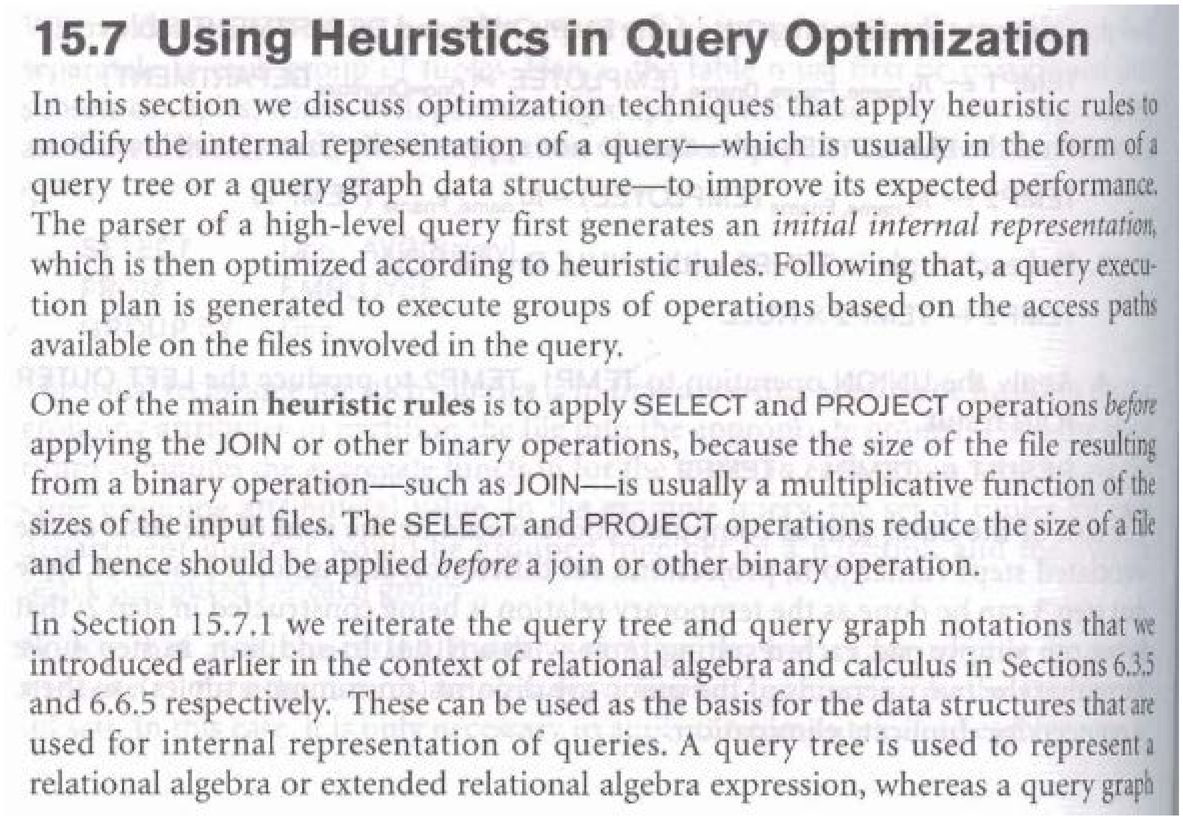
Answer:

The possible reasons for a transaction to fail in the middle of execution in DBMS are:

- **A computer failure:** A hardware, software, or network error occurs in the computer system during transaction execution.
- **A transaction or system error:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Moreover, the user may interrupt the transaction during its execution.
- **Local errors or exception conditions detected by the transaction:** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Insufficient account balance in banking database may cause a transaction, such as a fund withdrawal, to be cancelled.
- **Concurrency control enforcement:** the concurrency control method may decide to abort the transaction, to be restarted later, because several transactions are in a state of deadlock.
- **Disk failure:** Some disk blocks may lose their data because of read or write malfunction or a write operation of the transaction.
- **Physical problems and catastrophes:**This refers to an endless list of problems that includes power or air conditioner failure, fire, theft, etc.

Q.7 a. Explain heuristics in Query Optimization. (6)

Answer:



15.7 Using Heuristics in Query Optimization

In this section we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance. The parser of a high-level query first generates an *initial internal representation*, which is then optimized according to heuristic rules. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

In Section 15.7.1 we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 6.3.5 and 6.6.5 respectively. These can be used as the basis for the data structures that are used for internal representation of queries. A query tree is used to represent a relational algebra or extended relational algebra expression, whereas a query graph is used to represent a relational calculus expression. Then in Section 15.7.2 we show how heuristic optimization rules are applied to convert a query tree into an **equivalent query tree**, which represents a different relational algebra expression that is more efficient to execute but gives the same result as the original one. We also discuss the equivalence of various relational algebra expressions. Finally, Section 15.7.3 discusses the generation of query execution plans.

15.7.1 Notation for Query Trees and Query Graphs

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 15.4(a) shows a query tree (the same as shown in Figure 6.9) for query Q2 of Chapters 5 to 8: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT}))$$
$$\bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

This corresponds to the following SQL query:

```
Q2:  SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
     FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
             P.Plocation= 'Stafford';
```

In Figure 15.4(a) the three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E, while the relational algebra operations of the expression are represented by internal tree nodes. When this query tree is executed, the node marked (1) in Figure 15.4(a) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral representation of a query is the **query graph** notation. Figure 15.4(c) (the same as shown in Figure 6.13) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 15.4(c). Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

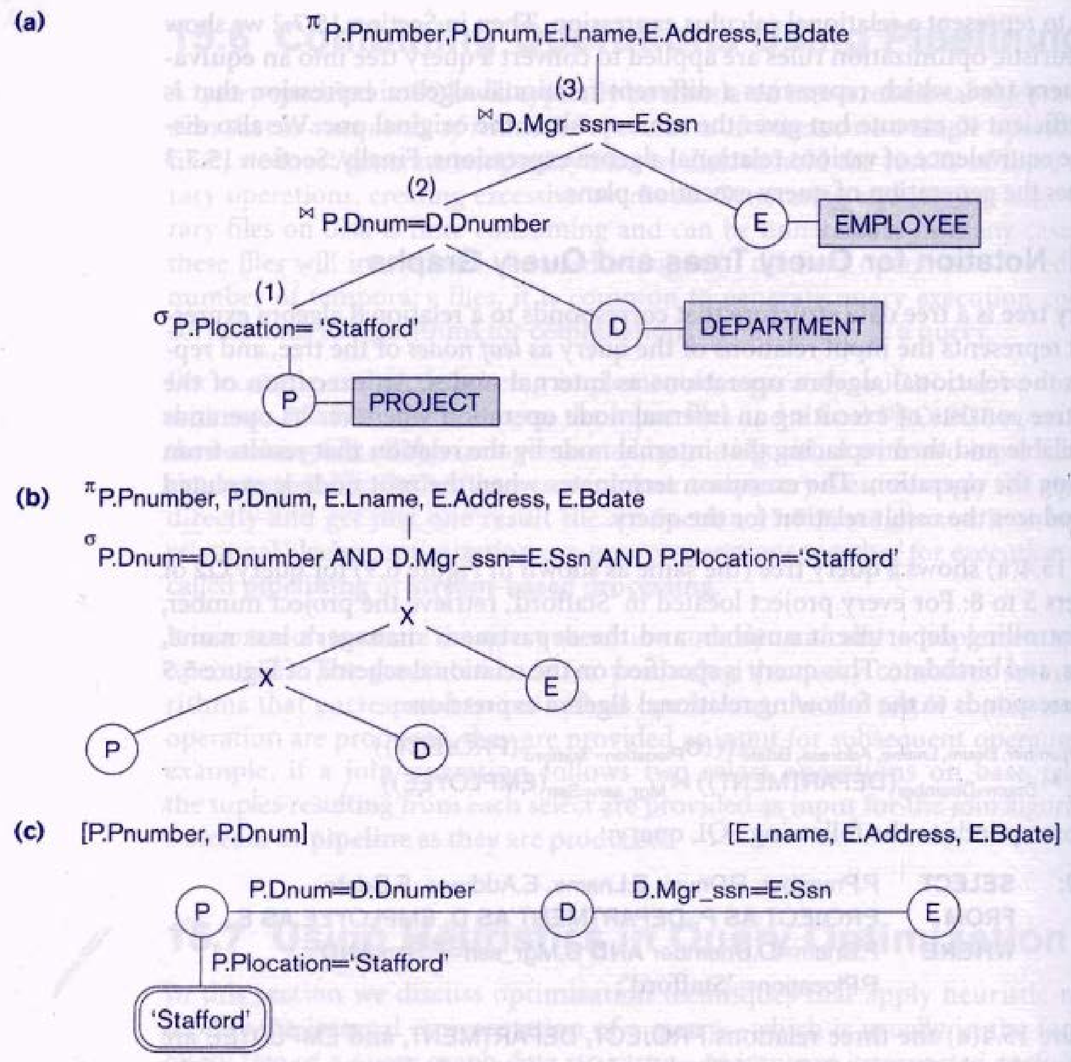

(a)
π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
⋈ D.Mgr_ssn=E.Ssn
(2)
⋈ P.Dnum=D.Dnumber
E
EMPLOYEE
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT
(b)
π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D
(c)
[P.Pnumber, P.Dnum]
[E.Lname, E.Address, E.Bdate]
P.Dnum=D.Dnumber
D.Mgr_ssn=E.Ssn
P
D
E
P.Plocation='Stafford'
'Stafford'

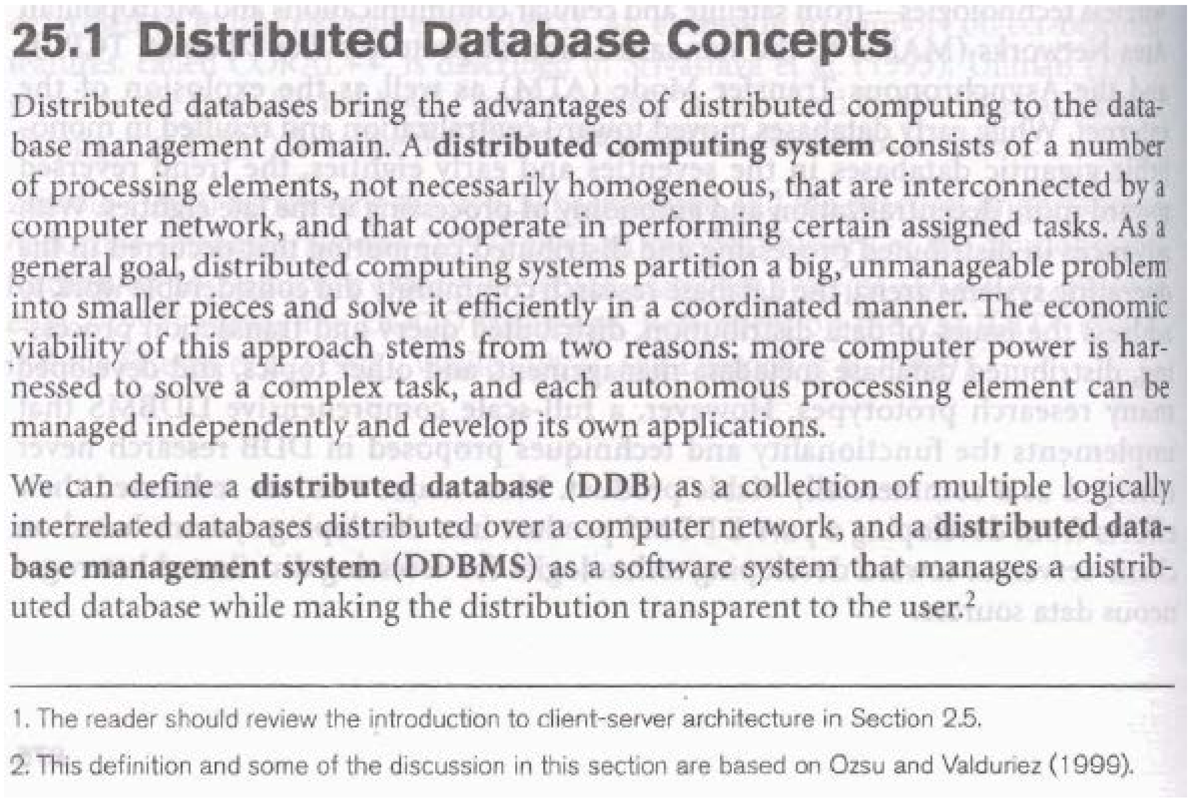
Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query.[15] Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

15. Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 6.6.5.

b. Explain the components of distributed database. (6)

Answer:



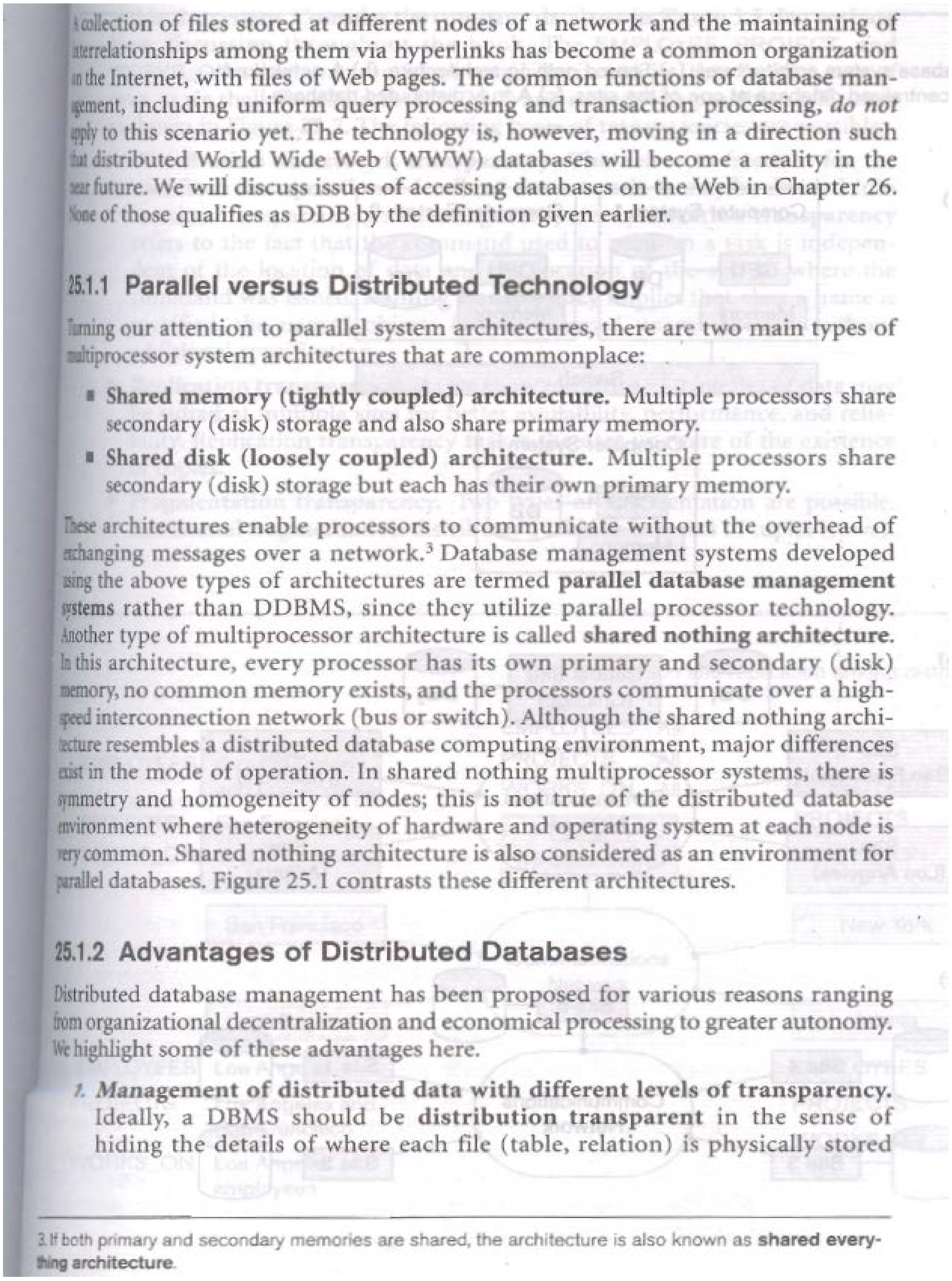
25.1 Distributed Database Concepts

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: more computer power is harnessed to solve a complex task, and 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]

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).



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. 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 will 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.

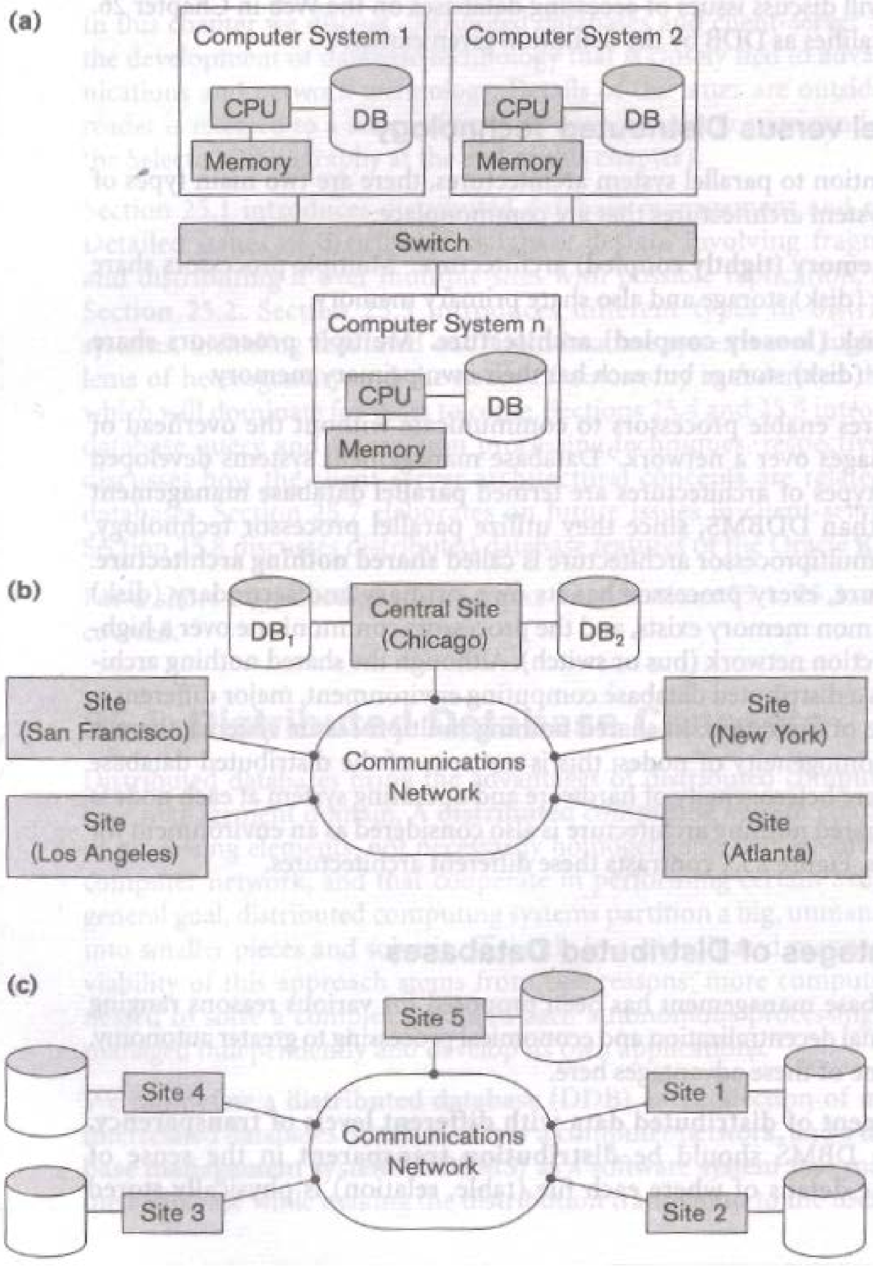
25.1.2 Advantages of Distributed Databases

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

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

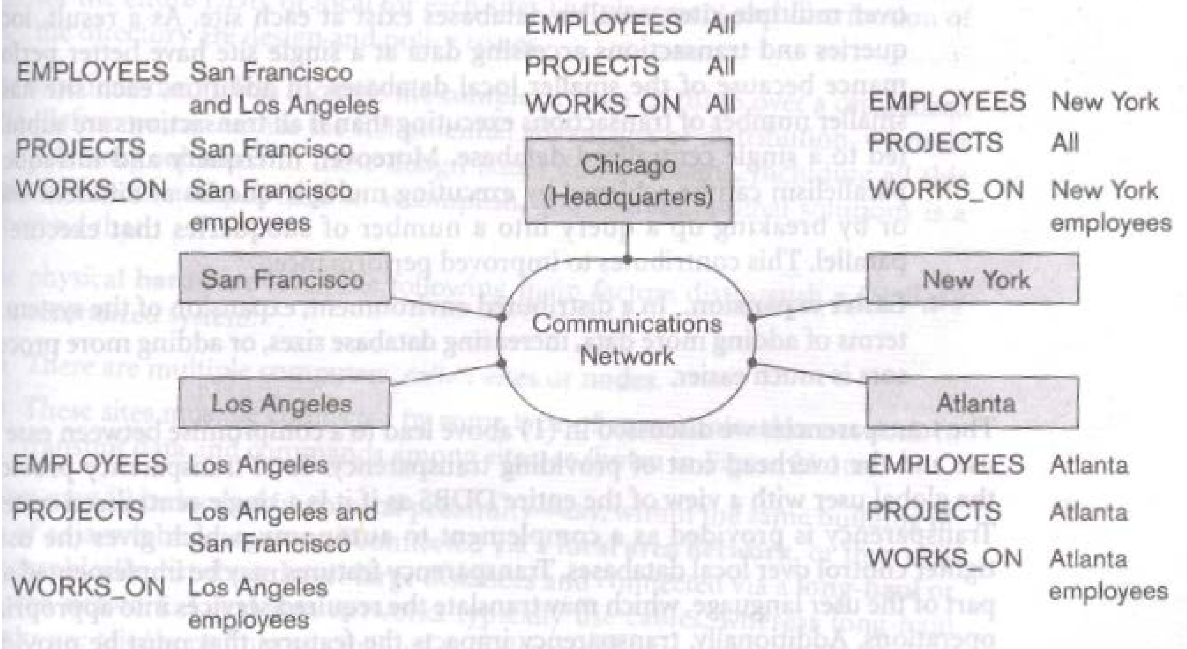
Figure 25.1
Some different database system architectures. (a) Shared nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.



within the system. Consider the company database in Figure 5.5 that we have been discussing throughout the 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).

Figure 25.2
Data distribution and replication among distributed databases.

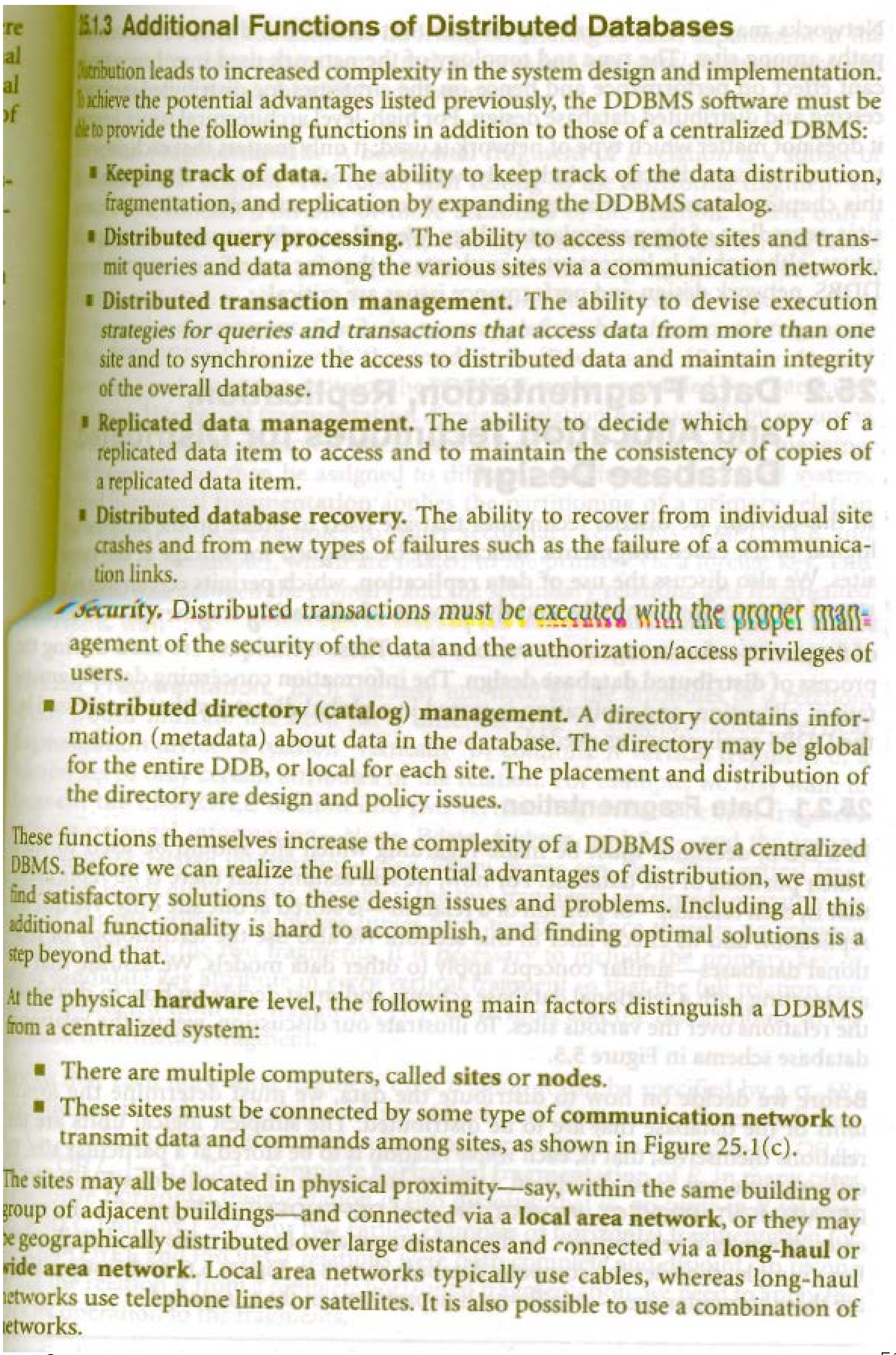


Vertical fragmentation distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. A global query by the user must be transformed into several fragment queries. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include **design transparency** and **execution transparency**—referring to freedom from knowing how the distributed database is designed and where a transaction executes.

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 local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations. Additionally, transparency impacts the features that must be provided by the operating system and the DBMS.



5.1.3 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:

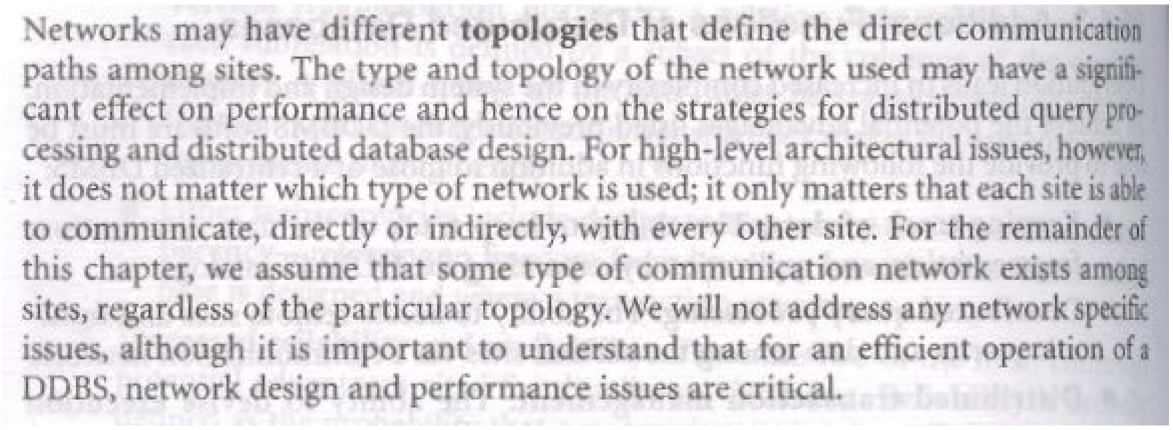
- **Keeping track of data.** The ability to keep track of the data distribution, fragmentation, and replication by expanding the DDBMS catalog.
- **Distributed query processing.** The ability to access remote sites and transmit queries and data among the various sites via a communication network.
- **Distributed 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 25.1(c).

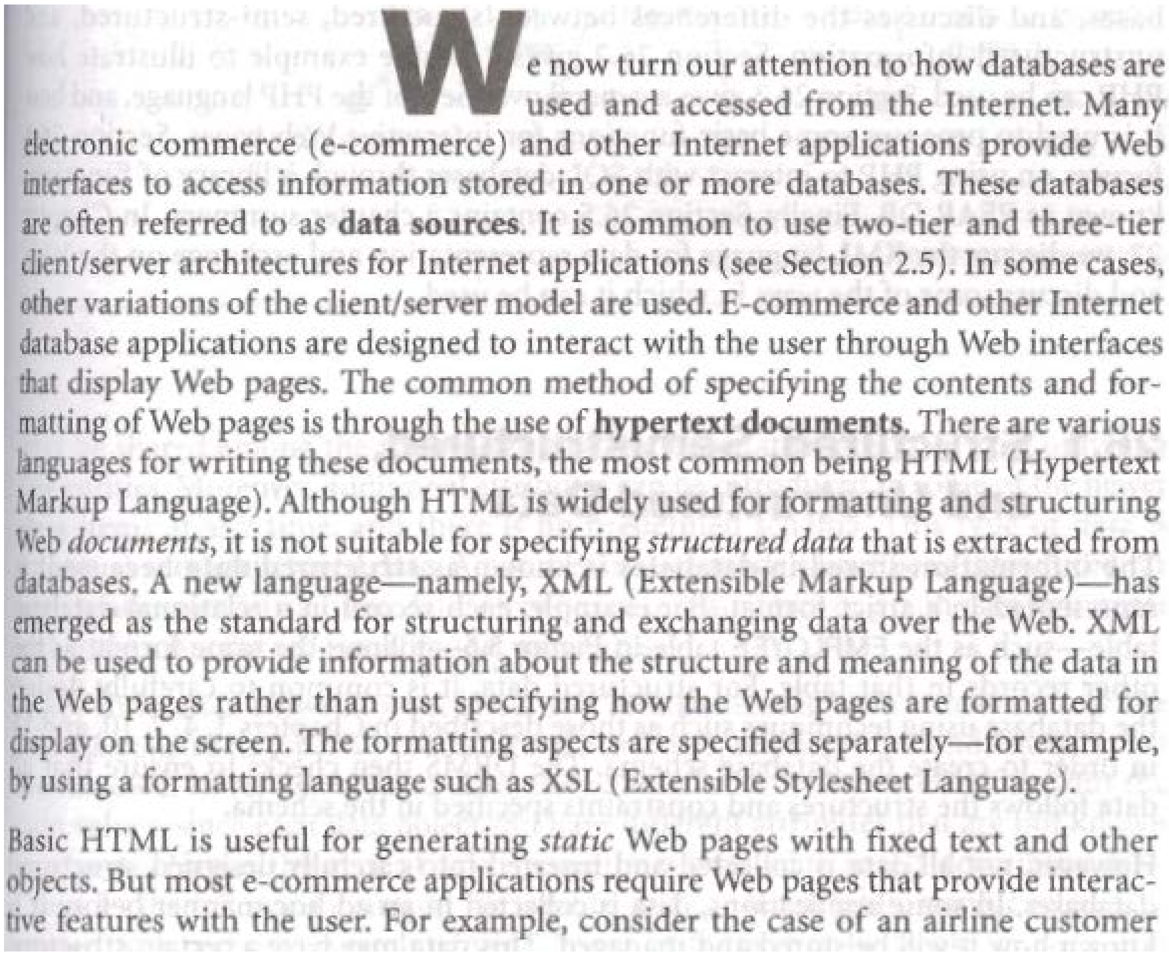
The sites may all be located in physical proximity—say, within the same building or group of adjacent buildings—and connected via a **local area network**, or they may e geographically distributed over large distances and connected via a **long-haul** or ide **area network**. Local area networks typically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



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 critical.

c. Explain Web database. **(6)**

Answer:



We now turn our attention to how databases are used and accessed from the Internet. Many electronic commerce (e-commerce) and other Internet applications provide Web interfaces to access information stored in one or more databases. These databases are often referred to as **data sources**. It is common to use two-tier and three-tier client/server architectures for Internet applications (see Section 2.5). In some cases, other variations of the client/server model are used. E-commerce and other Internet database applications are designed to interact with the user through Web interfaces that display Web pages. The common method of specifying the contents and formatting of Web pages is through the use of **hypertext documents**. There are various languages for writing these documents, the most common being HTML (Hypertext Markup Language). Although HTML is widely used for formatting and structuring Web *documents*, it is not suitable for specifying *structured data* that is extracted from databases. A new language—namely, XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web. XML can be used to provide information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen. The formatting aspects are specified separately—for example, by using a formatting language such as XSL (Extensible Stylesheet Language).

Basic HTML is useful for generating *static* Web pages with fixed text and other objects. But most e-commerce applications require Web pages that provide interactive features with the user. For example, consider the case of an airline customer who wants to check the arrival time and gate information of a particular flight. The user may enter information such as a date and flight number in certain form fields of the Web page. The Web program must now submit a query to the airline database to retrieve this information, and then display it. Such Web pages, where part of the information is extracted from databases or other data sources are called *dynamic* Web pages, because the data extracted and displayed each time will be for different flights and dates.

There are various techniques for programming dynamic features into Web pages. We will focus on one technique here, which is based on using the PHP open source scripting language. PHP has recently experienced widespread use. The interpreters for PHP are provided free of charge, and are written in the C language so they are available on most computer platforms. A PHP interpreter provides a Hypertext Preprocessor, which will execute PHP commands in a text file and create the desired HTML file. To access databases, a library of PHP functions needs to be included in the PHP interpreter as we will discuss in Section 26.4. PHP programs are executed on the Web server computer. This is in contrast to some scripting languages, such as JavaScript that are executed on the client computer.

This chapter is organized as follows. Section 26.1 discusses how the information displayed through Web pages differs from the information stored in structured databases, and discusses the differences between structured, semi-structured, and unstructured information. Section 26.2 gives a simple example to illustrate how PHP can be used. Section 26.3 gives a general overview of the PHP language, and how it is used to program some basic functions for interactive Web pages. Section 26.4 focuses on using PHP to interact with SQL databases through a library of functions known as PEAR DB. Finally, Section 26.5 contains a chapter summary. In Chapter 27, we discuss the XML language for data representation and exchange on the Web, and discuss some of the ways in which it can be used.

26.1 Structured, Semistructured, and Unstructured Data

The information stored in databases is known as **structured data** because it is represented in a strict format. For example, each record in a relational database table—such as the EMPLOYEE table in Figure 5.6—follows the same format as the other records in that table. For structured data, it is common to carefully design the database using techniques such as those described in Chapters 3, 4, 7, 10, and 11 in order to create the database schema. The DBMS then checks to ensure that all data follows the structures and constraints specified in the schema.

However, not all data is collected and inserted into carefully designed structured databases. In some applications, data is collected in an ad hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have identical structure. Some attributes

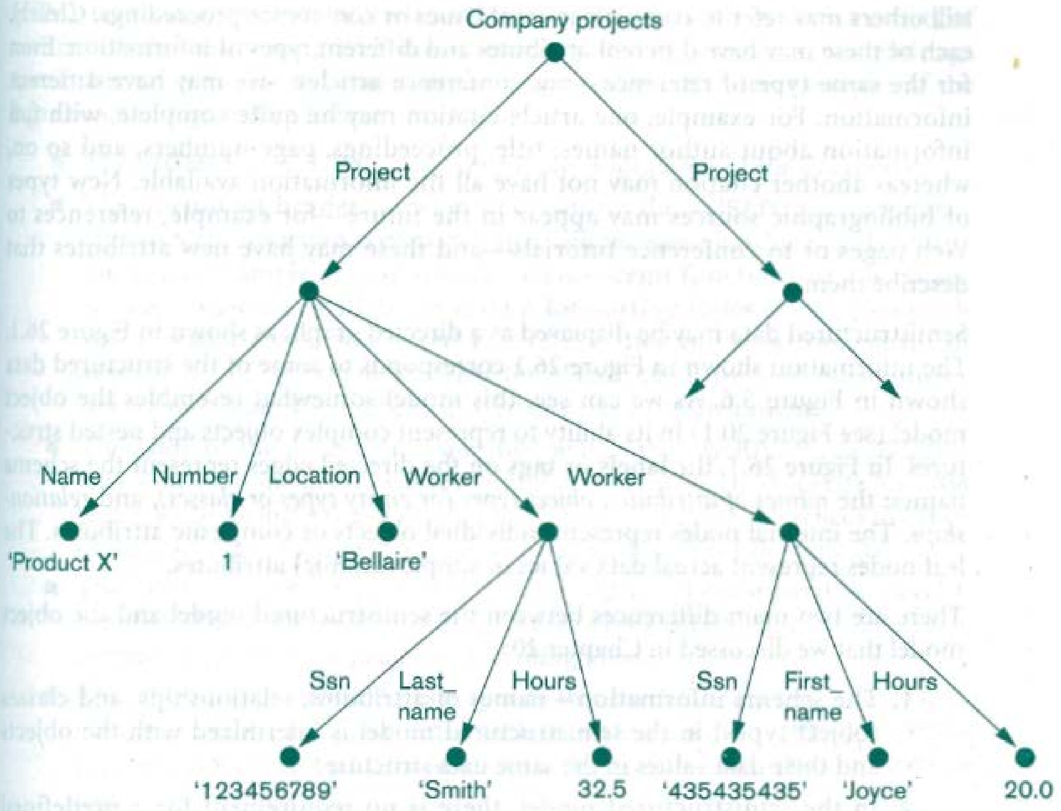


Figure 26.1
Representing semistructured data as a graph.

may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no predefined schema. This type of data is known as **semistructured data.** A number of data models have been introduced for representing semistructured data, often based on using tree or graph data structures rather than the flat relational model structures.

A key difference between structured and semistructured data concerns how the schema constructs (such as the names of attributes, relationships, and entity types) are handled. In semistructured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data.** Consider the following example. We want to collect a list of bibliographic references related to a certain research project. Some of these may be books or technical reports, others may be research articles in journals or conference proceedings, and still others may refer to complete journal issues or conference proceedings. Clearly, each of these may have different attributes and different types of information. Even for the same type of reference—say, conference articles—we may have different information. For example, one article citation may be quite complete, with full information about author names, title, proceedings, page numbers, and so on, whereas another citation may not have all the information available. New types of bibliographic sources may appear in the future—for example, references to Web pages or to conference tutorials—and these may have new attributes that describe them.

Semistructured data may be displayed as a directed graph, as shown in Figure 26.1. The information shown in Figure 26.1 corresponds to some of the structured data shown in Figure 5.6. As we can see, this model somewhat resembles the object model (see Figure 20.1) in its ability to represent complex objects and nested structures. In Figure 26.1, the **labels** or **tags** on the directed edges represent the schema names: the *names of attributes, object types (or entity types or classes)*, and *relationships*. The internal nodes represent individual objects or composite attributes. The leaf nodes represent actual data values of simple (atomic) attributes.

There are two main differences between the semistructured model and the object model that we discussed in Chapter 20:

1. The schema information—names of attributes, relationships, and classes (object types) in the semistructured model is intermixed with the objects and their data values in the same data structure.
2. In the semistructured model, there is no requirement for a predefined schema to which the data objects must conform.

In addition to structured and semistructured data, a third category exists, known as **unstructured data** because there is very limited indication of the type of data. A typical example is a text document that contains information embedded within it. Web pages in HTML that contain some data are considered to be unstructured data. Consider part of an HTML file, shown in Figure 26.2. Text that appears between angled brackets, <. . .>, is an **HTML tag**. A tag with a backslash, </. . .>, indicates an **end tag**, which represents the ending of the effect of a matching **start tag**. The tags **mark up** the document[1] in order to instruct an HTML processor how to display the text between a start tag and a matching end tag. Hence, the tags specify document formatting rather than the meaning of the various data elements in the document. HTML tags specify information, such as font size and style (boldface, italics, and so on), color, heading levels in documents, and so on. Some tags provide text structuring in documents, such as specifying a numbered or unnumbered list or a table. Even these structuring tags specify that the embedded textual data is to be displayed in a certain manner, rather than indicating the type of data represented in the table.

1. That is why it is known as Hypertext *Markup* Language.

HTML uses a large number of predefined tags, which are used to specify a variety of commands for formatting Web documents for display. The start and end tags specify the range of text to be formatted by each command. A few examples of the tags shown in Figure 26.2 follow:

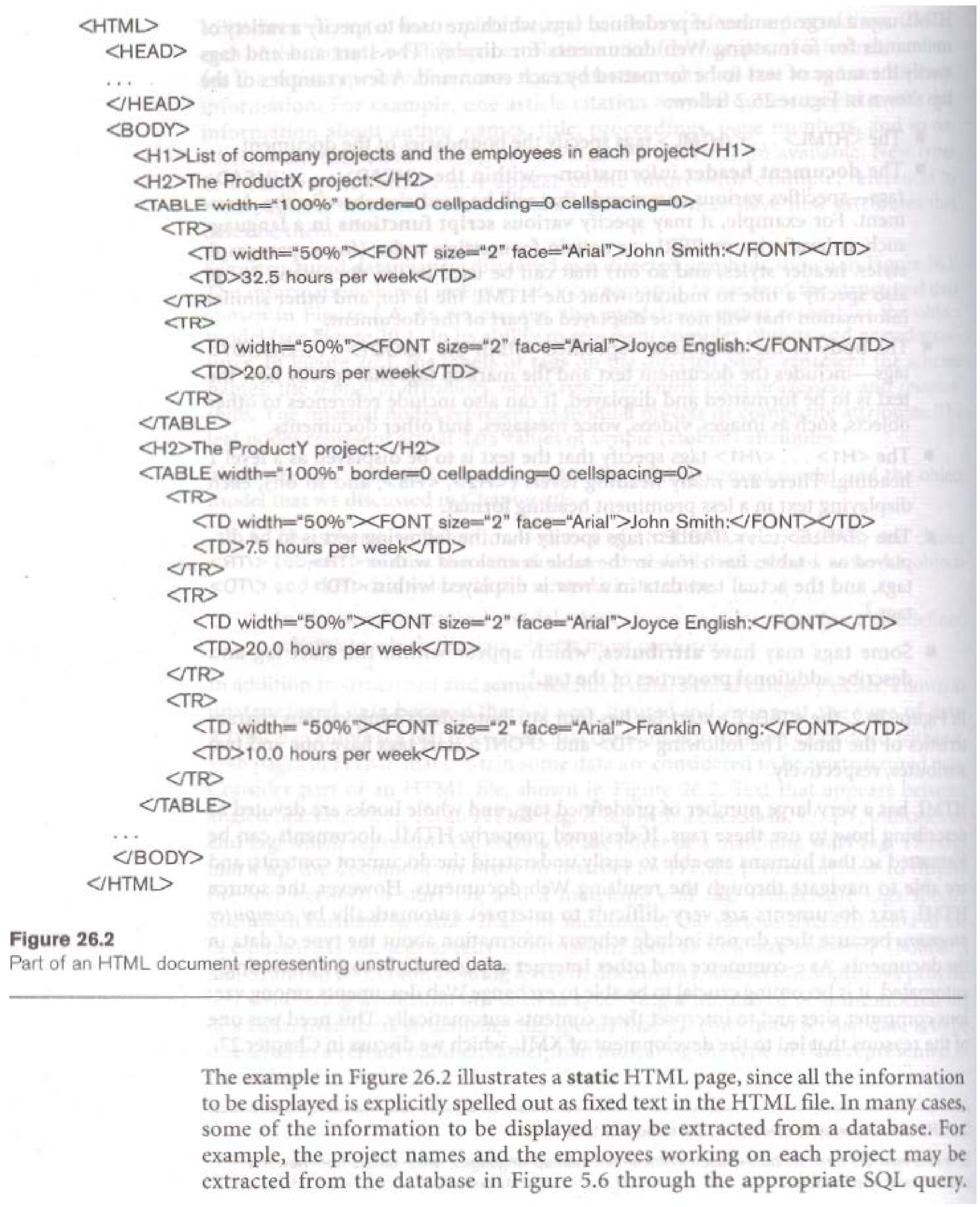
- The <HTML> . . . </HTML> tags specify the boundaries of the document.
- The **document header** information—within the <HEAD> . . . </HEAD> tags—specifies various commands that will be used elsewhere in the document. For example, it may specify various **script functions** in a language such as JavaScript or PERL, or certain **formatting styles** (fonts, paragraph styles, header styles, and so on) that can be used in the document. It can also specify a title to indicate what the HTML file is for, and other similar information that will not be displayed as part of the document.
- The **body** of the document—specified within the <BODY> . . . </BODY> tags—includes the document text and the markup tags that specify how the text is to be formatted and displayed. It can also include references to other objects, such as images, videos, voice messages, and other documents.
- The <H1> . . . </H1> tags specify that the text is to be displayed as a level 1 heading. There are many heading levels (<H2>, <H3>, and so on), each displaying text in a less prominent heading format.
- The <TABLE> . . . </TABLE> tags specify that the following text is to be displayed as a table. Each row in the table is enclosed within <TR> . . . </TR> tags, and the actual text data in a row is displayed within <TD> . . . </TD> tags.[2]
- Some tags may have **attributes**, which appear within the start tag and describe additional properties of the tag.[3]

In Figure 26.2, the <TABLE> start tag has four attributes describing various characteristics of the table. The following <TD> and <FONT> start tags have one and two attributes, respectively.

HTML has a very large number of predefined tags, and whole books are devoted to describing how to use these tags. If designed properly, HTML documents can be formatted so that humans are able to easily understand the document contents, and are able to navigate through the resulting Web documents. However, the source HTML text documents are very difficult to interpret automatically by *computer programs* because they do not include schema information about the type of data in the documents. As e-commerce and other Internet applications become increasingly automated, it is becoming crucial to be able to exchange Web documents among various computer sites and to interpret their contents automatically. This need was one of the reasons that led to the development of XML, which we discuss in Chapter 27.

2. <TR> stands for table row and <TD> stands for table data.

3. This is how the term *attribute* is used in document markup languages, which differs from how it is used in database models.



```
<HTML>
   <HEAD>
   ...
   </HEAD>
   <BODY>
      <H1>List of company projects and the employees in each project</H1>
      <H2>The ProductX project:</H2>
      <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
         <TR>
            <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
            <TD>32.5 hours per week</TD>
         </TR>
         <TR>
            <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
            <TD>20.0 hours per week</TD>
         </TR>
      </TABLE>
      <H2>The ProductY project:</H2>
      <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
         <TR>
            <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
            <TD>7.5 hours per week</TD>
         </TR>
         <TR>
            <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
            <TD>20.0 hours per week</TD>
         </TR>
         <TR>
            <TD width= "50%"><FONT size="2" face="Arial">Franklin Wong:</FONT></TD>
            <TD>10.0 hours per week</TD>
         </TR>
      </TABLE>
   ...
   </BODY>
</HTML>
```

Figure 26.2
Part of an HTML document representing unstructured data.

The example in Figure 26.2 illustrates a **static** HTML page, since all the information to be displayed is explicitly spelled out as fixed text in the HTML file. In many cases, some of the information to be displayed may be extracted from a database. For example, the project names and the employees working on each project may be extracted from the database in Figure 5.6 through the appropriate SQL query.

TEXT BOOK

I. Elmasri & Navathe, "Fundamental of Database Systems", Addison Wesley, 5th Edition, 2006
II. R Ramakrishnan & J Gehrke, Database Management Systems, McGraw Hill, Third Edition, 2002