chapter 14

Basics of Functional Dependencies and Normalization for Relational Databases

In Chapters 5 through 8, 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) 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 discussed in Chapter 9 are followed. However, we still need some formal way of analyzing why one grouping of attributes into a relation schema may be better than another. While discussing database design in Chapters 3, 4, and 9, we did not develop 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 **physical 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 14.1.

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. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes. 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 primarily to the top-down design approach, and as such is more appropriate when performing design of databases by analysis and decomposition of sets of attributes that appear together in files, in reports, and on forms in real-life situations.

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are *information preservation* and *minimum redundancy*. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships, which are described using a model such as the EER model. Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

We start this chapter by informally discussing some criteria for good and bad relation schemas in Section 14.1. In Section 14.2, 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. In Section 14.3, we discuss normal forms and the process of normalization using functional dependencies. Successive normal forms are defined to meet a set of desirable constraints expressed using primary keys and functional dependencies. The normalization procedure consists of applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary. In Section 14.4, we discuss more general definitions of normal forms that can be directly

[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).

applied to any given design and do not require step-by-step analysis and normalization. Sections 14.5 to 14.7 discuss further normal forms up to the fifth normal form. In Section 14.6 we introduce the multivalued dependency (MVD), followed by the join dependency (JD) in Section 14.7. Section 14.8 summarizes the chapter.

Chapter 15 continues the development of the theory related to the design of good relational schemas. We discuss desirable properties of relational decomposition—nonadditive join property and functional dependency preservation property. A general algorithm that tests whether or not a decomposition has the nonadditive (or *lossless*) join property (Algorithm 15.3 is also presented). We then discuss properties of functional dependencies and the concept of a minimal cover of dependencies. We consider the bottom-up approach to database design consisting of a set of algorithms to design relations in a desired normal form. These algorithms assume as input a given set of functional dependencies and achieve a relational design in a target normal form while adhering to the above desirable properties. In Chapter 15 we also define additional types of dependencies that further enhance the evaluation of the *goodness* of relation schemas.

If Chapter 15 is not covered in a course, we recommend a quick introduction to the desirable properties of decomposition from Section 15.2. and the importance of the non-additive join property during decomposition.

14.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information 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 will see.

14.1.1 Imparting Clear Semantics to Attributes in Relations

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. The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. In Chapter 5 we discussed how a relation can be interpreted as a set of facts. If the conceptual design described in Chapters 3 and 4 is done carefully and the mapping procedure in Chapter 9 is followed systematically, the relational schema design should have a clear meaning.

In general, the easier it is to explain the semantics of the relation—or in other words, what a relation exactly means and stands for—the better the relation schema design will be. To illustrate this, consider Figure 14.1, a simplified version of the COMPANY relational database schema in Figure 5.5, and Figure 14.2, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is 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 Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, whereas 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 attributes can be explained is an *informal measure* of how well the relation is designed.

Figure 14.1
A simplified COMPANY relational database schema.

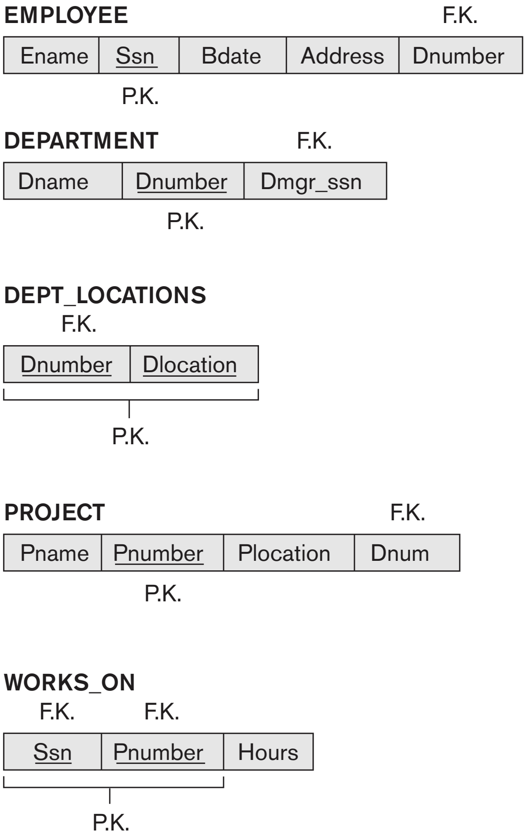


Figure 14.2
Sample database state for the relational database schema in Figure 14.1.

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	291Berry, 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

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
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

The semantics of the other two relation schemas in Figure 14.1 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 14.1 may be considered as easy to explain and therefore 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 relationship 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.

Examples of Violating Guideline 1. The relation schemas in Figures 14.3(a) and 14.3(b) 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 14.2.) A tuple in the EMP_DEPT relation schema in Figure 14.3(a) represents a single employee but includes, along with the Dnumber (the identifier for the department he/she works for), additional information—namely, the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager. For the EMP_PROJ relation in Figure 14.3(b), each tuple relates an employee to a project but also includes

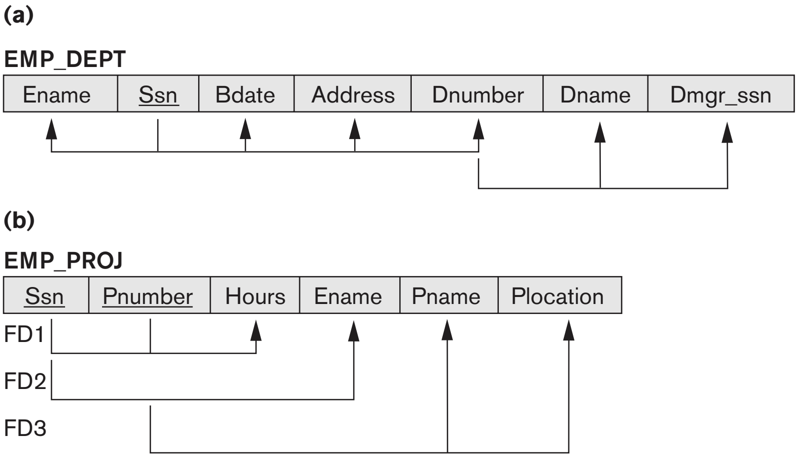


Figure 14.3
Two relation schemas suffering from update anomalies.
(a) EMP_DEPT and
(b) EMP_PROJ.

the employee name (Ename), project name (Pname), and project location (Plocation). Although there is nothing wrong logically with these two relations, 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 and the WORKS_ON relationship. Hence, they fare poorly against the above measure of design quality. They may be used as views, but they cause problems when used as base relations, as we discuss in the following section.

14.1.2 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 14.2 with that for an EMP_DEPT base relation in Figure 14.4, 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, Dmgr_ssn) 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 14.2. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 14.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.[2]

Insertion Anomalies. Insertion anomalies 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 all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT. In the design of Figure 14.2, 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

[2] These anomalies were identified by Codd (1972a) to justify the need for normalization of relations, as we shall discuss in Section 15.3.

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
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 14.4
Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 14.2. These may be stored as base relations for performance reasons.

attributes for employee. This violates the entity integrity for EMP_DEPT because its primary key Ssn cannot be null. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values anymore. This problem does not occur in the design of Figure 14.2 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 just discussed. 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 inadvertently from the database. This problem does not occur in the database of Figure 14.2 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]

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence, we can state the next guideline as 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,[4] 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 14.2 through 14.4 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. If EMP_DEPT is used as a stored relation (known otherwise as a *materialized view*) in addition to the base relations of EMPLOYEE and DEPARTMENT, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates). This way, 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 attributes frequently referenced in important queries.

14.1.3 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

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

[4] Other application considerations may dictate and make certain anomalies unavoidable. For example, the EMP_DEPT relation may correspond to a query or a report that is frequently required.

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. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.[6] Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we 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 with NULL values 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 15% 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.

14.1.4 Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 14.5(a), which can be used instead of the single EMP_PROJ relation in Figure 14.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on *at least one* project located at Plocation. A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works the given Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation. Figure 14.5(b) shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the EMP_PROJ relation in Figure 14.4, which are obtained by applying the appropriate PROJECT (π) operations to EMP_PROJ.

[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.

[6] In Section 5.5.1 we presented comparisons involving NULL values where the outcome (in three-valued logic) is TRUE, FALSE, and UNKNOWN.

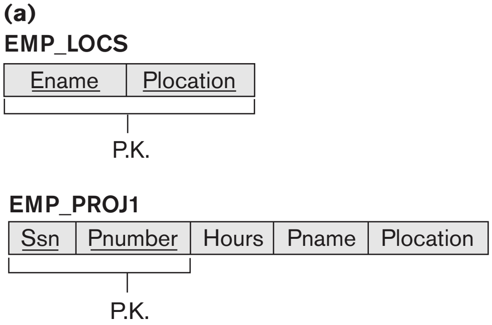


Figure 14.5
Particularly poor design for the EMP_PROJ relation in Figure 14.3(b). (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 14.4 onto the relations EMP_LOCS and EMP_PROJ1.

(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	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	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

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 14.6, the result of applying the join to only the tuples for employee with Ssn = "123456789" 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 information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 14.6. It is left to the reader to complete the result of NATURAL JOIN operation on the EMP_PROJ1 and EMP_LOCS tables in their entirety and to mark the spurious tuples in this result.

	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 14.6
Result of applying NATURAL JOIN to the tuples in EMP_PROJ1 and EMP_LOCS of Figure 14.5 just for employee with Ssn = "123456789". Generated spurious tuples are marked by asterisks.

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 happens to be 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 now informally state another design guideline.

Guideline 4. Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs 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 Section 15.2 we discuss a formal condition called the nonadditive (or lossless) join property that guarantees that certain joins do not produce spurious tuples.

14.1.5 Summary and Discussion of Design Guidelines

In Sections 14.1.1 through 14.1.4, 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 selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

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. First we discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas as the established and accepted standards of quality in relational design. The strategy for achieving a good design is to decompose a badly designed relation appropriately to achieve higher normal forms. We also briefly introduce additional normal forms that deal with additional dependencies. In Chapter 15, we discuss the properties of decomposition in detail and provide a variety of algorithms related to functional dependencies, goodness of decomposition, and the bottom-up design of relations by using the functional dependencies as a starting point.

14.2 Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. 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 14.3 we see how it can be used to define normal forms for relation schemas.

14.2.1 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, … , A_n; let us think of the whole database as being described by a single **universal** relation schema $R = \{A_1, A_2, \ldots , A_n\}$.[7] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[8]

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

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

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

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance $r(R)$—that is, X is a **candidate key** of R—this implies that $X \rightarrow Y$ 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 is a candidate key of R, then $X \rightarrow R$.
- 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. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R. Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} → Ssn should normally hold for any adult in the United States and hence should hold whenever these attributes appear in a relation.[9] It is also possible that certain functional

[7] This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 15.

[8] 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.

[9] Note that there are databases, such as those of credit card agencies or police departments, where this functional dependency may not hold because of fraudulent records resulting from the same driver's license number being used by two or more different individuals.

dependencies may cease to exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

Consider the relation schema EMP_PROJ in Figure 14.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. Ssn → Ename

b. Pnumber → {Pname, Plocation}

c. {Ssn, Pnumber} → Hours

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename,* and so on.

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*. For example, Figure 14.7 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 'Database Systems,' we can conclude that Teacher *does not* functionally determine Course.

Given a populated relation, we cannot determine which FDs hold and which do not unless we know the meaning of and the relationships among the attributes. All we can say is that a certain FD *may* exist if it holds in that particular extension. We cannot guarantee its existence until we understand the meaning of the corresponding attributes. We can, however, emphatically state that a certain FD *does not hold* if there are

TEACH

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

Figure 14.7
A relation state of TEACH with a *possible* functional dependency TEXT → COURSE. However, TEACHER → COURSE, TEXT → TEACHER and COURSE → TEXT are ruled out.

Figure 14.8
A relation *R*(A, B, C, D) with its extension.

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

tuples that show the violation of such an FD. See the illustrative example relation in Figure 14.8. Here, the following FDs *may hold* because the four tuples in the current extension have no violation of these constraints: $B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$. However, the following *do not* hold because we already have violations of them in the given extension: $A \rightarrow B$ (tuples 1 and 2 violate this constraint); $B \rightarrow A$ (tuples 2 and 3 violate this constraint); $D \rightarrow C$ (tuples 3 and 4 violate it).

Figure 14.3 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, whereas the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by *F* the set of functional dependencies that are specified on relation schema *R*. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in *F*. Those other dependencies can be *inferred* or *deduced* from the FDs in *F*. We defer the details of inference rules and properties of functional dependencies to Chapter 15.

14.3 Normal Forms Based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify how to use them to develop a formal methodology for testing and improving 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:

- Perform a conceptual schema design using a conceptual model such as ER or EER and 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 we 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 14.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 3 that are needed here. Then we discuss the first normal form (1NF) in Section 14.3.4, and we present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 14.3.5 and 14.3.6, respectively.

14.3.1 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 a single analytical tool: 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 briefly discussed in Sections 14.6 and 14.7.

Normalization of data can be considered 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 14.1.2. It can be considered as a “filtering” or “purification” process to make the design have successively better quality. An unsatisfactory relation schema that does not meet the condition for a normal form—the **normal form test**—is decomposed into smaller relation schemas that contain a subset of the attributes and meet the test that was otherwise not met by the original relation. 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

Definition. 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 **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem discussed in Section 14.1.4 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 15.2.2. We defer the discussion of the formal concepts and techniques that guarantee the above two properties to Chapter 15.

14.3.2 Practical Use of Normal Forms

Most practical design projects in commercial and governmental environment acquire existing designs of databases from previous designs, from designs in legacy models, or from existing files. They are certainly interested in assuring that the designs are good quality and sustainable over long periods of time. Existing designs are evaluated by applying the tests for normal forms, and 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 Sections 14.6 and 14.7, the practical utility of these normal forms becomes questionable. The reason is that the constraints on which they are based are rare and hard for the database designers and users to understand or to detect. Designers and users must either already know them or discover them as a part of the business. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 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 14.1.2. Doing so incurs the corresponding penalties of dealing with the anomalies.

Definition. Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

14.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema from Chapter 3.

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 anymore.

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 14.1, {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. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 14.1, {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 14.1, 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 independently attack different types of problems arising from problematic functional dependencies among attributes. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation *already satisfies* 2NF.

14.3.4 First Normal Form

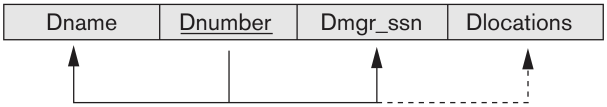
First normal form (1NF)is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (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 14.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 14.9(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and a sample relation state are shown in Figure 14.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 14.9(b). 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.

(a)

DEPARTMENT



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 14.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

- 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.[10]

In either case, the DEPARTMENT relation in Figure 14.9 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 3.1. 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 newly formed relation is the combination {Dnumber, Dlocation}, as shown in Figure 14.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

[10] 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.

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 14.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation and hence is rarely adopted.
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 spurious semantics about the ordering among the location values; that *ordering* 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. For all these reasons, it is best to avoid this alternative.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general; it places no maximum limit on the 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 14.10 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 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 12) and XML data (see Chapter 13) 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 14.10(a) and (b), whereas 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, as shown in Figure 14.10(c).

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

(a)

EMP_PROJ

		Projs	
Ssn	Ename	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 14.10
Normalizing nested relations into 1NF.
(a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample 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.

unnormalized relation schema with many levels of nesting into 1NF relations. As an example, consider the following:

CANDIDATE (Ssn, Name, {JOB_HIST (Company, Highest_position,
{SAL_HIST (Year, Max_sal)})})

The foregoing describes data about candidates applying for jobs with their job history as a nested relation within which the salary history is stored as a deeper nested relation. The first normalization using internal partial keys Company and Year, respectively, results in the following 1NF relations:

CANDIDATE_1 (Ssn, Name)
CANDIDATE_JOB_HIST (Ssn, Company, Highest_position)
CANDIDATE_SAL_HIST (Ssn, Company, Year, Max-sal)

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 strategy 2 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 that are typically discovered at a later stage of normalization and that are handled by multivalued dependencies and 4NF, which we will discuss in Section 14.6. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(Ss#, Car_lic#) and P2(Ss#, Phone#).

A note about the relations that involve attributes that go beyond just numeric and character string data. It is becoming common in today's databases to incorporate images, documents, video clips, audio clips, and so on. When these are stored in a relation, the entire object or file is treated as an atomic value, which is stored as a BLOB (binary large object) or CLOB (character large object) data type using SQL. For practical purposes, the object is treated as an atomic, single-valued attribute and hence it maintains the 1NF status of the relation.

14.3.5 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 anymore; that is, for any attribute $A\ \varepsilon\ X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A\ \varepsilon\ X$ can be removed from X and the dependency still holds; that is, for some $A\ \varepsilon\ X$, $(X - \{A\}) \rightarrow Y$. In Figure 14.3(b), {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 14.3(b) 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. Each of the functional dependencies FD2 and FD3 violates 2NF because Ename can be functionally determined by only Ssn, and both Pname and Plocation can be functionally determined by only Pnumber. Attributes Ssn and Pnumber are a part of 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. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure 14.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 14.11(a), each of which is in 2NF.

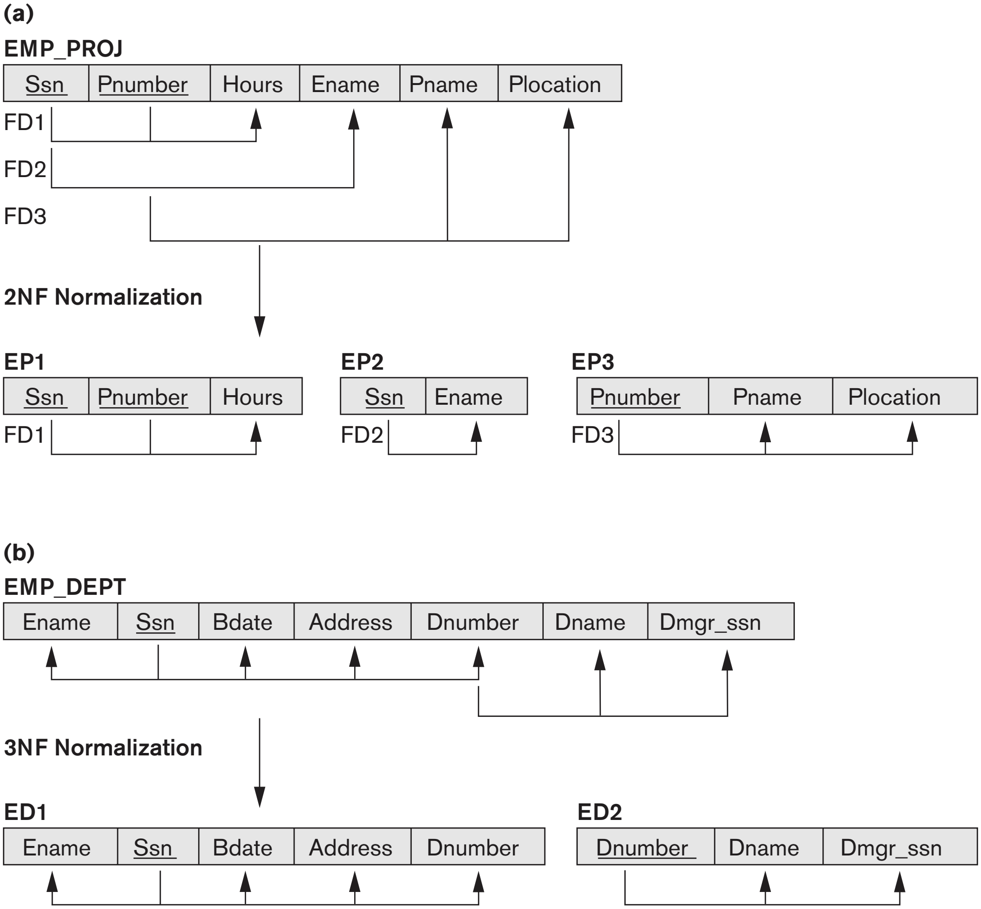


Figure 14.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

14.3.6 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 exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R,[11] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency Ssn → Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 14.3(a), because both the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn 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 Dmgr_ssn 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 14.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (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 14.11(b). Intuitively, we see that ED1 and ED2 represent independent facts about employees and departments, both of which are entities in their own right. 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 (a 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. Moreover, the general definition of 3NF we present in Section 14.4.2 automatically covers the condition that the relation also satisfies 2NF. Table 14.1 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.

14.4 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 14.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on

[11] 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 14.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued 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).

the *primary key*. The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

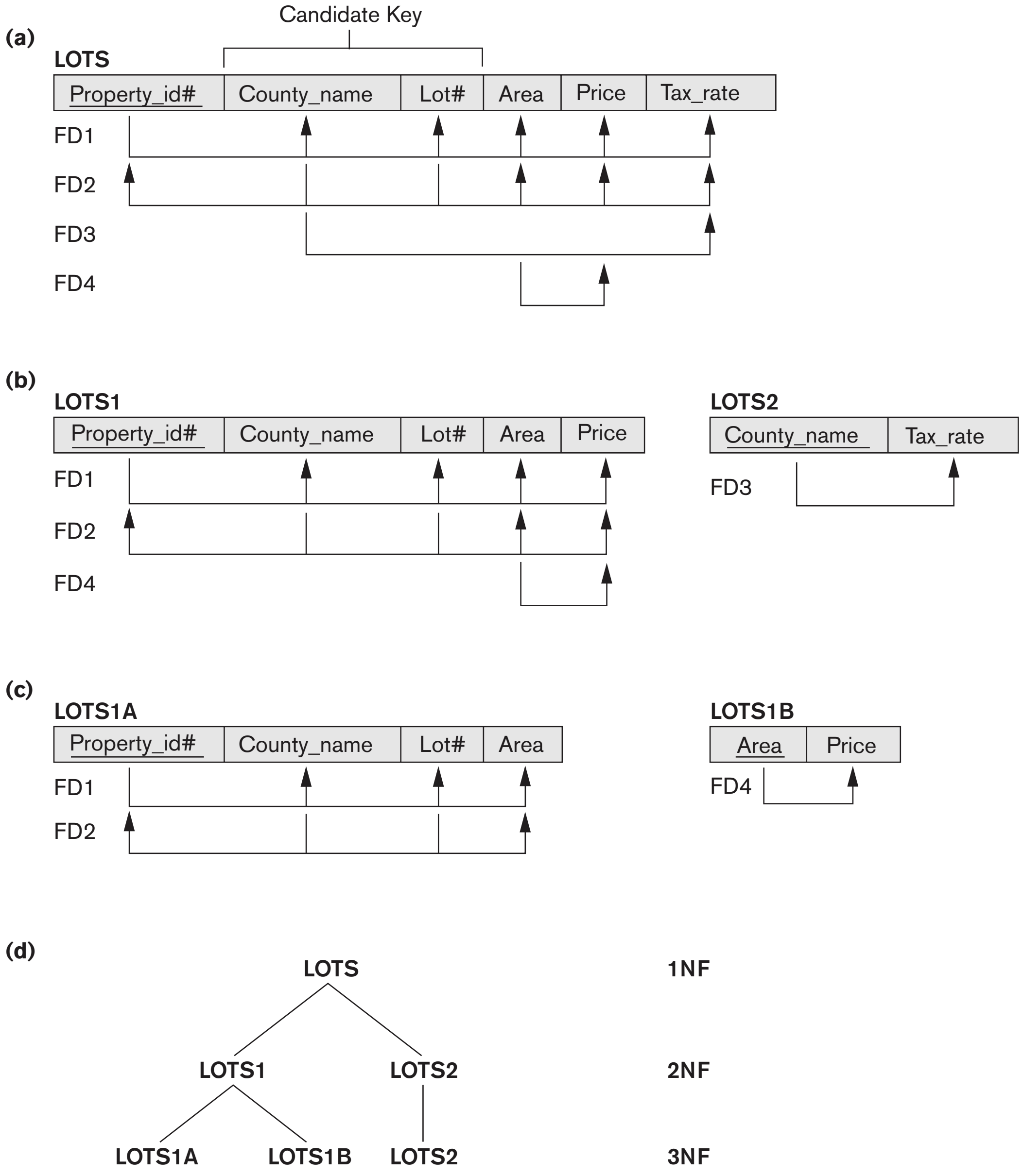
14.4.1 General Definition of Second Normal Form

> **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*.[12]

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 14.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

[12] 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 14.12
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) Progressive normalization of LOTS into a 3NF design.



Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure 14.12(a) hold. We choose Property_id# as the primary key, so it is underlined in Figure 14.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate
FD4: Area → Price

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), whereas 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 14.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

14.4.2 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.[13]

According to this definition, LOTS2 (Figure 14.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 14.12(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

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

- LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. In other words, if a relation passes the general 3NF test, then it automatically passes the 2NF test.

[13] Note that based on inferred f.d.'s (which are discussed in Section 15.1), the f.d. $Y \rightarrow YA$ also holds whenever $Y \rightarrow A$ is true. Therefore, a slightly better way of saying this statement is that $\{A\text{-}X\}$ is a prime attribute of R.

If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF by the general definition above because the LHS County_name in FD3 is not a superkey. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

14.4.3 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 meets either of the two conditions, namely (a) and (b). The first condition "catches" two types of problematic dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of *R* functionally determines a nonprime attribute. Here we have a partial dependency that violates 2NF.

Thus, condition (a) alone addresses the problematic dependencies that were causes for second and third normalization as we discussed.

Therefore, we can state a **general alternative definition of 3NF** as follows:

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

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

However, note the clause (b) in the general definition of 3NF. It allows certain functional dependencies to slip through or escape in that they are OK with the 3NF definition and hence are not "caught" by the 3NF definition even though they may be potentially problematic. The Boyce-Codd normal form "catches" these dependencies in that it does not allow them. We discuss that normal form next.

14.5 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. We pointed out in the last subsection that although 3NF allows functional dependencies that conform to the clause (b) in the 3NF definition, BCNF disallows them and hence is a stricter definition of a normal form.

Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 14.12(a) 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

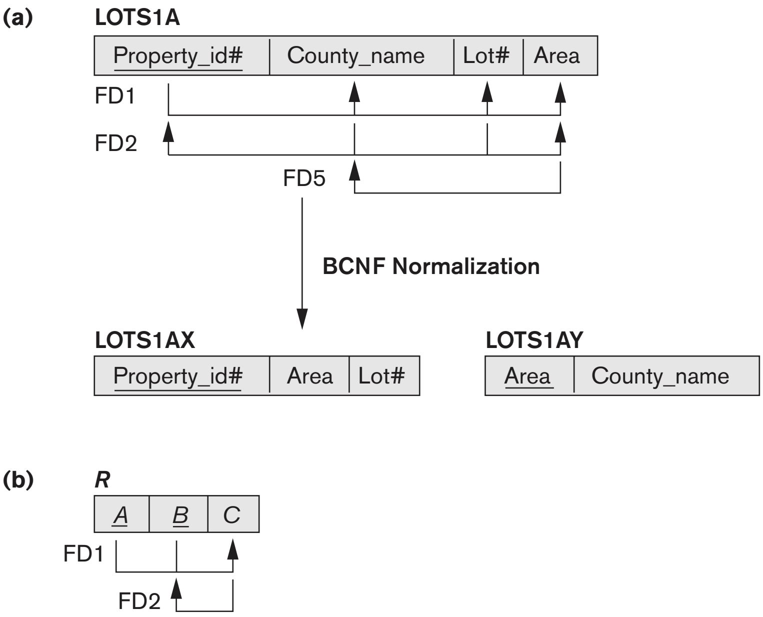


Figure 14.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. $C \rightarrow B$.

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 this f.d. conforms to clause (b) in the general definition of 3NF, County_name being 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 (see Figure 14.13). 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 $X \rightarrow A$ holds in R, then X is a superkey of R.

The formal definition of BCNF differs from the definition of 3NF in that clause (b) of 3NF, which allows f.d.'s having the RHS as a prime attribute, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because Area is not a superkey of LOTS1A. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 14.13(a). 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 there exists some f.d. $X \rightarrow A$ that 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 14.13(b) illustrates the general case of such a relation. Such an f.d. leads to potential redundancy of data, as we illustrated above in case of FD5: Area → County_name.in LOTS1A relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since both were developed historically to be intermediate normal forms as stepping stones to 3NF and BCNF.

14.5.1 Decomposition of Relations not in BCNF

As another example, consider Figure 14.14, which shows a relation TEACH with the following dependencies:

FD1: {Student, Course} → Instructor
FD2:[14] Instructor → Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.13(b), 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. R1 (Student, Instructor) and R2(Student, Course)
2. R1 (Course, Instructor) and R2(Course, Student)
3. R1 (Instructor, Course) and R2(Instructor, Student)

All three decompositions *lose* the functional dependency FD1. The question then becomes: Which of the above three is a *desirable decomposition*? As we pointed out earlier (Section 14.3.1), we strive to meet two properties of decomposition during

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
Narayan	Operating Systems	Ammar

Figure 14.14
A relation TEACH that is in 3NF but not BCNF.

[14] This dependency means that *each instructor teaches one course* is a constraint for this application.

the normalization process: the nonadditive join property and the functional dependency preservation property. We are not able to meet the functional dependency preservation for any of the above BCNF decompositions as seen above; but we must meet the nonadditive join property. A simple test comes in handy to test the binary decomposition of a relation into two relations:

> **NJB (Nonadditive 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^{+}[15], or
> - The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^{+}

If we apply this test to the above three decompositions, we find that only the third decomposition meets the test. In the third decomposition, the $R_1 \cap R_2$ for the above test is Instructor and $R_1 - R_2$ is Course. Because Instructor → Course, the NJB test is satisfied and the decomposition is nonadditive. (It is left as an exercise for the reader to show that the first two decompositions do not meet the NJB test.) Hence, the proper decomposition of TEACH into BCNF relations is:

> TEACH1 (Instructor, Course) and TEACH2 (Instructor, Student)

We make sure that we meet this property, because nonadditive decomposition is a must during normalization. You should verify that this property holds with respect to our informal successive normalization examples in Sections 14.3 and 14.4 and also by the decomposition of LOTS1A into two BCNF relations LOTS1AX and LOTS1AY.

In general, a relation R not in BCNF can be decomposed so as to meet the nonadditive join property by the following procedure.[16] It decomposes R successively into a set of relations that are in BCNF:

> Let R be the relation not in BCNF, let $X \subseteq R$, and let $X \rightarrow A$ be the FD that causes a violation of BCNF. R may be decomposed into two relations:
>
> $R - A$
>
> XA

If either $R - A$ or XA. is not in BCNF, repeat the process.

The reader should verify that if we applied the above procedure to LOTS1A, we obtain relations LOTS1AX and LOTS1AY as before. Similarly, applying this procedure to TEACH results in relations TEACH1 and TEACH2

[15] The notation F^+ refers to the cover of the set of functional dependencies and includes all f.d.'s implied by F. It is discussed in detail in Section 15.1. Here, it is enough to make sure that one of the two f.d.'s actually holds for the nonadditive decomposition into R_1 and R_2 to pass this test.

[16] Note that this procedure is based on Algorithm 15.5 from Chapter 15 for producing BCNF schemas by decomposition of a universal schema.

Note that if we designate (Student, Instructor) as a primary key of the relation TEACH, the FD instructor → Course causes a partial (non-fully-functional) dependency of Course on a part of this key. This FD may be removed as a part of second normalization (or by a direct application of the above procedure to achieve BCNF) yielding exactly the same two relations in the result. This is an example of a case where we may reach the same ultimate BCNF design via alternate paths of normalization.

14.6 Multivalued Dependency and Fourth Normal Form

Consider the relation EMP shown in Figure 14.15(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another.[17] To keep the relation state consistent and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. In the relation state shown in Figure 14.15(a), the employee with Ename Smith works on two projects 'X' and 'Y' and has two dependents 'John' and 'Anna', and therefore there are four tuples to represent these facts together. The relation EMP is an **all-key relation** (with key made up of all attributes) and therefore has no f.d.'s and as such qualifies to be a BCNF relation. We can see that there is an obvious redundancy in the relation EMP—the dependent information is repeated for every project and the project information is repeated for every dependent.

As illustrated by the EMP relation, some relations have constraints that cannot be specified as functional dependencies and hence are not in violation of BCNF. To address this situation, the concept of *multivalued dependency* (MVD) was proposed and, based on this dependency, the *fourth normal form* was defined. A more formal discussion of MVDs and their properties is deferred to Chapter 15. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 14.3.4), which disallows an attribute in a tuple to have a *set of values*. If more than one multivalued attribute is present, the second option of normalizing the relation (see Section 14.3.4) introduces a multivalued dependency. Informally, whenever two *independent* 1:N relationships A:B and A:C are mixed in the same relation, $R(A, B, C)$, an MVD may arise.[18]

14.6.1 Formal Definition of Multivalued Dependency

Definition. A multivalued dependency $X \rightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any

[17] In an ER diagram, each would be represented as a multivalued attribute or as a weak entity type (see Chapter 7).

[18] This MVD is denoted as $A \twoheadrightarrow B|C$.

(a) EMP

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

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

(c) SUPPLY

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

(d) R_1

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

R_2

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

R_3

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

Figure 14.15
Fourth and fifth normal forms.
(a) The EMP relation with two MVDs: Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.
(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3).
(d) Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3.

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,[19] where we use Z to denote $(R - (X \cup Y))$:[20]

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

[19] The tuples t_1, t_2, t_3, and t_4 are not necessarily distinct.

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

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 $X \twoheadrightarrow Y|Z$.

An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup \mathrm{Y} = R$. For example, the relation EMP_PROJECTS in Figure 14.15(b) has the trivial MVD Ename $\twoheadrightarrow$ Pname and the relation EMP_DEPENDENTS has the trivial MVD Ename $\twoheadrightarrow$ Dname. 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 14.15(a), 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 need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

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 4**NF** 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^+,[21] X is a superkey for R.

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.
- An all-key relation such as the EMP relation in Figure 14.15(a), which has no FDs but has the MVD Ename $\twoheadrightarrow$ Pname | Dname, is not in 4NF.
- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure 14.15(a). EMP is not in 4NF because in the nontrivial MVDs Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname,

[21] F^+ refers to the cover of functional dependencies F, or all dependencies that are implied by F. This is defined in Section 15.1.

and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 14.15(b). 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.

14.7 Join Dependencies and Fifth Normal Form

In our discussion so far, we have pointed out the problematic functional dependencies and shown how they were eliminated by a process of repeated binary decomposition during the process of normalization to achieve 1NF, 2NF, 3NF, and BCNF. These binary decompositions must obey the NJB property for which we introduced a test in Section 14.5 while discussing the decomposition to achieve BCNF. Achieving 4NF typically involves eliminating MVDs by repeated binary decompositions as well. However, in some cases there may be no nonadditive join decomposition of R into *two* relation schemas, but there may be a nonadditive 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 peculiar semantic constraint that is difficult to detect in practice; therefore, normalization into 5NF is rarely done in practice.

Definition. A **join dependency** (**JD**), denoted by JD($R_1, R_2, \ldots, R_n$), 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 $R_1, R_2, \ldots, R_n$. Hence, for every such r we have

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

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($R_1, R_2, \ldots, R_n$), specified on relation schema R, is a **trivial** JD if one of the relation schemas R_i in JD($R_1, R_2, \ldots, R_n$) 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 thus does not specify any constraint on R. We can now define the 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($R_1, R_2, \ldots, R_n$) in F^+ (that is, implied by F),[22] every R_i is a superkey of R.

[22] Again, F^+ refers to the cover of functional dependencies F, or all dependencies that are implied by F. This is defined in Section 15.1.

For an example of a JD, consider once again the SUPPLY all-key relation in Figure 14.15(c). 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(R_1, R_2, R_3) among the three projections R_1 (Sname, Part_name), R_2 (Sname, Proj_name), and R_3 (Part_name, Proj_name) of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure 14.15(c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line. Figure 14.15(d) shows how the SUPPLY relation *with the join dependency* is decomposed into three relations R_1, R_2, and R_3 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 sample relation in Figure 14.15(c) and its projections in Figure 14.15(d). This is because only the JD exists, but no MVDs are specified. Notice, too, that the JD(R_1, R_2, R_3) is specified on *all* legal relation states, not just on the one shown in Figure 14.15(c).

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. Therefore, the current practice of database design pays scant attention to them. One result due to Date and Fagin (1992) relates to conditions detected using f.d.'s alone and ignores JDs completely. It states: "If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF."

14.8 Summary

In this chapter we 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 we provided informal guidelines for a good design. These guidelines are based on doing a careful conceptual design in the ER and EER model, following the mapping procedure in Chapter 9 to map entities and relationships into relations. Proper enforcement of these guidelines and lack of redundancy will avoid the insertion/deletion/update anomalies and generation of spurious data. We recommended limiting NULL values, which cause problems during SELECT, JOIN, and aggregation operations. Then we 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 defined the concept of functional dependency, which is the basic tool for analyzing relational schemas, and we discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. Next we 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, and we 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.

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. We presented a test for the nonadditive join property of binary decompositions and also gave a general algorithm to convert any relation not in BCNF into a set of BCNF relations. We motivated the need for an additional constraint beyond the functional dependencies based on mixing of independent multivalued attributes into a single relation. We introduced multivalued dependency (MVD) to address such conditions and defined the fourth normal form based on MVDs. Finally, we introduced the fifth normal form, which is based on join dependency and which identifies a peculiar constraint that causes a relation to be decomposed into several components so that they always yield the original relation after a join. In practice, most commercial designs have followed the normal forms up to BCNF. The need to decompose into 5NF rarely arises in practice, and join dependencies are difficult to detect for most practical situations, making 5NF more of theoretical value.

Chapter 15 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *nonadditive* (or *lossless*) *join* and *dependency preservation,* which are enforced by some of these algorithms. Other topics in Chapter 15 include a more detailed treatment of functional and multivalued dependencies, and other types of dependencies.

Review Questions

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

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

14.3. Why should NULLs in a relation be avoided as much as possible? Discuss the problem of spurious tuples and how we may prevent it.

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

14.5. 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?

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