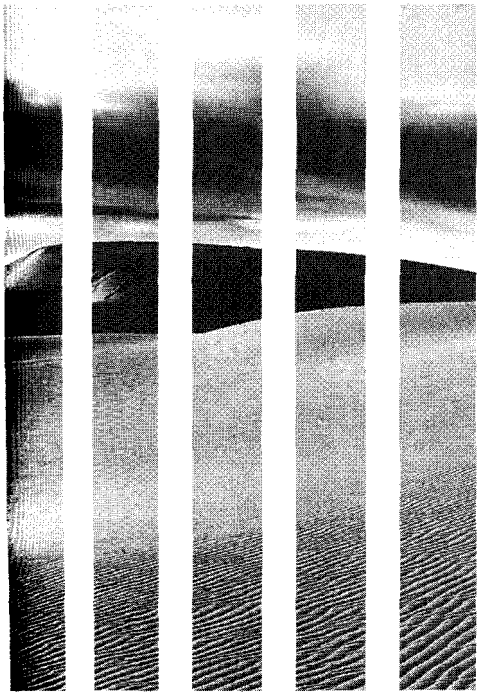


3

DATABASE DESIGN THEORY AND METHODOLOGY

10

Functional Dependencies and Normalization for Relational Databases

In Chapters 5 through 9, we presented various aspects of the relational model and the languages associated with it. Each *relation schema* consists of a number of attributes, and the *relational database schema* consists of a number of relation schemas. So far, we have assumed that attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or enhanced ER (EER) or some other conceptual data model. These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations when the mapping procedures in Chapter 7 are followed. However, we still need some formal measure of why one grouping of attributes into a relation schema may be better than another. So far in our discussion of conceptual design in Chapters 3 and 4 and its mapping into the relational model in Chapter 7, we have not developed any measure of appropriateness or "goodness" to measure the quality of the design, other than the intuition of the designer. In this chapter we discuss some of the theory that has been developed with the goal of evaluating relational schemas for design quality—that is, to measure formally why one set of groupings of attributes into relation schemas is better than another.

There are two levels at which we can discuss the "goodness" of relation schemas. The first is the **logical (or conceptual) level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their

queries correctly. The second is the **implementation** (or **storage**) **level**—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations). The relational database design theory developed in this chapter applies mainly to *base relations*, although some criteria of appropriateness also apply to views, as shown in Section 10.1.

As with many design problems, database design may be performed using two approaches: bottom-up or top-down. A **bottom-up design methodology** (also called *design by synthesis*) considers the basic relationships *among individual attributes* as the starting point and uses those to construct relation schemas. This approach is not very popular in practice[1] because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. In contrast, a **top-down design methodology** (also called *design by analysis*) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met. The theory described in this chapter is applicable to both the top-down and bottom-up design approaches, but is more practical when used with the top-down approach.

We start this chapter by informally discussing some criteria for good and bad relation schemas in Section 10.1. Then in Section 10.2 we define the concept of *functional dependency*, a formal constraint among attributes that is the main tool for formally measuring the appropriateness of attribute groupings into relation schemas. Properties of functional dependencies are also studied and analyzed. In Section 10.3 we show how functional dependencies can be used to group attributes into relation schemas that are in a *normal form*. A relation schema is in a normal form when it satisfies certain desirable properties. The process of *normalization* consists of analyzing relations to meet increasingly more stringent normal forms leading to progressively better groupings of attributes. Normal forms are specified in terms of functional dependencies—which are identified by the database designer—and key attributes of relation schemas. In Section 10.4 we discuss more general definitions of normal forms that can be directly applied to any given design and do not require step-by-step analysis and normalization.

Chapter 11 continues the development of the theory related to the design of good relational schemas. Whereas in Chapter 10 we concentrate on the normal forms for single relation schemas, in Chapter 11 we will discuss measures of appropriateness for a whole set of relation schemas that together form a *relational database schema*. We specify two such properties—the nonadditive (lossless) join property and the dependency preservation property—and discuss bottom-up design algorithms for relational database design that start off with a given set of functional dependencies and achieve certain normal forms while maintaining the aforementioned properties. A general algorithm that tests whether or not a decomposition has the lossless join property (Algorithm 11.1) is

1. An exception in which this approach is used in practice is based on a model called the binary relational model. An example is the NIAM methodology (Verheijen and VanBekkum 1982).

also presented. In Chapter 11 we also define additional types of dependencies and advanced normal forms that further enhance the "goodness" of relation schemas.

For the reader interested in only an informal introduction to normalization, Sections 10.2.3, 10.2.4, and 10.2.5 may be skipped. If Chapter 11 is not covered in a course, we recommend a quick introduction to the desirable properties of decomposition from Section 11.1 and a discussion of Property LJ1 in addition to Chapter 10.

10.1 Informal Design Guidelines for Relation Schemas

We discuss four *informal measures* of quality for relation schema design in this section:

- Semantics of the attributes
- Reducing the redundant values in tuples
- Reducing the null values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we shall see.

10.1.1 Semantics of the Relation Attributes

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. In Chapter 5 we discussed how each relation can be interpreted as a set of facts or statements. This meaning, or **semantics,** specifies how to interpret the attribute values stored in a tuple of the relation—in other words, how the attribute values in a tuple relate to one another. If the conceptual design is done carefully, followed by a systematic mapping into relations, most of the semantics will have been accounted for and the resulting design should have a clear meaning.

In general, the easier it is to explain the semantics of the relation, the better the relation schema design will be. To illustrate this, consider Figure 10.1, a simplified version of the COMPANY relational database schema of Figure 5.5, and Figure 10.2, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (ENAME), social security number (SSN), birth date (BDATE), and address (ADDRESS), and the number of the department that the employee works for (DNUMBER). The DNUMBER attribute is a foreign key that represents an *implicit relationship* between EMPLOYEE and DEPARTMENT. The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute DMGRSSN of DEPARTMENT relates a department to the employee who is its manager, while DNUM of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's atributes can be explained is an *informal measure* of how well the relation is designed.

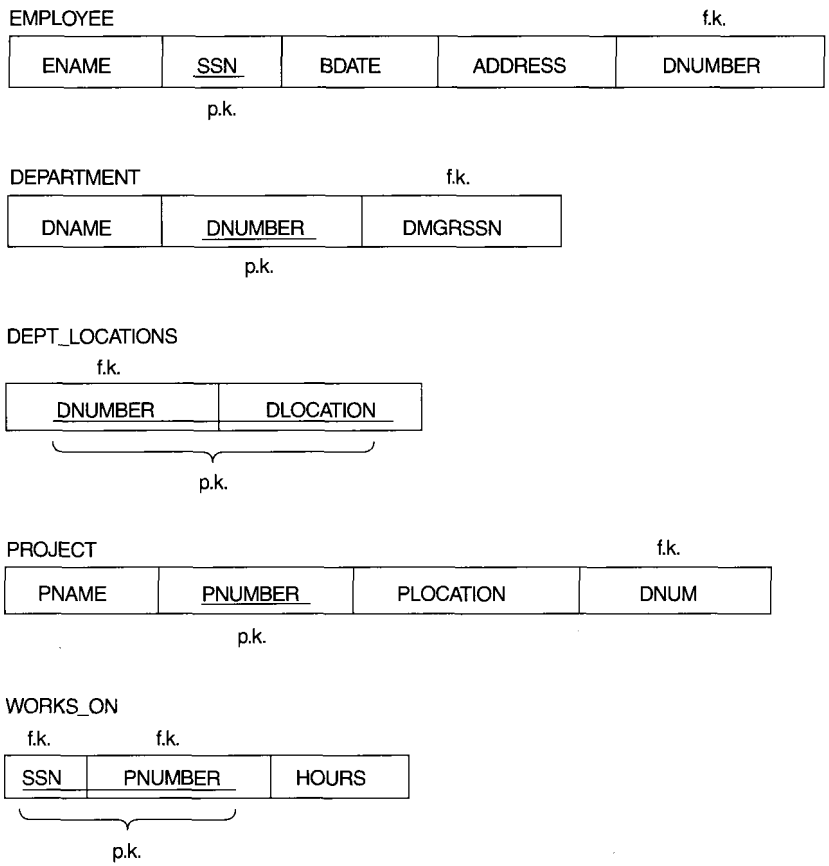


FIGURE 10.1 A simplified COMPANY relational database schema.

The semantics of the other two relation schemas in Figure 10.1 are slightly more complex. Each tuple in DEPT_LOCATIONS gives a department number (DNUMBER) and *one of* the locations of the department (DLOCATION). Each tuple in WORKS_ON gives an employee social security number (SSN), the project number of *one of* the projects that the employee works on (PNUMBER), and the number of hours per week that the employee works on that project (HOURS). However, both schemas have a well-defined and unambiguous interpretation. The schema DEPT_LOCATIONS represents a multivalued attribute of DEPARTMENT, whereas WORKS_ON represents an M:N relationship between EMPLOYEE and PROJECT. Hence, all the relation schemas in Figure 10.1 may be considered as easy to explain and hence good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

GUIDELINE 1. Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relation-

EMPLOYEE

ENAME	SSN	BDATE	ADDRESS	DNUMBER
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5
Zelaya,Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4
Narayan,Remesh K.	666884444	1962-09-15	975 Fire Oak,Humble,TX	5
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1

DEPARTMENT

DNAME	DNUMBER	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

DNUMBER	DLOCATION
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

SSN	PNUMBER	HOURS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	null

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

FIGURE 10.2 Example database state for the relational database schema of Figure 10.1.

ship type, it is straightforward to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

The relation schemas in Figures 10.3a and 10.3b also have clear semantics. (The reader should ignore the lines under the relations for now; they are used to illustrate functional dependency notation, discussed in Section 10.2.) A tuple in the EMP_DEPT

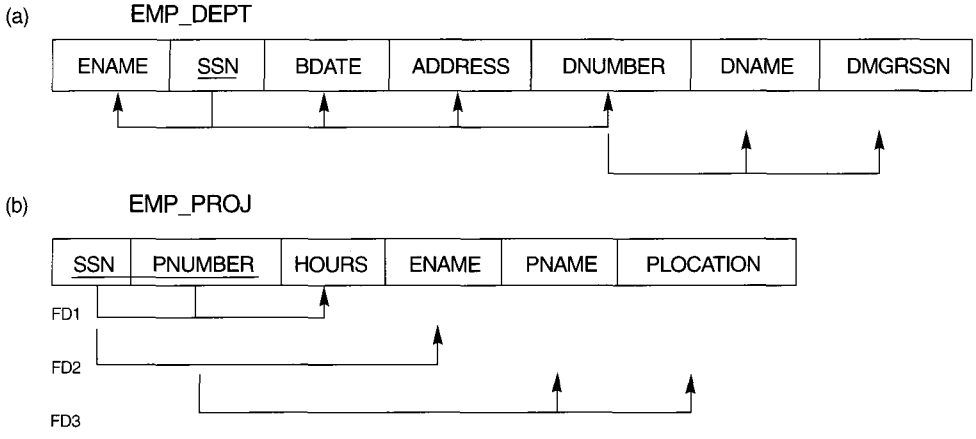


FIGURE 10.3 Two relation schemas suffering from update anomalies.

relation schema of Figure 10.3a represents a single employee but includes additional information—namely, the name (DNAME) of the department for which the employee works and the social security number (DMGRSSN) of the department manager. For the EMP_PROJ relation of Figure 10.3b, each tuple relates an employee to a project but also includes the employee name (ENAME), project name (PNAME), and project location (PLOCATION). Although there is nothing wrong logically with these two relations, they are considered poor designs because they violate Guideline 1 by mixing attributes from distinct real-world entities; EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects. They may be used as views, but they cause problems when used as base relations, as we discuss in the following section.

10.1.2 Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 10.2 with that for an EMP_DEPT base relation in Figure 10.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (DNUMBER, DNAME, DMGRSSN) are repeated for *every employee who works for that department*. In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 10.2. Only the department number (DNUMBER) is repeated in the EMPLOYEE relation for each employee who works in that department. Similar comments apply to the EMP_PROJ relation (Figure 10.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

redundancy

EMP_DEPT

ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Research	333445555
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4	Administration	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administration	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Research	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Research	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administration	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Headquarters	888665555

redundancy redundancy

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	32.5	Smith,John B.	ProductX	Bellaire
123456789	2	7.5	Smith,John B.	ProductY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProductZ	Houston
453453453	1	20.0	English,Joyce A.	ProductX	Bellaire
453453453	2	20.0	English,Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong,Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong,Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya,Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar,Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar,Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace,Jennifer S.	Reorganization	Houston
888665555	20	null	Borg,James E.	Reorganization	Houston

FIGURE 10.4 Example states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 10.2. These may be stored as base relations for performance reasons.

Another serious problem with using the relations in Figure 10.4 as base relations is the problem of **update anomalies.** These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.[2]

Insertion Anomalies. Insertion anomolies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or nulls (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter the attribute values of department 5 correctly so

2. These anomalies were identified by Codd (1972a) to justify the need for normalization of relations, as we shall discuss in Section 10.3.

that they are *consistent* with values for department 5 in other tuples in EMP_DEPT. In the design of Figure 10.2, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.

- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place null values in the attributes for employee. This causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity—not a department entity. Moreover, when the first employee is assigned to that department, we do not need this tuple with null values any more. This problem does not occur in the design of Figure 10.2, because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation discussed earlier. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the database of Figure 10.2 because DEPARTMENT tuples are stored separately.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.[3]

Based on the preceding three anomalies, we can state the guideline that follows.

GUIDELINE 2. Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines. Sections 10.2 through 10.4 provide these needed formal concepts. It is important to note that these guidelines may sometimes *have to be violated* in order to *improve the performance* of certain queries. For example, if an important query retrieves information concerning the department of an employee along with employee attributes, the EMP_DEPT schema may be used as a base relation. However, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates) so that, whenever the base relation is updated, we do not end up with inconsistencies. In general, it is advisable to use anomaly-free base relations and to specify views that include the joins for placing together the

3. This is not as serious as the other problems, because all tuples can be updated by a single SQL query.

attributes frequently referenced in important queries. This reduces the number of JOIN terms specified in the query, making it simpler to write the query correctly, and in many cases it improves the performance.[4]

10.1.3 Null Values in Tuples

In some schema designs we may group many attributes together into a "fat" relation. If many of the attributes do not apply to all tuples in the relation, we end up with many nulls in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.[5] Another problem with nulls is how to account for them when aggregate operations such as COUNT or SUM are applied. Moreover, nulls can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple.
- The attribute value for this tuple is *unknown*.
- The value is *known but absent*; that is, it has not been recorded yet.

Having the same representation for all nulls compromises the different meanings they may have. Therefore, we may state another guideline.

GUIDELINE 3. As far as possible, avoid placing attributes in a base relation whose values may frequently be null. If nulls are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Using space efficiently and avoiding joins are the two overriding criteria that determine whether to include the columns that may have nulls in a relation or to have a separate relation for those columns (with the appropriate key columns). For example, if only 10 percent of employees have individual offices, there is little justification for including an attribute OFFICE_NUMBER in the EMPLOYEE relation; rather, a relation EMP_OFFICES(ESSN, OFFICE_NUMBER) can be created to include tuples for only the employees with individual offices.

10.1.4 Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 10.5a, which can be used instead of the single EMP_PROJ relation of Figure 10.3b. A tuple in EMP_LOCS means that the employee whose name is ENAME works on *some project* whose location is PLOCATION. A tuple

4. The performance of a query specified on a view that is the join of several base relations depends on how the DBMS implements the view. Many RDBMSS materialize a frequently used view so that they do not have to perform the joins often. The DBMS remains responsible for updating the materialized view (either immediately or periodically) whenever the base relations are updated.

5. This is because inner and outer joins produce different results when nulls are involved in joins. The users must thus be aware of the different meanings of the various types of joins. Although this is reasonable for sophisticated users, it may be difficult for others.

(a)

EMP_LOCS

ENAME	PLOCATION

p.k.

EMP_PROJ1

SSN	PNUMBER	HOURS	PNAME	PLOCATION

p.k.

(b)

EMP_LOCS

ENAME	PLOCATION
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg,James E.	Houston

EMP_PROJ1

SSN	PNUMBER	HOURS	PNAME	PLOCATION
123456789	1	32.5	Product X	Bellaire
123456789	2	7.5	Product Y	Sugarland
666884444	3	40.0	Product Z	Houston
453453453	1	20.0	Product X	Bellaire
453453453	2	20.0	Product Y	Sugarland
333445555	2	10.0	Product Y	Sugarland
333445555	3	10.0	Product Z	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	null	Reorganization	Houston

FIGURE 10.5 Particularly poor design for the EMP_PROJ relation of Figure 10.3b. (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 10.4 onto the relations EMP_LOCS and EMP_PROJ1.

in EMP_PROJ1 means that the employee whose social security number is SSN works HOURS per week on the project whose name, number, and location are PNAME, PNUMBER, and PLOCATION. Figure 10.5b shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the EMP_PROJ relation of Figure 10.4, which are obtained by applying the appropriate PROJECT (π) operations to EMP_PROJ (ignore the dotted lines in Figure 10.5b for now).

Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design, because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ. In Figure 10.6, the result of applying the join to only the tuples *above* the dotted lines in Figure 10.5b is shown (to reduce the size of the resulting relation). Additional tuples that were not in EMP_PROJ are called **spurious tuples** because they represent spurious or *wrong* information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 10.6.

Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because, when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case PLOCATION is the attribute that relates EMP_LOCS and EMP_PROJ1, and PLOCATION is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1. We can now informally state another design guideline.

SSN	PNUMBER	HOURS	PNAME	PLOCATION	ENAME
123456789	1	32.5	ProductX	Bellaire	Smith,John B.
123456789	1	32.5	ProductX	Bellaire	English,Joyce A.
123456789	2	7.5	ProductY	Sugarland	Smith,John B.
123456789	2	7.5	ProductY	Sugarland	English,Joyce A.
123456789	2	7.5	ProductY	Sugarland	Wong,Franklin T.
666884444	3	40.0	ProductZ	Houston	Narayan,Ramesh K.
666884444	3	40.0	ProductZ	Houston	Wong,Franklin T.
453453453	1	20.0	ProductX	Bellaire	Smith,John B.
453453453	1	20.0	ProductX	Bellaire	English,Joyce A.
453453453	2	20.0	ProductY	Sugarland	Smith,John B.
453453453	2	20.0	ProductY	Sugarland	English,Joyce A.
453453453	2	20.0	ProductY	Sugarland	Wong,Franklin T.
333445555	2	10.0	ProductY	Sugarland	Smith,John B.
333445555	2	10.0	ProductY	Sugarland	English,Joyce A.
333445555	2	10.0	ProductY	Sugarland	Wong,Franklin T.
333445555	3	10.0	ProductZ	Houston	Narayan,Ramesh K.
333445555	3	10.0	ProductZ	Houston	Wong,Franklin T.
333445555	10	10.0	Computerization	Stafford	Wong,Franklin T.
333445555	20	10.0	Reorganization	Houston	Narayan,Ramesh K.
333445555	20	10.0	Reorganization	Houston	Wong,Franklin T.

FIGURE 10.6 Result of applying NATURAL JOIN to the tuples above the dotted lines in EMP_PROJ1 and EMP_LOCS of Figure 10.5. Generated spurious tuples are marked by asterisks.

GUIDELINE 4. Design relation schemas so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys in a way that guarantees that no spurious tuples are generated. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations, because joining on such attributes may produce spurious tuples.

This informal guideline obviously needs to be stated more formally. In Chapter 11 we discuss a formal condition, called the nonadditive (or lossless) join property, that guarantees that certain joins do not produce spurious tuples.

10.1.5 Summary and Discussion of Design Guidelines

In Sections 10.1.1 through 10.1.4, we informally discussed situations that lead to problematic relation schemas, and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to nulls and the difficulty of performing aggregation operations and joins due to null values
- Generation of invalid and spurious data during joins on improperly related base relations

In the rest of this chapter we present formal concepts and theory that may be used to define the "goodness" and "badness" of *individual* relation schemas more precisely. We first discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas. In Chapter 11, we define additional normal forms that which are based on additional types of data dependencies called multivalued dependencies and join dependencies.

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 $R = \{A_1,$

$A_2, \ldots, A_n\}$.[6] 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.[7]

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 $X \rightarrow Y$ 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 $X \rightarrow Y$ in R, this does not say whether or not $Y \rightarrow X$ 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. 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.

6. This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 11.

7. This assumption implies that every attribute in the database should have a *distinct name*. In Chapter 5 we prefixed attribute names by relation names to achieve uniqueness whenever attributes in distinct relations had the same name.

Consider the relation schema EMP_PROJ in Figure 10.3b; 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. Hence, 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 10.3a and 10.3b.

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 that satisfy the dependencies in F. Those other dependencies can be *inferred* or *deduced* from the FDs in F.

TEACH

TEACHER	COURSE	TEXT
Smith	Data Structures	Bartram
Smith	Data Management	Al-Nour
Hall	Compilers	Hoffman
Brown	Data Structures	Augenthaler

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

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 MANAGER_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 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 10.3a:

F = {SSN → {ENAME, BDATE, ADDRESS, DNUMBER},
DNUMBER → {DNAME, DMGRSSN}}

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

SSN → {DNAME, DMGRSSN}

SSN → SSN

DNUMBER → 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[8]): If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule[9]): $\{ X \rightarrow Y \} \models XZ \rightarrow YZ$.

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

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

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

9. 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.

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

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

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

PROOF OF IR5 (USING IR1 THROUGH IR3)

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

PROOF OF IR6 (USING IR1 THROUGH IR3)

1. $X \rightarrow Y$ (given).
2. $WY \rightarrow Z$ (given).
3. $WX \rightarrow WY$ (using IR2 on 1 by augmenting with W).
4. $WX \rightarrow Z$ (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.**[10]

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. Thus, 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^+.

10. 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
    oldX+ := X+;
    for each functional dependency Y → Z in F do
    if X+ ⊇ Y then X+ := X+ ∪ Z;
until (X+ = oldX+);
```

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 10.3b; 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 of each line represents all those attributes that are functionally dependent on the set of attributes in the left-hand side based on the given set F.

10.2.3 Equivalence of Sets of Functional Dependencies

In this section we discuss the equivalence of two sets of functional dependencies. First, we give some preliminary definitions.

Definition. A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is **covered by** F.

Definition. Two sets of functional dependencies E and F are **equivalent** if $E^+ = F^+$. Hence, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions E covers F *and* F covers E hold.

We can determine whether F covers E by calculating X^+ *with respect to* F for each FD $X \rightarrow Y$ *in* E, and then checking whether this X^+ includes the attributes in Y. If this is the

case for *every* FD in E, then F covers E. We determine whether E and F are equivalent by checking that E covers F and F covers E.

10.2.4 Minimal Sets of Functional Dependencies

Informally, a **minimal cover** of a set of functional dependencies E is a set of functional dependencies F that satisfies the property that every dependency in E is in the closure F^+ of F. In addition, this property is lost if any dependency from the set F is removed; F must have no redundancies in it, and the dependencies in E are in a standard form. To satisfy these properties, we can formally define a set of functional dependencies F to be **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

We can think of a minimal set of dependencies as being a set of dependencies in a *standard* or *canonical form* and with *no redundancies*. Condition 1 just represents every dependency in a canonical form with a single attribute on the right-hand side.[11] Conditions 2 and 3 ensure that there are no redundancies in the dependencies either by having redundant attributes on the left-hand side of a dependency (Condition 2) or by having a dependency that can be inferred from the remaining FDs in F (Condition 3). A **minimal cover** of a set of functional dependencies E is a minimal set of dependencies F that is equivalent to E. There can be several minimal covers for a set of functional dependencies. We can always find *at least one* minimal cover F for any set of dependencies E using Algorithm 10.2.

If several sets of FDs qualify as minimal covers of E by the definition above, it is customary to use additional criteria for "minimality." For example, we can choose the minimal set with the *smallest number of dependencies* or with the smallest *total length* (the total length of a set of dependencies is calculated by concatenating the dependencies and treating them as one long character string).

Algorithm 10.2: Finding a Minimal Cover F for a Set of Functional Dependencies E

1. Set $F := E$.
2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$.
3. For each functional dependency $X \rightarrow A$ in F

11. This is a standard form to simplify the conditions and algorithms that ensure no redundancy exists in F. By using the inference rule IR4, we can convert a single dependency with multiple attributes on the right-hand side into a set of dependencies with single attributes on the right-hand side.

for each attribute B that is an element of X
if $\{ \{ F - \{X \rightarrow A\} \} \cup \{(X - \{B\}) \rightarrow A\} \}$ is equivalent to F,
then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F.

4. For each remaining functional dependency $X \rightarrow A$ in F
if $\{ F - \{X \rightarrow A\} \}$ is equivalent to F,
then remove $X \rightarrow A$ from F.

In Chapter 11 we will see how relations can be synthesized from a given set of dependencies E by first finding the minimal cover F for E.

10.3 Normal Forms Based on Primary Keys

Having studied functional dependencies and some of their properties, we are now ready to use them to specify some aspects of the semantics of relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the *normalization process* for relational schema design. Most practical relational design projects take one of the following two approaches:

- First perform a conceptual schema design using a conceptual model such as ER or EER and then map the conceptual design into a set of relations.
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports.

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms, using the normalization theory presented in this chapter and the next. We focus in this section on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are deferred to Section 10.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 5 that are needed here. We then discuss first normal form (1NF) in Section 10.3.4, and present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 10.3.5 and 10.3.6 respectively.

10.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to "certify" whether it satisfies a certain **normal form.** The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as

relational design by analysis. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively; these are discussed in Chapter 11. At the beginning of Chapter 11, we also discuss how 3NF relations may be synthesized from a given set of FDs. This approach is called *relational design by synthesis*.

Normalization of data can be looked upon as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies discussed in Section 10.1.2. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
- A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized. Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The **lossless join** or **nonadditive join property,** which guarantees that the spurious tuple generation problem discussed in Section 10.1.4 does not occur with respect to the relation schemas created after decomposition
- The **dependency preservation property,** which ensures that each functional dependency is represented in some individual relation resulting after decomposition

The nonadditive join property is extremely critical and must be achieved at any cost, whereas the dependency preservation property, although desirable, is sometimes sacrificed, as we discuss in Section 11.1.2. We defer the presentation of the formal concepts and techniques that guarantee the above two properties to Chapter 11.

10.3.2 Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and

5NF that we discuss in Chapter 11, the practical utility of these normal forms becomes questionable when the constraints on which they are based are hard to understand or to detect by the database designers and users who must discover these constraints. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons, such as those discussed at the end of Section 10.1.2. The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form—is known as **denormalization.**

10.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let us look again at the definitions of keys of a relation schema from Chapter 5.

Definition. A **superkey** of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.

The difference between a key and a superkey is that a key has to be *minimal*; that is, if we have a key $K = \{A_1, A_2, \ldots, A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \leq i \leq k$. In Figure 10.1, {SSN} is a key for EMPLOYEE, whereas {SSN}, {SSN, ENAME}, {SSN, ENAME, BDATE}, and any set of attributes that includes SSN are all superkeys.

If a relation schema has more than one key, each is called a **candidate key.** One of the candidate keys is *arbitrarily* designated to be the **primary key,** and the others are called secondary keys. Each relation schema must have a primary key. In Figure 10.1, {SSN} is the only candidate key for EMPLOYEE, so it is also the primary key.

Definition. An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

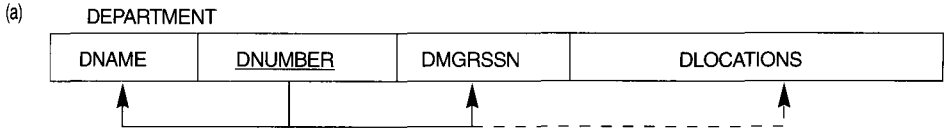
In Figure 10.1 both SSN and PNUMBER are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for historical reasons, it is customary to follow them in that sequence; hence we will assume that a 3NF relation *already satisfies* 2NF.

10.3.4 First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model;[12] historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple*. In other words, 1NF disallows "relations within relations" or "relations as attribute values within tuples." The only attribute values permitted by 1NF are single **atomic (or indivisible) values.**

Consider the DEPARTMENT relation schema shown in Figure 10.1, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute as shown in Figure 10.8a. We assume that each department can have *a number of* locations. The DEPARTMENT schema and an example relation state are shown in Figure 10.8. As we can see,



(b) DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c) DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

FIGURE 10.8 Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example state of relation DEPARTMENT. (c) 1NF version of same relation with redundancy.

12. This condition is removed in the *nested relational model* and in *object-relational systems* (ORDBMSs), both of which allow *unnormalized relations* (see Chapter 22).

this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure 10.8b. There are two ways we can look at the DLOCATIONS attribute:

- The domain of DLOCATIONS contains atomic values, but some tuples can have a set of these values. In this case, DLOCATIONS *is not* functionally dependent on the primary key DNUMBER.
- The domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, DNUMBER → DLOCATIONS, because each set is considered a single member of the attribute domain.[13]

In either case, the DEPARTMENT relation of Figure 10.8 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 5.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATION}, as shown in Figure 10.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 10.8c. In this case, the primary key becomes the combination {DNUMBER, DLOCATION}. This solution has the disadvantage of introducing *redundancy* in the relation.
3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2, and DLOCATION3. This solution has the disadvantage of introducing *null values* if most departments have fewer than three locations. It further introduces a spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: "List the departments that have "Bellaire" as one of their locations" in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it*. Figure 10.9 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(PNUMBER, HOURS) *within each*

13. In this case we can consider the domain of DLOCATIONS to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

(a) **EMP_PROJ**

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b) **EMP_PROJ**

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

(c) **EMP_PROJ1**

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

FIGURE 10.9 Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" attribute PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

tuple represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

```
EMP_PROJ(SSN, ENAME, {PROJS(PNUMBER, HOURS)})
```

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 20) and XML data (see Chapter 26) using the relational model attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that SSN is the primary key of the EMP_PROJ relation in Figures 10.9a and b, while PNUMBER is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of PNUMBER. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2 shown in Figure 10.9c.

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (SS#, {CAR_LIC#}, {PHONE#})

This relation represents the fact that a person has multiple cars and multiple phones. If a strategy like the second option above is followed, it results in an all-key relation:

PERSON_IN_1NF (SS#, CAR_LIC#, PHONE#)

To avoid introducing any extraneous relationship between CAR_LIC# and PHONE#, all possible combinations of values are represented for every SS#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we discuss in Chapter 11. The right way to deal with the two multivalued attributes in PERSON above is to decompose it into two separate relations, using strategy 1 discussed above: P1(SS#, CAR_LIC#) and P2(SS#, PHONE#).

10.3.5 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 10.3b, {SSN, PNUMBER} → HOURS is a full dependency (neither SSN → HOURS nor PNUMBER → HOURS holds). However, the dependency {SSN, PNUMBER} → ENAME is partial because SSN → ENAME holds.

Definition. A relation schema R is in **2NF** if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

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. The EMP_PROJ relation in Figure 10.3b is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the nonprime attributes PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3 make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be "second normalized" or "2NF normalized" into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies FD1, FD2, and FD3 in Figure 10.3b hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 10.10a, each of which is in 2NF.

10.3.6 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there is a set of

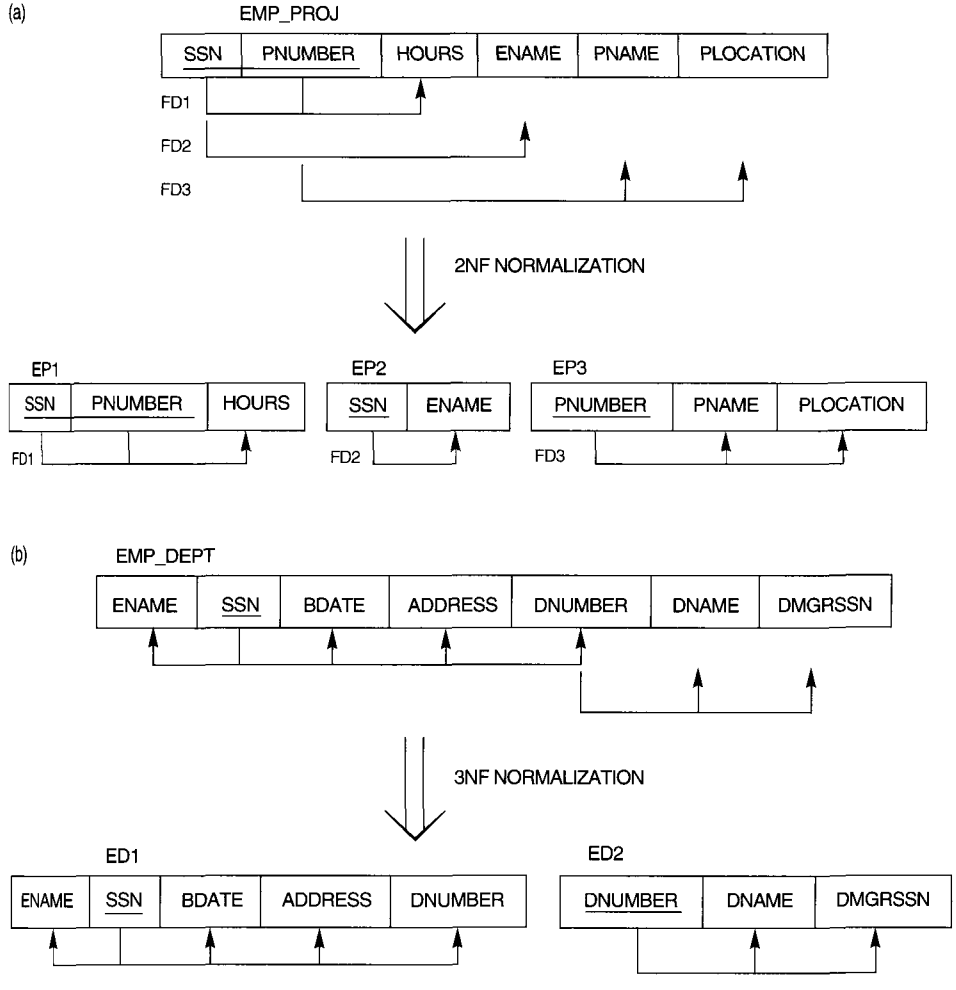


FIGURE 10.10 Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

attributes Z that is neither a candidate key nor a subset of any key of R,[14] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency SSN $\rightarrow$ DMGRSSN is transitive through DNUMBER in EMP_DEPT of Figure 10.3a because both the dependencies SSN $\rightarrow$ DNUMBER and DNUMBER $\rightarrow$ DMGRSSN hold *and* DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.

Definition. According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF *and* no nonprime attribute of R is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 10.3a is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 10.10b. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute is a "problematic" FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 10.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding "remedy" or normalization performed to achieve the normal form.

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.

14. This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where X is the primary key but Z may be (a subset of) a candidate key.

TABLE 10.1 SUMMARY OF NORMAL FORMS BASED ON PRIMARY KEYS AND CORRESPONDING NORMALIZATION

NORMAL FORM	TEST	REMEDY (NORMALIZATION)
First (1NF)	Relation should have no nonatomic attributes or nested relations.	Form new relations for each nonatomic attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes.) That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

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.[15]

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 10.11a, 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 10.11a hold. We choose PROPERTY_ID# as the primary key, so it is underlined in Figure 10.11a, but no special consideration will

15. 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.

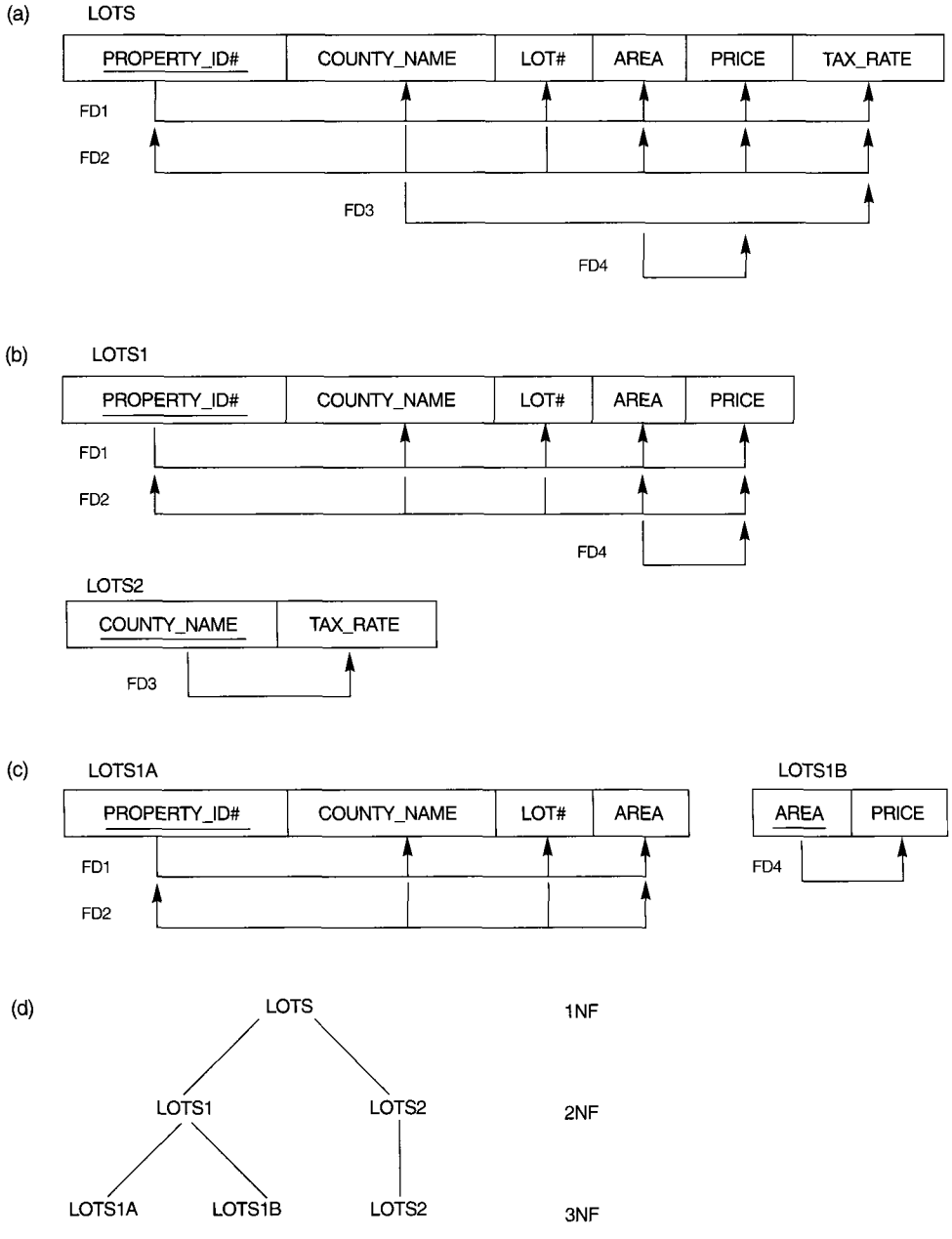


FIGURE 10.11 Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.

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. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 10.11b. 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 10.11b) 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 10.11c. 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. We could hence 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). Hence, we can state a **general alternative definition of 3NF** as follows: 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.

10.5 Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema of Figure 10.11a with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: Dekalb and Fulton. Suppose also that lot sizes in Dekalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, . . . , 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: AREA → COUNTY_NAME. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation R(AREA, COUNTY_NAME), since there are only 16 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

Definition. A relation schema R is in **BCNF** if whenever a *nontrivial* functional dependency X → A holds in R, then X is a superkey of R.

The formal definition of BCNF differs slightly from the definition of 3NF. The only difference between the definitions of BCNF and 3NF is that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because COUNTY_NAME is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 10.12a. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if X → A holds in a relation schema R with X not being a superkey *and* A being a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in Figure 10.12b illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization

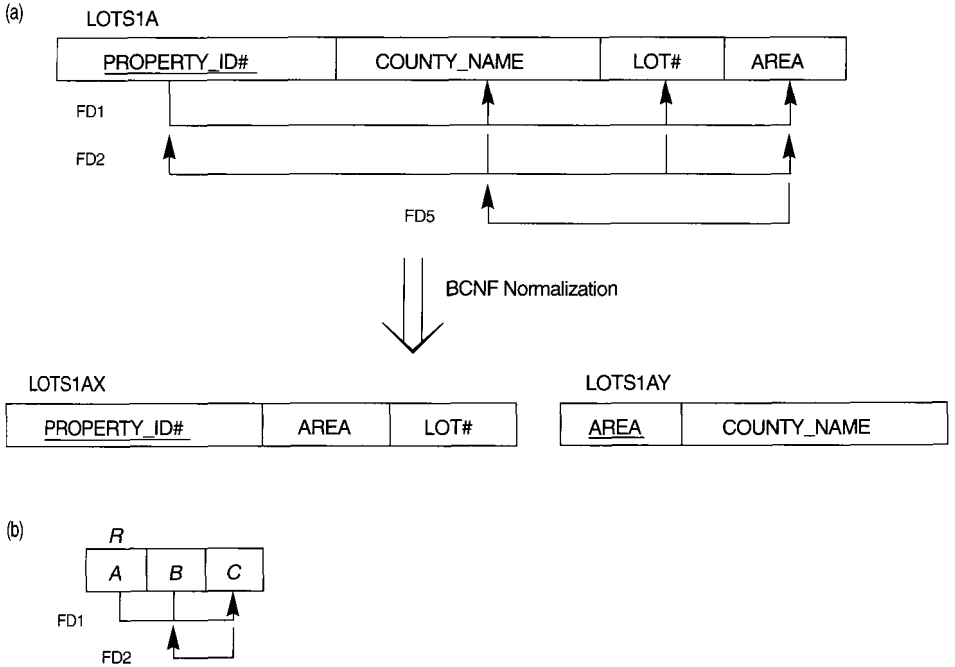


FIGURE 10.12 Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 10.13, which shows a relation TEACH with the following dependencies:

FD1: { STUDENT, COURSE} → INSTRUCTOR

FD2:[16] INSTRUCTOR → COURSE

Note that {STUDENT, COURSE} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12b, with STUDENT as A, COURSE as B, and INSTRUCTOR as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. {STUDENT, INSTRUCTOR} and {STUDENT, COURSE}.
2. {COURSE, INSTRUCTOR } and {COURSE, STUDENT}.
3. {INSTRUCTOR, COURSE } and {INSTRUCTOR, STUDENT}.

16. This dependency means that "each instructor teaches one course" is a constraint for this application.

TEACH

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe

FIGURE 10.13 A relation TEACH that is in 3NF but not BCNF.

All three decompositions "lose" the functional dependency FD1. The *desirable decomposition* of those just shown is 3, because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (lossless) is discussed in Section 11.1.4 under Property LJ1. In general, a relation not in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 11.3 does that and could be used above to give decomposition 3 for TEACH.

10.6 SUMMARY

In this chapter we first discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating whether a relation schema is "good" or "bad," and provided informal guidelines for a good design. We then presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We discussed the problems of update anomalies that occur when redundancies are present in relations. Informal measures of good relation schemas include simple and clear attribute semantics and few nulls in the extensions (states) of relations. A good decomposition should also avoid the problem of generation of spurious tuples as a result of the join operation.

We defined the concept of functional dependency and discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. We showed how from a given set of functional dependencies, additional dependencies can be inferred using a set of inference rules. We defined the concepts of closure and cover related to functional dependencies. We then defined

minimal cover of a set of dependencies, and provided an algorithm to compute a minimal cover. We also showed how to check whether two sets of functional dependencies are equivalent.

We then described the normalization process for achieving good designs by testing relations for undesirable types of "problematic" functional dependencies. We provided a treatment of successive normalization based on a predefined primary key in each relation, then relaxed this requirement and provided more general definitions of second normal form (2NF) and third normal form (3NF) that take all candidate keys of a relation into account. We presented examples to illustrate how by using the general definition of 3NF a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.

Finally, we presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement.

Chapter 11 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *lossless (nonadditive) join* and *dependency preservation*, which are enforced by some of these algorithms. Other topics in Chapter 11 include multivalued dependencies, join dependencies, and fourth and fifth normal forms, which take these dependencies into account.

Review Questions

10.1. Discuss attribute semantics as an informal measure of goodness for a relation schema.

10.2. Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

10.3. Why should nulls in a relation be avoided as far as possible? Discuss the problem of spurious tuples and how we may prevent it.

10.4. State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.

10.5. What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?

10.6. Why can we not infer a functional dependency automatically from a particular relation state?

10.7. What role do Armstrong's inference rules—the three inference rules IR1 through IR3—play in the development of the theory of relational design?

10.8. What is meant by the completeness and soundness of Armstrong's inference rules?

10.9. What is meant by the closure of a set of functional dependencies? Illustrate with an example.

10.10. When are two sets of functional dependencies equivalent? How can we determine their equivalence?

10.11. What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set? Is it always unique?

10.12. What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?

10.13. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

10.14. What undesirable dependencies are avoided when a relation is in 2NF?

10.15. What undesirable dependencies are avoided when a relation is in 3NF?

10.16. Define Boyce-Codd normal form. How does it differ from 3NF? Why is it considered a stronger form of 3NF?

Exercises

10.17. Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:

a. The university keeps track of each student's name (SNAME), student number (SNUM), social security number (SSN), current address (SCADDR) and phone (SCPHONE), permanent address (SPADDR) and phone (SPPHONE), birth date (BDATE), sex (SEX), class (CLASS) (freshman, sophomore, . . . , graduate), major department (MAJORCODE), minor department (MINORCODE) (if any), and degree program (PROG) (B.A., B.S., . . . , PH.D.). Both SSSN and student number have unique values for each student.

b. Each department is described by a name (DNAME), department code (DCODE), office number (DOFFICE), office phone (DPHONE), and college (DCOLLEGE). Both name and code have unique values for each department.

c. Each course has a course name (CNAME), description (CDESC), course number (CNUM), number of semester hours (CREDIT), level (LEVEL), and offering department (CDEPT). The course number is unique for each course.

d. Each section has an instructor (INAME), semester (SEMESTER), year (YEAR), course (SECCOURSE), and section number (SECNUM). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, . . . , up to the total number of sections taught during each semester.

e. A grade record refers to a student (SSN), a particular section, and a grade (GRADE).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

10.18. Prove or disprove the following inference rules for functional dependencies. A proof can be made either by a proof argument or by using inference rules IR1 through IR3. A disproof should be performed by demonstrating a relation instance that satisfies the conditions and functional dependencies in the left-hand side of the inference rule but does not satisfy the dependencies in the right-hand side.

a. $\{W \rightarrow Y, X \rightarrow Z\} \models \{WX \rightarrow Y\}$

b. $\{X \rightarrow Y\}$ and $Y \supseteq Z \models \{X \rightarrow Z\}$

c. $\{X \rightarrow Y, X \rightarrow W, WY \rightarrow Z\} \models \{X \rightarrow Z\}$
d. $\{XY \rightarrow Z, Y \rightarrow W\} \models \{XW \rightarrow Z\}$
e. $\{X \rightarrow Z, Y \rightarrow Z\} \models \{X \rightarrow Y\}$
f. $\{X \rightarrow Y, XY \rightarrow Z\} \models \{X \rightarrow Z\}$
g. $\{X \rightarrow Y, Z \rightarrow W\} \models \{XZ \rightarrow YW\}$
h. $\{XY \rightarrow Z, Z \rightarrow X\} \models \{Z \rightarrow Y\}$
i. $\{X \rightarrow Y, Y \rightarrow Z\} \models \{X \rightarrow YZ\}$
j. $\{XY \rightarrow Z, Z \rightarrow W\} \models \{X \rightarrow W\}$

10.19. Consider the following two sets of functional dependencies: $F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$ and $G = \{A \rightarrow CD, E \rightarrow AH\}$. Check whether they are equivalent.

10.20. Consider the relation schema EMP_DEPT in Figure 10.3a and the following set G of functional dependencies on EMP_DEPT: G = {SSN → {ENAME, BDATE, ADDRESS, DNUMBER}, DNUMBER → {DNAME, DMGRSSN}}. Calculate the closures $\{SSN\}^+$ and $\{DNUMBER\}^+$ with respect to G.

10.21. Is the set of functional dependencies G in Exercise 10.20 minimal? If not, try to find a minimal set of functional dependencies that is equivalent to G. Prove that your set is equivalent to G.

10.22. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 10.3 and 10.4?

10.23. In what normal form is the LOTS relation schema in Figure 10.11a with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

10.24. Prove that any relation schema with two attributes is in BCNF.

10.25. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations of Figure 10.5 (result shown in Figure 10.6)?

10.26. Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $F = \{\{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\}\}$. What is the key for R? Decompose R into 2NF and then 3NF relations.

10.27. Repeat Exercise 10.26 for the following different set of functional dependencies $G = \{\{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\}\}$.

10.28. Consider the following relation:

A	B	C	TUPLE#
10	b1	c1	#1
10	b2	c2	#2
11	b4	c1	#3
12	b3	c4	#4
13	b1	c1	#5
14	b3	c4	#6

a. Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation*.

i. A → B, ii. B → C, iii. C → B, iv. B → A, v. C → A

b. Does the above relation have a *potential* candidate key? If it does, what is it? If it does not, why not?

10.29. Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$AB \rightarrow C, CD \rightarrow E, DE \rightarrow B$

Is AB a candidate key of this relation? If not, is ABD? Explain your answer.

10.30. Consider the relation R, which has attributes that hold schedules of courses and sections at a university; R = {CourseNo, SecNo, OfferingDept, Credit–Hours, CourseLevel, InstructorSSN, Semester, Year, Days_Hours, RoomNo, NoOfStudents}. Suppose that the following functional dependencies hold on R:

{CourseNo} → {OfferingDept, CreditHours, CourseLevel}

{CourseNo, SecNo, Semester, Year} → {Days_Hours, RoomNo, NoOfStudents, InstructorSSN}

{RoomNo, Days_Hours, Semester, Year} → {Instructorssn, CourseNo, SecNo}

Try to determine which sets of attributes form keys of R. How would you normalize this relation?

10.31. Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (O#, Odate, Cust#, Total_amount)

ORDER-ITEM(O#, I#, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The TOTAL_PRICE refers to one item, ODATE is the date on which the order was placed, and the TOTAL_AMOUNT is the amount of the order. If we apply a natural join on the relations ORDER-ITEM and ORDER in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

10.32. Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesman#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salesmen, and hence {CAR#, SALESMAN#} is the primary key. Additional dependencies are

Date_sold → Discount_amt

and

Salesman# → Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

10.33. Consider the following relation for published books:

BOOK (Book_title, Authorname, Book_type, Listprice, Author_affil, Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title → Publisher, Book_type

Book_type → Listprice

Authorname → Author-affil

a. What normal form is the relation in? Explain your answer.
b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies.

Armstrong (1974) shows the soundness and completeness of the inference rules IR1 through IR3. Additional references to relational design theory are given in Chapter 11.

11

Relational Database Design Algorithms and Further Dependencies

In this chapter, we describe some of the relational database design algorithms that utilize functional dependency and normalization theory, as well as some other types of dependencies. In Chapter 10, we introduced the two main approaches for relational database design. The first approach utilizes a **top-down design** technique, and is currently used most extensively in commercial database application design. This involves designing a conceptual schema in a high-level data model, such as the EER model, and then mapping the conceptual schema into a set of relations using mapping procedures such as the ones discussed in Chapter 7. Following this, each of the relations is analyzed based on the functional dependencies and assigned primary keys. By applying the normalization procedure in Section 10.3, we can remove any remaining partial and transitive dependencies from the relations. In some design methodologies, this analysis is applied directly during conceptual design to the attributes of the entity types and relationship types. In this case, undesirable dependencies are discovered during conceptual design, and the relation schemas resulting from the mapping procedures would automatically be in higher normal forms, so there would be no need for additional normalization.

The second approach utilizes a **bottom-up design** technique, and is a more purist approach that views relational database schema design strictly in terms of functional and other types of dependencies specified on the database attributes. It is also known as relational synthesis. After the database designer specifies the dependencies, a **normalization algorithm** is applied to synthesize the relation schemas. Each individual relation schema should possess the measures of goodness associated with 3NF or BCNF or with some higher normal form.

In this chapter, we describe some of these normalization algorithms as well as the other types of dependencies. We also describe the two desirable properties of nonadditive (lossless) joins and dependency preservation in more detail. The normalization algorithms typically start by synthesizing one giant relation schema, called the **universal relation,** which is a theoretical relation that includes all the database attributes. We then perform **decomposition**—breaking up into smaller relation schemas—until it is no longer feasible or no longer desirable, based on the functional and other dependencies specified by the database designer.

We first describe in Section 11.1 the two desirable **properties of decompositions,** namely, the dependency preservation property and the lossless (or nonadditive) join property, which are both used by the design algorithms to achieve desirable decompositions. It is important to note that it is *insufficient* to test the relation schemas *independently of one another* for compliance with higher normal forms like 2NF, 3NF, and BCNF. The resulting relations must collectively satisfy these two additional properties to qualify as a good design. Section 11.2 presents several normalization algorithms based on functional dependencies alone that can be used to design 3NF and BCNF schemas.

We then introduce other types of data dependencies, including multivalued dependencies and join dependencies, that specify constraints that *cannot* be expressed by functional dependencies. Presence of these dependencies leads to the definition of fourth normal form (4NF) and fifth normal form (5NF), respectively. We also define inclusion dependencies and template dependencies (which have not led to any new normal forms so far). We then briefly discuss domain-key normal form (DKNF), which is considered the most general normal form.

It is possible to skip some or all of Sections 11.4, 11.5, and 11.6 in an introductory database course.

11.1 Properties of Relational Decompositions

In Section 11.1.1 we give examples to show that looking at an *individual* relation to test whether it is in a higher normal form does not, on its own, guarantee a good design; rather, a *set of relations* that together form the relational database schema must possess certain additional properties to ensure a good design. In Sections 11.1.2 and 11.1.3 we discuss two of these properties: the dependency preservation property and the lossless or nonadditive join property. Section 11.1.4 discusses binary deecompositions, and Section 11.1.5 discusses successive nonadditive join decompositions.

11.1.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 11.2 start from a single **universal relation schema** $R = \{A_1, A_2, \ldots, A_n\}$ that includes *all* the attributes of the

database. We implicitly make the **universal relation assumption,** which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, \ldots, R_m\}$ that will become the relational database schema; D is called a **decomposition** of R.

We must make sure that each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are "lost"; formally, we have

$$\bigcup_{i=1}^{m} R_i = R$$

This is called the **attribute preservation** condition of a decomposition.

Another goal is to have each individual relation R_i in the decomposition D be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations. To illustrate this point, consider the EMP_LOCS(ENAME, PLOCATION) relation of Figure 10.5, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF.[1] Although EMP_LOCS is in BCNF, it still gives rise to spurious tuples when joined with EMP_PROJ (SSN, PNUMBER, HOURS, PNAME, PLOCATION), which is not in BCNF (see the result of the natural join in Figure 10.6). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted semantics by which PLOCATION gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(PNAME, PNUMBER, PLOCATION, DNUM) of Figure 10.2—which *is* in BCNF—also gives rise to spurious tuples. This underscores the need for other criteria that, together with the conditions of 3NF or BCNF, prevent such bad designs. In the next three subsections we discuss such additional conditions that should hold on a decomposition D as a whole.

11.1.2 Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i. Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation; instead, we have to join two or more of the relations in the decomposition and then check that the functional dependency holds in the result of the JOIN operation. This is clearly an inefficient and impractical procedure.

1. As an exercise, the reader should prove that this statement is true.

It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D. It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F. We now define these concepts more formally.

Definition. Given a set of dependencies F on R, the **projection** of F on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand-side attributes are in R_i. We say that a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is,

$$((\pi_{R_1}(F)) \cup \ldots \cup (\pi_{R_m}(F)))^+ = F^+$$

If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. As we mentioned earlier, to check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left- and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

An example of a decomposition that does not preserve dependencies is shown in Figure 10.12a, in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 10.11, however, are dependency-preserving. Similarly, for the example in Figure 10.13, no matter what decomposition is chosen for the relation TEACH(STUDENT, COURSE, INSTRUCTOR) from the three provided in the text, one or both of the dependencies originally present are lost. We state a claim below related to this property without providing any proof.

CLAIM 1

It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation R_i in D is in 3NF.

In Section 11.2.1, we describe Algorithm 11.2, which creates a dependency-preserving decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF.

11.1.3 Lossless (Nonadditive) Join Property of a Decomposition

Another property that a decomposition D should possess is the lossless join or nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations in the decomposition. We already illustrated this problem in Section 10.1.4 with the example of Figures 10.5 and 10.6. Because this is a property of a decomposition of relation *schemas*, the condition of no spurious tuples

should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in F. Hence, the lossless join property is always defined with respect to a specific set F of dependencies.

Definition. Formally, a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R has the **lossless (nonadditive) join property** with respect to the set of dependencies F on R if, for *every* relation state r of R that satisfies F, the following holds, where * is the NATURAL JOIN of all the relations in D:

$$* (\pi_{R_1}(r), \ldots, \pi_{R_m}(r)) = r$$

The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous information. We prefer the term nonadditive join because it describes the situation more accurately. If the property holds on a decomposition, we are guaranteed that no spurious tuples bearing wrong information are added to the result after the project and natural join operations are applied.

The decomposition of EMP_PROJ(SSN, PNUMBER, HOURS, ENAME, PNAME, PLOCATION) from Figure 10.3 into EMP_LOCS(ENAME, PLOCATION) and EMP_PROJ1(SSN, PNUMBER, HOURS, PNAME, PLOCATION) in Figure 10.5 obviously does not have the lossless join property, as illustrated by Figure 10.6. We will use a general procedure for testing whether any decomposition D of a relation into n relations is lossless (nonadditive) with respect to a set of given functional dependencies F in the relation; it is presented as Algorithm 11.1 below. It is possible to apply a simpler test to check if the decomposition is nonadditive for binary decompositions; that test is described in Section 11.1.4.

Algorithm 11.1: Testing for Lossless (nonadditive) Join Property

Input: A universal relation R, a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R, and a set F of functional dependencies.

1. Create an initial matrix S with one row i for each relation R_i in D, and one column j for each attribute A_j in R.
2. Set $S(i, j) := b_{ij}$ for all matrix entries.
   (* each b_{ij} is a distinct symbol associated with indices (i, j) *)
3. For each row i representing relation schema R_i
   {for each column j representing attribute A_j
   {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};};
   (* each a_j is a distinct symbol associated with index (j) *)
4. Repeat the following loop until a *complete loop execution* results in no changes to S
   {for each functional dependency $X \rightarrow Y$ in F
   {for all rows in S *that have the same symbols* in the columns corresponding to attributes in X
   {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an "a" symbol for the

column, set the other rows to that *same* "*a*" symbol in the column. If no "*a*" symbol exists for the attribute in any of the rows, choose one of the "*b*" symbols that appears in one of the rows for the attribute and set the other rows to that same "*b*" symbol in the column ;};};};

5. If a row is made up entirely of "*a*" symbols, then the decomposition has the lossless join property; otherwise, it does not.

Given a relation R that is decomposed into a number of relations $R_1, R_2, \ldots, R_m$, Algorithm 11.1 begins the matrix S that we consider to be some relation state r of R. Row i in S represents a tuple t_i (corresponding to relation R_i) that has "*a*" symbols in the columns that correspond to the attributes of R_i and "*b*" symbols in the remaining columns. The algorithm then transforms the rows of this matrix (during the loop of step 4) so that they represent tuples that satisfy all the functional dependencies in F. At the end of step 4, any two rows in S—which represent two tuples in r—that agree in their values for the left-hand-side attributes X of a functional dependency $X \rightarrow Y$ in F will also agree in their values for the right-hand-side attributes Y. It can be shown that after applying the loop of step 4, if any row in S ends up with all "*a*" symbols, then the decomposition D has the lossless join property with respect to F.

If, on the other hand, no row ends up being all "*a*" symbols, D does not satisfy the lossless join property. In this case, the relation state r represented by S at the end of the algorithm will be an example of a relation state r of R that satisfies the dependencies in F but does not satisfy the lossless join condition. Thus, this relation serves as a **counterexample** that proves that D does not have the lossless join property with respect to F. Note that the "*a*" and "*b*" symbols have no special meaning at the end of the algorithm

Figure 11.1a shows how we apply Algorithm 11.1 to the decomposition of the EMP_PROJ relation schema from Figure 10.3b into the two relation schemas EMP_PROJ1 and EMP_LOCS of Figure 10.5a. The loop in step 4 of the algorithm cannot change any "*b*" symbols to "*a*" symbols; hence, the resulting matrix S does not have a row with all "*a*" symbols, and so the decomposition does not have the lossless join property.

Figure 11.1b shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the lossless join property, and Figure 11.1c shows how we apply the algorithm to that decomposition. Once a row consists only of "*a*" symbols, we know that the decomposition has the lossless join property, and we can stop applying the functional dependencies (step 4 of the algorithm) to the matrix S.

11.1.4 Testing Binary Decompositions for the Nonadditive Join Property

Algorithm 11.1 allows us to test whether a particular decomposition D into n relations obeys the lossless join property with respect to a set of functional dependencies F. There is a special case of a decomposition called a **binary decomposition**—decomposition of a relation R into two relations. We give an easier test to apply than Algorithm 11.1, but while it is very handy to use, it is *limited* to binary decompositions only.

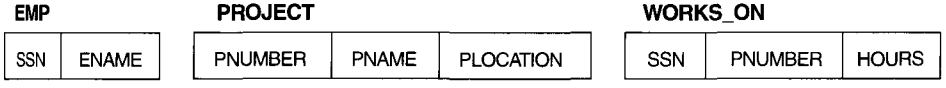
(a) R={SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS} D={R_1, R_2}
R_1=EMP_LOCS={ENAME, PLOCATION}
R_2=EMP_PROJ1={SSN, PNUMBER, HOURS, PNAME, PLOCATION}

F={SSN→ENAME;PNUMBER→{PNAME, PLOCATION} ;{SSN,PNUMBER}→HOURS}

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(no changes to matrix after applying functional dependencies)

(b)



(c) R={SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS} D={R_1, R_2, R_3}
R_1=EMP={SSN, ENAME}
R_2=PROJ={PNUMBER, PNAME, PLOCATION}
R_3=WORKS_ON={SSN, PNUMBER, HOURS}

F={SSN→{ENAME;PNUMBER→{PNAME, PLOCATION} ;{SSN,PNUMBER}→HOURS}

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(original matrix S at start of algorithm)

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32} a_2	a_3	b_{34} a_4	b_{35} a_5	a_6

(matrix S after applying the first two functional dependencies - last row is all "a" symbols, so we stop)

FIGURE 11.1 Lossless (nonadditive) join test for n-ary decompositions. (a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property. (c) Case 2: Decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON satisfies test.

PROPERTY LJ1 (LOSSLESS JOIN TEST FOR BINARY DECOMPOSITIONS)

A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R *if and only if* either

- The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+, or
- The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+

You should verify that this property holds with respect to our informal successive normalization examples in Sections 10.3 and 10.4.

11.1.5 Successive Lossless (Nonadditive) Join Decompositions

We saw the successive decomposition of relations during the process of second and third normalization in Sections 10.3 and 10.4. To verify that these decompositions are nonadditive, we need to ensure another property, as set forth in Claim 2.

CLAIM 2 (Preservation of Nonadditivity in Successive Decompositions)

If a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $D_i = \{Q_1, Q_2, \ldots, Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, \ldots, R_{i-1}, Q_1, Q_2, \ldots, Q_k, R_{i+1}, \ldots, R_m\}$ of R has the nonadditive join property with respect to F.

11.2 Algorithms for Relational Database Schema Design

We now give three algorithms for creating a relational decomposition. Each algorithm has specific properties, as we discuss below.

11.2.1 Dependency-Preserving Decomposition into 3NF Schemas

Algorithm 11.2 creates a dependency-preserving decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF. It guarantees only the dependency-preserving property; it does *not* guarantee the lossless join property. The first step of Algorithm 11.2 is to find a minimal cover G for F; Algorithm 10.2 can be used for this step.

Algorithm 11.2: Relational Synthesis into 3NF with Dependency Preservation

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F (use Algorithm 10.2);
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, where $X \rightarrow A_1$, $X \rightarrow A_2, \ldots, X \rightarrow A_k$ are the only dependencies in G with X as the left-hand-side (X is the key of this relation);
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

CLAIM 3

Every relation schema created by Algorithm 11.2 is in 3NF. (We will not provide a formal proof here;[2] the proof depends on G being a minimal set of dependencies.)

It is obvious that all the dependencies in G are preserved by the algorithm because each dependency appears in one of the relations R_i in the decomposition D. Since G is equivalent to F, all the dependencies in F are either preserved directly in the decomposition or are derivable using the inference rules from Section 10.2.2 from those in the resulting relations, thus ensuring the dependency preservation property. Algorithm 11.2 is called the relational synthesis algorithm, because each relation schema R_i in the decomposition is synthesized (constructed) from the set of functional dependencies in G with the same left-hand-side X.

11.2.2 Lossless (Nonadditive) Join Decomposition into BCNF Schemas

The next algorithm decomposes a universal relation schema $R = \{A_1, A_2, \ldots, A_n\}$ into a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ such that each R_i is in BCNF *and* the decomposition D has the lossless join property with respect to F. Algorithm 11.3 utilizes Property LJ1 and Claim 2 (preservation of nonadditivity in successive decompositions) to create a nonadditive join decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in BCNF.

Algorithm 11.3: Relational Decomposition into BCNF with Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Set $D := \{R\}$;
2. While there is a relation schema Q in D that is not in BCNF do
   {
   choose a relation schema Q in D that is not in BCNF;
   find a functional dependency $X \rightarrow Y$ in Q that violates BCNF;
   replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
   };

2. See Maier (1983) or Ullman (1982) for a proof.

Each time through the loop in Algorithm 11.3, we decompose one relation schema Q that is not in BCNF into two relation schemas. According to Property LJ1 for binary decompositions and Claim 2, the decomposition D has the nonadditive join property. At the end of the algorithm, all relation schemas in D will be in BCNF. The reader can check that the normalization example in Figures 10.11 and 10.12 basically follows this algorithm. The functional dependencies FD3, FD4, and later FD5 violate BCNF, so the LOTS relation is decomposed appropriately into BCNF relations, and the decomposition then satisfies the nonadditive join property. Similarly, if we apply the algorithm to the TEACH relation schema from Figure 10.13, it is decomposed into TEACH1(INSTRUCTOR, STUDENT) and TEACH2(INSTRUCTOR, COURSE) because the dependency FD2: INSTRUCTOR → COURSE violates BCNF.

In step 2 of Algorithm 11.3, it is necessary to determine whether a relation schema Q is in BCNF or not. One method for doing this is to test, for each functional dependency $X \rightarrow Y$ in Q, whether X^+ fails to include all the attributes in Q, thereby determining whether or not X is a (super)key in Q. Another technique is based on an observation that whenever a relation schema Q violates BCNF, there exists a pair of attributes A and B in Q such that $\{Q - \{A, B\}\} \rightarrow A$; by computing the closure $\{Q - \{A, B\}\}^+$ for each pair of attributes $\{A, B\}$ of Q, and checking whether the closure includes A (or B), we can determine whether Q is in BCNF.

11.2.3 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

If we want a decomposition to have the nonadditive join property *and* to preserve dependencies, we have to be satisfied with relation schemas in 3NF rather than BCNF. A simple modification to Algorithm 11.2, shown as Algorithm 11.4, yields a decomposition D of R that does the following:

- Preserves dependencies
- Has the nonadditive join property
- Is such that each resulting relation schema in the decomposition is in 3NF

Algorithm 11.4: Relational Synthesis into 3NF with Dependency Preservation and Nonadditive (Lossless) Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F (use Algorithm 10.2).
2. For each left-hand-side X of a functional dependency that appears in G create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R.

It can be shown that the decomposition formed from the set of relation schemas created by the preceding algorithm is dependency-preserving *and* has the nonadditive join property. In addition, each relation schema in the decomposition is in 3NF. This algorithm is an improvement over Algorithm 11.2 in that the former guaranteed only dependency preservation.[3]

Step 3 of Algorithm 11.4 involves identifying a key K of R. Algorithm 11.4a can be used to identify a key K of R based on the set of given functional dependencies F. We start by setting K to all the attributes of R; we then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice that the set of functional dependencies used to determine a key in Algorithm 11.4a could be either F or G, since they are equivalent. Notice, too, that Algorithm 11.4a determines only *one key* out of the possible candidate keys for R; the key returned depends on the order in which attributes are removed from R in step 2.

Algorithm 11.4a: Finding a Key K for R Given a set F of Functional Dependencies

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Set $K := R$.
2. For each attribute A in K
   {compute $(K - A)^+$ with respect to F;
   If $(K - A)^+$ contains all the attributes in R, then set $K := K - \{A\}$};

It is important to note that the theory of nonadditive join decompositions is based on the assumption that *no null values are allowed for the join attributes*. The next section discusses some of the problems that nulls may cause in relational decompositions.

11.2.4 Problems with Null Values and Dangling Tuples

We must carefully consider the problems associated with nulls when designing a relational database schema. There is no fully satisfactory relational design theory as yet that includes null values. One problem occurs when some tuples have null values for attributes that will be used to join individual relations in the decomposition. To illustrate this, consider the database shown in Figure 11.2a, where two relations EMPLOYEE and DEPARTMENT are shown. The last two employee tuples—Berger and Benitez—represent newly hired employees who have not yet been assigned to a department (assume that this does not violate any integrity constraints). Now suppose that we want to retrieve a list of (ENAME, DNAME) values for all the employees. If we apply the NATURAL JOIN operation on EMPLOYEE and DEPARTMENT (Figure 11.2b), the two aforementioned tuples will *not* appear in the result.

3. Step 3 of Algorithm 11.2 is not needed in Algorithm 11.4 to preserve attributes because the key will include any unplaced attributes; these are the attributes that do not participate in any functional dependency.

(a)

EMPLOYEE

ENAME	SSN	BDATE	ADDRESS	DNUM
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	**null**
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	**null**

DEPARTMENT

DNAME	DNUM	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

(b)

ENAME	SSN	BDATE	ADDRESS	DNUM	DNAME	DMGRSSN
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

(c)

ENAME	SSN	BDATE	ADDRESS	DNUM	DNAME	DMGRSSN
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	**null**	**null**	**null**
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	**null**	**null**	**null**

FIGURE 11.2 Issues with null-value joins. (a) Some EMPLOYEE tuples have null for the join attribute DNUM. (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

The OUTER JOIN operation, discussed in Chapter 6, can deal with this problem. Recall that if we take the LEFT OUTER JOIN of EMPLOYEE with DEPARTMENT, tuples in EMPLOYEE that have null for the join attribute will still appear in the result, joined with an "imaginary" tuple in DEPARTMENT that has nulls for all its attribute values. Figure 11.2c shows the result.

In general, whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be devoted to watching for potential null values in foreign keys. This can cause unexpected loss of information in queries that involve joins on that foreign key. Moreover, if nulls occur in other attributes, such as SALARY, their effect on built-in functions such as SUM and AVERAGE must be carefully evaluated.

A related problem is that of *dangling tuples*, which may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation of Figure 11.2a further into EMPLOYEE_1 and EMPLOYEE_2, shown in Figure 11.3a and 11.3b.[4] If we apply the NATURAL JOIN operation to EMPLOYEE_1 AND EMPLOYEE_2, we get the original EMPLOYEE relation. However, we may use the alternative representation, shown in Figure 11.3c, where we *do not include a tuple* in EMPLOYEE_3 if the employee has not been assigned a department (instead of including a tuple with null for DNUM as in EMPLOYEE_2). If we use EMPLOYEE_3 instead of EMPLOYEE_2 and apply a NATURAL JOIN on EMPLOYEE_1 and EMPLOYEE_3, the tuples for Berger and Benitez will not appear in the result; these are called **dangling tuples** because they are represented in only one of the two relations that represent employees and hence are lost if we apply an (INNER) JOIN operation.

11.2.5 Discussion of Normalization Algorithms

One of the problems with the normalization algorithms we described is that the database designer must first specify *all* the relevant functional dependencies among the database attributes. This is not a simple task for a large database with hundreds of attributes. Failure to specify one or two important dependencies may result in an undesirable design. Another problem is that these algorithms are *not deterministic* in general. For example, the *synthesis algorithms* (Algorithms 11.2 and 11.4) require the specification of a minimal cover G for the set of functional dependencies F. Because there may be in general many minimal covers corresponding to F, the algorithm can give different designs depending on the particular minimal cover used. Some of these designs may not be desirable. The *decomposition algorithm* (Algorithm 11.3) depends on the order in which the functional dependencies are supplied to the algorithm to check for BCNF violation. Again, it is possible that many different designs may arise corresponding to the same set of functional dependencies, depending on the order in which such dependencies are considered for violation of BCNF. Some of the designs may be quite superior, whereas others may be undesirable.

4. This sometimes happens when we apply vertical fragmentation to a relation in the context of a distributed database (see Chapter 25).

(a) **EMPLOYEE_1**

ENAME	SSN	BDATE	ADDRESS
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX

(b) **EMPLOYEE_2**

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	null
888664444	null

(c) **EMPLOYEE_3**

SSN	DNUM
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

FIGURE 11.3 The "dangling tuple" problem. (a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from Figure 11.2a except DNUM). (b) The relation EMPLOYEE_2 (includes DNUM attribute with null values). (c) The relation EMPLOYEE_3 (includes DNUM attribute but does not include tuples for which DNUM has null values).

It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (instead of 3NF as in Algorithm 11.4). We can check the 3NF relation schemas in the decomposition individually to see whether each satisfies BCNF. If some relation schema R_i is not in BCNF, we can choose to decompose it further or to leave it as it is in 3NF (with some possible update anomalies). The fact that we cannot always find a decomposition into relation schemas in BCNF that preserves dependencies can be illustrated by the examples in Figures 10.12 and 10.13. The relations LOTS1A (Figure 10.12a) and TEACH (Figure 10.13) are not in BCNF but are in 3NF. Any attempt to decompose either relation further into BCNF relations results in loss of the dependency FD2: {COUNTY_NAME, LOT#} → {PROPERTY_ID#, AREA} in LOTS1A or loss of FD1: {STUDENT, COURSE} → INSTRUCTOR in TEACH.

Table 11.1 summarizes the properties of the algorithms discussed in this chapter so far.

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
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 11.4a. 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

(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	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) **R1**

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

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(R1, R2, R3). (d) Decomposing the relation SUPPLY into the 5NF relations R1, R2, R3.

dependents are independent of one another.[5] 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, an MVD may arise.

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

11.3.1 Formal Definition of Multivalued Dependency

Definition. A multivalued dependency $X \twoheadrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties,[6] where we use Z to denote $(R - (X \cup Y))$:[7]

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Whenever $X \twoheadrightarrow Y$ holds, we say that X **multidetermines** Y. Because of the symmetry in the definition, whenever $X \twoheadrightarrow Y$ holds in R, so does $X \twoheadrightarrow Z$. Hence, $X \twoheadrightarrow Y$ implies $X \twoheadrightarrow Z$, and therefore it is sometimes written as $X \twoheadrightarrow Y \mid Z$.

The formal definition specifies that given a particular value of X, the set of values of Y determined by this value of X is completely determined by X alone and *does not depend* on the values of the remaining attributes Z of R. Hence, whenever two tuples exist that have distinct values of Y but the same value of X, these values of Y must be repeated in separate tuples with *every distinct value of* Z that occurs with that same value of X. This informally corresponds to Y being a multivalued attribute of the entities represented by tuples in R.

In Figure 11.4a the MVDs ENAME $\twoheadrightarrow$ PNAME and ENAME $\twoheadrightarrow$ DNAME (or ENAME $\twoheadrightarrow$ PNAME | DNAME) hold in the EMP relation. The employee with ENAME 'SMITH' works on projects with PNAME 'X' and 'Y' and has two dependents with DNAME 'John' and 'Anna'. If we stored only the first two tuples in EMP (<'Smith', 'X', 'John'> and <'Smith', 'Y', 'Anna'>), we would incorrectly show associations between project 'X' and 'John' and between project 'Y' and 'Anna'; these should not be conveyed, because no such meaning is intended in this relation. Hence, we must store the other two tuples (<'Smith', 'X', 'Anna'> and <'Smith', 'Y', 'John'>) to show that {'X', 'Y'} and {'John', 'Anna'} are associated only with 'Smith'; that is, there is no association between PNAME and DNAME–which means that the two attributes are independent.

An MVD $X \twoheadrightarrow Y$ in R is called a **trivial** MVD if (a) Y is a subset of X, or (b) $X \cup Y = R$. For example, the relation EMP_PROJECTS in Figure 11.4b has the trivial MVD ENAME $\twoheadrightarrow$ PNAME. An MVD that satisfies neither (a) nor (b) is called a **nontrivial** MVD. A trivial MVD will hold in *any* relation state r of R; it is called trivial because it does not specify any significant or meaningful constraint on R.

If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 11.4a, the values 'X' and 'Y' of PNAME are repeated with each value of DNAME (or, by symmetry, the values 'John' and 'Anna' of DNAME are repeated with each value of PNAME). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because *no* functional dependencies hold in EMP. Therefore, we

6. The tuples t_1, t_2, t_3, and t_4 are not necessarily distinct.

7. Z is shorthand for the attributes remaining in R after the attributes in $(X \cup Y)$ are removed from R.

need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. We first discuss some of the properties of MVDs and consider how they are related to functional dependencies. Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together.

11.3.2 Inference Rules for Functional and Multivalued Dependencies

As with functional dependencies (FDs), inference rules for multivalued dependencies (MVDs) have been developed. It is better, though, to develop a unified framework that includes both FDs and MVDs so that both types of constraints can be considered together. The following inference rules IR1 through IR8 form a sound and complete set for inferring functional and multivalued dependencies from a given set of dependencies. Assume that all attributes are included in a "universal" relation schema $R = \{A_1, A_2, \ldots, A_n\}$ and that X, Y, Z, and W are subsets of R.

IR1 (reflexive rule for FDs): If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule for FDs): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

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

IR4 (complementation rule for MVDs): $\{X \twoheadrightarrow Y\} \models \{X \twoheadrightarrow (R - (X \cup Y))\}$.

IR5 (augmentation rule for MVDs): If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.

IR6 (transitive rule for MVDs): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$.

IR7 (replication rule for FD to MVD): $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

IR8 (coalescence rule for FDs and MVDs): If $X \twoheadrightarrow Y$ and there exists W with the properties that (a) $W \cap Y$ is empty, (b) $W \rightarrow Z$, and (c) $Y \supseteq Z$, then $X \rightarrow Z$.

IR1 through IR3 are Armstrong's inference rules for FDs alone. IR4 through IR6 are inference rules pertaining to MVDs only. IR7 and IR8 relate FDs and MVDs. In particular, IR7 says that a functional dependency is a *special case* of a multivalued dependency; that is, every FD is also an MVD because it satisfies the formal definition of an MVD. However, this equivalence has a catch: An FD $X \rightarrow Y$ is an MVD $X \twoheadrightarrow Y$ with the *additional implicit restriction* that at most one value of Y is associated with each value of X.[8] Given a set F of functional and multivalued dependencies specified on $R = \{A_1, A_2, \ldots, A_n\}$, we can use IR1 through IR8 to infer the (complete) set of all dependencies (functional or multivalued) F^+ that will hold in every relation state r of R that satisfies F. We again call F^+ the **closure** of F.

8. That is, the set of values of Y determined by a value of X is restricted to being a *singleton set* with only one value. Hence, in practice, we never view an FD as an MVD.

11.3.3 Fourth Normal Form

We now present the definition of **fourth normal form** (4NF), which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations.

Definition. A relation schema R is in **4NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^+, X is a superkey for R.

The EMP relation of Figure 11.4a is not in 4NF because in the nontrivial MVDs ENAME $\twoheadrightarrow$ PNAME and ENAME $\twoheadrightarrow$ DNAME, ENAME is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 11.4b. Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs ENAME $\twoheadrightarrow$ PNAME in EMP_PROJECTS and ENAME $\twoheadrightarrow$ DNAME in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

To illustrate the importance of 4NF, Figure 11.5a shows the EMP relation with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('W', 'X', 'Y', and 'Z'). There are 16 tuples in EMP in Figure 11.5a. If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 11.5b, we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but the update anomalies associated with multivalued dependencies would also be avoided. For example, if Brown starts working on a new

(a) **EMP**

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

ENAME	PNAME
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

ENAME	DNAME
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

FIGURE 11.5 Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

project P, we must insert *three* tuples in EMP–one for each dependent. If we forget to insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent.

If the relation has nontrivial MVDs, then insert, delete, and update operations on single tuples may cause additional tuples besides the one in question to be modified. If the update is handled incorrectly, the meaning of the relation may change. However, after normalization into 4NF, these update anomalies disappear. For example, to add the information that Brown will be assigned to project P, only a single tuple need be inserted in the 4NF relation EMP_PROJECTS.

The EMP relation in Figure 11.4a is not in 4NF because it represents two *independent* 1:N relationships—one between employees and the projects they work on and the other between employees and their dependents. We sometimes have a relationship among three entities that depends on all three participating entities, such as the SUPPLY relation shown in Figure 11.4c. (Consider only the tuples in Figure 11.4c *above* the dotted line for now.) In this case a tuple represents a supplier supplying a specific part *to a particular project*, so there are *no* nontrivial MVDs. The SUPPLY relation is already in 4NF and should not be decomposed.

11.3.4 Lossless (Nonadditive) Join Decomposition into 4NF Relations

Whenever we decompose a relation schema R into $R_1 = (X \cup Y)$ and $R_2 = (R - Y)$ based on an MVD $X \twoheadrightarrow Y$ that holds in R, the decomposition has the nonadditive join property. It can be shown that this is a necessary and sufficient condition for decomposing a schema into two schemas that have the nonadditive join property, as given by property LJ1′ which is a further generalization of Property LJ1 given earlier. Property LJ1 dealt with FDs only, whereas LJ1′ deals with both FDs and MVDs (recall that an FD is also an MVD).

PROPERTY LJ1′

The relation schemas R_1 and R_2 form a nonadditive join decomposition of R with respect to a set F of functional *and* multivalued dependencies if and only if

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$$

or, by symmetry, if and only if

$$(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1).$$

We can use a slight modification of Algorithm 11.3 to develop Algorithm 11.5, which creates a nonadditive join decomposition into relation schemas that are in 4NF (rather than in BCNF). As with Algorithm 11.3, Algorithm 11.5 does *not* necessarily produce a decomposition that preserves FDs.

Algorithm 11.5: Relational Decomposition into 4NF Relations with Nonadditive Join Property

Input: A universal relation R and a set of functional and multivalued dependencies F.

1. Set $D := \{ R \}$;
2. While there is a relation schema Q in D that is not in 4NF, do
   {choose a relation schema Q in D that is not in 4NF;
   find a nontrivial MVD $X \twoheadrightarrow Y$ in Q that violates 4NF;
   replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
   };

11.4 Join Dependencies and Fifth Normal Form

We saw that LJ1 and LJ1′ give the condition for a relation schema R to be decomposed into two schemas R_1 and R_2, where the decomposition has the nonadditive join property. However, in some cases there may be no nonadditive join decomposition of R into *two* relation schemas, but there may be a nonadditive (lossless) join decomposition into *more than two* relation schemas. Moreover, there may be no functional dependency in R that violates any normal form up to BCNF, and there may be no nontrivial MVD present in R either that violates 4NF. We then resort to another dependency called the *join dependency* and, if it is present, carry out a *multiway decomposition* into fifth normal form (5NF). It is important to note that such a dependency is a very peculiar semantic constraint that is very difficult to detect in practice; therefore, normalization into 5NF is very rarely done in practice.

Definition. A **join dependency** (JD), denoted by JD(R_1, R_2, . . . , R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, . . . , R_n; that is, for every such r we have

$$* (\pi_{R_1}(r), \pi_{R_2}(r), ..., \pi_{R_n}(r)) = r$$

Notice that an MVD is a special case of a JD where $n = 2$. That is, a JD denoted as JD(R_1, R_2) implies an MVD $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (or, by symmetry, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$). A join dependency JD($R_1$, R_2, . . . , R_n), specified on relation schema R, is a **trivial** JD if one of the relation schemas R_i in JD(R_1, R_2, . . . , R_n) is equal to R. Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and hence does not specify any constraint on R. We can now define fifth normal form, which is also called project-join normal form.

Definition. A relation schema R is in **fifth normal form (5NF)** (or **project-join normal form [PJNF]**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, . . . , R_n) in F^+ (that is, implied by F), every R_i is a superkey of R.

For an example of a JD, consider once again the SUPPLY all-key relation of Figure 11.4c. Suppose that the following additional constraint always holds: Whenever a supplier s supplies part p, *and* a project j uses part p, *and* the supplier s supplies *at least one* part to project j, *then* supplier s will also be supplying part p to project j. This constraint can be restated in other ways and specifies a join dependency JD(R1, R2, R3) among the three projections R1(SNAME, PARTNAME), R2(SNAME, PROJNAME), and R3(PARTNAME, PROJNAME) of SUPPLY. If this constraint holds, the tuples below the dotted line in Figure 11.4c must exist in any legal state of the SUPPLY relation that also contains the tuples above the dotted line. Figure 11.4d shows how the SUPPLY relation *with the join dependency* is decomposed into three relations R1, R2, and R3 that are each in 5NF. Notice that applying a natural join to *any two* of these relations *produces spurious tuples*, but applying a natural join to *all three together* does not. The reader should verify this on the example relation of Figure 11.4c and its projections in Figure 11.4d. This is because only the JD exists, but no MVDs are specified. Notice, too, that the JD(R1, R2, R3) is specified on *all* legal relation states, not just on the one shown in Figure 11.4c.

Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer. Hence, the current practice of database design pays scant attention to them.

11.5 Inclusion Dependencies

Inclusion dependencies were defined in order to formalize two types of interrelational constraints:

- The foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations.
- The constraint between two relations that represent a class/subclass relationship (see Chapter 4 and Section 7.2) also has no formal definition in terms of the functional, multivalued, and join dependencies.

Definition. An **inclusion dependency** $R.X < S.Y$ between two sets of attributes—X of relation schema R, and Y of relation schema S—specifies the constraint that, at any specific time when r is a relation state of R and s a relation state of S, we must have

$$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

The $\subseteq$ (subset) relationship does not necessarily have to be a proper subset. Obviously, the sets of attributes on which the inclusion dependency is specified—X of R and Y of S—must have the same number of attributes. In addition, the domains for each pair of corresponding attributes should be compatible. For example, if $X = \{A_1, A_2, \ldots, A_n\}$

and $Y = \{B_1, B_2, \ldots, B_n\}$, one possible correspondence is to have dom(A_i) *Compatible With* dom(B_i) for $1 \leq i \leq n$. In this case, we say that A_i **corresponds to** B_i.

For example, we can specify the following inclusion dependencies on the relational schema in Figure 10.1:

```
DEPARTMENT.DMGRSSN < EMPLOYEE.SSN

WORKS_ON.SSN < EMPLOYEE.SSN

EMPLOYEE.DNUMBER < DEPARTMENT.DNUMBER

PROJECT.DNUM < DEPARTMENT.DNUMBER

WORKS_ON.PNUMBER < PROJECT.PNUMBER

DEPT_LOCATIONS.DNUMBER < DEPARTMENT.DNUMBER
```

All the preceding inclusion dependencies represent **referential integrity constraints.** We can also use inclusion dependencies to represent **class/subclass relationships.** For example, in the relational schema of Figure 7.5, we can specify the following inclusion dependencies:

```
EMPLOYEE.SSN < PERSON.SSN

ALUMNUS.SSN < PERSON.SSN

STUDENT.SSN < PERSON.SSN
```

As with other types of dependencies, there are *inclusion dependency inference rules* (IDIRs). The following are three examples:

IDIR1 (reflexivity): $R.X < R.X$.

IDIR2 (attribute correspondence): If $R.X < S.Y$, where $X = \{A_1, A_2, \ldots, A_n\}$ and $Y = \{B_1, B_2, \ldots, B_n\}$ and A_i *Corresponds to* B_i, then $R.A_i < S.B_i$ for $1 \leq i \leq n$.

IDIR3 (transitivity): If $R.X < S.Y$ and $S.Y < T.Z$, then $R.X < T.Z$.

The preceding inference rules were shown to be sound and complete for inclusion dependencies. So far, no normal forms have been developed based on inclusion dependencies.

11.6 Other Dependencies and Normal Forms

11.6.1 Template Dependencies

Template dependencies provide a technique for representing constraints in relations that typically have no easy and formal definitions. No matter how many types of dependencies we develop, some peculiar constraint may come up based on the semantics of attributes within relations that cannot be represented by any of them. The idea behind template dependencies is to specify a template—or example—that defines each constraint or dependency.

There are two types of templates: tuple-generating templates and constraint-generating templates. A template consists of a number of **hypothesis tuples** that are meant to show an example of the tuples that may appear in one or more relations. The other part of the template is the **template conclusion.** For tuple-generating templates, the conclusion is a *set*

of tuples that must also exist in the relations if the hypothesis tuples are there. For constraint-generating templates, the template conclusion is a *condition* that must hold on the hypothesis tuples.

Figure 11.6 shows how we may define functional, multivalued, and inclusion dependencies by templates. Figure 11.7 shows how we may specify the constraint that "an

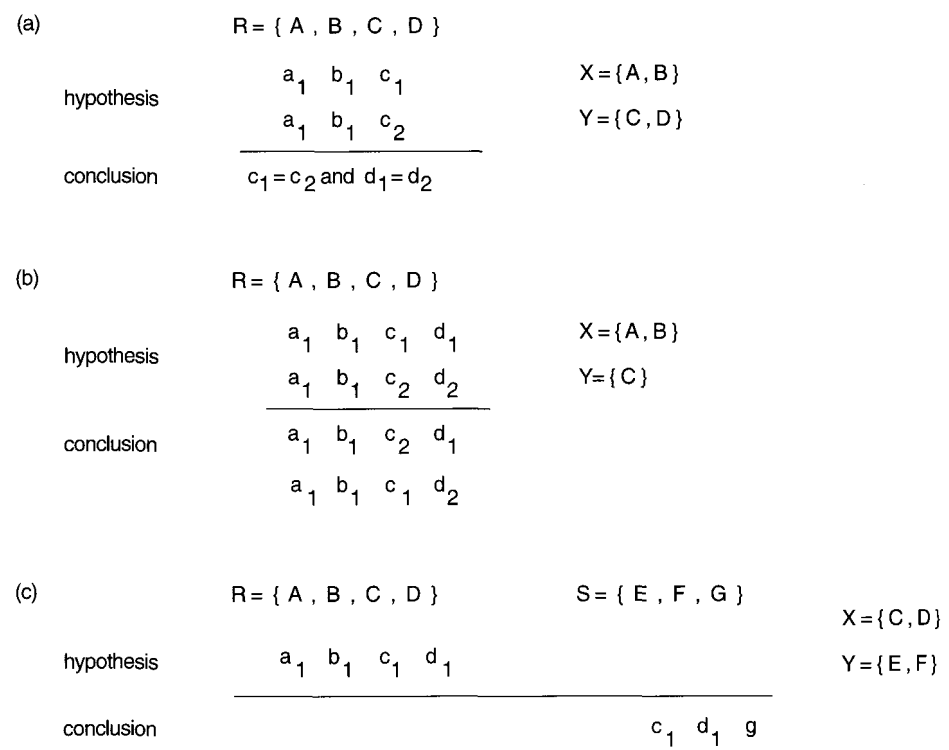


FIGURE 11.6 Templates for some common type of dependencies. (a) Template for functional dependency $X \rightarrow Y$. (b) Template for the multivalued dependency $X \twoheadrightarrow Y$. (c) Template for the inclusion dependency $R.X < S.Y$.

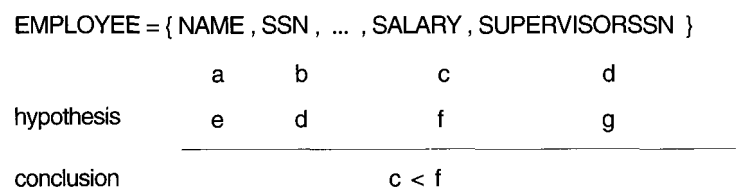


FIGURE 11.7 Templates for the constraint that an employee's salary must be less than the supervisor's salary.

employee's salary cannot be higher than the salary of his or her direct supervisor" on the relation schema EMPLOYEE in Figure 5.5.

11.6.2 Domain-Key Normal Form

There is no hard and fast rule about defining normal forms only up to 5NF. Historically, the process of normalization and the process of discovering undesirable dependencies was carried through 5NF, but it has been possible to define stricter normal forms that take into account additional types of dependencies and constraints. The idea behind **domain-key normal form (DKNF)** is to specify (theoretically, at least) the "ultimate normal form" that takes into account all possible types of dependencies and constraints. A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation. For a relation in DKNF, it becomes very straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.

However, because of the difficulty of including complex constraints in a DKNF relation, its practical utility is limited, since it may be quite difficult to specify general integrity constraints. For example, consider a relation CAR(MAKE, VIN#) (where VIN# is the vehicle identification number) and another relation MANUFACTURE(VIN#, COUNTRY) (where COUNTRY is the country of manufacture). A general constraint may be of the following form: "If the MAKE is either Toyota or Lexus, then the first character of the VIN# is a "J" if the country of manufacture is Japan; if the MAKE is Honda or Acura, the second character of the VIN# is a "J" if the country of manufacture is Japan." There is no simplified way to represent such constraints short of writing a procedure (or general assertions) to test them.

11.7 SUMMARY

In this chapter we presented several normalization algorithms. The *relational synthesis algorithms* create 3NF relations from a universal relation schema based on a given set of functional dependencies that has been specified by the database designer. The relational decomposition algorithms create BCNF (or 4NF) relations by successive nonadditive decomposition of unnormalized relations into two component relations at a time. We first discussed two important properties of decompositions: the lossless (nonadditive) join property, and the dependency-preserving property. An algorithm to test for lossless decomposition, and a simpler test for checking the losslessness of binary decompositions, were described. We saw that it is possible to synthesize 3NF relation schemas that meet both of the above properties; however, in the case of BCNF, it is possible to aim only for the nonadditiveness of joins—dependency preservation *cannot* be necessarily guaranteed. If one has to aim for one of these two, the nonadditive join condition is an absolute must.

We then defined additional types of dependencies and some additional normal forms. Multivalued dependencies, which arise from an improper combination of two or more independent multivalued attributes in the same relation, are used to define fourth normal

form (4NF). Join dependencies, which indicate a lossless multiway decomposition of a relation, lead to the definition of fifth normal form (5NF), which is also known as project-join normal form (PJNF). We also discussed inclusion dependencies, which are used to specify referential integrity and class/subclass constraints, and template dependencies, which can be used to specify arbitrary types of constraints. We concluded with a brief discussion of the domain-key normal form (DKNF).

Review Questions

11.1. What is meant by the attribute preservation condition on a decomposition?
11.2. Why are normal forms alone insufficient as a condition for a good schema design?
11.3. What is the dependency preservation property for a decomposition? Why is it important?
11.4. Why can we not guarantee that BCNF relation schemas will be produced by dependency-preserving decompositions of non-BCNF relation schemas? Give a counterexample to illustrate this point.
11.5. What is the lossless (or nonadditive) join property of a decomposition? Why is it important?
11.6. Between the properties of dependency preservation and losslessness, which one must definitely be satisfied? Why?
11.7. Discuss the null value and dangling tuple problems.
11.8. What is a multivalued dependency? What type of constraint does it specify? When does it arise?
11.9. Illustrate how the process of creating first normal form relations may lead to multivalued dependencies. How should the first normalization be done properly so that MVDs are avoided?
11.10. Define fourth normal form. When is it violated? Why is it useful?
11.11. Define join dependencies and fifth normal form. Why is 5NF also called project-join normal form (PJNF)?
11.12. What types of constraints are inclusion dependencies meant to represent?
11.13. How do template dependencies differ from the other types of dependencies we discussed?
11.14. Why is the domain-key normal form (DKNF) known as the ultimate normal form?

Exercises

11.15. Show that the relation schemas produced by Algorithm 11.2 are in 3NF.
11.16. Show that, if the matrix S resulting from Algorithm 11.1 does not have a row that is all "a" symbols, projecting S on the decomposition and joining it back will always produce at least one spurious tuple.
11.17. Show that the relation schemas produced by Algorithm 11.3 are in BCNF.
11.18. Show that the relation schemas produced by Algorithm 11.4 are in 3NF.
11.19. Specify a template dependency for join dependencies.
11.20. Specify all the inclusion dependencies for the relational schema of Figure 5.5.

11.21. Prove that a functional dependency satisfies the formal definition of multivalued dependency.

11.22. Consider the example of normalizing the LOTS relation in Section 10.4. Determine whether the decomposition of LOTS into {LOTS1AX, LOTS1AY, LOTS1B, LOTS2} has the lossless join property, by applying Algorithm 11.1 and also by using the test under Property LJ1.

11.23. Show how the MVDs ENAME $\twoheadrightarrow$ PNAME and ENAME $\twoheadrightarrow$ DNAME in Figure 11.4a may arise during normalization into 1NF of a relation, where the attributes PNAME and DNAME are multivalued.

11.24. Apply Algorithm 11.4a to the relation in Exercise 10.26 to determine a key for R. Create a minimal set of dependencies G that is equivalent to F, and apply the synthesis algorithm (Algorithm 11.4) to decompose R into 3NF relations.

11.25. Repeat Exercise 11.24 for the functional dependencies in Exercise 10.27.

11.26. Apply the decomposition algorithm (Algorithm 11.3) to the relation R and the set of dependencies F in Exercise 10.26. Repeat for the dependencies G in Exercise 10.27.

11.27. Apply Algorithm 11.4a to the relations in Exercises 10.29 and 10.30 to determine a key for R. Apply the synthesis algorithm (Algorithm 11.4) to decompose R into 3NF relations and the decomposition algorithm (Algorithm 11.3) to decompose R into BCNF relations.

11.28. Write programs that implement Algorithms 11.3 and 11.4.

11.29. Consider the following decompositions for the relation schema R of Exercise 10.26. Determine whether each decomposition has (i) the dependency preservation property, and (ii) the lossless join property, with respect to F. Also determine which normal form each relation in the decomposition is in.

a. $D_1 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C\}$, $R_2 = \{A, D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

b. $D_2 = \{R_1, R_2, R_3\}$; $R_1 = \{A, B, C, D, E\}$, $R_2 = \{B, F, G, H\}$, $R_3 = \{D, I, J\}$

c. $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C, D\}$, $R_2 = \{D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

11.30. Consider the relation REFRIG(MODEL#, YEAR, PRICE, MANUF_PLANT, COLOR), which is abbreviated as REFRIG(M, Y, P, MP, C), and the following set F of functional dependencies: F = {M $\rightarrow$ MP, {M, Y} $\rightarrow$ P, MP $\rightarrow$ C}

a. Evaluate each of the following as a candidate key for REFRIG, giving reasons why it can or cannot be a key: {M}, {M, Y}, {M, C}.

b. Based on the above key determination, state whether the relation REFRIG is in 3NF and in BCNF, giving proper reasons.

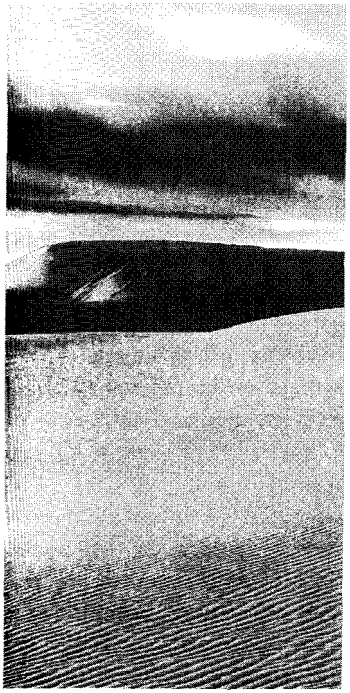
c. Consider the decomposition of REFRIG into D = {R1(M, Y, P), R2(M, MP, C)}. Is this decomposition lossless? Show why. (You may consult the test under Property LJ1 in Section 11.1.4.)

Selected Bibliography

The books by Maier (1983) and Atzeni and De Antonellis (1992) include a comprehensive discussion of relational dependency theory. The decomposition algorithm (Algorithm 11.3) is due to Bernstein (1976). Algorithm 11.4 is based on the normalization algorithm presented in Biskup et al. (1979). Tsou and Fischer (1982) give a polynomial-time algorithm for BCNF decomposition.

The theory of dependency preservation and lossless joins is given in Ullman (1988), where proofs of some of the algorithms discussed here appear. The lossless join property is analyzed in Aho et al. (1979). Algorithms to determine the keys of a relation from functional dependencies are given in Osborn (1976); testing for BCNF is discussed in Osborn (1979). Testing for 3NF is discussed in Tsou and Fischer (1982). Algorithms for designing BCNF relations are given in Wang (1990) and Hernandez and Chan (1991).

Multivalued dependencies and fourth normal form are defined in Zaniolo (1976) and Nicolas (1978). Many of the advanced normal forms are due to Fagin: the fourth normal form in Fagin (1977), PJNF in Fagin (1979), and DKNF in Fagin (1981). The set of sound and complete rules for functional and multivalued dependencies was given by Beeri et al. (1977). Join dependencies are discussed by Rissanen (1977) and Aho et al. (1979). Inference rules for join dependencies are given by Sciore (1982). Inclusion dependencies are discussed by Casanova et al. (1981) and analyzed further in Cosmadakis et al. (1990). Their use in optimizing relational schemas is discussed in Casanova et al. (1989). Template dependencies are discussed by Sadri and Ullman (1982). Other dependencies are discussed in Nicolas (1978), Furtado (1978), and Mendelzon and Maier (1979). Abiteboul et al. (1995) provides a theoretical treatment of many of the ideas presented in this chapter and Chapter 10.



12

Practical Database Design Methodology and Use of UML Diagrams

In this chapter we move from the theory to the practice of database design. We have already described in several chapters material that is relevant to the design of actual databases for practical real-world applications. This material includes Chapters 3 and 4 on database conceptual modeling; Chapters 5 through 9 on the relational model, the SQL language, relational algebra and calculus, mapping a high-level conceptual ER or EER schema into a relational schema, and programming in relational systems (RDBMSs); and Chapters 10 and 11 on data dependency theory and relational normalization algorithms.

The overall database design activity has to undergo a systematic process called the **design methodology,** whether the target database is managed by an RDBMS, object database management systems (ODBMS), or object relational database management systems (ORDBMS). Various design methodologies are implicit in the database design tools currently supplied by vendors. Popular tools include Designer 2000 by Oracle; ERWin, BPWin, and Paradigm Plus by Platinum Technology; Sybase Enterprise Application Studio; ER Studio by Embarcadero Technologies; and System Architect by Popkin Software, among many others. Our goal in this chapter is to discuss not one specific methodology but rather database design in a broader context, as it is undertaken in large organizations for the design and implementation of applications catering to hundreds or thousands of users.

Generally, the design of small databases with perhaps up to 20 users need not be very complicated. But for medium-sized or large databases that serve several diverse application groups, each with tens or hundreds of users, a systematic approach to the

overall database design activity becomes necessary. The sheer size of a populated database does not reflect the complexity of the design; it is the schema that is more important. Any database with a schema that includes more than 30 or 40 entity types and a similar number of relationship types requires a careful design methodology.

Using the term **large database** for databases with several tens of gigabytes of data and a schema with more than 30 or 40 distinct entity types, we can cover a wide array of databases in government, industry, and financial and commercial institutions. Service sector industries, including banking, hotels, airlines, insurance, utilities, and communications, use databases for their day-to-day operations 24 hours a day, 7 days a week—known in industry as *24 by 7* operations. Application systems for these databases are called *transaction processing systems* due to the large transaction volumes and rates that are required. In this chapter we will be concentrating on the database design for such medium- and large- scale databases where transaction processing dominates.

This chapter has a variety of objectives. Section 12.1 discusses the information system life cycle within organizations with a particular emphasis on the database system. Section 12.2 highlights the phases of a database design methodology in the organizational context. Section 12.3 introduces UML diagrams and gives details on the notations of some of them that are particularly helpful in collecting requirements, and performing coneptual and logical design of databases. An illustrative partial example of designing a university database is presented. Section 12.4 introduces the popular software development tool called Rational Rose which has UML diagrams as its main specification technique. Features of Rational Rose that are specific to database requirements modeling and schema design are highlighted. Section 12.5 briefly discusses automated database design tools.

12.1 The Role of Information Systems in Organizations

12.1.1 The Organizational Context for Using Database Systems

Database systems have become a part of the information systems of many organizations. In the 1960s information systems were dominated by file systems, but since the early 1970s organizations have gradually moved to database systems. To accommodate such systems, many organizations have created the position of database administrator (DBA) or even database administration departments to oversee and control database life-cycle activities. Similarly, information technology (IT), and information resource management (IRM) have been recognized by large organizations to be a key to successful management of the business. There are several reasons for this:

- Data is regarded as a corporate resource, and its management and control is considered central to the effective working of the organization.
- More functions in organizations are computerized, increasing the need to keep large volumes of data available in an up-to-the-minute current state.

- As the complexity of the data and applications grows, complex relationships among the data need to be modeled and maintained.
- There is a tendency toward consolidation of information resources in many organizations.
- Many organizations are reducing their personnel costs by letting the end-user perform business transactions. This is evident in the form of travel services, financial services, online retail goods outlet and customer-to-business electronic commerce examples such as amazon.com or Ebay. In these instances, a publicly accessible and updatable operational database must be designed and made available for these transactions.

Database systems satisfy the preceding requirements in large measure. Two additional characteristics of database systems are also very valuable in this environment:

- *Data independence* protects application programs from changes in the underlying logical organization and in the physical access paths and storage structures.
- *External schemas* (views) allow the same data to be used for multiple applications, with each application having its own view of the data.

New capabilities provided by database systems and the following key features that they offer have made them integral components in computer-based information systems:

- Integration of data across multiple applications into a single database.
- Simplicity of developing new applications using high-level languages like SQL.
- Possibility of supporting casual access for browsing and querying by managers while supporting major production-level transaction processing.

From the early 1970s through the mid-1980s, the move was toward creating large centralized repositories of data managed by a single centralized DBMS. Over the last 10 to 15 years, this trend has been reversed because of the following developments:

1. Personal computers and database system-like software products, such as EXCEL, FOXPRO, ACCESS (all of Microsoft), or SQL Anywhere (of Sybase), and public domain products such as MYSQL are being heavily utilized by users who previously belonged to the category of casual and occasional database users. Many administrators, secretaries, engineers, scientists, architects, and the like belong to this category. As a result, the practice of creating **personal databases** is gaining popularity. It is now possible to check out a copy of part of a large database from a mainframe computer or a database server, work on it from a personal workstation, and then re-store it on the mainframe. Similarly, users can design and create their own databases and then merge them into a larger one.
2. The advent of distributed and client-server DBMSs (see Chapter 25) is opening up the option of distributing the database over multiple computer systems for better local control and faster local processing. At the same time, local users can access remote data using the facilities provided by the DBMS as a client, or through the Web. Application development tools such as PowerBuilder or Developer 2000 (by Oracle) are being used heavily with built-in facilities to link applications to multiple back-end database servers.

3. Many organizations now use **data dictionary systems** or **information repositories,** which are mini DBMSs that manage **metadata**—that is, data that describes the database structure, constraints, applications, authorizations, and so on. These are often used as an integral tool for information resource management. A useful data dictionary system should store and manage the following types of information:
   a. Descriptions of the schemas of the database system.
   b. Detailed information on physical database design, such as storage structures, access paths, and file and record sizes.
   c. Descriptions of the database users, their responsibilities, and their access rights.
   d. High-level descriptions of the database transactions and applications and of the relationships of users to transactions.
   e. The relationship between database transactions and the data items referenced by them. This is useful in determining which transactions are affected when certain data definitions are changed.
   f. Usage statistics such as frequencies of queries and transactions and access counts to different portions of the database.

This metadata is available to DBAs, designers, and authorized users as online system documentation. This improves the control of DBAs over the information system and the users' understanding and use of the system. The advent of data warehousing technology has highlighted the importance of metadata.

When designing high-performance **transaction processing systems,** which require around-the-clock nonstop operation, performance becomes critical. These databases are often accessed by hundreds of transactions per minute from remote and local terminals. Transaction performance, in terms of the average number of transactions per minute and the average and maximum transaction response time, is critical. A careful physical database design that meets the organization's transaction processing needs is a must in such systems.

Some organizations have committed their information resource management to certain DBMS and data dictionary products. Their investment in the design and implementation of large and complex systems makes it difficult for them to change to newer DBMS products, which means that the organizations become locked in to their current DBMS system. With regard to such large and complex databases, we cannot overemphasize the importance of a careful design that takes into account the need for possible system modifications—called tuning—to respond to changing requirements. We will discuss tuning in conjunction with query optimization in Chapter 16. The cost can be very high if a large and complex system cannot evolve, and it becomes necessary to move to other DBMS products.

12.1.2 The Information System Life Cycle

In a large organization, the database system is typically part of the **information system,** which includes all resources that are involved in the collection, management, use, and dissemination of the information resources of the organization. In a computerized environment, these resources include the data itself, the DBMS software, the computer system hardware and storage media, the personnel who use and manage the data (DBA, end users,

parametric users, and so on), the applications software that accesses and updates the data, and the application programmers who develop these applications. Thus the database system is part of a much larger organizational information system.

In this section we examine the typical life cycle of an information system and how the database system fits into this life cycle. The information system life cycle is often called the **macro life cycle,** whereas the database system life cycle is referred to as the **micro life cycle.** The distinction between these two is becoming fuzzy for information systems where databases are a major integral component. The macro life cycle typically includes the following phases:

1. *Feasibility analysis:* This phase is concerned with analyzing potential application areas, identifying the economics of information gathering and dissemination, performing preliminary cost-benefit studies, determining the complexity of data and processes, and setting up priorities among applications.
2. *Requirements collection and analysis:* Detailed requirements are collected by interacting with potential users and user groups to identify their particular problems and needs. Interapplication dependencies, communication, and reporting procedures are identified.
3. *Design:* This phase has two aspects: the design of the database system, and the design of the application systems (programs) that use and process the database.
4. *Implementation:* The information system is implemented, the database is loaded, and the database transactions are implemented and tested.
5. *Validation and acceptance testing:* The acceptability of the system in meeting users' requirements and performance criteria is validated. The system is tested against performance criteria and behavior specifications.
6. *Deployment, operation and maintenance:* This may be preceded by conversion of users from an older system as well as by user training. The operational phase starts when all system functions are operational and have been validated. As new requirements or applications crop up, they pass through all the previous phases until they are validated and incorporated into the system. Monitoring of system performance and system maintenance are important activities during the operational phase.

12.1.3 The Database Application System Life Cycle

Activities related to the database application system (micro) life cycle include the following:

1. *System definition:* The scope of the database system, its users, and its applications are defined. The interfaces for various categories of users, the response time constraints, and storage and processing needs are identified.
2. *Database design:* At the end of this phase, a complete logical and physical design of the database system on the chosen DBMS is ready.

3. *Database implementation:* This comprises the process of specifying the conceptual, external, and internal database definitions, creating empty database files, and implementing the software applications.
4. *Loading or data conversion:* The database is populated either by loading the data directly or by converting existing files into the database system format.
5. *Application conversion:* Any software applications from a previous system are converted to the new system.
6. *Testing and validation:* The new system is tested and validated.
7. *Operation:* The database system and its applications are put into operation. Usually, the old and the new systems are operated in parallel for some time.
8. *Monitoring and maintenance:* During the operational phase, the system is constantly monitored and maintained. Growth and expansion can occur in both data content and software applications. Major modifications and reorganizations may be needed from time to time.

Activities 2, 3, and 4 together are part of the design and implementation phases of the larger information system life cycle. Our emphasis in Section 12.2 is on activities 2 and 3, which cover the database design and implementation phases. Most databases in organizations undergo all of the preceding life-cycle activities. The conversion activities (4 and 5) are not applicable when both the database and the applications are new. When an organization moves from an established system to a new one, activities 4 and 5 tend to be the most time-consuming and the effort to accomplish them is often underestimated. In general, there is often feedback among the various steps because new requirements frequently arise at every stage. Figure 12.1 shows the feedback loop affecting the conceptual and logical design phases as a result of system implementation and tuning.

12.2 The Database Design and Implementation Process

We now focus on activities 2 and 3 of the database application system life cycle, which are database design and implementation. The problem of database design can be stated as follows:

> Design the logical and physical structure of one or more databases to accommodate the information needs of the users in an organization for a defined set of applications.

The goals of database design are multiple:

- Satisfy the information content requirements of the specified users and applications.
- Provide a natural and easy-to-understand structuring of the information.
- Support processing requirements and any performance objectives, such as response time, processing time, and storage space.

These goals are very hard to accomplish and measure, and they involve an inherent tradeoff: if one attempts to achieve more "naturalness" and "understandability" of the model, it may be at the cost of performance. The problem is aggravated because the database design process often begins with informal and poorly defined requirements. In contrast, the result of the design activity is a rigidly defined database schema that cannot easily be modified once the database is implemented. We can identify six main phases of the overall database design and implementation process:

1. Requirements collection and analysis.
2. Conceptual database design.
3. Choice of a DBMS.
4. Data model mapping (also called logical database design).
5. Physical database design.
6. Database system implementation and tuning.

The design process consists of two parallel activities, as illustrated in Figure 12.1. The first activity involves the design of the **data content and structure** of the database; the second relates to the design of **database applications.** To keep the figure simple, we have avoided showing most of the interactions among these two sides, but the two activities are closely intertwined. For example, by analyzing database applications, we can identify data items that will be stored in the database. In addition, the physical database design phase, during which we choose the storage structures and access paths of database files, depends on the applications that will use these files. On the other hand, we usually specify the design of database applications by referring to the database schema constructs, which are specified during the first activity. Clearly, these two activities strongly influence one another. Traditionally, database design methodologies have primarily focused on the first of these activities whereas software design has focused on the second; this may be called **data-driven** versus **process-driven design.** It is rapidly being recognized by database designers and software engineers that the two activities should proceed hand in hand, and design tools are increasingly combining them.

The six phases mentioned previously do not have to proceed strictly in sequence. In many cases we may have to modify the design from an earlier phase during a later phase. These **feedback loops** among phases—and also within phases—are common. We show only a couple of feedback loops in Figure 12.1, but many more exist between various pairs of phases. We have also shown some interaction between the data and the process sides of the figure; many more interactions exist in reality. Phase 1 in Figure 12.1 involves collecting information about the intended use of the database, and Phase 6 concerns database implementation and redesign. The heart of the database design process comprises Phases 2, 4, and 5; we briefly summarize these phases:

- *Conceptual database design (Phase 2)*: The goal of this phase is to produce a conceptual schema for the database that is independent of a specific DBMS. We often use a high-level data model such as the ER or EER model (see Chapters 3 and 4) during this phase. In addition, we specify as many of the known database applications or transactions as possible, using a notation that is independent of any specific DBMS. Often,

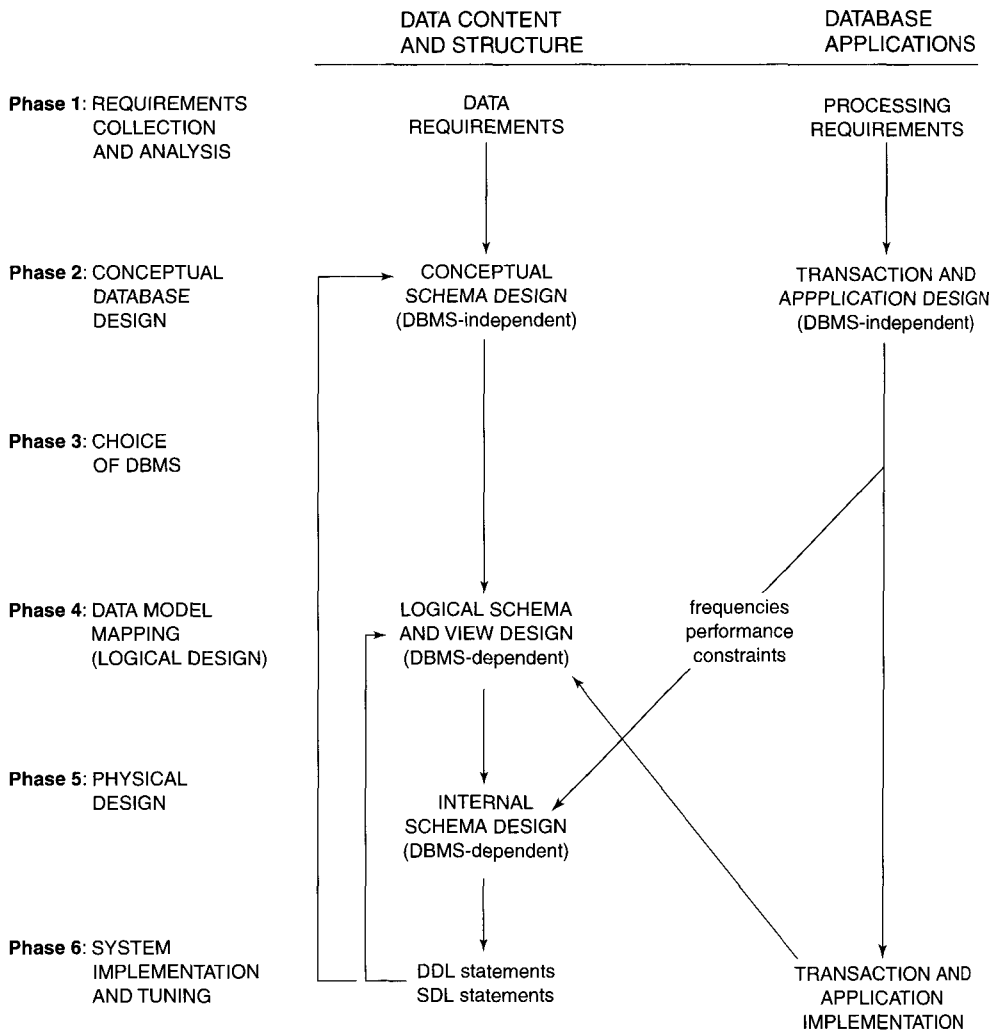


FIGURE 12.1 Phases of database design and implementation for large databases.

the DBMS choice is already made for the organization; the intent of conceptual design is still to keep it as free as possible from implementation considerations.

- *Data model mapping (Phase 4)*: During this phase, which is also called **logical database design,** we **map** (or **transform**) the conceptual schema from the high-level data model used in Phase 2 into the data model of the chosen DBMS. We can start this phase after choosing a specific type of DBMS—for example, if we decide to use some relational DBMS but have not yet decided on which particular one. We call the latter *system-independent* (but *data model-dependent*) logical design. In terms of the three-

level DBMS architecture discussed in Chapter 2, the result of this phase is a *conceptual schema* in the chosen data model. In addition, the design of *external schemas* (views) for specific applications is often done during this phase.

- *Physical database design (Phase 5)*: During this phase, we design the specifications for the stored database in terms of physical storage structures, record placement, and indexes. This corresponds to designing the *internal schema* in the terminology of the three-level DBMS architecture.
- *Database system implementation and tuning (Phase 6)*: During this phase, the database and application programs are implemented, tested, and eventually deployed for service. Various transactions and applications are tested individually and then in conjunction with each other. This typically reveals opportunities for physical design changes, data indexing, reorganization, and different placement of data—an activity referred to as **database tuning.** Tuning is an ongoing activity—a part of system maintenance that continues for the life cycle of a database as long as the database and applications keep evolving and performance problems are detected.

In the following subsections we discuss each of the six phases of database design in more detail.

12.2.1 Phase 1: Requirements Collection and Analysis[1]

Before we can effectively design a database, we must know and analyze the expectations of the users and the intended uses of the database in as much detail as possible. This process is called **requirements collection and analysis.** To specify the requirements, we must first identify the other parts of the information system that will interact with the database system. These include new and existing users and applications, whose requirements are then collected and analyzed. Typically, the following activities are part of this phase:

1. The major application areas and user groups that will use the database or whose work will be affected by it are identified. Key individuals and committees within each group are chosen to carry out subsequent steps of requirements collection and specification.
2. Existing documentation concerning the applications is studied and analyzed. Other documentation—policy manuals, forms, reports, and organization charts—is reviewed to determine whether it has any influence on the requirements collection and specification process.
3. The current operating environment and planned use of the information is studied. This includes analysis of the types of transactions and their frequencies as well as of the flow of information within the system. Geographic characteristics regarding users, origin of transactions, destination of reports, and so forth, are studied. The input and output data for the transactions are specified.

1. A part of this section has been contributed by Colin Potts.

4. Written responses to sets of questions are sometimes collected from the potential database users or user groups. These questions involve the users' priorities and the importance they place on various applications. Key individuals may be interviewed to help in assessing the worth of information and in setting up priorities.

Requirement analysis is carried out for the final users, or "customers," of the database system by a team of analysts or requirement experts. The initial requirements are likely to be informal, incomplete, inconsistent, and partially incorrect. Much work therefore needs to be done to transform these early requirements into a specification of the application that can be used by developers and testers as the starting point for writing the implementation and test cases. Because the requirements reflect the initial understanding of a system that does not yet exist, they will inevitably change. It is therefore important to use techniques that help customers converge quickly on the implementation requirements.

There is a lot of evidence that customer participation in the development process increases customer satisfaction with the delivered system. For this reason, many practitioners now use meetings and workshops involving all stakeholders. One such methodology of refining initial system requirements is called Joint Application Design (JAD). More recently, techniques have been developed, such as Contextual Design, that involve the designers becoming immersed in the workplace in which the application is to be used. To help customer representatives better understand the proposed system, it is common to walk through workflow or transaction scenarios or to create a mock-up prototype of the application.

The preceding modes help structure and refine requirements but leave them still in an informal state. To transform requirements into a better structured form, **requirements specification techniques** are used. These include OOA (object-oriented analysis), DFDs (data flow diagrams), and the refinement of application goals. These methods use diagramming techniques for organizing and presenting information-processing requirements. Additional documentation in the form of text, tables, charts, and decision requirements usually accompanies the diagrams. There are techniques that produce a formal specification that can be checked mathematically for consistency and "what-if" symbolic analyses. These methods are hardly used now but may become standard in the future for those parts of information systems that serve mission-critical functions and which therefore must work as planned. The model-based formal specification methods, of which the Z-notation and methodology is the most prominent, can be thought of as extensions of the ER model and are therefore the most applicable to information system design.

Some computer-aided techniques—called "Upper CASE" tools—have been proposed to help check the consistency and completeness of specifications, which are usually stored in a single repository and can be displayed and updated as the design progresses. Other tools are used to trace the links between requirements and other design entities, such as code modules and test cases. Such *traceability databases* are especially important in conjunction with enforced change-management procedures for systems where the requirements change frequently. They are also used in contractual projects where the development organization must provide documentary evidence to the customer that all the requirements have been implemented.

The requirements collection and analysis phase can be quite time-consuming, but it is crucial to the success of the information system. Correcting a requirements error is much more expensive than correcting an error made during implementation, because the effects of a requirements error are usually pervasive, and much more downstream work has to be re-implemented as a result. Not correcting the error means that the system will not satisfy the customer and may not even be used at all. Requirements gathering and analysis have been the subject of entire books.

12.2.2 Phase 2: Conceptual Database Design

The second phase of database design involves two parallel activities.[2] The first activity, **conceptual schema design,** examines the data requirements resulting from Phase 1 and produces a conceptual database schema. The second activity, **transaction and application design,** examines the database applications analyzed in Phase 1 and produces high-level specifications for these applications.

Phase 2a: Conceptual Schema Design. The conceptual schema produced by this phase is usually contained in a DBMS-independent high-level data model for the following reasons:

1. The goal of conceptual schema design is a complete understanding of the database structure, meaning (semantics), interrelationships, and constraints. This is best achieved independently of a specific DBMS because each DBMS typically has idiosyncrasies and restrictions that should not be allowed to influence the conceptual schema design.
2. The conceptual schema is invaluable as a *stable description* of the database contents. The choice of DBMS and later design decisions may change without changing the DBMS-independent conceptual schema.
3. A good understanding of the conceptual schema is crucial for database users and application designers. Use of a high-level data model that is more expressive and general than the data models of individual DBMSs is hence quite important.
4. The diagrammatic description of the conceptual schema can serve as an excellent vehicle of communication among database users, designers, and analysts. Because high-level data models usually rely on concepts that are easier to understand than lower-level DBMS-specific data models, or syntactic definitions of data, any communication concerning the schema design becomes more exact and more straightforward.

In this phase of database design, it is important to use a conceptual high-level data model with the following characteristics:

2. This phase of design is discussed in great detail in the first seven chapters of Batini et al. (1992); we summarize that discussion here.

1. *Expressiveness:* The data model should be expressive enough to distinguish different types of data, relationships, and constraints.
2. *Simplicity and understandability:* The model should be simple enough for typical nonspecialist users to understand and use its concepts.
3. *Minimality:* The model should have a small number of basic concepts that are distinct and nonoverlapping in meaning.
4. *Diagrammatic representation:* The model should have a diagrammatic notation for displaying a conceptual schema that is easy to interpret.
5. *Formality:* A conceptual schema expressed in the data model must represent a formal unambiguous specification of the data. Hence, the model concepts must be defined accurately and unambiguously.

Many of these requirements—the first one in particular—sometimes conflict with other requirements. Many high-level conceptual models have been proposed for database design (see the selected bibliography for Chapter 4). In the following discussion, we will use the terminology of the Enhanced Entity-Relationship (EER) model presented in Chapter 4, and we will assume that it is being used in this phase. Conceptual schema design, including data modeling, is becoming an integral part of object-oriented analysis and design methodologies. The UML has class diagrams that are largely based on extensions of the EER model.

Approaches to Conceptual Schema Design. For conceptual schema design, we must identify the basic components of the schema: the entity types, relationship types, and attributes. We should also specify key attributes, cardinality and participation constraints on relationships, weak entity types, and specialization/generalization hierarchies/lattices. There are two approaches to designing the conceptual schema, which is derived from the requirements collected during Phase 1.

The first approach is the **centralized** (or **one-shot**) **schema design approach,** in which the requirements of the different applications and user groups from Phase 1 are merged into a single set of requirements before schema design begins. A single schema corresponding to the merged set of requirements is then designed. When many users and applications exist, merging all the requirements can be an arduous and time-consuming task. The assumption is that a centralized authority, the DBA, is responsible for deciding how to merge the requirements and for designing the conceptual schema for the whole database. Once the conceptual schema is designed and finalized, external schemas for the various user groups and applications can be specified by the DBA.

The second approach is the **view integration approach,** in which the requirements are not merged. Rather a schema (or view) is designed for each user group or application based only on its own requirements. Thus we develop one high-level schema (view) for each such user group or application. During a subsequent **view integration** phase, these schemas are merged or integrated into a **global conceptual schema** for the entire database. The individual views can be reconstructed as external schemas after view integration.

The main difference between the two approaches lies in the manner and stage in which multiple views or requirements of the many users and applications are reconciled and merged. In the centralized approach, the reconciliation is done manually by the DBA's staff prior to designing any schemas and is applied directly to the requirements collected in Phase 1. This places the burden to reconcile the differences and conflicts among user groups on the DBA's staff. The problem has been typically dealt with by using external consultants/design experts to bring in their own ways of resolving these conflicts. Because of the difficulties of managing this task, the view integration approach is now gaining more acceptance.

In the view integration approach, each user group or application actually designs its own conceptual (EER) schema from its requirements. Then an integration process is applied to these schemas (views) by the DBA to form the global integrated schema. Although view integration can be done manually, its application to a large database involving tens of user groups requires a methodology and the use of automated tools to help in carrying out the integration. The correspondences among the attributes, entity types, and relationship types in various views must be specified before the integration can be applied. In addition, problems such as integrating conflicting views and verifying the consistency of the specified interschema correspondences must be dealt with.

Strategies for Schema Design. Given a set of requirements, whether for a single user or for a large user community, we must create a conceptual schema that satisfies these requirements. There are various strategies for designing such a schema. Most strategies follow an incremental approach— that is, they start with some schema constructs derived from the requirements and then they incrementally modify, refine, or build on them. We now discuss some of these strategies:

1. *Top-down strategy:* We start with a schema containing high-level abstractions and then apply successive top-down refinements. For example, we may specify only a few high-level entity types and then, as we specify their attributes, split them into lower-level entity types and relationships. The process of specialization to refine an entity type into subclasses that we illustrated in Sections 4.2 and 4.3 (see Figures 4.1, 4.4, and 4.5) is another example of a top-down design strategy.
2. *Bottom-up strategy:* Start with a schema containing basic abstractions and then combine or add to these abstractions. For example, we may start with the attributes and group these into entity types and relationships. We may add new relationships among entity types as the design progresses. The process of generalizing entity types into higher-level generalized superclasses (see Sections 4.2 and 4.3, Figure 4.3) is another example of a bottom-up design strategy.
3. *Inside-out strategy:* This is a special case of a bottom-up strategy, where attention is focused on a central set of concepts that are most evident. Modeling then *spreads outward* by considering new concepts in the vicinity of existing ones. We could specify a few clearly evident entity types in the schema and continue by adding other entity types and relationships that are related to each.

4. *Mixed strategy:* Instead of following any particular strategy throughout the design, the requirements are partitioned according to a top-down strategy, and part of the schema is designed for each partition according to a bottom-up strategy. The various schema parts are then combined.

Figures 12.2 and 12.3 illustrate top-down and bottom-up refinement, respectively. An example of a top-down refinement primitive is decomposition of an entity type into several entity types. Figure 12.2(a) shows a COURSE being refined into COURSE and SEMINAR, and the TEACHES relationship is correspondingly split into TEACHES and OFFERS. Figure 12.2(b) shows a COURSE_OFFERING entity type being refined into two entity types (COURSE and INSTRUCTOR) and a relationship between them. Refinement typically forces a designer to ask more questions and extract more constraints and details: for example, the (min, max) cardinality ratios between COURSE and INSTRUCTOR are obtained during refinement. Figure 12.3(a) shows the bottom-up refinement primitive of generating new relationships among

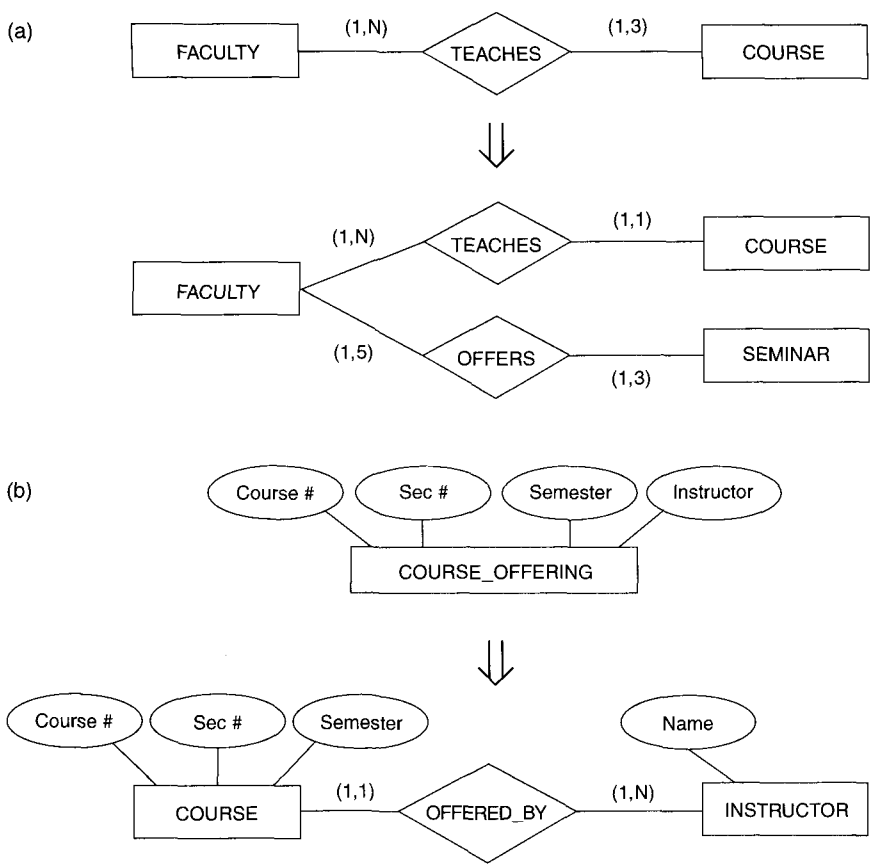


FIGURE 12.2 Examples of top-down refinement. (a) Generating a new entity type. (b) Decomposing an entity type into two entity types and a relationship type.

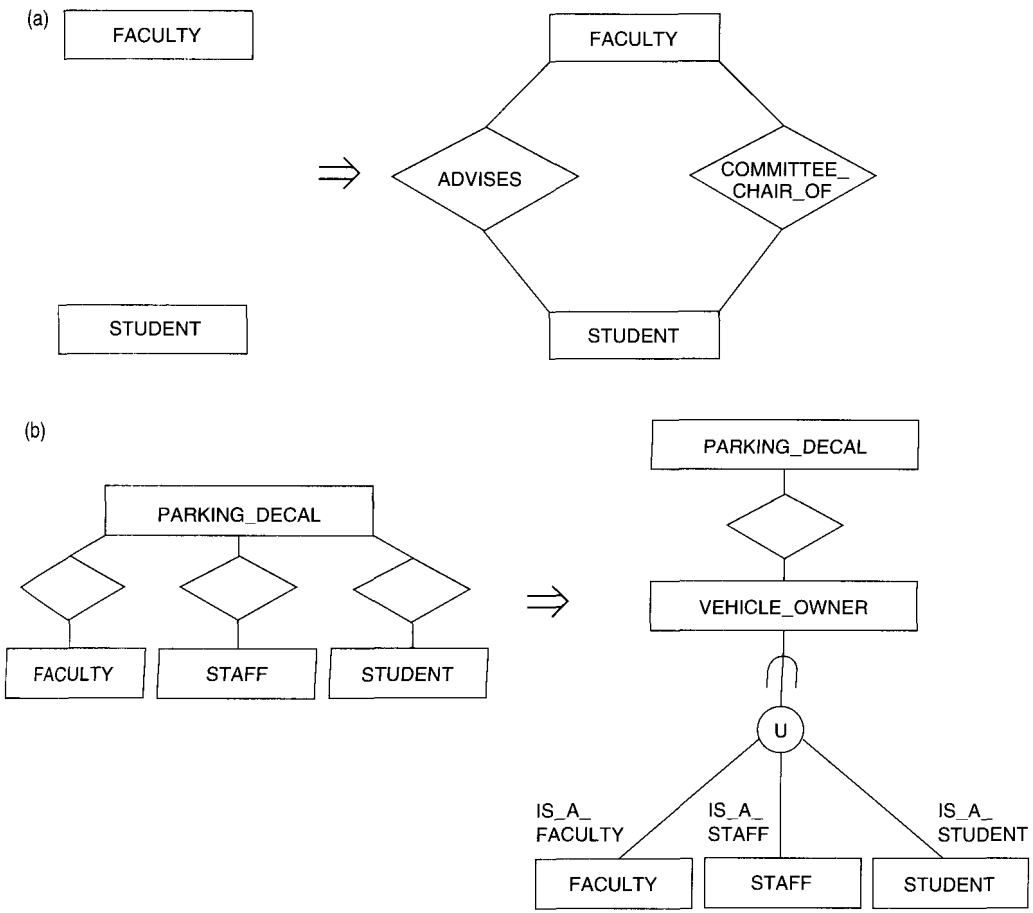


FIGURE 12.3 Examples of bottom-up refinement. (a) Discovering and adding new relationships. (b) Discovering a new category (union type) and relating it.

entity types. The bottom-up refinement using categorization (union type) is illustrated in Figure 12.3(b), *where the new concept of* VEHICLE_OWNER *is discovered from the existing entity types* FACULTY, STAFF, *and* STUDENT; *this process of creating a category and the related diagrammatic notation follows what we introduced in Section 4.4.*

Schema (View) Integration. For large databases with many expected users and applications, the view integration approach of designing individual schemas and then merging them can be used. Because the individual views can be kept relatively small, design of the schemas is simplified. However, a methodology for integrating the views into a global database schema is needed. Schema integration can be divided into the following subtasks:

1. *Identifying correspondences and conflicts among the schemas:* Because the schemas are designed individually, it is necessary to specify constructs in the schemas that represent the same real-world concept. These correspondences must be identified before integration can proceed. During this process, several types of conflicts among the schemas may be discovered:
   a. *Naming conflicts:* These are of two types: synonyms and homonyms. A **synonym** occurs when two schemas use different names to describe the same concept; for example, an entity type CUSTOMER in one schema may describe the same concept as an entity type CLIENT in another schema. A **homonym** occurs when two schemas use the same name to describe different concepts; for example, an entity type PART may represent computer parts in one schema and furniture parts in another schema.
   b. *Type conflicts:* The same concept may be represented in two schemas by different modeling constructs. For example, the concept of a DEPARTMENT may be an entity type in one schema and an attribute in another.
   c. *Domain (value set) conflicts:* An attribute may have different domains in two schemas. For example, SSN may be declared as an integer in one schema and as a character string in the other. A conflict of the unit of measure could occur if one schema represented WEIGHT in pounds and the other used kilograms.
   d. *Conflicts among constraints:* Two schemas may impose different constraints; for example, the key of an entity type may be different in each schema. Another example involves different structural constraints on a relationship such as TEACHES; one schema may represent it as 1:N (a course has one instructor), while the other schema represents it as M:N (a course may have more than one instructor).
2. *Modifying views to conform to one another:* Some schemas are modified so that they conform to other schemas more closely. Some of the conflicts identified in the first subtask are resolved during this step.
3. *Merging of views:* The global schema is created by merging the individual schemas. Corresponding concepts are represented only once in the global schema, and mappings between the views and the global schema are specified. This is the most difficult step to achieve in real-life databases involving hundreds of entities and relationships. It involves a considerable amount of human intervention and negotiation to resolve conflicts and to settle on the most reasonable and acceptable solutions for a global schema.
4. *Restructuring:* As a final optional step, the global schema may be analyzed and restructured to remove any redundancies or unnecessary complexity.

Some of these ideas are illustrated by the rather simple example presented in Figures 12.4 and 12.5. In Figure 12.4, two views are merged to create a bibliographic database. During identification of correspondences between the two views, we discover that RESEARCHER and AUTHOR are synonyms (as far as this database is concerned), as are CONTRIBUTED_BY and WRITTEN_BY. Further, we decide to modify VIEW 1 to include a SUBJECT for ARTICLE, as shown in Figure 12.4, *to conform to* VIEW 2. Figure 12.5 shows the result of merging MODIFIED VIEW 1 with VIEW 2. We generalize the entity types ARTICLE and BOOK into

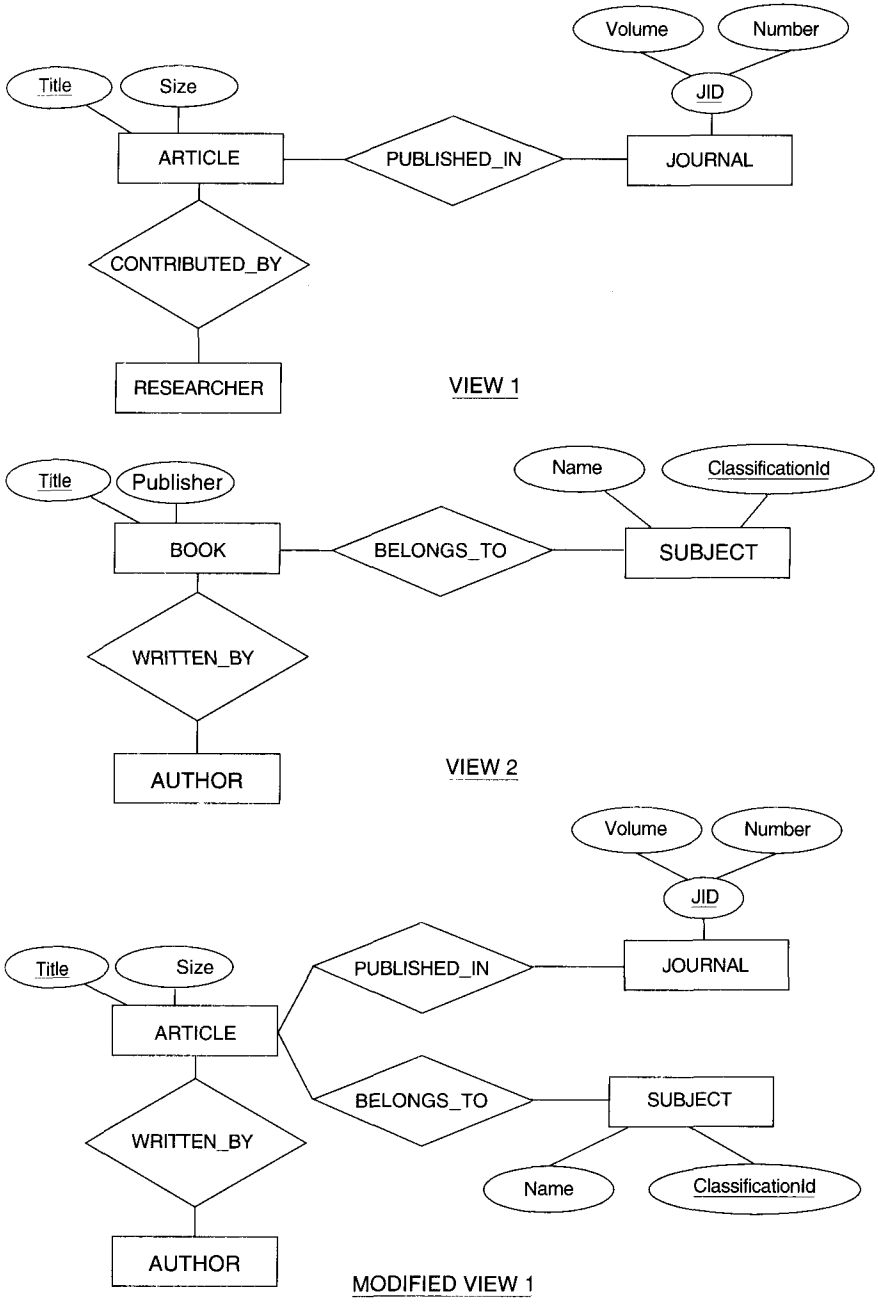


FIGURE 12.4 Modifying views to conform before integration.

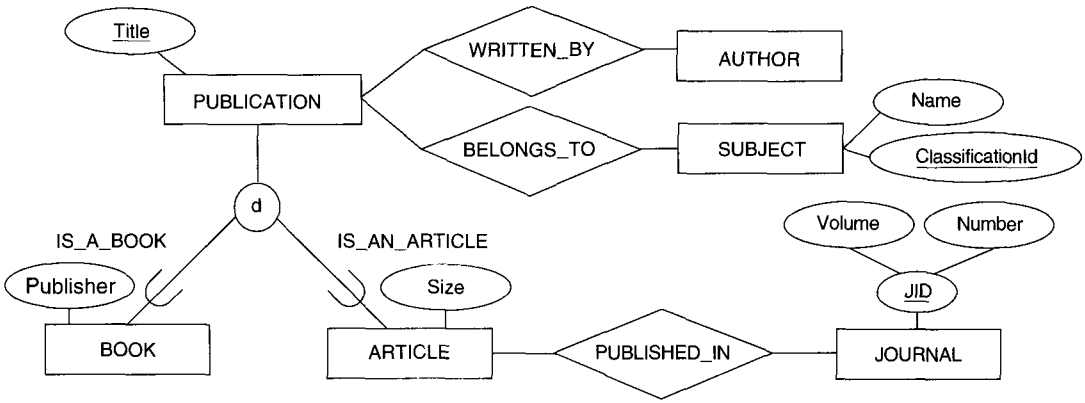


FIGURE 12.5 Integrated schema after merging views 1 and 2.

the entity type PUBLICATION, with their common attribute `Title`. The relationships CONTRIBUTED_BY and WRITTEN_BY are merged, as are the entity types RESEARCHER and AUTHOR. The attribute `Publisher` applies only to the entity type BOOK, whereas the attribute `Size` and the relationship type PUBLISHED_IN apply only to ARTICLE.

The above example illustrates the complexity of the merging process and how the meaning of the various concepts must be accounted for in simplifying the resultant schema design. For real-life designs, the process of schema integration requires a more disciplined and systematic approach. Several strategies have been proposed for the view integration process (Figure 12.6):

1. *Binary ladder integration:* Two schemas that are quite similar are integrated first. The resulting schema is then integrated with another schema, and the process is repeated until all schemas are integrated. The ordering of schemas for integration can be based on some measure of schema similarity. This strategy is suitable for manual integration because of its step-by-step approach.
2. *N-ary integration:* All the views are integrated in one procedure after an analysis and specification of their correspondences. This strategy requires computerized tools for large design problems. Such tools have been built as research prototypes but are not yet commercially available.
3. *Binary balanced strategy:* Pairs of schemas are integrated first; then the resulting schemas are paired for further integration; the procedure is repeated until a final global schema results.
4. *Mixed strategy:* Initially, the schemas are partitioned into groups based on their similarity, and each group is integrated separately. The intermediate schemas are grouped again and integrated, and so on.

Phase 2b: Transaction Design. The purpose of Phase 2b, which proceeds in parallel with Phase 2a, is to design the characteristics of known database transactions (applications) in a DBMS-independent way. When a database system is being designed,

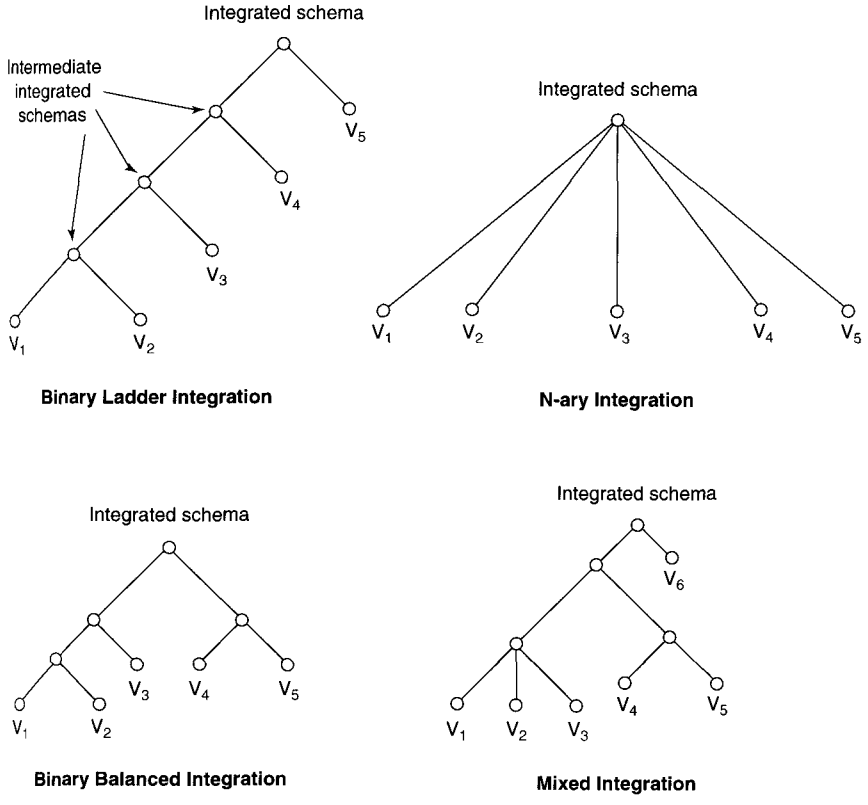


FIGURE **12.6** Different strategies for the view integration process.

the designers are aware of many known applications (or **transactions**) that will run on the database once it is implemented. An important part of database design is to specify the functional characteristics of these transactions early on in the design process. This ensures that the database schema will include all the information required by these transactions. In addition, knowing the relative importance of the various transactions and the expected rates of their invocation plays a crucial part in physical database design (Phase 5). Usually, only some of the database transactions are known at design time; after the database system is implemented, new transactions are continuously identified and implemented. However, the most important transactions are often known in advance of system implementation and should be specified at an early stage. The informal "80–20 rule" typically applies in this context: 80 percent of the workload is represented by 20 percent of the most frequently used transactions, which govern the design. In applications that are of the ad-hoc querying or batch processing variety, queries and applications that process a substantial amount of data must be identified.

A common technique for specifying transactions at a conceptual level is to identify their **input/output** and **functional behavior.** By specifying the input and output

parameters (arguments), and internal functional flow of control, designers can specify a transaction in a conceptual and system-independent way. Transactions usually can be grouped into three categories: (1) **retrieval transactions,** which are used to retrieve data for display on a screen or for production of a report; (2) **update transactions,** which are used to enter new data or to modify existing data in the database; (3) **mixed transactions,** which are used for more complex applications that do some retrieval and some update. For example, consider an airline reservations database. A retrieval transaction could list all morning flights on a given date between two cities. An update transaction could be to book a seat on a particular flight. A mixed transaction may first display some data, such as showing a customer reservation on some flight, and then update the database, such as canceling the reservation by deleting it, or by adding a flight segment to an existing reservation. Transactions (applications) may originate in a front-end tool such as PowerBuilder 9.0 (from Sybase) or Developer 2000 (from Oracle), which collect parameters on-line and then send a transaction to the DBMS as a backend.[3]

Several techniques for requirements specification include notation for specifying **processes,** which in this context are more complex operations that can consist of several transactions. Process modeling tools like BPWin as well as workflow modeling tools are becoming popular to identify information flows in organizations. The UML language, which provides for data modeling via class and object diagrams, has a variety of process modeling diagrams including state transition diagrams, activity diagrams, sequence diagrams, and collaboration diagrams. All of these refer to activities, events, and operations within the information system, the inputs and outputs of the processes, and the sequencing or synchronization requirements, and other conditions. It is possible to refine these specifications and extract individual transactions from them. Other proposals for specifying transactions include TAXIS, GALILEO, and GORDAS (see the selected bibliography at the end of this chapter). Some of these have been implemented into prototype systems and tools. Process modeling still remains an active area of research.

Transaction design is just as important as schema design, but it is often considered to be part of software engineering rather than database design. Many current design methodologies emphasize one over the other. One should go through Phases 2a and 2b in parallel, using feedback loops for refinement, until a stable design of schema and transactions is reached.[4]

12.2.3 Phase 3: Choice of a DBMS

The choice of a DBMS is governed by a number of factors—some technical, others economic, and still others concerned with the politics of the organization. The technical factors are concerned with the suitability of the DBMS for the task at hand. Issues to consider

3. This philosophy has been followed for over 20 years in popular products like CICS, which serves as a tool to generate transactions for legacy DBMSs like IMS.

4. High-level transaction modeling is covered in Batini et al. (1992, chaps. 8, 9, and 11). The joint functional and data analysis philosophy is advocated throughout that book.

here are the type of DBMS (relational, object-relational, object, other), the storage structures and access paths that the DBMS supports, the user and programmer interfaces available, the types of high-level query languages, the availability of development tools, ability to interface with other DBMSs via standard interfaces, architectural options related to client-server operation, and so on. Nontechnical factors include the financial status and the support organization of the vendor. In this section we concentrate on discussing the economic and organizational factors that affect the choice of DBMS. The following costs must be considered:

1. *Software acquisition cost:* This is the "up-front" cost of buying the software, including language options, different interface options such as forms, menu, and Web-based graphic user interface (GUI) tools, recovery/backup options, special access methods, and documentation. The correct DBMS version for a specific operating system must be selected. Typically, the development tools, design tools, and additional language support are not included in basic pricing.
2. *Maintenance cost:* This is the recurring cost of receiving standard maintenance service from the vendor and for keeping the DBMS version up to date.
3. *Hardware acquisition cost:* New hardware may be needed, such as additional memory, terminals, disk drives and controllers, or specialized DBMS storage and archival storage.
4. *Database creation and conversion cost:* This is the cost of either creating the database system from scratch or converting an existing system to the new DBMS software. In the latter case it is customary to operate the existing system in parallel with the new system until all the new applications are fully implemented and tested. This cost is hard to project and is often underestimated.
5. *Personnel cost:* Acquisition of DBMS software for the first time by an organization is often accompanied by a reorganization of the data-processing department. Positions of DBA and staff exist in most companies that have adopted DBMSs.
6. *Training cost:* Because DBMSs are often complex systems, personnel must often be trained to use and program the DBMS. Training is required at all levels, including programming, application development, and database administration.
7. *Operating cost:* The cost of continued operation of the database system is typically not worked into an evaluation of alternatives because it is incurred regardless of the DBMS selected.

The benefits of acquiring a DBMS are not so easy to measure and quantify. A DBMS has several intangible advantages over traditional file systems, such as ease of use, consolidation of company-wide information, wider availability of data, and faster access to information. With Web-based access, certain parts of the data can be made globally accessible to employees as well as external users. More tangible benefits include reduced application development cost, reduced redundancy of data, and better control and security. Although databases have been firmly entrenched in most organizations, the decision of whether to move an application from a file-based to a database-centered approach comes up frequently. This move is generally driven by the following factors:

1. *Data complexity:* As data relationships become more complex, the need for a DBMS is felt more strongly.
2. *Sharing among applications:* The greater the sharing among applications, the more the redundancy among files, and hence the greater the need for a DBMS.
3. *Dynamically evolving or growing data:* If the data changes constantly, it is easier to cope with these changes using a DBMS than using a file system.
4. *Frequency of ad hoc requests for data:* File systems are not at all suitable for ad hoc retrieval of data.
5. *Data volume and need for control:* The sheer volume of data and the need to control it sometimes demands a DBMS.

It is difficult to develop a generic set of guidelines for adopting a single approach to data management within an organization—whether relational, object-oriented, or object-relational. If the data to be stored in the database has a high level of complexity and deals with multiple data types, the typical approach may be to consider an object or object-relational DBMS.[5] Also, the benefits of inheritance among classes and the corresponding advantage of reuse favor these approaches. Finally, several economic and organizational factors affect the choice of one DBMS over another:

1. *Organization-wide adoption of a certain philosophy:* This is often a dominant factor affecting the acceptability of a certain data model (for example, relational versus object), a certain vendor, or a certain development methodology and tools (for example, use of an object-oriented analysis and design tool and methodology may be required of all new applications).
2. *Familiarity of personnel with the system:* If the programming staff within the organization is familiar with a particular DBMS, it may be favored to reduce training cost and learning time.
3. *Availability of vendor services:* The availability of vendor assistance in solving problems with the system is important, since moving from a non-DBMS to a DBMS environment is generally a major undertaking and requires much vendor assistance at the start.

Another factor to consider is the DBMS portability among different types of hardware. Many commercial DBMSs now have versions that run on many hardware/software configurations (or **platforms**). The need of applications for backup, recovery, performance, integrity, and security must also be considered. Many DBMSs are currently being designed as *total solutions* to the information-processing and information resource management needs within organizations. Most DBMS vendors are combining their products with the following options or built-in features:

- Text editors and browsers.
- Report generators and listing utilities.
- Communication software (often called teleprocessing monitors).

5. See the discussion in Chapter 22 concerning this issue.

- Data entry and display features such as forms, screens, and menus with automatic editing features.
- Inquiry and access tools that can be used on the World Wide Web (Web enabling tools).
- Graphical database design tools.

A large amount of "third-party" software is available that provides added functionality to a DBMS in each of the above areas. In rare cases it may be preferable to develop in-house software rather than use a DBMS— for example, if the applications are very well defined and are *all* known beforehand. Under such circumstances, an in-house custom-designed system may be appropriate to implement the known applications in the most efficient way. In most cases, however, new applications that were not foreseen at design time come up *after* system implementation. This is precisely why DBMSs have become very popular: They facilitate the incorporation of new applications with only incremental modifications to the existing design of a database. Such design evolution—or **schema evolution**—is a feature present to various degrees in commercial DBMSs.

12.2.4 Phase 4: Data Model Mapping (Logical Database Design)

The next phase of database design is to create a conceptual schema and external schemas in the data model of the selected DBMS by mapping those schemas produced in Phase 2a. The mapping can proceed in two stages:

1. *System-independent mapping:* In this stage, the mapping does not consider any specific characteristics or special cases that apply to the DBMS implementation of the data model. We already discussed DBMS-independent mapping of an ER schema to a relational schema in Section 7.1 and of EER schemas to relational schemas in Section 7.2.
2. *Tailoring the schemas to a specific* DBMS: Different DBMSs implement a data model by using specific modeling features and constraints. We may have to adjust the schemas obtained in Step 1 to conform to the specific implementation features of a data model as used in the selected DBMS.

The result of this phase should be DDL statements in the language of the chosen DBMS that specify the conceptual and external level schemas of the database system. But if the DDL statements include some physical design parameters, a complete DDL specification must wait until after the physical database design phase is completed. Many automated CASE (computer-assisted software engineering) design tools (see Section 12.5) can generate DDL for commercial systems from a conceptual schema design.

12.2.5 Phase 5: Physical Database Design

Physical database design is the process of choosing specific storage structures and access paths for the database files to achieve good performance for the various database

applications. Each DBMS offers a variety of options for file organization and access paths. These usually include various types of indexing, clustering of related records on disk blocks, linking related records via pointers, and various types of hashing. Once a specific DBMS is chosen, the physical database design process is restricted to choosing the most appropriate structures for the database files from among the options offered by that DBMS. In this section we give generic guidelines for physical design decisions; they hold for any type of DBMS. The following criteria are often used to guide the choice of physical database design options:

1. *Response time*: This is the elapsed time between submitting a database transaction for execution and receiving a response. A major influence on response time that is under the control of the DBMS is the database access time for data items referenced by the transaction. Response time is also influenced by factors not under DBMS control, such as system load, operating system scheduling, or communication delays.
2. *Space utilization*: This is the amount of storage space used by the database files and their access path structures on disk, including indexes and other access paths.
3. *Transaction throughput*: This is the average number of transactions that can be processed per minute; it is a critical parameter of transaction systems such as those used for airline reservations or banking. Transaction throughput must be measured under peak conditions on the system.

Typically, average and worst-case limits on the preceding parameters are specified as part of the system performance requirements. Analytical or experimental techniques, which can include prototyping and simulation, are used to estimate the average and worst-case values under different physical design decisions, to determine whether they meet the specified performance requirements.

Performance depends on record size and number of records in the file. Hence, we must estimate these parameters for each file. In addition, we should estimate the update and retrieval patterns for the file cumulatively from all the transactions. Attributes used for selecting records should have primary access paths and secondary indexes constructed for them. Estimates of file growth, either in the record size because of new attributes or in the number of records, should also be taken into account during physical database design.

The result of the physical database design phase is an *initial* determination of storage structures and access paths for the database files. It is almost always necessary to modify the design on the basis of its observed performance after the database system is implemented. We include this activity of **database tuning** in the next phase and cover it in the context of query optimization in Chapter 16.

12.2.6 Phase 6: Database System Implementation and Tuning

After the logical and physical designs are completed, we can implement the database system. This is typically the responsibility of the DBA and is carried out in conjunction with

the database designers. Language statements in the DDL (data definition language) including the SDL (storage definition language) of the selected DBMS are compiled and used to create the database schemas and (empty) database files. The database can then be **loaded** (populated) with the data. If data is to be converted from an earlier computerized system, **conversion routines** may be needed to reformat the data for loading into the new database.

Database transactions must be implemented by the application programmers by referring to the conceptual specifications of transactions, and then writing and testing program code with embedded DML commands. Once the transactions are ready and the data is loaded into the database, the design and implementation phase is over and the operational phase of the database system begins.

Most systems include a monitoring utility to collect performance statistics, which are kept in the system catalog or data dictionary for later analysis. These include statistics on the number of invocations of predefined transactions or queries, input/output activity against files, counts of file pages or index records, and frequency of index usage. As the database system requirements change, it often becomes necessary to add or remove existing tables and to reorganize some files by changing primary access methods or by dropping old indexes and constructing new ones. Some queries or transactions may be rewritten for better performance. Database tuning continues as long as the database is in existence, as long as performance problems are discovered, and while the requirements keep changing.

12.3 Use of UML Diagrams as an Aid to Database Design Specification[6]

12.3.1 UML As a Design Specification Standard

In the first section of this chapter, we discussed in detail how organizations work with information systems and elaborated the various activities in the information system life cycle. Databases are an integral part of information systems in most organizations. The phases of database design starting with requirements analysis up to system implementation and tuning were introduced at the end of Section 12.1 and discussed in detail in Section 12.2. Industry is always in the need of some standard approaches to cover this entire spectrum of requirements analysis, modeling, design, implementation and deployment. The approach that is receiving a wide attention and acceptability and that is also proposed as a standard by the OMG (Object Management Group) is the **Unified Modeling Language** (UML) approach. It provides a mechanism in the form of diagrammatic notation and associated language synatx to cover the entire lifecycle. Presently UML is used by software developers, data modelers, data designers, database architects, etc. to define the detailed specification of an application. They also use it to specify the environment consisting of software, communications and hardware to implement and deploy the application.

6. The contribution of Abrar Ul-Haque to the UML and Rational Rose sections is much appreciated.

UML combines commonly accepted concepts from many OO methods and methodologies (see bibliographic notes for the contributing methodologies that led to UML). It is applicable to any domain, and is language- and platform-independent; so software architects can model any type of application, running on any operating system, programming language or network in UML. That has made the approach very widely applicable. Tools like Rational Rose are currently popular for drawing UML diagrams – they enable software developers to develop clear and easy-to-understand models for specifying, visualizing, constructing and documenting components of software systems. Since the scope of UML extends to software and application development at large, we will not cover all aspects of UML here. Our goal is to show some relevant UML notations that are commonly used in the requirements collection and analysis as well as the conceptual design phases (phases 1 and 2 in Figure 12.1). A detailed application development methodology using UML is outside the scope of this book and may be found in various textbooks devoted to object-oriented design, software engineering, and UML (see bibliographic notes).

Class diagrams, which are the end result of conceptual database design have already been discussed in Sections 3.8 and 4.6. To arrive at the class diagrams, the information may be gathered and specified using use case diagrams, sequence diagrams and state chart diagrams. In the rest of this section we will first introduce the different types of UML diagrams briefly to give the reader an idea of the scope of UML. Then we will present a small sample application to illustrate the use of use case, sequence and statechart diagrams and show how they lead to the eventual class diagram as the final conceptual design. The diagrams presented in this section pertain to the standard UML notation and have been drawn using the tool Rational Rose. Section 12.4 will be devoted to a general discussion of the use of Rational Rose in database application design.

12.3.2 UML for Database Application Design

The database community has started embracing UML, and now many database designers and developers are using UML for data modeling as well as for subsequent phases of database design. The advantage of UML is that even though its concepts are based on object-oriented techniques, the resulting models of structure and behavior can be used to design both relational, object-oriented and object-relational databases (see Chapters 20 to 22 for definition of object databases and object-relational databases). We already introduced UML **Class Diagrams,** which are similar to the ER and EER diagrams in Sections 3.8 and 4.6, respectively. They give a structural specification of the database schemas in an object-oriented sense by showing the name, attributes and operations of each class. Their normal use is to describe the collections of data objects and their inter-relationships which is consistent with the goal of conceptual database design.

One of the major contributions of the UML approach has been to bring the traditional database modelers, analysts and designers together with the software application developers. In Figure 12.1 we showed the phases of database design and implementation and how they apply to these two groups. UML has been able to propose a common notation or a meta model that can be adopted by both of these communities and

tailored to their needs. Whereas we dwelt solely on the structural aspect of modeling in Chapters 3 and 4, UML also allows us to do behavioral or/and dynamic modeling by introducing various types of diagrams. This results in a more complete specification/description of the overall database application. In the next sections we will first summarize the different UML diagrams and then give an example of the use case, sequence and statechart diagrams in a sample application. A complete case study of a database application development is presented in Appendix B.

12.3.3 Different Diagrams in UML

UML defines nine types of diagrams divided into two categories.

Structural Diagrams. These describe the structural or static relationships among components. They include Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram.

Behavioral Diagrams. Their purpose is to describe the behavioral or dynamic relationships among components. They include Use Case Diagram, Sequence Diagram, Collaboration Diagram, Statechart Diagram, and Activity Diagram.

We introduce the nine types briefly below. The structural diagrams include:

A. Class Diagrams

Class diagrams capture the static structure of the system and act as foundation for other models. They show Classes, Interfaces, Collaborations, Dependencies, Generalizations, Association and other relationships. Class diagrams are a very useful way to model the conceptual database schema. We showed examples of class diagrams for the company database schema in Figure 3.16 and for a generalization hierarchy in Figure 4.10.

Package Diagrams. Package diagrams are a subset of class diagrams. They organize elements of the system into related groups called packages. A package may be a collection of related classes and the relationships between them. Package diagrams help minimize dependencies in a system.

B. Object Diagrams

Object diagrams show a set of objects and their relationships. They correspond to what we called instance diagrams in chapters 3 and 4. They give a static view of a system at a particular time and are normally used to test class diagrams for accuracy.

C. Component Diagrams

Component diagrams illustrate the organizations and dependencies among software components. A component diagram typically consists of components, interfaces and dependency relationships. A component may be a source code component, a run-time component or an executable component. It is a physical building block in the system and is

represented as a rectangle with two small rectangles or tabs overlaid on its left side. An interface is a group of operations used or created by a component and is usually represented by a small circle. Dependency relationship is used to model the relationship between two components is represented by a dotted arrow pointing from a component to the component it depends on. For databases, component diagrams stand for stored data such as tablespaces or partitions. Interfaces refer to applications that use the stored data.

D. Deployment Diagrams

Deployment diagrams represent the distribution of components (executables, libraries, tables, files) across the hardware topology. They depict the physical resources in a system, including nodes, components and connections, and are basically used to show the configuration of run-time processing elements (the nodes) and the software processes that reside on them (the threads).

Now we will describe the behavioral diagrams and expand on those that are of particular interest.

E. Use Case Diagrams

Use case diagrams are used to model the functional interactions between users and the system. A **scenario** is a sequence of steps describing an interaction between a user and a system. A **use case** is a set of scenarios that have a common goal. The use case diagram was introduced by Jacobson[7] to visualize use cases. The **use case diagram** shows actors interacting with use cases and can be understood easily without the knowledge of any notation. An individual use case is shown as an oval and stands for a specific task performed by the system. An **actor**, shown with a stick person symbol, represents an external user, which may be a human user, a representative group of users, a certain role of a person in the organization, or anything external to the system. The use case diagram shows possible interactions of the system (in our case, a database system) and describes as use cases the specific tasks the system performs. Since they do not specify any implementation detail and are very easy to understand, they are a good vehicle for communicating between the end users and developers and help in easier user validation at an early stage. Test plans can also be easily generated using use cases diagrams. Figure 12.7 shows the use case diagram notation. The **include** relationship is used to factor out some common behavior from two or more of the original use cases – it is a form of reuse. For example, in a university environment shown in Figure 12.8, the use cases "register for courses" and "enter grades" in which actors student and professor are involved, include a common use case called "validate user." If a use case incorporates two or more significantly different scenarios, based on circumstances or varying conditions, the **extend** relationship is used to show the subcases attached to the base case (see Figure 12.7)

Interaction diagrams. Interaction diagrams are used to model the dynamic aspects of a system. They basically consist of a set of messages exchanged between a set of Objects. There are two types of interaction diagrams, Sequence and Collaboration.

7. See Jacobson et al. (1992)

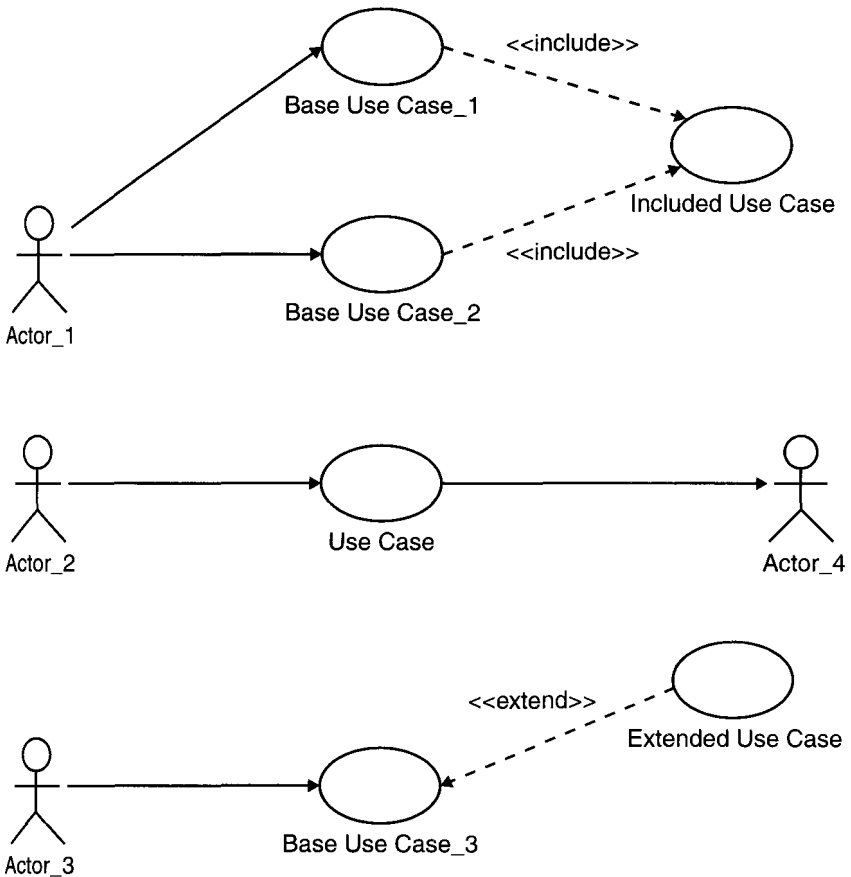


FIGURE **12.7** The use-case diagram notation.

F. Sequence Diagrams

Sequence diagrams describe the interactions between various objects over time. They basically give a dynamic view of the system by showing the flow of messages between objects. Within the sequence diagram, an object or an actor is shown as a box at the top of a dashed vertical line, which is called the **object's lifeline**. For a database, this object is typically something physical (like a book in the warehouse) that would be contained in the database, an external document or form such as an order form, or an external visual screen which may be part of a user interface. The lifeline represents the existence of object over time. **Activation,** which indicates when an object is performing an action, is represented as a rectangular box on a lifeline. Each message is represented as an arrow between the lifelines of two objects. A message bears a name and may have arguments and control information to explain the nature of the interaction. The order of messages is read from top to bottom. A sequence diagram also gives the option of self-call, which is

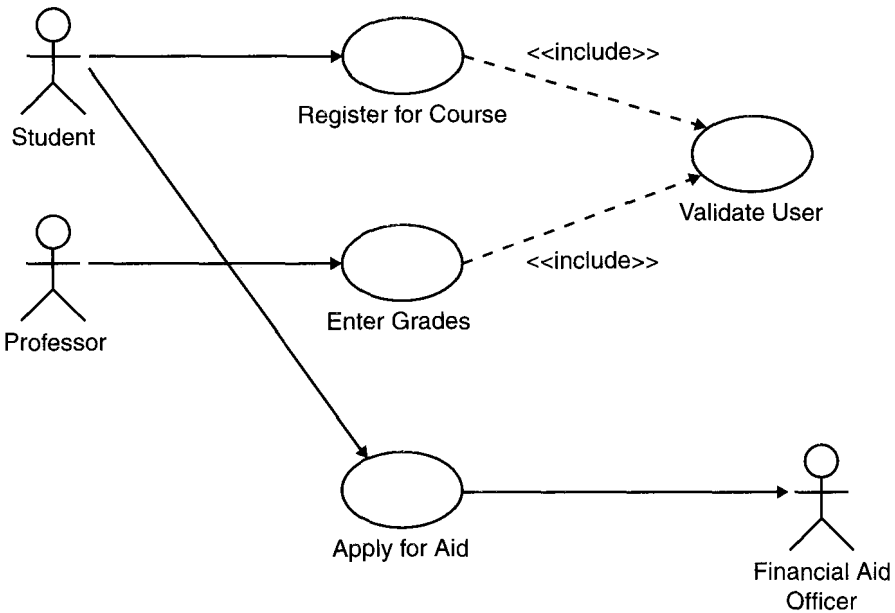


FIGURE 12.8 An example use case diagram for a University Database.

basically just a message from an object to itself. **Condition** and **Iteration markers** can also be shown in sequence diagrams to specify when the message should be sent and to specify the condition to send multiple markers. A return dashed line shows a return from the message and is optional unless it carries a special meaning. Object deletion is shown with a large X. Figure 12.9 explains the notation of the sequence diagram.

G. Collaboration Diagrams

Collaboration diagrams represent interactions between objects as a series of sequenced messages. In Collaboration Diagrams the emphasis is on the structural organization of the objects that send and receive messages whereas in Sequence Diagrams the emphasis is on the time-ordering of the messages. Collaboration diagrams show objects as icons and number the messages; numbered messages represent an ordering. The spatial layout of collaboration diagrams allows linkages among objects that show their structural relationships. Use of collaboration and sequence diagrams to represent interactions is a matter of choice; we will hereafter use only sequence diagrams.

H. Statechart Diagram

Statechart diagrams describe how an object's state changes in response to external events.

To describe the behavior of an object, it is common in most object-oriented techniques to draw a state diagram to show all the possible states an object can get into in

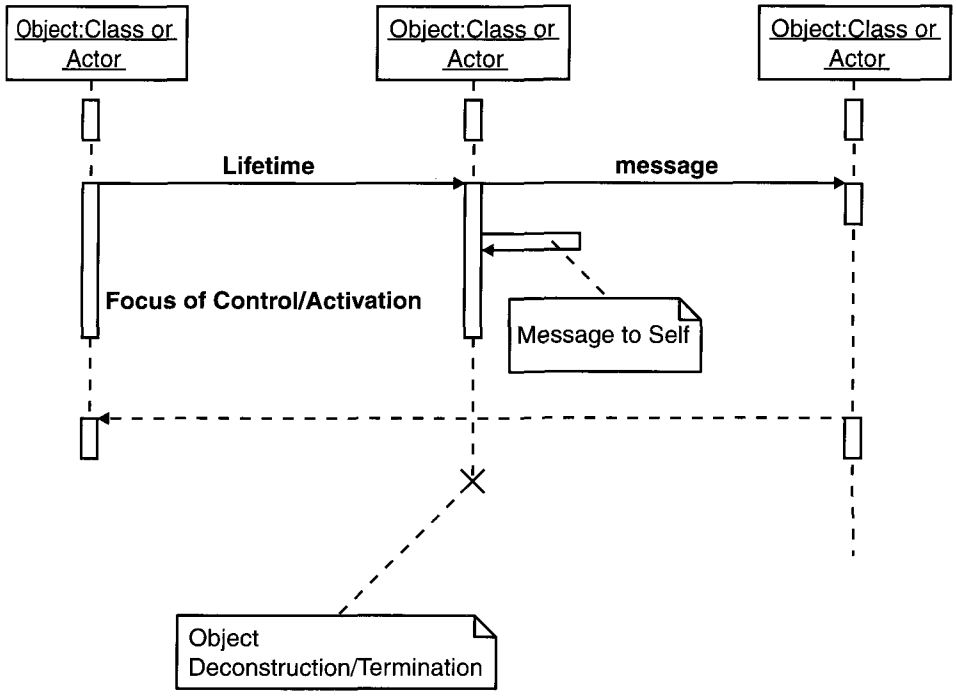


FIGURE 12.9 The sequence diagram notation.

its lifetime. The UML statecharts are based on David Harel's[8] statecharts. They basically show a state machine consisting of states, transitions, events and actions and are very useful in the conceptual design of the application that works against the database of stored objects.

The important elements of a statechart diagram shown in Figure 12.10 are as follows.

- States: shown as boxes with rounded corners, represent situations in the lifetime of an object.
- Transitions: shown as solid arrows between the states, they represent the paths between different states of an object. They are labeled by the eventname [guard] /action; the event triggers the transition and the action results from it. The guard is an additional and optional condition that specifies a condition under which the change of state may not occur.
- Start/Initial State: shown by a solid circle with an outgoing arrow to a state.
- Stop/Final State: shown as a double-lined filled circle with an arrow pointing into it from a state.

8. See Harel (1987).

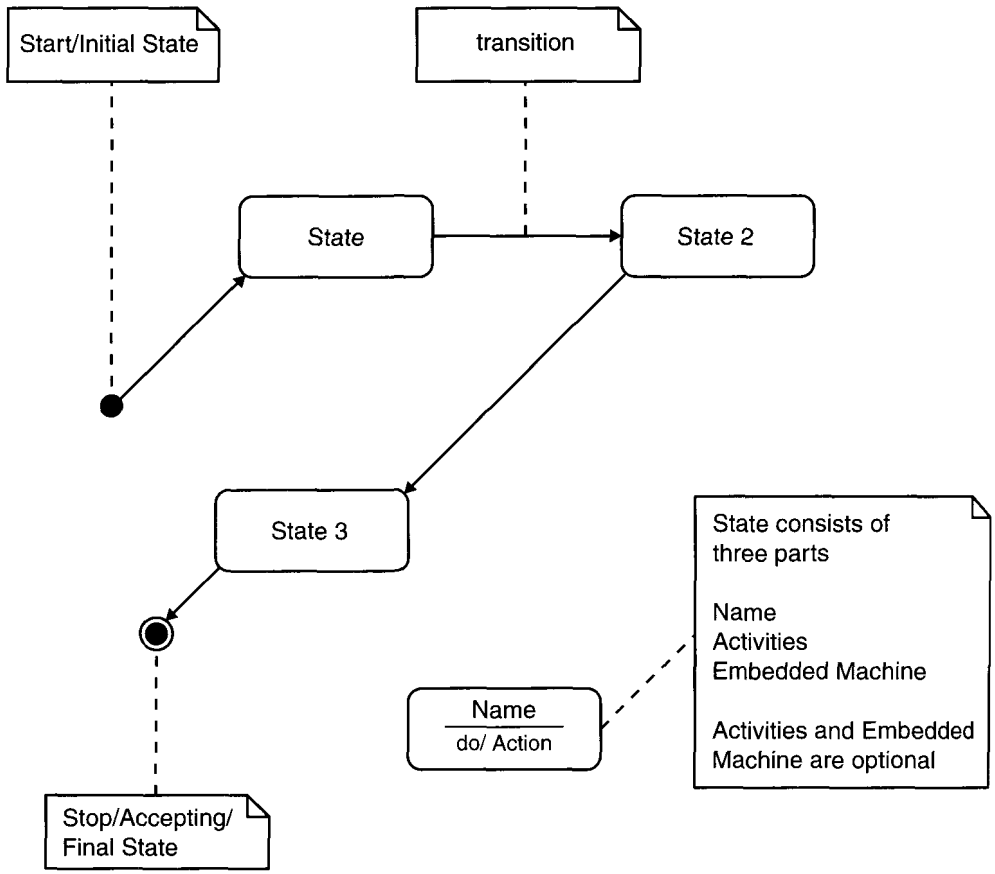


FIGURE 12.10 The statechart diagram notation.

Statechart diagrams are useful in specifying how an object's reaction to a message depends on its state. An event is something done to an object such as being sent a message; an action is something that an object does such as sending a message.

I. Activity Diagrams

Activity diagrams present a dynamic view of the system by modeling the flow of control from activity to activity. They can be considered as flowcharts with states. An activity is a state of doing something, which could be a real-world process or an operation on some class in the database. Typically, activity diagrams are used to model workflow and internal business operations for an application.

12.3.4 A Modeling and Design Example: University Database

In this section we will briefly illustrate the use of the UML diagrams we presented above to design a sample relational database in a university setting. A large number of details are left out to conserve space; only a stepwise use of these diagrams that leads towards a conceptual design and the design of program components is illustrated. As we indicated before, the eventual DBMS on which this database gets implemented may be relational, object-oriented or object-relational. That will not change the stepwise analysis and modeling of the application using the UML diagrams.

Imagine a scenario with students enrolling in courses which are offered by professors. The registrar's office is in charge of maintaining a schedule of courses in a course catalog. They have the authority to add and delete courses and to do schedule changes. They also set enrollment limits on courses. The financial aid office is in charge of processing student's aid applications for which the students have to apply. Assume that we have to design a database that maintains the data about students, professors, courses, aid, etc. We also want to design the application that enables us to do the course registration, financial-aid application processing, and maintaining of the university-wide course catalog by the registrar's office. The above requirements may be depicted by a series of UML diagrams as shown below.

As mentioned previously one of the first steps involved in designing a database is to gather customer requirements and the best way to do this is by using use case diagrams. Suppose one of the requirements in the University Database is to allow the professors to enter grades for the courses they are teaching and for the students to be able to register for courses and apply for financial aid. The use case diagram corresponding to these use cases can be drawn as shown in Figure 12.8.

Another helpful thing while designing a system is to graphically represent some of the states the system can be in. This helps in visualizing the various states the system can be in during the course of the application. For example, in our university database the various states which the system goes through when the registration for a course with 50 seats is opened can be represented by the statechart diagram in Figure 12.11. Note that it shows the states of a course while enrollment is in process. During the enrolling state, the "Enroll Student" transition continues as long as the count of enrolled students is less than 50.

Now having made the use case and state chart diagram we can make a sequence diagram to visualize the execution of the use cases. For the university database, the sequence diagram corresponding to the use case: student requests to register and selects a particular course to register is shown in Figure 12.12. The prerequisites and course capacity are then checked and the course is then added to the student's schedule if the prerequisites are met and there is space in the course.

The above UML diagrams are *not* the complete specification of the University database. There will be other use cases with the Registrar as the actor or the student

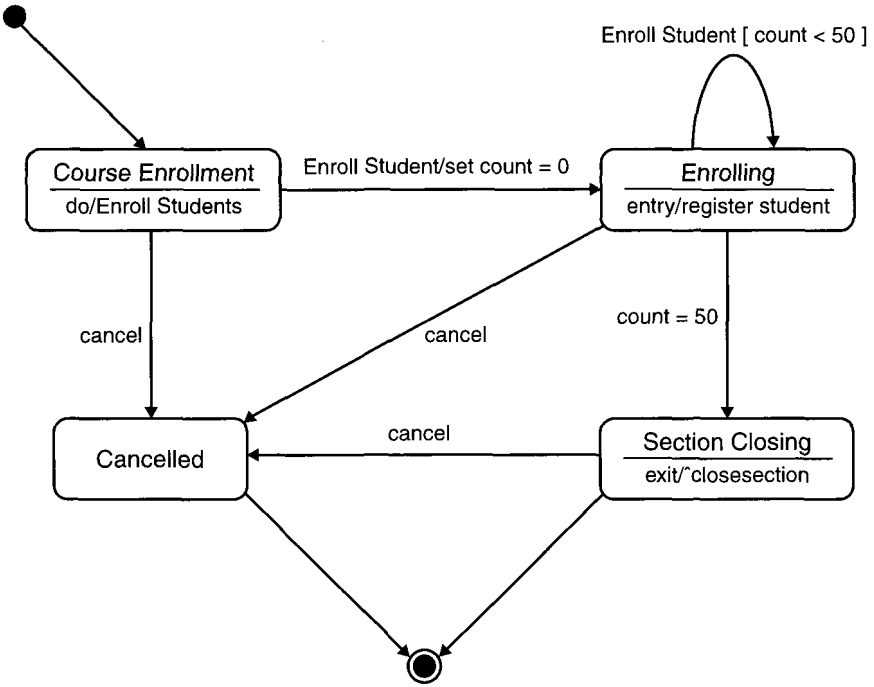


FIGURE 12.11 An example statechart diagram for the University Database.

appearing for a test for a course and receiving a grade in the course, etc. A complete methodology for how to arrive at the class diagrams from the various diagrams we illustrated above is outside our scope here. It is explained further in the case study (Appendix B). Design methodologies remain a matter of judgement, personal preferences, etc. However, we can make sure that the class diagram will account for all the specifications that have been given in the form of the use cases, statechart and sequence diagrams. The class diagram in Figure 12.13 shows the classes with the structural relationships and the operations within the classes that are derived from these diagrams. These classes will need to be implemented to develop the Universiy Database and together with the operations, it will implement the complete class schedule/enrollment/aid application. For clear understanding only some of the important attributes are shown in classes with certain methods that originate from the shown diagrams. It is conceivable that these class diagrams can be constantly upgraded as more details get specified and more functions evolve in the University Application.

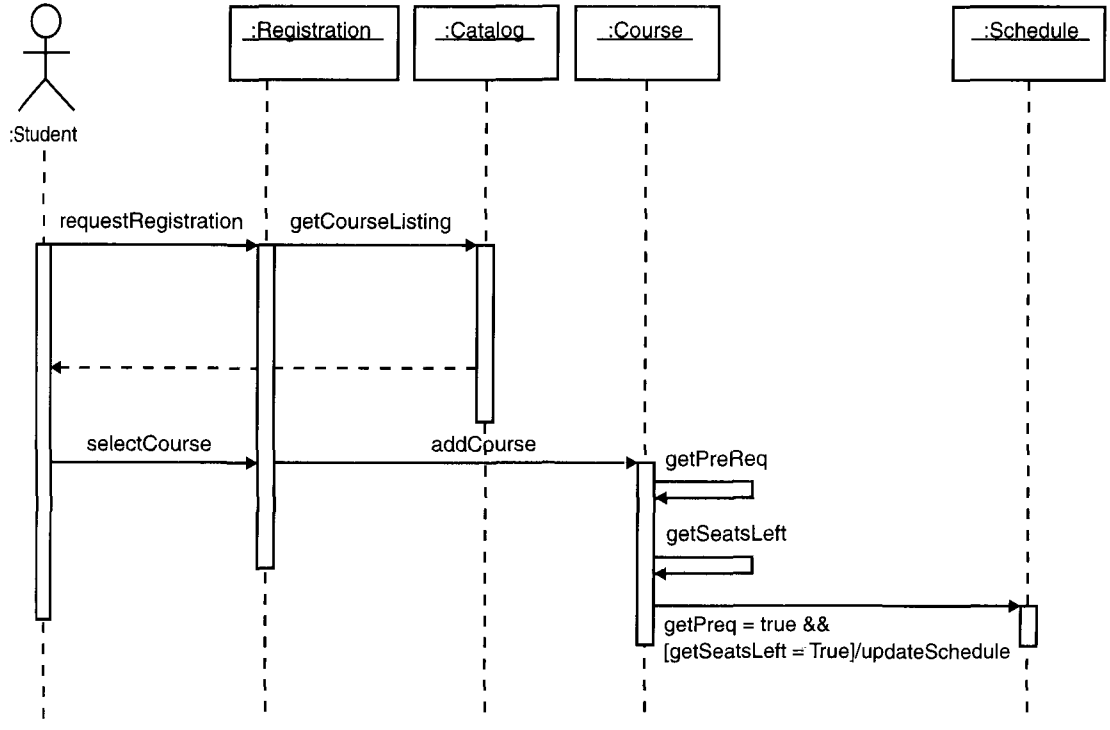


FIGURE 12.12 A sequence diagram for the University Database.

12.4 Rational Rose, A UML Based Design Tool

12.4.1 Rational Rose for Database Design

Rational Rose is one of the most important modeling tools used in the industry to develop information systems. As we pointed out in the first two sections of this chapter, database is a central component of most information systems, and hence, Rational Rose provides the initial specification in UML that eventually leads to the database development. Many extensions have been made in the latest versions of Rose for data modeling and now Rational Rose provides support for conceptual, logical and physical database modeling and design.

12.4.2 Rational Rose Data Modeler

Rational Rose Data Modeler is a visual modeling tool for designing databases. One of the reasons for its popularity is that unlike other data modeling tools it is UML based; it

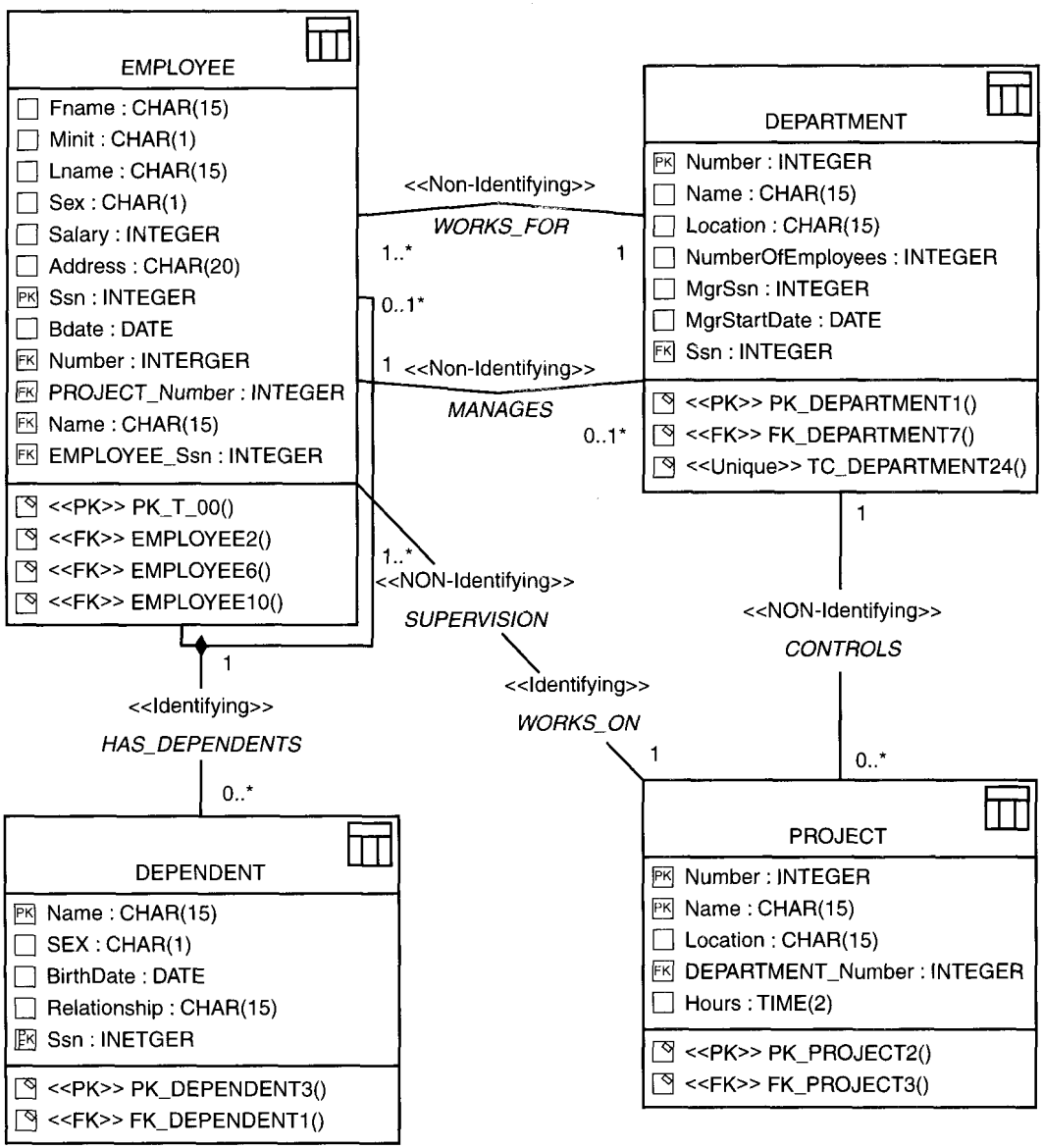


FIGURE 12.13 A graphical data model diagram in Rational Rose.

provides a common tool and language to bridge the communication gap between database designers and application developers. It makes it possible for database designers, developers and analysts to work together, capture and share business requirements and track them as they change throughout the process. Also, by allowing the designers to

model and design all specifications on the same platform using the same notation it improves the design process and reduces the risk of errors.

Another major advantage of Rose is its process modeling capabilities that allow the modeling of the behavior of database as we saw in the short example above in the form of use cases, sequence diagrams, and statechart diagrams. There is the additional machinery of collaboration diagrams to show interactions between objects and activity diagrams to model the flow of control which we did not elaborate upon. The eventual goal is to generate the database specification and application code as much as possible. With the Rose Data Modeler we can capture triggers, stored procedures etc. (see Chapter 24 where active databases contain these features) explicitly on the diagram rather than representing them with hidden tagged values behind the scenes. The Data Modeler also provides the capability to forward engineer a database in terms of constantly changing requirements and reverse engineer an existing implemented database into its conceptual design.

12.4.3 Data Modeling Using Rational Rose Data Modeler

There are many tools and options available in Rose Data Modeler for data modeling. Rational Rose Data Modeler allows creating a data model based on the database structure or creating a database based on the data model.

Reverse Engineering. Reverse Engineering of the database allows the user to create a data model based on the database structure. If we have an existing DBMS database or DDL file we can use the reverse engineering wizard in Rational Rose Data Modeler to generate a conceptual data model. The reverse engineering wizard basically reads the schema in the database or DDL file, and recreates it in a data model. While doing so, it also includes the names of all quoted identifier entities.

Forward Engineering and DDL Generation. We can also create a data model[9] directly from scratch in Rational Rose. Having created the data model we can also use it to generate the DDL in a specific DBMS from the data model. There is a Forward Engineering Wizard in Modeler, which reads the schema in the data model or reads both the schema in the data model and the tablespaces in the data storage model and generates the appropriate DDL code in a DDL file. The wizard also provides the option of generating a database by executing the generated DDL file.

Conceptual Design in UML Notation. As mentioned earlier, one of the major advantages of Rose is that it allows modeling of databases using UML notation. ER

9. The term data model used by Rational Rose Modelre corresponds to our notion of an application model.

diagrams most often used in the conceptual design of databases can be easily built using the UML notation as class diagrams in Rational Rose, e.g. the ER schema of our company example in Chapter 3 can be redrawn in Rational Rose using UML notation as follows.

This can then be converted into a graphical form by using the data model diagram option in Rose.

The above diagrams correspond partly to a relational (logical) schema although they are at a conceptual level. They show the relationships among tables via the primary key (PK)–foreign key (FK) relationships. **Identifying relationships** specify that a child table cannot exist without the parent table (Dependent tables), whereas **non-identifying relationships** specify a regular association between two independent tables. For better and clear understanding, foreign keys automatically appear as one of the attributes in the child entities. It is possible to update the schemas directly in their text or graphical form. For example, the relationship between the EMPLOYEE and PROJECT called WORKS-ON may be deleted and Rose automatically takes care of all the foreign keys, etc. in the table.

Supported Databases. Some of the DBMSs that are currently supported by Rational Rose include the following:

- IBM DB2 versions MVS and UDB 5.x, 6.x, and 7.0.
- Oracle DBMS versions 7.x and 8.x.
- SQL Server QL Server DBMS versions 6.5, 7.0 & 2000.
- Sybase Adaptive Server version 12.x.

The SQL 92 Data Modeler does not reverse engineer ANSI SQL 92 DDLs, however it can forward engineer SQL 92 data models to DDLs.

Converting Logical Data Model to Object Model and Vice Versa. Rational Rose Data Modeler also provides the option of converting a logical database design to an object model design and vice versa. For example the logical data model shown in Figure 12.14 can be converted to an object model. This sort of mapping allows a deep understanding of the relationships between the logical model and database and helps in keeping them both up to date with changes made during the development process. Figure 12.16 shows the Employee table after converting it to a class in an object model. The various tabs in the window can then be used to enter/display different types of information. They include operations, attributes and relationships for that class.

Synchronization Between the Conceptual Design and the Actual Database. Rose Data Modeler allows keeping the data model and database synchronized. It allows visualizing both the data model and the database and then, based on the differences, it gives the option to update the model or change the database.

Extensive Domain Support. The Data Modeler allows database designers to create a standard set of user-defined data types and assign them to any column in the data

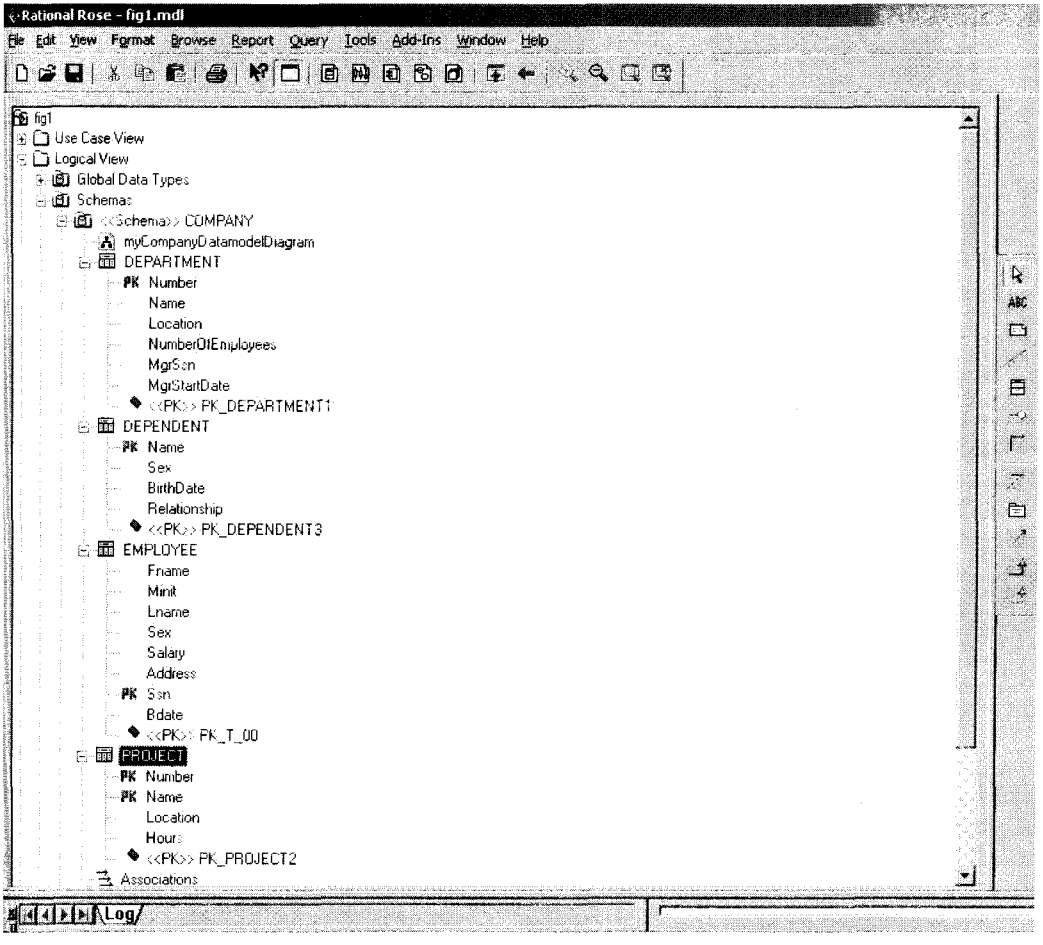


FIGURE 12.14 A logical data model diagram definition in Rational Rose.

model. Properties of the domain are then cascaded to assigned columns. These domains can then be maintained by a standard group and deployed to all modelers when they begin creating new models by using the Rational Rose Framework.

Easy Communication Among Design Teams. As mentioned earlier, using a common tool allows easy communication between teams. In Data Modeler an application developer can access both the object and data models and see how they are related and thus make informed and better choices about how to build data access methods. There is also the option of using ***Rational Rose Web Publisher*** to allow the models and the metadata beneath these models to be available to everyone on the team.

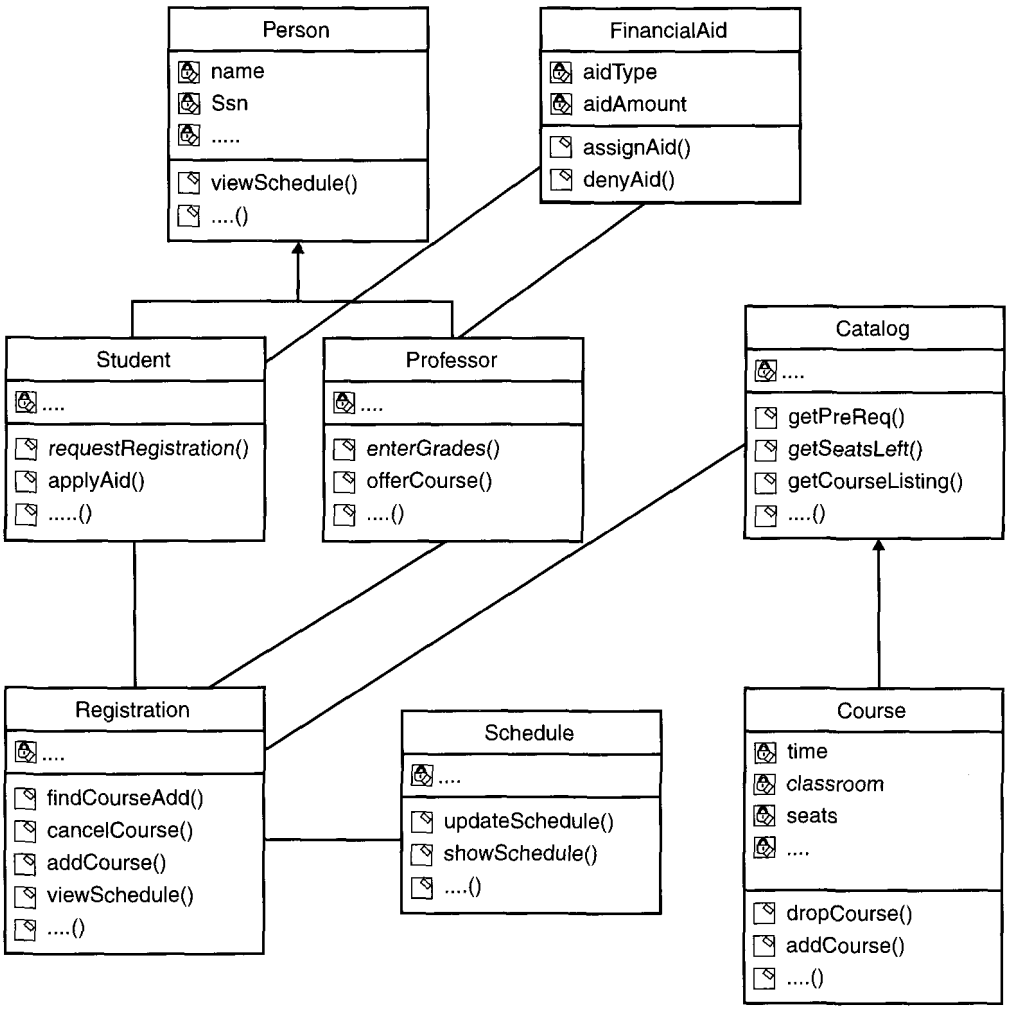


FIGURE 12.15 The design of the university database as a class diagram.

What we have described above is a partial description of the capabilities of the tool as it related to the conceptual and logical design phases in Figure 12.1. The entire range of UML diagrams we described in Section 12.3 can be developed and maintained in Rose. For further details the reader is referred to the product literature. Appendix B develops a full case study with the help of UML diagrams and shows the progression of design through different phases. Figure 12.17 gives a version of the class diagram in Figure 3.16 drawn using Rational Rose.

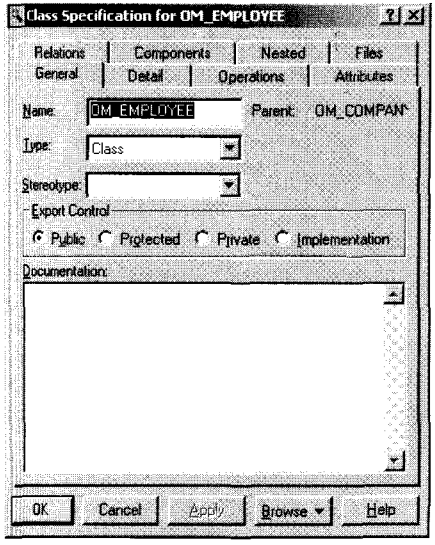


FIGURE 12.16 The class OM_EMPLOYEE corresponding to the table Employee in Figure 12.14.

12.5 AUTOMATED DATABASE DESIGN TOOLS

The database design activity predominantly spans Phase 2 (conceptual design), Phase 4 (data model mapping, or logical design) and Phase 5 (physical database design) in the design process that we discussed in Section 12.2. Discussion of Phase 5 is deferred to Chapter 16 in the context of query optimization. We discussed Phases 2 and 4 in detail with the use of the UML notation in Section 12.3 and pointed out the features of the tool Rational Rose, which support these phases. As we pointed out before, Rational Rose is more than just a database design tool. It is a software development tool and does database modeling and schema design in the form of class diagrams as part of its overall object-oriented application development methodology. In this section, we summarize the features and shortcomings of the set of commercial tools that are focussed on automating the process of conceptual, logical and physical design of databases.

When database technology was first introduced, most database design was carried out manually by expert designers, who used their experience and knowledge in the design process. However, at least two factors indicated that some form of automation had to be utilized if possible:

1. As an application involves more and more complexity of data in terms of relationships and constraints, the number of options or different designs to model the same information keeps increasing rapidly. It becomes difficult to deal with this complexity and the corresponding design alternatives manually.

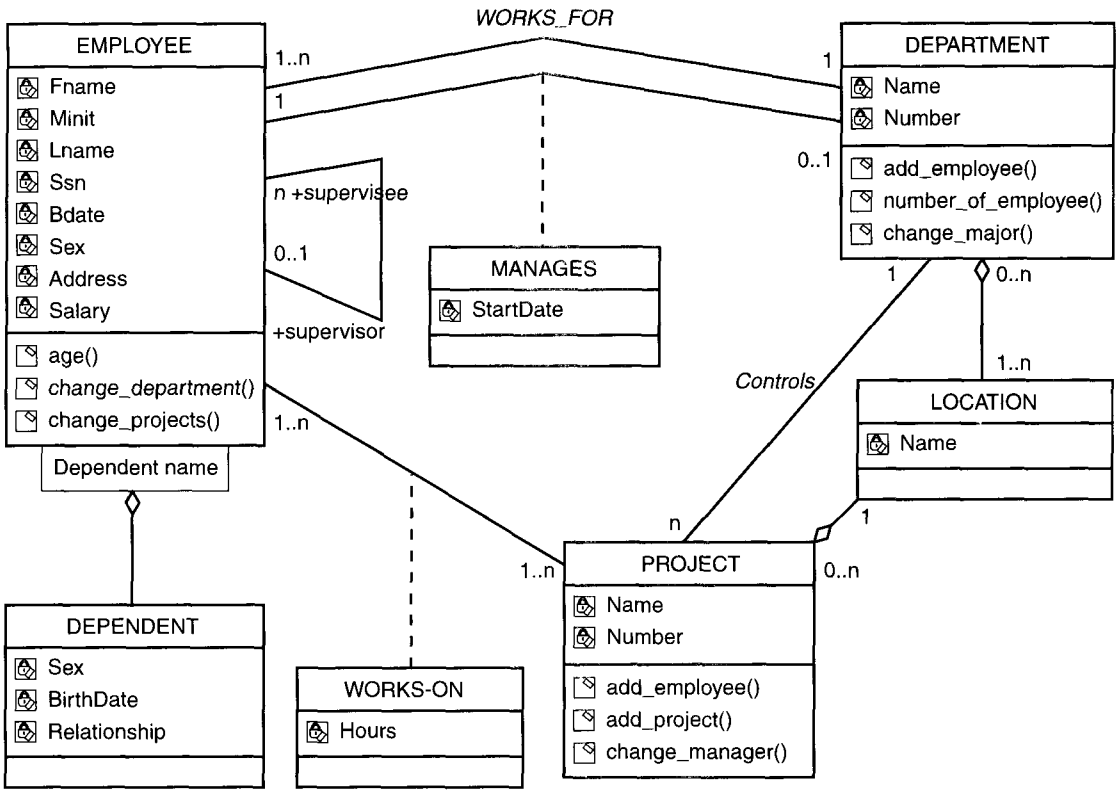


FIGURE 12.17 The Company Database Class Diagram (Fig.3.16) drawn in Rational Rose.

2. The sheer size of some databases runs into hundreds of entity types and relationship types making the task of manually managing these designs almost impossible. The meta information related to the design process we described in Section 12.2 yields another database that must be created, maintained, and queried as a database in its own right.

The above factors have given rise to many tools on the market that come under the general category of CASE (Computer-Aided Software Engineering) tools for database design. Rational Rose is a good example of a modern CASE tool. Typically these tools consist of a combination of the following facilities:

1. *Diagramming:* This allows the designer to draw a conceptual schema diagram, in some tool-specific notation. Most notations include entity types, relationship types that are shown either as separate boxes or simply as directed or undirected lines, cardinality constraints shown alongside the lines or in terms of the different types of

arrowheads or min/max constraints, attributes, keys, and so on.[10] Some tools display inheritance hierarchies and use additional notation for showing the partial versus total and disjoint versus overlapping nature of the generalizations. The diagrams are internally stored as conceptual designs and are available for modification as well as generation of reports, cross reference listings, and other uses.

2. *Model mapping:* This implements mapping algorithms similar to the ones we presented in Sections 9.1 and 9.2. The mapping is system-specific—most tools generate schemas in SQL DDL for Oracle, DB2, Informix, Sybase, and other RDBMSs. This part of the tool is most amenable to automation. The designer can edit the produced DDL files if needed.
3. *Design normalization:* This utilizes a set of functional dependencies that are supplied at the conceptual design or after the relational schemas are produced during logical design. The design decomposition algorithms from Chapter 15 are applied to decompose existing relations into higher normal form relations. Typically, tools lack the approach of generating alternative 3NF or BCNF designs and allowing the designer to select among them based on some criteria like the minimum number of relations or least amount of storage.

Most tools incorporate some form of physical design including the choice of indexes. A whole range of separate tools exists for performance monitoring and measurement. The problem of tuning a design or the database implementation is still mostly handled as a human decision-making activity. Out of the phases of design described in this chapter, one area where there is hardly any commercial tool support is view integration (see Section 12.2.2).

We will not survey database design tools here, but only mention the following characteristics that a good design tool should possess:

1. *An easy-to-use interface:* This is critical because it enables designers to focus on the task at hand, not on understanding the tool. Graphical and point and click interfaces are commonly used. A few tools like the SECSI tool from France use natural language input. Different interfaces may be tailored to beginners or to expert designers.
2. *Analytical components:* Tools should provide analytical components for tasks that are difficult to perform manually, such as evaluating physical design alternatives or detecting conflicting constraints among views. This area is weak in most current tools.
3. *Heuristic components:* Aspects of the design that cannot be precisely quantified can be automated by entering heuristic rules in the design tool to evaluate design alternatives.

10. We showed the ER, EER, and UML class diagram notations in Chapters 3 and 4. See Appendix A for an idea of the different types of diagrammatic notations used.

4. *Trade-off analysis:* A tool should present the designer with adequate comparative analysis whenever it presents multiple alternatives to choose from. Tools should ideally incorporate an analysis of a design change at the conceptual design level down to physical design. Because of the many alternatives possible for physical design in a given system, such tradeoff analysis is difficult to carry out and most current tools avoid it.
5. *Display of design results:* Design results, such as schemas, are often displayed in diagrammatic form. Aesthetically pleasing and well laid out diagrams are not easy to generate automatically. Multipage design layouts that are easy to read are another challenge. Other types of results of design may be shown as tables, lists, or reports that can be easily interpreted.
6. *Design verification:* This is a highly desirable feature. Its purpose is to verify that the resulting design satisfies the initial requirements. Unless the requirements are captured and internally represented in some analyzable form, the verification cannot be attempted.

Currently there is increasing awareness of the value of design tools, and they are becoming a must for dealing with large database design problems. There is also an increasing awareness that schema design and application design should go hand in hand, and the current trend among CASE tools is to address both areas. The popularity of Rational Rose is due to the fact that it approaches the two arms of the design process shown in Figure 12.1 concurrently, approaching database design and application design as a unified activity. Some vendors like Platinum provide a tool for data modeling and schema design (ERWin) and another for process modeling and functional design (BPWin). Other tools (for example, SECSI) use expert system technology to guide the design process by including design expertise in the form of rules. Expert system technology is also useful in the requirements collection and analysis phase, which is typically a laborious and frustrating process. The trend is to use both metadata repositories and design tools to achieve better designs for complex databases. Without a claim of being exhaustive, Table 12.1 lists some popular database design and application modeling tools. Companies in the table are listed in alphabetical order.

12.6 SUMMARY

We started this chapter by discussing the role of information systems in organizations; database systems are looked upon as a part of information systems in large-scale applications. We discussed how databases fit within an information system for information resource management in an organization and the life cycle they go through. We then discussed the six phases of the design process. The three phases commonly included as a part of database design are conceptual design, logical design (data model mapping), and physical design. We also discussed the initial phase of requirements collection and analysis, which is often considered to be a *predesign phase*. In addition, at some point during the design, a specific DBMS package must be chosen. We discussed some of the organizational

TABLE 12.1 SOME OF THE CURRENTLY AVAILABLE AUTOMATED DATABASE DESIGN TOOLS

COMPANY	TOOL	FUNCTIONALITY
Embarcadero Technologies	ER Studio	Database Modeling in ER and IDEF1X
	DB Artisan	Database administration and space and security management
Oracle	Developer 2000 and Designer 2000	Database modeling, application development
Popkin Software	System Architect 2001	Data modeling, object modeling, process modeling, structured analysis/design
Platinum Technology	Platinum Enterprise Modeling Suite: ERwin, BPWin, Paradigm Plus	Data, process, and business component modeling
Persistence Inc.	Powertier	Mapping from O-O to relational model
Rational	Rational Rose	Modeling in UML and application generation in C++ and JAVA
Rogue Ware	RW Metro	Mapping from O-O to relational model
Resolution Ltd.	XCase	Conceptual modeling up to code maintenance
Sybase	Enterprise Application Suite	Data modeling, business logic modeling
Visio	Visio Enterprise	Data modeling, design and reengineering Visual Basic and Visual C++

criteria that come into play in selecting a DBMS. As performance problems are detected, and as new applications are added, designs have to be modified. The importance of designing both the schema and the applications (or transactions) was highlighted. We discussed different approaches to conceptual schema design and the difference between centralized schema design and the view integration approach.

We introduced UML diagrams as an aid to the specification of database models and designs. We introduced the entire range of structural and behavioral diagrams and then described the notational detail about the following types of diagrams: use case, sequence, statechart. Class diagrams have already been discussed in Sections 3.8 and 4.6, respectively. We showed how requirements for a university database are specified using these diagrams and can be used to develop the conceptual design of the database. Only

illustrative details and not the complete specification were supplied. Appendix B develops a complete case study of the design and implementation of a database. Then we discussed the currently popular software development tool—Rational Rose and the Rose Data Modeler—that provides support for the conceptual design and logical design phases of database design. Rose is a much broader tool for design of information systems at large. Finally, we briefly discussed the functionality and desirable features of commercial automated database design tools that are more focussed on database design as opposed to Rose. A tabular summary of features was pesented.

Review Questions

12.1. What are the six phases of database design? Discuss each phase.
12.2. Which of the six phases are considered the main activities of the database design process itself? Why?
12.3. Why is it important to design the schemas and applications in parallel?
12.4. Why is it important to use an implementation-independent data model during conceptual schema design? What models are used in current design tools? Why?
12.5. Discuss the importance of Requirements Collection and Analysis.
12.6. Consider an actual application of a database system of interest. Define the requirements of the different levels of users in terms of data needed, types of queries, and transactions to be processed.
12.7. Discuss the characteristics that a data model for conceptual schema design should possess.
12.8. Compare and contrast the two main approaches to conceptual schema design.
12.9. Discuss the strategies for designing a single conceptual schema from its requirements.
12.10. What are the steps of the view integration approach to conceptual schema design? What are the difficulties during each step?
12.11. How would a view integration tool work? Design a sample modular architecture for such a tool.
12.12. What are the different strategies for view integration.
12.13. Discuss the factors that influence the choice of a DBMS package for the information system of an organization.
12.14. What is system-independent data model mapping? How is it different from system-dependent data model mapping?
12.15. What are the important factors that influence physical database design?
12.16. Discuss the decisions made during physical database design.
12.17. Discuss the macro and micro life cycles of an information system.
12.18. Discuss the guidelines for physical database design in RDBMSs.
12.19. Discuss the types of modifications that may be applied to the logical database design of a relational database.
12.20. What functions do the typical database design tools provide?
12.21. What type of functionality would be desirable in automated tools to support optimal design of large databases?

Selected Bibliography

There is a vast amount of literature on database design. We first list some of the books that address database design. Batini et al. (1992) is a comprehensive treatment of conceptual and logical database design. Wiederhold (1986) covers all phases of database design, with an emphasis on physical design. O'Neil (1994) has a detailed discussion of physical design and transaction issues in reference to commercial RDBMSs. A large body of work on conceptual modeling and design was done in the eighties. Brodie et al. (1984) gives a collection of chapters on conceptual modeling, constraint specification and analysis, and transaction design. Yao (1985) is a collection of works ranging from requirements specification techniques to schema restructuring. Teorey (1998) emphasizes EER modeling and discusses various aspects of conceptual and logical database design. McFadden and Hoffer (1997) is a good introduction to the business applications issues of database management.

Navathe and Kerschberg (1986) discuss all phases of database design and point out the role of data dictionaries. Goldfine and Konig (1988) and ANSI (1989) discuss the role of data dictionaries in database design. Rozen and Shasha (1991) and Carlis and March (1984) present different models for the problem of physical database design. Object-oriented database design is discussed in Schlaer and Mellor (1988), Rumbaugh et al. (1991), Martin and Odell (1991), and Jacobson (1992). Recent books by Blaha and Premerlani (1998) and Rumbaugh et al. (1999) consolidate the existing techniques in object-oriented design. Fowler and Scott (1997) is a quick introduction to UML.

Requirements collection and analysis is a heavily researched topic. Chatzoglu et al. (1997) and Lubars et al. (1993) present surveys of current practices in requirements capture, modeling, and analysis. Carroll (1995) provides a set of readings on the use of scenarios for requirements gathering in early stages of system development. Wood and Silver (1989) gives a good overview of the official Joint Application Design (JAD) process. Potter et al. (1991) describes the Z notation and methodology for formal specification of software. Zave (1997) has classified the research efforts in requirements engineering.

A large body of work has been produced on the problems of schema and view integration, which is becoming particularly relevant now because of the need to integrate a variety of existing databases. Navathe and Gadgil (1982) defined approaches to view integration. Schema integration methodologies are compared in Batini et al. (1986). Detailed work on n-ary view integration can be found in Navathe et al. (1986), Elmasri et al. (1986), and Larson et al. (1989). An integration tool based on Elmasri et al. (1986) is described in Sheth et al. (1988). Another view integration system is discussed in Hayne and Ram (1990). Casanova et al. (1991) describes a tool for modular database design. Motro (1987) discusses integration with respect to preexisting databases. The binary balanced strategy to view integration is discussed in Teorey and Fry (1982). A formal approach to view integration, which uses inclusion dependencies, is given in Casanova and Vidal (1982). Ramesh and Ram (1997) describe a methodology for integration of relationships in schemas utilizing the knowledge of integrity constraints; this extends the previous work of Navathe et al. (1984a). Sheth at al. (1993) describe the issues of building global schemas by reasoning about attribute relationships and entity equivalences. Navathe and Savasere (1996) describe a practical approach to building

global schemas based on operators applied to schema components. Santucci (1998) provides a detailed treatment of refinement of EER schemas for integration. Castano et al. (1999) present a comprehensive survey of conceptual schema analysis techniques.

Transaction design is a relatively less thoroughly researched topic. Mylopoulos et al. (1980) proposed the TAXIS language, and Albano et al. (1987) developed the GALILEO system, both of which are comprehensive systems for specifying transactions. The GORDAS language for the ECR model (Elmasri et al. 1985) contains a transaction specification capability. Navathe and Balaraman (1991) and Ngu (1991) discuss transaction modeling in general for semantic data models. Elmagarmid (1992) discusses transaction models for advanced applications. Batini et al. (1992, chaps. 8, 9, and 11) discuss high level transaction design and joint analysis of data and functions. Shasha (1992) is an excellent source on database tuning.

Information about some well-known commercial database design tools can be found at the Web sites of the vendors (see company names in Table 12.1). Principles behind automated design tools are discussed in Batini et al. (1992, chap. 15). The SECSI tool from France is described in Metais et al. (1998). DKE (1997) is a special issue on natural language issues in databases.