FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

part 6
Database Design Theory and Normalization

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.

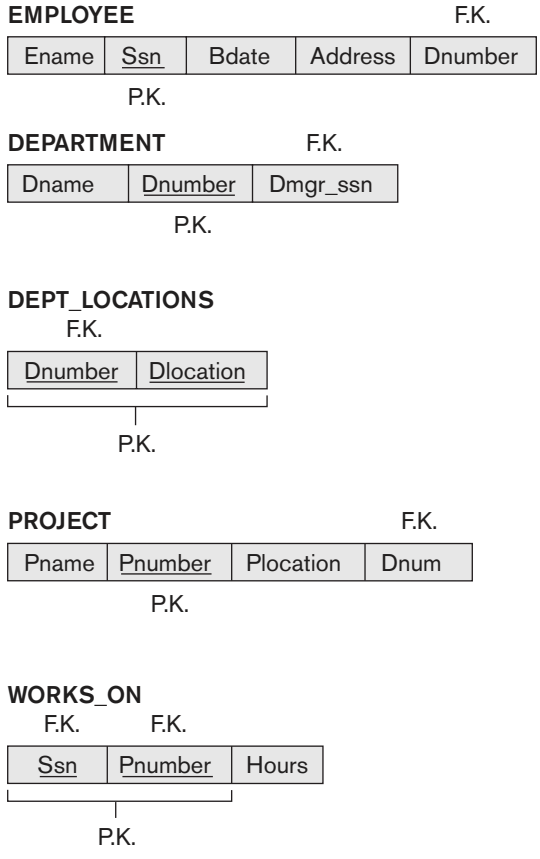


Figure 14.1
A simplified COMPANY relational database schema.

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

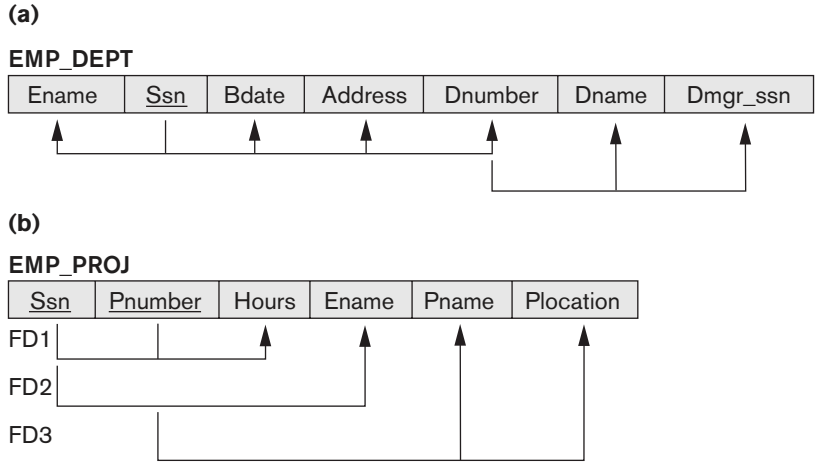


Figure 14.3
Two relation schemas suffering from update anomalies.
(a) EMP_DEPT and
(b) EMP_PROJ.

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

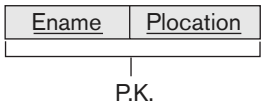
Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 14.5(a), which can be used instead of the single EMP_PROJ relation in Figure 14.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on *at least one* project located at Plocation. A tuple in EMP_PROJ1 refers to 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.

(a)

EMP_LOCS



EMP_PROJ1

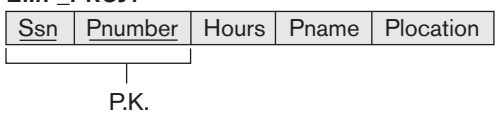


Figure 14.5
Particularly poor design for the EMP_PROJ relation in Figure 14.3(b). (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 14.4 onto the relations EMP_LOCS and EMP_PROJ1.

(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} $\rightarrow$ 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

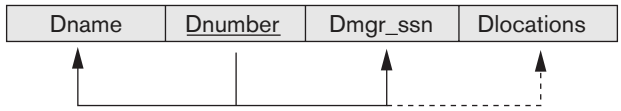
First normal form (1NF)is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (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 $A \in X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 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.

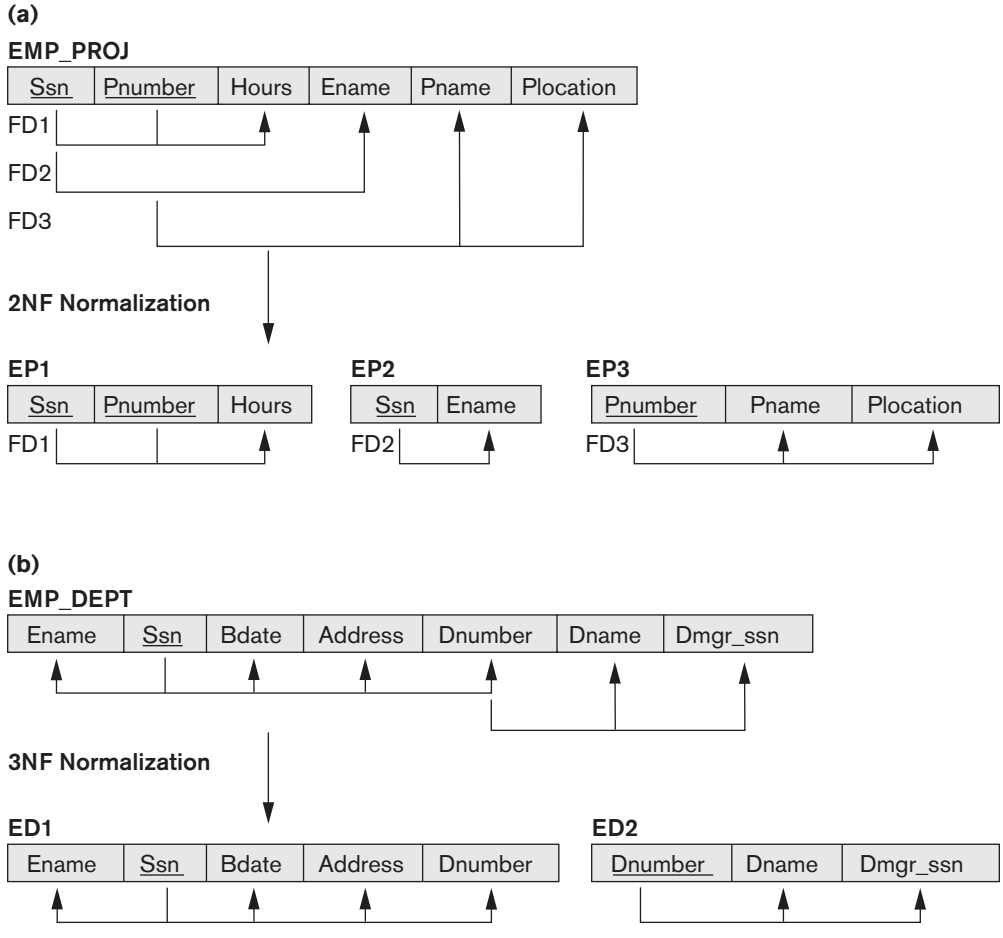


Figure 14.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

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 $\rightarrow$ Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 14.3(a), because both the dependencies Ssn $\rightarrow$ Dnumber and Dnumber $\rightarrow$ 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.

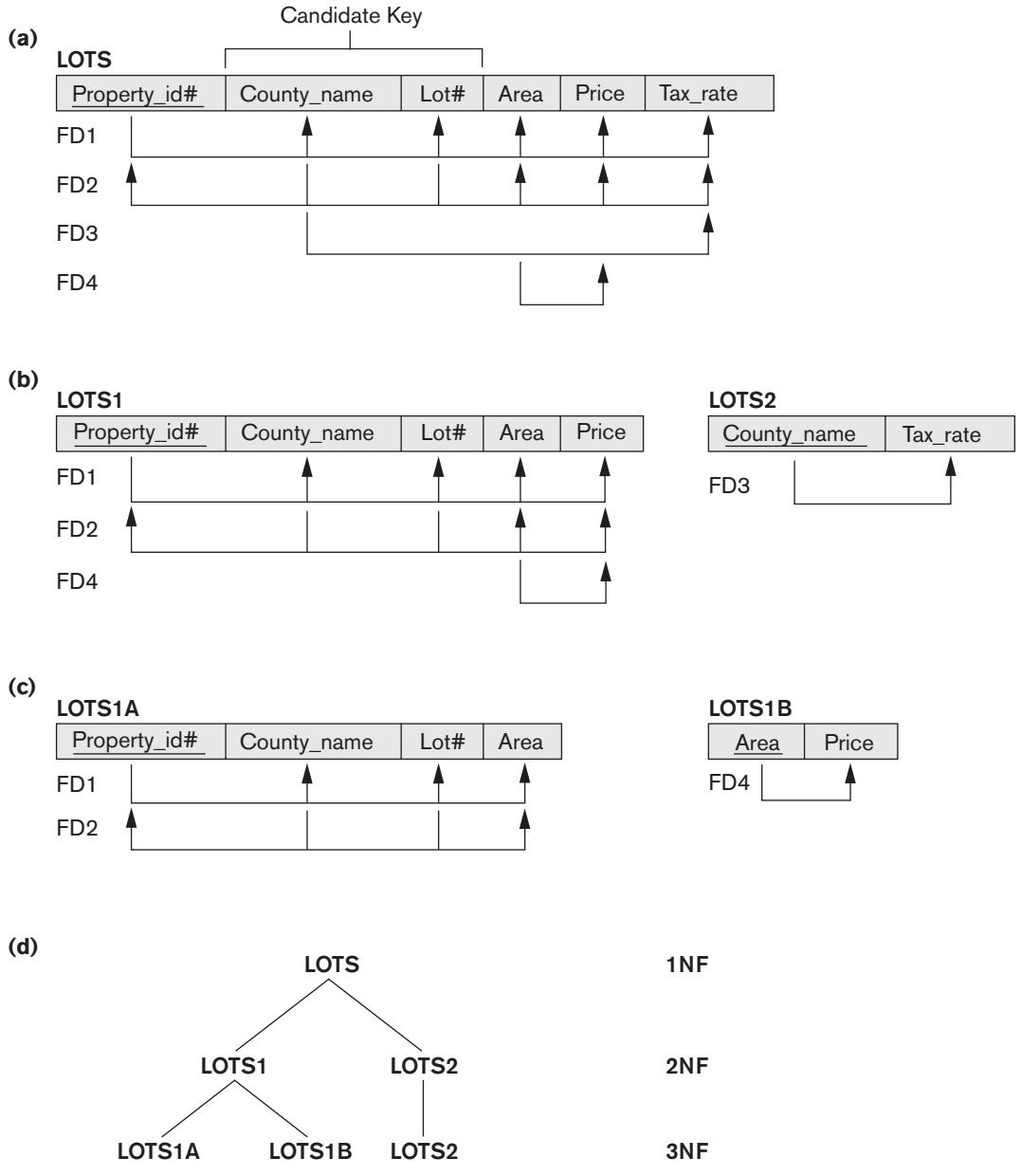
14.4.1 General Definition of Second Normal Form

> **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

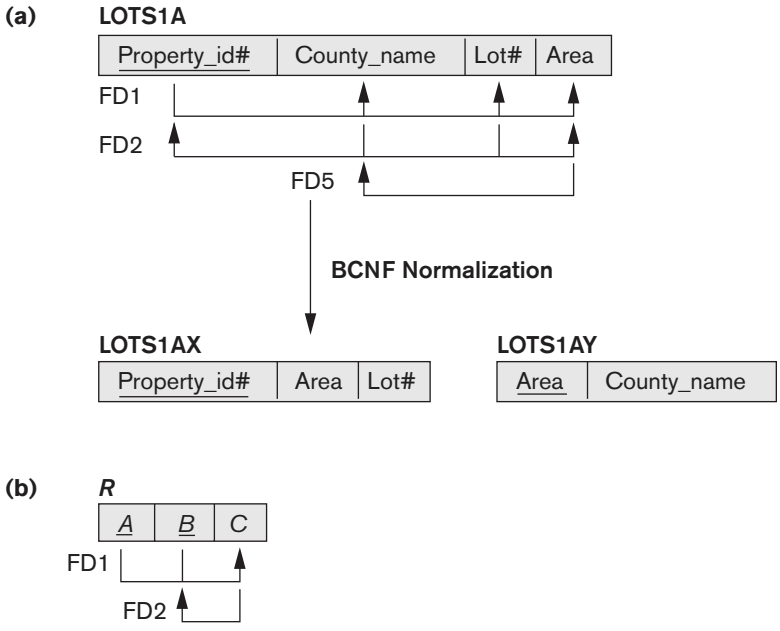


Figure 14.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. $C \rightarrow B$.

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 $\rightarrow$ 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) $X \cup Y = R$. 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, … , R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, … , R_n. 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, … , R_n), specified on relation schema R, is a **trivial** JD if one of the relation schemas R_i in JD(R_1, R_2, … , R_n) is equal to R. Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and 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, … , 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?

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

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

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

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

14.11. In what way do the generalized definitions of 2NF and 3NF extend the definitions beyond primary keys?

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

14.13. What is multivalued dependency? When does it arise?

14.14. Does a relation with two or more columns always have an MVD? Show with an example.

14.15. Define *fourth normal form*. When is it violated? When is it typically applicable?

14.16. Define *join dependency* and *fifth normal form*.

14.17. Why is 5NF also called project-join normal form (PJNF)?

14.18. Why do practical database designs typically aim for BCNF and not aim for higher normal forms?

Exercises

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

a. The university keeps track of each student's name (Sname), student number (Snum), Social Security number (Ssn), current address (Sc_addr) and phone (Sc_phone), permanent address (Sp_addr) and phone (Sp_phone), birth date (Bdate), sex (Sex), class (Class) ('freshman', 'sophomore', … , 'graduate'), major department (Major_code), minor department (Minor_code) (if any), and degree program (Prog) ('b.a.', 'b.s.', … , 'ph.d.'). Both Ssn and student number have unique values for each student.

b. Each department is described by a name (Dname), department code (Dcode), office number (Doffice), office phone (Dphone), and college (Dcollege). Both name and code have unique values for each department.

c. Each course has a course name (Cname), description (Cdesc), course number (Cnum), number of semester hours (Credit), level (Level), and offering department (Cdept). The course number is unique for each course.

d. Each section has an instructor (Iname), semester (Semester), year (Year), course (Sec_course), and section number (Sec_num). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, … , up to the total number of sections taught during each semester.

e. A grade record refers to a student (Ssn), a particular section, and a grade (Grade).

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

14.20. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 14.3 and 14.4?

14.21. In what normal form is the LOTS relation schema in Figure 14.12(a) with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

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

14.23. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations in Figure 14.5 (result shown in Figure 14.6)?

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

14.25. Repeat Exercise 14.24 for the following different set of functional dependencies $G = \{\{A, B\}\rightarrow\{C\}, \{B, D\}\rightarrow\{E, F\}, \{A, D\}\rightarrow\{G, H\}, \{A\}\rightarrow\{I\}, \{H\}\rightarrow\{J\}\}$.

14.26. Consider the following relation:

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

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

i. $A \rightarrow B$, ii. $B \rightarrow C$, iii. $C \rightarrow B$, iv. $B \rightarrow A$, v. $C \rightarrow A$

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

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

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

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

14.28. Consider the relation R, which has attributes that hold schedules of courses and sections at a university; R = {Course_no, Sec_no, Offering_dept, Credit_hours, Course_level, Instructor_ssn, Semester, Year, Days_hours, Room_no, No_of_students}. Suppose that the following functional dependencies hold on R:

{Course_no} → {Offering_dept, Credit_hours, Course_level}
{Course_no, Sec_no, Semester, Year} → {Days_hours, Room_no, No_of_students, Instructor_ssn}
{Room_no, Days_hours, Semester, Year} → {Instructor_ssn, Course_no, Sec_no}

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

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

ORDER (O#, Odate, Cust#, Total_amount)
ORDER_ITEM(O#, I#, Qty_ordered, Total_price, Discount%)

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

14.30. Consider the following relation:

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

Assume that a car may be sold by multiple salespeople, and hence {Car#, Salesperson#} is the primary key. Additional dependencies are

Date_sold → Discount_amt and
Salesperson# → Commission%

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

14.31. Consider the following relation for published books:

BOOK (Book_title, Author_name, Book_type, List_price, Author_affil, Publisher)

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

Book_title → Publisher, Book_type
Book_type → List_price
Author_name → Author_affil

a. What normal form is the relation in? Explain your answer.

b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

14.32. This exercise asks you to convert business statements into dependencies. Consider the relation DISK_DRIVE (Serial_number, Manufacturer, Model, Batch, Capacity, Retailer). Each tuple in the relation DISK_DRIVE contains information about a disk drive with a unique Serial_number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain storage capacity and is sold by a certain retailer. For example, the tuple Disk_drive ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') specifies that WesternDigital made a disk drive with serial number 1978619 and model number A2235X, released in batch 765234; it is 500GB and sold by CompUSA.

Write each of the following dependencies as an FD:

a. The manufacturer and serial number uniquely identifies the drive.

b. A model number is registered by a manufacturer and therefore can't be used by another manufacturer.

c. All disk drives in a particular batch are the same model.

d. All disk drives of a certain model of a particular manufacturer have exactly the same capacity.

14.33. Consider the following relation:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In the above relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

14.34. Consider the following relation:

CAR_SALE (Car_id, Option_type, Option_listprice, Sale_date, Option_discountedprice)

This relation refers to options installed in cars (e.g., cruise control) that were sold at a dealership, and the list and discounted prices of the options.

If CarID → Sale_date and Option_type → Option_listprice and CarID, Option_type → Option_discountedprice, argue using the generalized definition of the 3NF

that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not even in 2NF.

14.35. Consider the relation:

BOOK (Book_Name, Author, Edition, Year)

with the data:

Book_Name	Author	Edition	Copyright_Year
DB_fundamentals	Navathe	4	2004
DB_fundamentals	Elmasri	4	2004
DB_fundamentals	Elmasri	5	2007
DB_fundamentals	Navathe	5	2007

a. Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?

b. Justify that this relation has the MVD {Book} $\twoheadrightarrow$ {Author} | {Edition, Year}.

c. What would be the decomposition of this relation based on the above MVD? Evaluate each resulting relation for the highest normal form it possesses.

14.36. Consider the following relation:

TRIP (Trip_id, Start_date, Cities_visited, Cards_used)

This relation refers to business trips made by company salespeople. Suppose the TRIP has a single Start_date but involves many Cities and salespeople may use multiple credit cards on the trip. Make up a mock-up population of the table.

a. Discuss what FDs and/or MVDs exist in this relation.

b. Show how you will go about normalizing the relation.

Laboratory Exercises

Note: The following exercises use the DBD (Data Base Designer) system that is described in the laboratory manual.

The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for this problem are coded as:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

14.37. Using the DBD system, verify your answers to the following exercises:

a. 14.24 (3NF only)

b. 14.25

c. 14.27

d. 14.28

Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies. Date and Fagin (1992) give some simple and practical results related to higher normal forms.

Additional references to relational design theory are given in Chapter 15.

chapter 15

Relational Database Design Algorithms and Further Dependencies

Chapter 14 presented a **top-down relational design** technique and related concepts used extensively in commercial database design projects today. The procedure involves designing an ER or EER conceptual schema and then mapping it to the relational model by a procedure such as the one described in Chapter 9. Primary keys are assigned to each relation based on known functional dependencies. In the subsequent process, which may be called **relational design by analysis**, initially designed relations from the above procedure—or those inherited from previous files, forms, and other sources—are analyzed to detect undesirable functional dependencies. These dependencies are removed by the successive normalization procedure that we described in Section 14.3 along with definitions of related normal forms, which are successively better states of design of individual relations. In Section 14.3 we assumed that primary keys were assigned to individual relations; in Section 14.4 a more general treatment of normalization was presented where all candidate keys are considered for each relation, and Section 14.5 discussed a further normal form called BCNF. Then in Sections 14.6 and 14.7 we discussed two more types of dependencies—multivalued dependencies and join dependencies—that can also cause redundancies and showed how they can be eliminated with further normalization.

In this chapter, we use the theory of normal forms and functional, multivalued, and join dependencies developed in the last chapter and build upon it while maintaining three different thrusts. First, we discuss the concept of inferring new functional dependencies from a given set and discuss notions including closure, cover, minimal cover, and equivalence. Conceptually, we need to capture the semantics of

attibutes within a relation completely and succinctly, and the minimal cover allows us to do it. Second, we discuss the desirable properties of nonadditive (lossless) joins and preservation of functional dependencies. A general algorithm to test for nonadditivity of joins among a set of relations is presented. Third, we present an approach to **relational design by synthesis** of functional dependencies. This is a **bottom-up approach to design** that presupposes that the known functional dependencies among sets of attributes in the Universe of Discourse (UoD) have been given as input. We present algorithms to achieve the desirable normal forms, namely 3NF and BCNF, and achieve one or both of the desirable properties of nonadditivity of joins and functional dependency preservation. Although the synthesis approach is theoretically appealing as a formal approach, it has not been used in practice for large database design projects because of the difficulty of providing all possible functional dependencies up front before the design can be attempted. Alternately, with the approach presented in Chapter 14, successive decompositions and ongoing refinements to design become more manageable and may evolve over time. The final goal of this chapter is to discuss further the multivalued dependency (MVD) concept we introduced in Chapter 14 and briefly point out other types of dependencies that have been identified.

In Section 15.1 we discuss the rules of inference for functional dependencies and use them to define the concepts of a cover, equivalence, and minimal cover among functional dependencies. In Section 15.2, first we describe the two desirable **properties of decompositions**, namely, the dependency preservation property and the nonadditive (or lossless) join property, which are both used by the design algorithms to achieve desirable decompositions. It is important to note that it is *insufficient* to test the relation schemas *independently of one another* for compliance with higher normal forms like 2NF, 3NF, and BCNF. The resulting relations must collectively satisfy these two additional properties to qualify as a good design. Section 15.3 is devoted to the development of relational design algorithms that start off with one giant relation schema called the **universal relation**, which is a hypothetical relation containing all the attributes. This relation is decomposed (or in other words, the given functional dependencies are synthesized) into relations that satisfy a certain normal form like 3NF or BCNF and also meet one or both of the desirable properties.

In Section 15.5 we discuss the multivalued dependency (MVD) concept further by applying the notions of inference, and equivalence to MVDs. Finally, in Section 15.6 we complete the discussion on dependencies among data by introducing inclusion dependencies and template dependencies. Inclusion dependencies can represent referential integrity constraints and class/subclass constraints across relations. We also describe some situations where a procedure or function is needed to state and verify a functional dependency among attributes. Then we briefly discuss domain-key normal form (DKNF), which is considered the most general normal form. Section 15.7 summarizes this chapter.

It is possible to skip some or all of Sections 15.3, 15.4, and 15.5 in an introductory database course.

15.1 Further Topics in Functional Dependencies: Inference Rules, Equivalence, and Minimal Cover

We introduced the concept of functional dependencies (FDs) in Section 14.2, illustrated it with some examples, and developed a notation to denote multiple FDs over a single relation. We identified and discussed problematic functional dependencies in Sections 14.3 and 14.4 and showed how they can be eliminated by a proper decomposition of a relation. This process was described as *normalization*, and we showed how to achieve the first through third normal forms (1NF through 3NF) given primary keys in Section 14.3. In Sections 14.4 and 14.5 we provided generalized tests for 2NF, 3NF, and BCNF given any number of candidate keys in a relation and showed how to achieve them. Now we return to the study of functional dependencies and show how new dependencies can be inferred from a given set and discuss the concepts of closure, equivalence, and minimal cover that we will need when we later consider a synthesis approach to design of relations given a set of FDs.

15.1.1 Inference Rules for Functional Dependencies

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

> **Definition:** An FD $X \rightarrow Y$ is **inferred from** or **implied by** a set of dependencies F specified on R if $X \rightarrow Y$ holds in *every* legal relation state r of R; that is, whenever r satisfies all the dependencies in F, $X \rightarrow Y$ also holds in r.

In real life, it is impossible to specify all possible functional dependencies for a given situation. For example, if each department has one manager, so that Dept_no uniquely determines Mgr_ssn (Dept_no $\rightarrow$ Mgr_ssn), and a manager has a unique phone number called Mgr_phone (Mgr_ssn $\rightarrow$ Mgr_phone), then these two dependencies together imply that Dept_no $\rightarrow$ Mgr_phone. This is an inferred or implied FD and need *not* be explicitly stated in addition to the two given FDs. Therefore, it is useful to define a concept called *closure* formally that includes all possible dependencies that can be inferred from the given set F.

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

For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema in Figure 14.3(a):

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

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

Ssn → {Dname, Dmgr_ssn}
Ssn → Ssn
Dnumber → Dname

The closure F^+ of F is the set of all functional dependencies that can be inferred from F. To determine a systematic way to infer dependencies, we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies. We consider some of these inference rules next. We use the notation $F \models X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies F.

In the following discussion, we use an abbreviated notation when discussing functional dependencies. We concatenate attribute variables and drop the commas for convenience. Hence, the FD $\{X,Y\} \rightarrow Z$ is abbreviated to $XY \rightarrow Z$, and the FD $\{X, Y, Z\} \rightarrow \{U, V\}$ is abbreviated to $XYZ \rightarrow UV$. We present below three rules IR1 through IR3 that are well-known inference rules for functional dependencies. They were proposed first by Armstrong (1974) and hence are known as **Armstrong's axioms**.[1]

IR1 (reflexive rule)[2]: If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule)[3]: $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

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

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

The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious. Because IR1 generates dependencies that are always true, such dependencies are called *trivial*. Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**. The augmentation rule (IR2) says that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency. According to IR3, functional dependencies are transitive.

---

[1]They are actually inference rules rather than axioms. In the strict mathematical sense, the *axioms* (given facts) are the functional dependencies in F, since we assume that they are correct, whereas IR1 through IR3 are the *inference rules* for inferring new functional dependencies (new facts).

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

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

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

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

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

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

There are three other inference rules that follow from IR1, IR2 and IR3. They are as follows:

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

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

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

The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ into the set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$. The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$ into the single FD $X \rightarrow \{A_1, A_2, \ldots, A_n\}$. The pseudotransitive rule (IR6) allows us to replace a set of attributes Y on the left-hand side of a dependency with another set X that functionally determines Y, and can be derived from IR2 and IR3 if we augment the first functional dependency $X \rightarrow Y$ with W (the augmentation rule) and then apply the transitive rule.

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

Using similar proof arguments, we can prove the inference rules IR4 to IR6 and any additional valid inference rules. However, a simpler way to prove that an inference rule for functional dependencies is valid is to prove it by using inference rules that have already been shown to be valid. Thus IR4, IR5, and IR6 are regarded as a corollary of the Armstrong's basic inference rules. For example, we can prove IR4 through IR6 by using *IR1 through IR3*. We present the proof of IR5 below. Proofs of IR4 and IR6 using IR1 through IR3 are left as an exercise for the reader.

Proof of IR5 (using IR1 through IR3).

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

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

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

Algorithm 15.1 can be used to calculate X^+.

Algorithm 15.1. Determining X^+, the Closure of X under F

Input: A set F of FDs on a relation schema R, and a set of attributes X, which is a subset of R.

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

Algorithm 15.1 starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the *repeat* loop) until no more attributes are added to X^+ *during a complete cycle* (of the *for* loop) through the dependencies in F. The closure concept is useful in understanding the meaning and implications of attributes or sets of attributes in a relation. For example, consider the following relation schema about classes held at a university in a given academic year.

CLASS (Classid, Course#, Instr_name, Credit_hrs, Text, Publisher,
Classroom, Capacity).
Let F, the set of functional dependencies for the above relation include the following f.d.s:
FD1: Sectionid → Course#, Instr_name, Credit_hrs, Text, Publisher,
Classroom, Capacity;

FD2: Course# → Credit_hrs;
FD3: {Course#, Instr_name} → Text, Classroom;
FD4: Text → Publisher
FD5: Classroom → Capacity

Note that the above FDs express certain semantics about the data in the relation CLASS. For example, FD1 states that each class has a unique Classid. FD3 states that when a given course is offered by a certain instructor, the text is fixed and the instructor teaches that class in a fixed room. Using the inference rules about the FDs and applying the definition of closure, we can define the following closures:

{ Classid } $^{+}$ = { Classid , Course#, Instr_name, Credit_hrs, Text, Publisher, Classroom, Capacity } = CLASS
{ Course#} $^{+}$ = { Course#, Credit_hrs}
{ Course#, Instr_name } $^{+}$ = { Course#, Credit_hrs, Text, Publisher, Classroom, Capacity }

Note that each closure above has an interpretation that is revealing about the attribute(s) on the left-hand side. For example, the closure of Course# has only Credit_hrs besides itself. It does not include Instr_name because different instructors could teach the same course; it does not include Text because different instructors may use different texts for the same course. Note also that the closure of {Course#, Instr_nam} does not include Classid, which implies that it is not a candidate key. This further implies that a course with given Course# could be offered by different instructors, which would make the courses distinct classes.

15.1.2 Equivalence of Sets of Functional Dependencies

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

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

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

We can determine whether F covers E by calculating X^{+} *with respect to* F for each FD $X \rightarrow Y$ *in* E, and then checking whether this X^{+} includes the attributes in Y. If this is the case for *every* FD in E, then F covers E. We determine whether E and F are equivalent by checking that E covers F and F covers E. It is left to the reader as an exercise to show that the following two sets of FDs are equivalent:

$F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$
and $G = \{A \rightarrow CD, E \rightarrow AH\}$

15.1.3 Minimal Sets of Functional Dependencies

Just as we applied inference rules to expand on a set F of FDs to arrive at F^+, its closure, it is possible to think in the opposite direction to see if we could shrink or reduce the set F to its *minimal form* so that the minimal set is still equivalent to the original set F. Informally, a **minimal cover** of a set of functional dependencies E is a set of functional dependencies F that satisfies the property that every dependency in E is in the closure F^+ of F. In addition, this property is lost if any dependency from the set F is removed; F must have no redundancies in it, and the dependencies in F are in a standard form.

We will use the concept of an extraneous attribute in a functional dependency for defining the minimum cover.

> **Definition:** An attribute in a functional dependency is considered an **extraneous attribute** if we can remove it without changing the closure of the set of dependencies. Formally, given F, the set of functional dependencies, and a functional dependency $X \rightarrow A$ in F, attribute Y is extraneous in X if $Y \subset X$, and F logically implies $(F - (X \rightarrow A) \cup \{ (X - Y) \rightarrow A \})$.

We can formally define a set of functional dependencies F to be **minimal** if it satisfies the following conditions:

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

We can think of a minimal set of dependencies as being a set of dependencies in a *standard* or *canonical form* and with *no redundancies.* Condition 1 just represents every dependency in a canonical form with a single attribute on the right-hand side, and it is a preparatory step before we can evaluate if conditions 2 and 3 are met.[4] Conditions 2 and 3 ensure that there are no redundancies in the dependencies either by having redundant attributes (referred to as extraneous attributes) on the left-hand side of a dependency (Condition 2) or by having a dependency that can be inferred from the remaining FDs in F (Condition 3).

> **Definition.** A **minimal cover** of a set of functional dependencies E is a minimal set of dependencies (in the standard canonical form[5] and without redundancy) that is equivalent to E. We can always find *at least one* minimal cover F for any set of dependencies E using Algorithm 15.2.

---

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

[5]It is possible to use the inference rule IR5 and combine the FDs with the same left-hand side into a single FD in the minimum cover in a nonstandard form. The resulting set is still a minimum cover, as illustrated in the example.

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

Algorithm 15.2. Finding a Minimal Cover F for a Set of Functional Dependencies E

Input: A set of functional dependencies E.

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (⋆*comment*⋆).

1. Set $F := E$.
2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$. (⋆This places the FDs in a canonical form for subsequent testing⋆)
3. For each functional dependency $X \rightarrow A$ in F
   for each attribute B that is an element of X
   if $\{ \{F - \{X \rightarrow A\}\} \cup \{ (X - \{B\}) \rightarrow A\} \}$ is equivalent to F
   then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F.
   (⋆This constitutes removal of an extraneous attribute B contained in the left-hand side X of a functional dependency $X \rightarrow A$ when possible⋆)
4. For each remaining functional dependency $X \rightarrow A$ in F
   if $\{F - \{X \rightarrow A\}\}$ is equivalent to F,
   then remove $X \rightarrow A$ from F. (⋆This constitutes removal of a redundant functional dependency $X \rightarrow A$ from F when possible⋆)

We illustrate the above algorithm with the following examples:

Example 1: Let the given set of FDs be E: $\{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$. We have to find the minimal cover of E.

- All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm 15.2 and can proceed to step 2. In step 2 we need to determine if $AB \rightarrow D$ has any redundant (extraneous) attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?
- Since B → A, by augmenting with B on both sides (IR2), we have $BB \rightarrow AB$, or $B \rightarrow AB$ (i). However, $AB \rightarrow D$ as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), $B \rightarrow D$. Thus $AB \rightarrow D$ may be replaced by $B \rightarrow D$.
- We now have a set equivalent to original E, say E': $\{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.

- In step 3 we look for a redundant FD in E'. By using the transitive rule on $B \rightarrow D$ and $D \rightarrow A$, we derive $B \rightarrow A$. Hence $B \rightarrow A$ is redundant in E' and can be eliminated.
- Therefore, the minimal cover of E is F: $\{B \rightarrow D, D \rightarrow A\}$.

The reader can verify that the original set F can be inferred from E; in other words, the two sets F and E are equivalent.

Example 2: Let the given set of FDs be G: $\{A \rightarrow BCDE, CD \rightarrow E\}$.

- Here, the given FDs are NOT in the canonical form. So we first convert them into:

$$E: \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, CD \rightarrow E\}.$$

- In step 2 of the algorithm, for $CD \rightarrow E$, neither C nor D is extraneous on the left-hand side, since we cannot show that $C \rightarrow E$ or $D \rightarrow E$ from the given FDs. Hence we cannot replace it with either.
- In step 3, we want to see if any FD is redundant. Since $A \rightarrow CD$ and $CD \rightarrow E$, by transitive rule (IR3), we get $A \rightarrow E$. Thus, $A \rightarrow E$ is redundant in G.
- So we are left with the set F, equivalent to the original set G as: $\{A \rightarrow B, A \rightarrow C, A \rightarrow D, CD \rightarrow E\}$. F is the minimum cover. As we pointed out in footnote 6, we can combine the first three FDs using the union rule (IR5) and express the minimum cover as:

$$\text{Minimum cover of } G, F: \{A \rightarrow BCD, CD \rightarrow E\}.$$

In Section 15.3, we will show algorithms that synthesize 3NF or BCNF relations from a given set of dependencies E by first finding the minimal cover F for E.

Next, we provide a simple algorithm to determine the key of a relation:

Algorithm 15.2(a). Finding a Key K for R Given a Set F of Functional Dependencies

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

1. Set $K := R$.
2. For each attribute A in K

   {compute $(K - A)^+$ with respect to F;

   if $(K - A)^+$ contains all the attributes in R, then set $K := K - \{A\}$ };

In Algorithm 15.2(a), we start by setting K to all the attributes of R; we can say that R itself is always a **default superkey**. We then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice, too, that Algorithm 15.2(a) determines only *one key* out of the possible candidate keys for R; the key returned depends on the order in which attributes are removed from R in step 2.

15.2 Properties of Relational Decompositions

We now turn our attention to the process of decomposition that we used throughout Chapter 14 to get rid of unwanted dependencies and achieve higher normal forms. In Section 15.2.1, we give examples to show that looking at an *individual* relation to test whether it is in a higher normal form does not, on its own, guarantee a good design; rather, a *set of relations* that together form the relational database schema must possess certain additional properties to ensure a good design. In Sections 15.2.2 and 15.2.3, we discuss two of these properties: the dependency preservation property and the nonadditive (or lossless) join property. Section 15.2.4 discusses binary decompositions, and Section 15.2.5 discusses successive nonadditive join decompositions.

15.2.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 15.3 start from a single **universal relation schema** $R = \{A_1, A_2, \ldots, A_n\}$ that includes *all* the attributes of the database. We implicitly make the **universal relation assumption**, which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, \ldots, R_m\}$ that will become the relational database schema; D is called a **decomposition** of R.

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

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

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

Another goal is to have each individual relation R_i in the decomposition D be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations. To illustrate this point, consider the EMP_LOCS(Ename, Plocation) relation in Figure 14.5, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF.[6] Although EMP_LOCS is in BCNF, it still gives rise to spurious tuples when joined with EMP_PROJ (Ssn, Pnumber, Hours, Pname, Plocation), which is not in BCNF (see the partial result of the natural join in Figure 14.6). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted

[6]As an exercise, the reader should prove that this statement is true.

semantics by which Plocation gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(Pname, Pnumber, Plocation, Dnum) in Figure 14.2—which *is* in BCNF—using Plocation as a joining attribute also gives rise to spurious tuples. This underscores the need for other criteria that, together with the conditions of 3NF or BCNF, prevent such bad designs. In the next three subsections we discuss such additional conditions that should hold on a decomposition D as a whole.

15.2.2 Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i. Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

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

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

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

An example of a decomposition that does not preserve dependencies is shown in Figure 14.13(a), in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 14.12, however, are dependency-preserving. Similarly, for the example in Figure 14.14, no

matter what decomposition is chosen for the relation TEACH(Student, Course, Instructor) from the three provided in the text, one or both of the dependencies originally present are bound to be lost. We now state a claim related to this property without providing any proof.

Claim 1. It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation R_i in D is in 3NF.

15.2.3 Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition. We already illustrated this problem in Section 14.1.4 with the example in Figures 14.5 and 14.6. Because this is a property of a decomposition of relation *schemas,* the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in F. Hence, the lossless join property is always defined with respect to a specific set F of dependencies.

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

The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN ($*$) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately. Although the term *lossless join* has been popular in the literature, *we* used the term *nonadditive join* in describing the NJB property in Section 14.5.1. We *will henceforth use the term nonadditive join*, which is self-explanatory and unambiguous. The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information. The decomposition of EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation) in Figure 14.3 into EMP_LOCS(Ename, Plocation) and EMP_PROJ1(Ssn, Pnumber, Hours, Pname, Plocation) in Figure 14.5 obviously does not have the nonadditive join property, as illustrated by the partial result of NATURAL JOIN in Figure 14.6. We provided a simpler test in case of binary decompositions to check if the decomposition is nonadditive—it was called the NJB property in Section 14.5.1. We provide a general procedure for testing whether any decomposition D of a relation into n relations is nonadditive with respect to a set of given functional dependencies F in the relation; it is presented as Algorithm 15.3.

Algorithm 15.3. Testing for Nonadditive Join Property

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

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (⋆*comment*⋆).

1. Create an initial matrix S with one row i for each relation R_i in D, and one column j for each attribute A_j in R.
2. Set $S(i, j) := b_{ij}$ for all matrix entries. (⋆Each b_{ij} is a distinct symbol associated with indices (i, j)⋆)
3. For each row i representing relation schema R_i
   {for each column j representing attribute A_j
   {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};}; (⋆Each a_j is a distinct symbol associated with index (j)⋆)
4. Repeat the following loop until a *complete loop execution* results in no changes to S
   {for each functional dependency $X \rightarrow Y$ in F
   {for all rows in S *that have the same symbols* in the columns corresponding to attributes in X
   {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that *same a* symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column ;} ; } ;};
5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

Given a relation R that is decomposed into a number of relations $R_1, R_2, \ldots, R_m$, Algorithm 15.3 begins the matrix S that we consider to be some relation state r of R. Row i in S represents a tuple t_i (corresponding to relation R_i) that has a symbols in the columns that correspond to the attributes of R_i and b symbols in the remaining columns. The algorithm then transforms the rows of this matrix (during the loop in step 4) so that they represent tuples that satisfy all the functional dependencies in F. At the end of step 4, any two rows in S—which represent two tuples in r—that agree in their values for the left-hand-side attributes X of a functional dependency $X \rightarrow Y$ in F will also agree in their values for the right-hand-side attributes Y. It can be shown that after applying the loop of step 4, if any row in S ends up with all a symbols, then the decomposition D has the nonadditive join property with respect to F.

If, on the other hand, no row ends up being all a symbols, D does not satisfy the lossless join property. In this case, the relation state r represented by S at the end of

the algorithm will be an example of a relation state r of R that satisfies the dependencies in F but does not satisfy the nonadditive join condition. Thus, this relation serves as a **counterexample** that proves that D does not have the nonadditive join property with respect to F. Note that the a and b symbols have no special meaning at the end of the algorithm.

Figure 15.1(a) shows how we apply Algorithm 15.3 to the decomposition of the EMP_PROJ relation schema from Figure 14.3(b)into the two relation schemas EMP_PROJ1 and EMP_LOCS in Figure 14.5(a). The loop in step 4 of the algorithm cannot change any b symbols to a symbols; hence, the resulting matrix S does not have a row with all a symbols, and so the decomposition does not have the nonadditive join property.

Figure 15.1(b) shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the nonadditive join property, and Figure 15.1(c) shows how we apply the algorithm to that decomposition. Once a row consists only of a symbols, we conclude that the decomposition has the nonadditive join property, and we can stop applying the functional dependencies (step 4 in the algorithm) to the matrix S.

15.2.4 Testing Binary Decompositions for the Nonadditive Join Property

Algorithm 15.3 allows us to test whether a particular decomposition D into n relations obeys the nonadditive join property with respect to a set of functional dependencies F. There is a special case of a decomposition called a **binary decomposition**—decomposition of a relation R into two relations. A test called the NJB property test, which is easier to apply than Algorithm 15.3 but is *limited* only to binary decompositions, was given in Section 14.5.1. It was used to do binary decomposition of the TEACH relation, which met 3NF but did not meet BCNF, into two relations that satisfied this property.

15.2.5 Successive Nonadditive Join Decompositions

We saw the successive decomposition of relations during the process of second and third normalization in Sections 14.3 and 14.4. To verify that these decompositions are nonadditive, we need to ensure another property, as set forth in Claim 2.

> **Claim 2 (Preservation of Nonadditivity in Successive Decompositions).** If a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $D_i = \{Q_1, Q_2, \ldots, Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, \ldots, R_{i-1}, Q_1, Q_2, \ldots, Q_k, R_{i+1}, \ldots, R_m\}$ of R has the nonadditive join property with respect to F.

Figure 15.1
Nonadditive join test for *n*-ary decompositions. (a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property. (c) Case 2: Decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON satisfies test.

(a) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2\}$
R_1 = EMP_LOCS = {Ename, Plocation}
R_2 = EMP_PROJ1 = {Ssn, Pnumber, Hours, Pname, Plocation}

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

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)

EMP

Ssn	Ename

PROJECT

Pnumber	Pname	Plocation

WORKS_ON

Ssn	Pnumber	Hours

(c) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2, R_3\}$
R_1 = EMP = {Ssn, Ename}
R_2 = PROJ = {Pnumber, Pname, Plocation}
R_3 = WORKS_ON = {Ssn, Pnumber, Hours}

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

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(Original matrix S at start of algorithm)

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	~~b_{32}~~ a_2	a_3	~~b_{34}~~ a_4	~~b_{35}~~ a_5	a_6

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

15.3 Algorithms for Relational Database Schema Design

We now give two algorithms for creating a relational decomposition from a universal relation. The first algorithm decomposes a universal relation into dependency-preserving 3NF relations that also possess the nonadditive join property. The second algorithm decomposes a universal relation schema into BCNF schemas that possess the nonadditive join property. It is not possible to design an algorithm to produce BCNF relations that satisfy both dependency preservation and nonadditive join decomposition

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

By now we know that it is *not possible to have all three of the following*: (1) guaranteed nonlossy (nonadditive) design, (2) guaranteed dependency preservation, and (3) all relations in BCNF. As we have stressed repeatedly, the first condition is a must and cannot be compromised. The second condition is desirable, but not a must, and may have to be relaxed if we insist on achieving BCNF. The original lost FDs can be recovered by a JOIN operation over the results of decomposition. Now we give an algorithm where we achieve conditions 1 and 2 and only guarantee 3NF. Algorithm 15.4 yields a decomposition D of R that does the following:

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

Algorithm 15.4 Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

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

1. Find a minimal cover G for F (use Algorithm 15.2).
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R. (Algorithm 15.2(a) may be used to find a key.)
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S.[7]

[7]Note that there is an additional type of dependency: R is a projection of the join of two or more relations in the schema. This type of redundancy is considered join dependency, as we discussed in Section 15.7. Hence, technically, it may continue to exist without disturbing the 3NF status for the schema.

Step 3 of Algorithm 15.4 involves identifying a key K of R. Algorithm 15.2(a) can be used to identify a key K of R based on the set of given functional dependencies F. Notice that the set of functional dependencies used to determine a key in Algorithm 15.2(a) could be either F or G, since they are equivalent.

Example 1 of Algorithm 15.4. Consider the following universal relation:

U (Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

Emp_ssn, Esal, and Ephone refer to the Social Security number, salary, and phone number of the employee. Pno, Pname, and Plocation refer to the number, name, and location of the project. Dno is the department number.

The following dependencies are present:

FD1: Emp_ssn → {Esal, Ephone, Dno}

FD2: Pno → { Pname, Plocation}

FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

By virtue of FD3, the attribute set {Emp_ssn, Pno} represents a key of the universal relation. Hence F, the set of given FDs, includes {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation; Emp_ssn, Pno → Esal, Ephone, Dno, Pname, Plocation}.

By applying the minimal cover Algorithm 15.2, in step 3 we see that Pno is an extraneous attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is extraneous in Emp_ssn, Pno → Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows (if we group attributes with the same left-hand side into one FD):

Minimal cover G: {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation}

The second step of Algorithm 15.4 produces relations R_1 and R_2 as:

R_1 (<u>Emp_ssn</u>, Esal, Ephone, Dno)

R_2 (<u>Pno</u>, Pname, Plocation)

In step 3, we generate a relation corresponding to the key {Emp_ssn, Pno} of U. Hence, the resulting design contains:

R_1 (<u>Emp_ssn</u>, Esal, Ephone, Dno)

R_2 (<u>Pno</u>, Pname, Plocation)

R_3 (<u>Emp_ssn</u>, <u>Pno</u>)

This design achieves both the desirable properties of dependency preservation and nonadditive join.

Example 2 of Algorithm 15.4 (Case *X*). Consider the relation schema LOTS1A shown in Figure 14.13(a).

Assume that this relation is given as a universal relation U (Property_id, County, Lot#, Area) with the following functional dependencies:

FD1: Property_id → Lot#, County, Area

FD2: Lot#, County → Area, Property_id

FD3: Area → County

These were called FD1, FD2, and FD5 in Figure 14.13(a). The meanings of the above attributes and the implication of the above functional dependencies were explained in Section 14.4.For ease of reference, let us abbreviate the above attributes with the first letter for each and represent the functional dependencies as the set

F: { P → LCA, LC → AP, A → C }

The universal relation with abbreviated attributes is U (P, C, L, A). If we apply the minimal cover Algorithm 15.2 to *F*, (in step 2) we first represent the set *F* as

F: {P → L, P → C, P → A, LC → A, LC → P, A → C}

In the set *F*, P → A can be inferred from P → LC and LC → A; hence P → A by transitivity and is therefore redundant. Thus, one possible minimal cover is

Minimal cover *GX*: {P → LC, LC → AP, A → C}

In step 2 of Algorithm 15.4, we produce design *X* (before removing redundant relations) using the above minimal cover as

Design *X*: R_1 (P, L, C), R_2 (L, C, A, P), and R_3 (A, C)

In step 4 of the algorithm, we find that R_3 is subsumed by R_2 (that is, R_3 is always a projection of R_2 and R_1 is a projection of R_2 as well). Hence both of those relations are redundant. Thus the 3NF schema that achieves both of the desirable properties is (after removing redundant relations)

Design *X*: R_2 (L, C, A, P).

or, in other words it is identical to the relation LOTS1A (Property_id, Lot#, County, Area) that we had determined to be in 3NF in Section 14.4.2.

Example 2 of Algorithm 15.4 (Case *Y*). Starting with LOTS1A as the universal relation and with the same given set of functional dependencies, the second step of the minimal cover Algorithm 15.2 produces, as before,

F: {P → C, P → A, P → L, LC → A, LC → P, A → C}

The FD LC → A may be considered redundant because LC → P and P → A implies LC → A by transitivity. Also, P → C may be considered to be redundant because P → A and A → C implies P → C by transitivity. This gives a different minimal cover as

Minimal cover *GY*: { P → LA, LC → P, A → C }

The alternative design *Y* produced by the algorithm now is

Design *Y*: S_1 (P, A, L), S_2 (L, C, P), and S_3 (A, C)

Note that this design has three 3NF relations, none of which can be considered as redundant by the condition in step 4. All FDs in the original set *F* are preserved. The

reader will notice that of the above three relations, relations S_1 and S_3 were produced as the BCNF design by the procedure given in Section 14.5 (implying that S_2 is redundant in the presence of S_1 and S_3). However, we cannot eliminate relation S_2 from the set of three 3NF relations above since it is not a projection of either S_1 or S_3. It is easy to see that S_2 is a valid and meaningful relation that has the two candidate keys (L, C), and P placed side-by-side. Notice further that S_2 preserves the FD LC → P, which is lost if the final design contains only S_1 and S_3. Design Y therefore remains as one possible final result of applying Algorithm 15.4 to the given universal relation that provides relations in 3NF.

The above two variations of applying Algorithm 15.4 to the same universal relation with a given set of FDs have illustrated two things:

- It is possible to generate alternate 3NF designs by starting from the same set of FDs.
- It is conceivable that in some cases the algorithm actually produces relations that satisfy BCNF and may include relations that maintain the dependency preservation property as well.

15.3.2 Nonadditive Join Decomposition into BCNF Schemas

The next algorithm decomposes a universal relation schema $R = \{A_1, A_2, \ldots, A_n\}$ into a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ such that each R_i is in BCNF *and* the decomposition D has the lossless join property with respect to F. Algorithm 15.5 utilizes property NJB and claim 2 (preservation of nonadditivity in successive decompositions) to create a nonadditive join decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in BCNF.

Algorithm 15.5. Relational Decomposition into BCNF with Nonadditive Join Property

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

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

Each time through the loop in Algorithm 15.5, we decompose one relation schema Q that is not in BCNF into two relation schemas. According to property NJB for binary decompositions and claim 2, the decomposition D has the nonadditive join property. At the end of the algorithm, all relation schemas in D will be in

BCNF. We illustrated the application of this algorithm to the TEACH relation schema from Figure 14.14; it is decomposed into TEACH1(Instructor, Student) and TEACH2(Instructor, Course) because the dependency FD2 Instructor → Course violates BCNF.

In step 2 of Algorithm 15.5, it is necessary to determine whether a relation schema Q is in BCNF or not. One method for doing this is to test, for each functional dependency $X \rightarrow Y$ in Q, whether X^+ fails to include all the attributes in Q, thereby determining whether or not X is a (super) key in Q. Another technique is based on an observation that whenever a relation schema Q has a BCNF violation, there exists a pair of attributes A and B in Q such that $\{Q - \{A, B\}\} \rightarrow A$; by computing the closure $\{Q - \{A, B\}\}^+$ for each pair of attributes $\{A, B\}$ of Q and checking whether the closure includes A (or B), we can determine whether Q is in BCNF.

It is important to note that the theory of nonadditive join decompositions is based on the assumption that *no NULL values are allowed for the join attributes*. The next section discusses some of the problems that NULLs may cause in relational decompositions and provides a general discussion of the algorithms for relational design by synthesis presented in this section.

15.4 About Nulls, Dangling Tuples, and Alternative Relational Designs

In this section, we discuss a few general issues related to problems that arise when relational design is not approached properly.

15.4.1 Problems with NULL Values and Dangling Tuples

We must carefully consider the problems associated with NULLs when designing a relational database schema. There is no fully satisfactory relational design theory as yet that includes NULL values. One problem occurs when some tuples have NULL values for attributes that will be used to join individual relations in the decomposition. To illustrate this, consider the database shown in Figure 15.2(a), where two relations EMPLOYEE and DEPARTMENT are shown. The last two employee tuples—'Berger' and 'Benitez'—represent newly hired employees who have not yet been assigned to a department (assume that this does not violate any integrity constraints). Now suppose that we want to retrieve a list of (Ename, Dname) values for all the employees. If we apply the NATURAL JOIN operation on EMPLOYEE and DEPARTMENT (Figure 15.2(b)), the two aforementioned tuples will *not* appear in the result. The OUTER JOIN operation, discussed in Chapter 8, can deal with this problem. Recall that if we take the LEFT OUTER JOIN of EMPLOYEE with DEPARTMENT, tuples in EMPLOYEE that have NULL for the join attribute will still appear in the result, joined with an *imaginary* tuple in DEPARTMENT that has NULLs for all its attribute values. Figure 15.2(c) shows the result.

In general, whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be devoted to

watching for potential NULL values in foreign keys. This can cause unexpected loss of information in queries that involve joins on that foreign key. Moreover, if NULLs occur in other attributes, such as Salary, their effect on built-in functions such as SUM and AVERAGE must be carefully evaluated.

A related problem is that of *dangling tuples*, which may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation in Figure 15.2(a) further into EMPLOYEE_1 and EMPLOYEE_2, shown in Figures 15.3(a) and 15.3(b). If we apply the NATURAL JOIN operation to EMPLOYEE_1 and EMPLOYEE_2, we get the original EMPLOYEE relation. However, we may use the alternative representation, shown in Figure 15.3(c), where we *do not include a tuple* in EMPLOYEE_3 if the employee has not been assigned a department (instead of including a tuple with NULL for Dnum as in EMPLOYEE_2). If we use EMPLOYEE_3 instead of EMPLOYEE_2 and apply a NATURAL JOIN on EMPLOYEE_1 and EMPLOYEE_3, the tuples for Berger and Benitez will not appear in the result; these are called **dangling tuples** in EMPLOYEE_1 because they are represented in only one of the two relations that represent employees, and hence they are lost if we apply an (INNER) JOIN operation.

15.4.2 Discussion of Normalization Algorithms and Alternative Relational Designs

One of the problems with the normalization algorithms we described is that the database designer must first specify *all* the relevant functional dependencies among the database attributes. This is not a simple task for a large database with hundreds of attributes. Failure to specify one or two important dependencies may result in an undesirable design. Another problem is that these algorithms are *not deterministic* in general. For example, the *synthesis algorithms* (Algorithms 15.4 and 15.5) require the specification of a minimal cover G for the set of functional dependencies F. Because there may be, in general, many minimal covers corresponding to F, as we illustrated in Example 2 of Algorithm 15.4 above, the algorithm can give different designs depending on the particular minimal cover used. Some of these designs may not be desirable. The decomposition algorithm to achieve BCNF (Algorithm 15.5) depends on the order in which the functional dependencies are supplied to the algorithm to check for BCNF violation. Again, it is possible that many different designs may arise. Some of the designs may be preferred, whereas others may be undesirable.

It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (instead of 3NF, as in Algorithm 15.4). We can check the 3NF relation schemas in the decomposition individually to see whether each satisfies BCNF. If some relation schema R_i is not in BCNF, we can choose to decompose it further or to leave it as it is in 3NF (with some possible update anomalies). We showed by using the bottom-up approach to design that different minimal covers in cases X and Y of Example 2 under Algorithm 15.4 produced different sets of relations

(a)

EMPLOYEE

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

DEPARTMENT

Dname	Dnum	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

(b)

Ename	Ssn	Bdate	Address	Dnum	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 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

(c)

Ename	Ssn	Bdate	Address	Dnum	Dname	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 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL	NULL	NULL
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX	NULL	NULL	NULL

Figure 15.2
Issues with NULL-value joins. (a) Some EMPLOYEE tuples have NULL for the join attribute Dnum. (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

(a) EMPLOYEE_1

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

(b) EMPLOYEE_2

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	NULL
888664444	NULL

(c) EMPLOYEE_3

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Figure 15.3
The dangling tuple problem.
(a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from Figure 15.2(a) except Dnum).
(b) The relation EMPLOYEE_2 (includes Dnum attribute with NULL values).
(c) The relation EMPLOYEE_3 (includes Dnum attribute but does not include tuples for which Dnum has NULL values).

based on minimal cover. The design X produced the 3NF design as LOTS1A (Property_id, County, Lot#, Area) relation, which is in 3NF but not BCNF. Alternately, design Y produced three relations: S_1 (Property_id, Area, Lot#), S_2 (Lot#, County, Property_id), and S_3 (Area, County). If we test each of these three relations, we find that they are in BCNF. We also saw previously that if we apply Algorithm 15.5 to LOTS1Y to decompose it into BCNF relations, the resulting design contains only S_1 and S_3 as a BCNF design. In summary, the above examples of cases (called Case X and Case Y) driven by different minimum covers for the same universal schema amply illustrate that alternate designs will result by the application of the bottom-up design algorithms we presented in Section 15.3.

Table 15.1 summarizes the properties of the algorithms discussed in this chapter so far.

Table 15.1 Summary of the Algorithms Discussed in This Chapter

Algorithm	Input	Output	Properties/Purpose	Remarks
15.1	An attribute or a set of attributes X, and a set of FDs F	A set of attributes in the closure of X with respect to F	Determine all the attributes that can be functionally determined from X	The closure of a key is the entire relation
15.2	A set of functional dependencies F	The minimal cover of functional dependencies	To determine the minimal cover of a set of dependencies F	Multiple minimal covers may exist—depends on the order of selecting functional dependencies
15.2a	Relation schema R with a set of functional dependencies F	Key K of R	To find a key K (that is a subset of R)	The entire relation R is always a default superkey
15.3	A decomposition D of R and a set F of functional dependencies	Boolean result: yes or no for nonadditive join property	Testing for nonadditive join decomposition	See a simpler test NJB in Section 14.5 for binary decompositions
15.4	A relation R and a set of functional dependencies F	A set of relations in 3NF	Nonadditive join and dependency-preserving decomposition	May not achieve BCNF, but achieves *all* desirable properties and 3NF
15.5	A relation R and a set of functional dependencies F	A set of relations in BCNF	Nonadditive join decomposition	No guarantee of dependency preservation
15.6	A relation R and a set of functional and multivalued dependencies	A set of relations in 4NF	Nonadditive join decomposition	No guarantee of dependency preservation

15.5 Further Discussion of Multivalued Dependencies and 4NF

We introduced and defined the concept of multivalued dependencies and used it to define the fourth normal form in Section 14.6. In this section, we discuss MVDs to make our treatment complete by stating the rules of inference with MVDs.

15.5.1 Inference Rules for Functional and Multivalued Dependencies

As with functional dependencies (FDs), inference rules for MVDs have been developed. It is better, though, to develop a unified framework that includes both FDs and MVDs so that both types of constraints can be considered together. The

following inference rules IR1 through IR8 form a sound and complete set for inferring functional and multivalued dependencies from a given set of dependencies. Assume that all attributes are included in a *universal* relation schema $R = \{A_1, A_2, \ldots, A_n\}$ and that X, Y, Z, and W are subsets of R.

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

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

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

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

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

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

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

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

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

15.5.2 Fourth Normal Form Revisited

We restate the definition of **fourth normal form** (**4NF**) from Section 14.6:

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^+, X in F^+, X is a superkey for R.

To illustrate the importance of 4NF, Figure 15.4(a) shows the EMP relation in Figure 14.15 with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('W', 'X', 'Y', and 'Z'). There are 16 tuples in EMP in Figure 15.4(a). If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 15.4(b), we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but the update anomalies associated with multivalued dependencies would also be avoided. For example, if 'Brown' starts working on a new additional project 'P', we must insert *three* tuples in EMP—one for each dependent. If we forget to

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

(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

Ename	Dname
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

Figure 15.4
Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent.

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

The EMP relation in Figure 14.15(a) is not in 4NF because it represents two *independent* 1:N relationships—one between employees and the projects they work on and the other between employees and their dependents. We sometimes have a relationship among three entities that is a legitimate three-way relationship and not a combination of two binary relationships among three participating entities, such as the SUPPLY relation shown in Figure 14.15(c). (Consider only the tuples in Figure 14.5(c) *above* the dashed line for now.) In this case a tuple represents a supplier supplying a specific part *to a particular project,* so there are *no* nontrivial MVDs. Hence, the SUPPLY all-key relation is already in 4NF and should not be decomposed.

15.5.3 Nonadditive Join Decomposition into 4NF Relations

Whenever we decompose a relation schema R into $R_1 = (X \cup Y)$ and $R_2 = (R - Y)$ based on an MVD $X \twoheadrightarrow Y$ that holds in R, the decomposition has the nonadditive join property. It can be shown that this is a necessary and sufficient condition for decomposing a schema into two schemas that have the nonadditive join property, as given by Property NJB′ that is a further generalization of Property NJB given earlier in Section 14.5.1. Property NJB dealt with FDs only, whereas NJB′ deals with both FDs and MVDs (recall that an FD is also an MVD).

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

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

or, by symmetry, if and only if

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

We can use a slight modification of Algorithm 15.5 to develop Algorithm 15.7, which creates a nonadditive join decomposition into relation schemas that are in 4NF (rather than in BCNF). As with Algorithm 15.5, Algorithm 15.7 does *not* necessarily produce a decomposition that preserves FDs.

Algorithm 15.7. Relational Decomposition into 4NF Relations with Nonadditive Join Property

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

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

15.6 Other Dependencies and Normal Forms

15.6.1 Join Dependencies and the Fifth Normal Form

We already introduced another type of dependency called join dependency (JD) in Section 14.7. It arises when a relation is decomposable into a set of projected relations that can be joined back to yield the original relation. After defining JD, we defined the fifth normal form based on it in Section 14.7. Fifth normal form has also been known as project join normal form or PJNF (Fagin, 1979). A practical problem with this and some additional dependencies (and related normal forms such as DKNF, which is defined in Section 15.6.3) is that they are difficult to discover.

Furthermore, there are no sets of sound and complete inference rules to reason about them. In the remaining part of this section, we introduce some other types of dependencies that have been identified. Among them, the inclusion dependencies and those based on arithmetic or similar functions are used frequently.

15.6.2 Inclusion Dependencies

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

- The foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations.
- The constraint between two relations that represent a class/subclass relationship (see Chapters 4 and 9) also has no formal definition in terms of the functional, multivalued, and join dependencies.

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

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

The $\subseteq$ (subset) relationship does not necessarily have to be a proper subset. Obviously, the sets of attributes on which the inclusion dependency is specified—X of R and Y of S—must have the same number of attributes. In addition, the domains for each pair of corresponding attributes should be compatible. For example, if $X = \{A_1, A_2, \ldots, A_n\}$ and $Y = \{B_1, B_2, \ldots, B_n\}$, one possible correspondence is to have $\text{dom}(A_i)$ *compatible with* $\text{dom}(B_i)$ for $1 \leq i \leq n$. In this case, we say that A_i **corresponds to** B_i.

For example, we can specify the following inclusion dependencies on the relational schema in Figure 14.1:

DEPARTMENT.Dmgr_ssn < EMPLOYEE.Ssn

WORKS_ON.Ssn < EMPLOYEE.Ssn

EMPLOYEE.Dnumber < DEPARTMENT.Dnumber

PROJECT.Dnum < DEPARTMENT.Dnumber

WORKS_ON.Pnumber < PROJECT.Pnumber

DEPT_LOCATIONS.Dnumber < DEPARTMENT.Dnumber

All the preceding inclusion dependencies represent **referential integrity constraints**. We can also use inclusion dependencies to represent **class/subclass relationships**. For example, in the relational schema of Figure 9.6, we can specify the following inclusion dependencies:

EMPLOYEE.Ssn < PERSON.Ssn

ALUMNUS.Ssn < PERSON.Ssn

STUDENT.Ssn < PERSON.Ssn

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

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

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

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

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

15.6.3 Functional Dependencies Based on Arithmetic Functions and Procedures

Sometimes some attributes in a relation may be related via some arithmetic function or a more complicated functional relationship. As long as a unique value of Y is associated with every X, we can still consider that the FD $X \rightarrow Y$ exists. For example, in the relation

ORDER_LINE (Order#, Item#, Quantity, Unit_price, Extended_price, Discounted_price)

each tuple represents an item from an order with a particular quantity, and the price per unit for that item. In this relation, (Quantity, Unit_price) $\rightarrow$ Extended_price by the formula

Extended_price = Unit_price * Quantity

Hence, there is a unique value for Extended_price for every pair (Quantity, Unit_price), and thus it conforms to the definition of functional dependency.

Moreover, there may be a procedure that takes into account the quantity discounts, the type of item, and so on and computes a discounted price for the total quantity ordered for that item. Therefore, we can say

(Item#, Quantity, Unit_price) $\rightarrow$ Discounted_price, or
(Item#, Quantity, Extended_price) $\rightarrow$ Discounted_price

To check the above FDs, a more complex procedure COMPUTE_TOTAL_PRICE may have to be called into play. Although the above kinds of FDs are technically present in most relations, they are not given particular attention during normalization. They may be relevant during the loading of relations and during query processing because populating or retrieving the attribute on the right-hand side of the dependency requires the execution of a procedure such as the one mentioned above.

15.6.4 Domain-Key Normal Form

There is no hard-and-fast rule about defining normal forms only up to 5NF. Historically, the process of normalization and the process of discovering undesirable

dependencies were carried through 5NF, but it has been possible to define stricter normal forms that take into account additional types of dependencies and constraints. The idea behind **domain-key normal form** (**DKNF**) is to specify (theoretically, at least) the *ultimate normal form* that takes into account all possible types of dependencies and constraints. A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation. For a relation in DKNF, it becomes straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.

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

For these reasons, although the concept of DKNF is appealing and appears straightforward, it cannot be directly tested or implemented with any guarantees of consistency or non-redundancy of design. Hence it is not used much in practice.

15.7 Summary

In this chapter we presented a further set of topics related to dependencies, a discussion of decomposition, and several algorithms related to them as well as to the process of designing 3NF, BCNF, and 4NF relations from a given set of functional dependencies and multivalued dependencies. In Section 15.1 we presented inference rules for functional dependencies (FDs), the notion of closure of an attribute, the notion of closure of a set of functional dependencies, equivalence among sets of functional dependencies, and algorithms for finding the closure of an attribute (Algorithm 15.1) and the minimal cover of a set of FDs (Algorithm 15.2). We then discussed two important properties of decompositions: the nonadditive join property and the dependency-preserving property. An algorithm to test for an n-way nonadditive decomposition of a relation (Algorithm 15.3) was presented. A simpler test for checking for nonadditive binary decompositions (property NJB) has already been described in Section 14.5.1. We then discussed relational design by synthesis, based on a set of given functional dependencies. The *relational synthesis algorithm* (Algorithm 15.4) creates 3NF relations from a universal relation schema based on a given set of functional dependencies that has been specified by

the database designer. The *relational decomposition algorithms* (such as Algorithms 15.5 and 15.6) create BCNF (or 4NF) relations by successive nonadditive decomposition of unnormalized relations into two component relations at a time. We saw that it is possible to synthesize 3NF relation schemas that meet both of the above properties; however, in the case of BCNF, it is possible to aim only for the nonadditiveness of joins—dependency preservation *cannot* be necessarily guaranteed. If the designer has to aim for one of these two, the nonadditive join condition is an absolute must. In Section 15.4 we showed how certain difficulties arise in a collection of relations due to null values that may exist in relations in spite of the relations being individually in 3NF or BCNF. Sometimes when decomposition is improperly carried too far, certain "dangling tuples" may result that do not participate in results of joins and hence may become invisible. We also showed how algorithms such as 15.4 for 3NF synthesis could lead to alternative designs based on the choice of minimum cover. We revisited multivalued dependencies (MVDs) in Section 15.5. MVDs arise from an improper combination of two or more independent multivalued attributes in the same relation, and MVDs result in a combinational expansion of the tuples used to define fourth normal form (4NF). We discussed inference rules applicable to MVDs and discussed the importance of 4NF. Finally, in Section 15.6 we discussed inclusion dependencies, which are used to specify referential integrity and class/subclass constraints, and pointed out the need for arithmetic functions or more complex procedures to enforce certain functional dependency constraints. We concluded with a brief discussion of the domain-key normal form (DKNF).

Review Questions

15.1. What is the role of Armstrong's inference rules (inference rules IR1 through IR3) in the development of the theory of relational design?

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

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

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

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

15.6. What is meant by the attribute preservation condition on a decomposition?

15.7. Why are normal forms alone insufficient as a condition for a good schema design?

15.8. What is the dependency preservation property for a decomposition? Why is it important?

15.9. Why can we *not* guarantee that BCNF relation schemas will be produced by dependency-preserving decompositions of non-BCNF relation schemas? Give a counterexample to illustrate this point.

15.10. What is the lossless (or nonadditive) join property of a decomposition? Why is it important?

15.11. Between the properties of dependency preservation and losslessness, which one must definitely be satisfied? Why?

15.12. Discuss the NULL value and dangling tuple problems.

15.13. Illustrate how the process of creating first normal form relations may lead to multivalued dependencies. How should the first normalization be done properly so that MVDs are avoided?

15.14. What types of constraints are inclusion dependencies meant to represent?

15.15. How do template dependencies differ from the other types of dependencies we discussed?

15.16. Why is the domain-key normal form (DKNF) known as the ultimate normal form?

Exercises

15.17. Show that the relation schemas produced by Algorithm 15.4 are in 3NF.

15.18. Show that, if the matrix S resulting from Algorithm 15.3 does not have a row that is all a symbols, projecting S on the decomposition and joining it back will always produce at least one spurious tuple.

15.19. Show that the relation schemas produced by Algorithm 15.5 are in BCNF.

15.20. Write programs that implement Algorithms 15.4 and 15.5.

15.21. Consider the relation REFRIG(Model#, Year, Price, Manuf_plant, Color), which is abbreviated as REFRIG(M, Y, P, MP, C), and the following set F of functional dependencies: $F = \{\text{M} \rightarrow \text{MP}, \{\text{M, Y}\} \rightarrow \text{P}, \text{MP} \rightarrow \text{C}\}$

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

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

c. Consider the decomposition of REFRIG into $D = \{R_1(\text{M, Y, P}), R_2(\text{M, MP, C})\}$. Is this decomposition lossless? Show why. (You may consult the test under Property NJB in Section 14.5.1.)

15.22. Specify all the inclusion dependencies for the relational schema in Figure 5.5.

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

15.24. Consider the example of normalizing the LOTS relation in Sections 14.4 and 14.5. Determine whether the decomposition of LOTS into {LOTS1AX, LOTS1AY, LOTS1B, LOTS2} has the lossless join property by applying Algorithm 15.3 and also by using the test under property NJB from Section 14.5.1.

15.25. Show how the MVDs Ename $\twoheadrightarrow$ and Ename $\twoheadrightarrow$ Dname in Figure 14.15(a) may arise during normalization into 1NF of a relation, where the attributes Pname and Dname are multivalued.

15.26. Apply Algorithm 15.2(a) to the relation in Exercise 14.24 to determine a key for R. Create a minimal set of dependencies G that is equivalent to F, and apply the synthesis algorithm (Algorithm 15.4) to decompose R into 3NF relations.

15.27. Repeat Exercise 15.26 for the functional dependencies in Exercise 14.25.

15.28. Apply the decomposition algorithm (Algorithm 15.5) to the relation R and the set of dependencies F in Exercise 15.24. Repeat for the dependencies G in Exercise 15.25.

15.29. Apply Algorithm 15.2(a) to the relations in Exercises 14.27 and 14.28 to determine a key for R. Apply the synthesis algorithm (Algorithm 15.4) to decompose R into 3NF relations and the decomposition algorithm (Algorithm 15.5) to decompose R into BCNF relations.

15.31. Consider the following decompositions for the relation schema R of Exercise 14.24. Determine whether each decomposition has (1) the dependency preservation property, and (2) the lossless join property, with respect to F. Also determine which normal form each relation in the decomposition is in.

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

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

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

Laboratory Exercises

Note: These exercises use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for Problem 14.24 are coded as:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

15.33. Using the DBD system, verify your answers to the following exercises:

a. 15.24
b. 15.26
c. 15.27
d. 15.28
e. 15.29
f. 15.31 (a) and (b)
g. 15.32 (a) and (c)

Selected Bibliography

The books by Maier (1983) and Atzeni and De Antonellis (1993) include a comprehensive discussion of relational dependency theory. Algorithm 15.4 is based on the normalization algorithm presented in Biskup et al. (1979). The decomposition algorithm (Algorithm 15.5) is due to Bernstein (1976). Tsou and Fischer (1982) give a polynomial-time algorithm for BCNF decomposition.

The theory of dependency preservation and lossless joins is given in Ullman (1988), where proofs of some of the algorithms discussed here appear. The lossless join property is analyzed in Aho et al. (1979). Algorithms to determine the keys of a relation from functional dependencies are given in Osborn (1977); testing for BCNF is discussed in Osborn (1979). Testing for 3NF is discussed in Tsou and Fischer (1982). Algorithms for designing BCNF relations are given in Wang (1990) and Hernandez and Chan (1991).

Multivalued dependencies and fourth normal form are defined in Zaniolo (1976) and Nicolas (1978). Many of the advanced normal forms are due to Fagin: the fourth normal form in Fagin (1977), PJNF in Fagin (1979), and DKNF in Fagin (1981). The set of sound and complete rules for functional and multivalued dependencies was given by Beeri et al. (1977). Join dependencies are discussed by Rissanen (1977) and Aho et al. (1979). Inference rules for join dependencies are given by Sciore (1982). Inclusion dependencies are discussed by Casanova et al. (1981) and analyzed further in Cosmadakis et al. (1990). Their use in optimizing relational schemas is discussed in Casanova et al. (1989). Template dependencies, which are a general form of dependencies based on hypotheses and conclusion tuples, are discussed by Sadri and Ullman (1982). Other dependencies are discussed in Nicolas (1978), Furtado (1978), and Mendelzon and Maier (1979). Abiteboul et al. (1995) provides a theoretical treatment of many of the ideas presented in this chapter and Chapter 14.

This page intentionally left blank