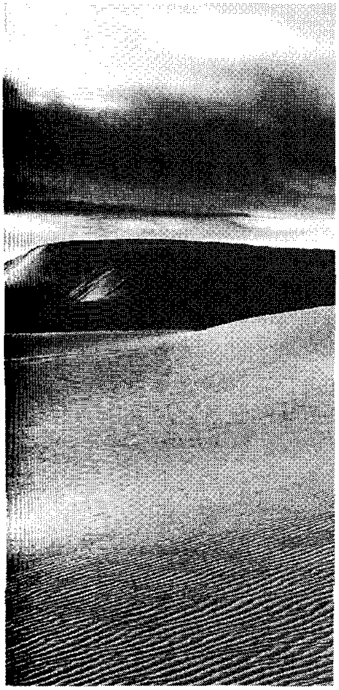


10

Functional Dependencies and Normalization for Relational Databases

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

There are two levels at which we can discuss the "goodness" of relation schemas. The first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly. The second is the **implementation** (or **storage**) **level**—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations). The relational database design theory developed in this chapter applies mainly to *base relations*, although some criteria of appropriateness also apply to views, as shown in Section 10.1.

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

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

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

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

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

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

10.1 Informal Design Guidelines for Relation Schemas

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

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

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

10.1.1 Semantics of the Relation Attributes

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

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

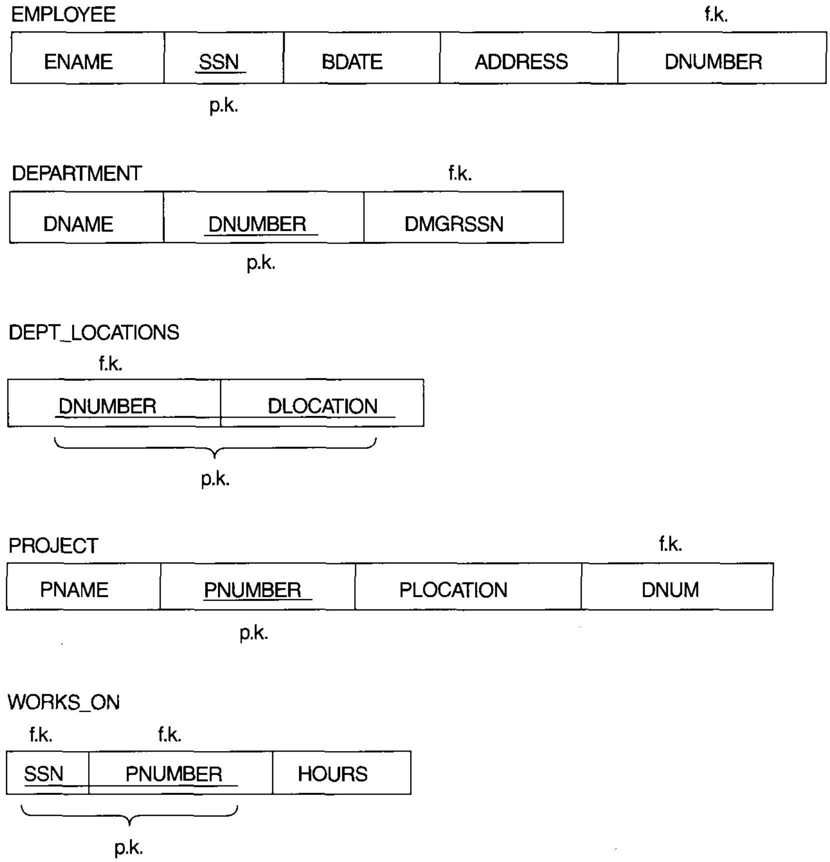


FIGURE 10.1 A simplified COMPANY relational database schema.

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

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

EMPLOYEE

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

DEPARTMENT

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

DEPT_LOCATIONS

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

WORKS_ON

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

PROJECT

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

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

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

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

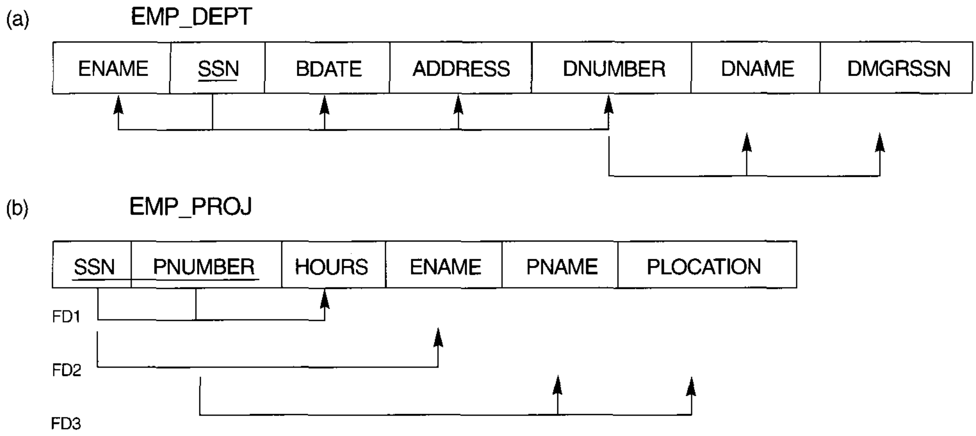


FIGURE 10.3 Two relation schemas suffering from update anomalies.

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

10.1.2 Redundant Information in Tuples and Update Anomalies

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

EMP_DEPT

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

redundancy (DNAME, DMGRSSN)

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

redundancy (HOURS, ENAME) redundancy (PNAME, PLOCATION)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.1.3 Null Values in Tuples

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

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

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

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

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

10.1.4 Generation of Spurious Tuples

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

---

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

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

(a)

EMP_LOCS

ENAME	PLOCATION

p.k.

EMP_PROJ1

SSN	PNUMBER	HOURS	PNAME	PLOCATION

p.k.

(b)

EMP_LOCS

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

EMP_PROJ1

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

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

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

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

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

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

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

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

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

10.1.5 Summary and Discussion of Design Guidelines

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

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

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

10.2 Functional Dependencies

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

10.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes $A_1, A_2, \ldots, A_n$; let us think of the whole database as being described by a single **universal** relation schema $R = \{A_1,$

$A_2, \ldots, A_n\}$.[6] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[7]

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

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

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

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

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

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

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

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

a. SSN → ENAME
b. PNUMBER → {PNAME, PLOCATION}
c. {SSN, PNUMBER} → HOURS

These functional dependencies specify that (a) the value of an employee's social security number (SSN) uniquely determines the employee name (ENAME), (b) the value of a project's number (PNUMBER) uniquely determines the project name (PNAME) and location (PLOCATION), and (c) a combination of SSN and PNUMBER values uniquely determines the number of hours the employee currently works on the project per week (HOURS). Alternatively, we say that ENAME is functionally determined by (or functionally dependent on) SSN, or "given a value of SSN, we know the value of ENAME," and so on.

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

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

10.2.2 Inference Rules for Functional Dependencies

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

TEACH

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

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

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

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

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

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

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

SSN → {DNAME, DMGRSSN}

SSN → SSN

DNUMBER → DNAME

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

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

IR1 (reflexive rule[8]): If $X \supseteq Y$, then $X \rightarrow Y$.

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

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

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

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

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

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

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

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

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

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

PROOF OF IR1

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

PROOF OF IR2 (BY CONTRADICTION)

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

PROOF OF IR3

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

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

PROOF OF IR4 (USING IR1 THROUGH IR3)

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

PROOF OF IR5 (USING IR1 THROUGH IR3)

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

PROOF OF IR6 (USING IR1 THROUGH IR3)

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

It has been shown by Armstrong (1974) that inference rules IR1 through IR3 are sound and complete. By **sound,** we mean that given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R that *satisfies the dependencies* in F. By **complete,** we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of *all possible dependencies* that can be inferred from F. In other words, the set of dependencies F^+, which we called the **closure** of F, can be determined from F by using only inference rules IR1 through IR3. Inference rules IR1 through IR3 are known as **Armstrong's inference rules.**[10]

Typically, database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then IR1, IR2, and IR3 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X. Thus, for each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the **closure of** X under F. Algorithm 10.1 can be used to calculate X^+.

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

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

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

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

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

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

{SSN }+ = {SSN, ENAME}
{PNUMBER }+ = {PNUMBER, PNAME, PLOCATION}
{SSN, PNUMBER}+ = {SSN, PNUMBER, ENAME, PNAME, PLOCATION, HOURS}

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

10.2.3 Equivalence of Sets of Functional Dependencies

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

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

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

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

10.2.4 Minimal Sets of Functional Dependencies

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

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

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

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

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

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

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

for each attribute B that is an element of X

if $\{ \{ F - \{X \rightarrow A\} \} \cup \{(X - \{B\}) \rightarrow A\} \}$ is equivalent to F,

then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F.

4. For each remaining functional dependency $X \rightarrow A$ in F

   if $\{ F - \{X \rightarrow A\} \}$ is equivalent to F,

   then remove $X \rightarrow A$ from F.

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

10.3 Normal Forms Based on Primary Keys

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

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

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

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

10.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to "certify" whether it satisfies a certain **normal form.** The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis*. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively; these are discussed in Chapter 11. At the beginning of Chapter 11, we also discuss how 3NF relations may be synthesized from a given set of FDs. This approach is called *relational design by synthesis*.

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

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

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

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

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

10.3.2 Practical Use of Normal Forms

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

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

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

10.3.3 Definitions of Keys and Attributes Participating in Keys

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

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

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

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

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

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

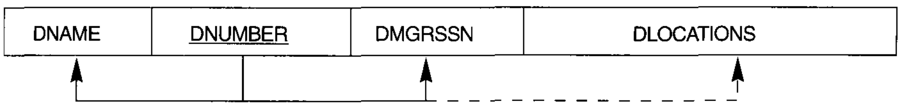
We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for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

10.3.4 First Normal Form

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

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

(a) DEPARTMENT



(b) DEPARTMENT

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

(c) DEPARTMENT

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

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

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

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

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

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

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

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

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

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

(a) **EMP_PROJ**

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b) **EMP_PROJ**

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

(c) **EMP_PROJ1**

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

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

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

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

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

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

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

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

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

PERSON_IN_1NF (SS#, CAR_LIC#, PHONE#)

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

10.3.5 Second Normal Form

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

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

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 10.3b is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the nonprime attributes PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3 make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ, thus violating the 2NF test.

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

10.3.6 Third Normal Form

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

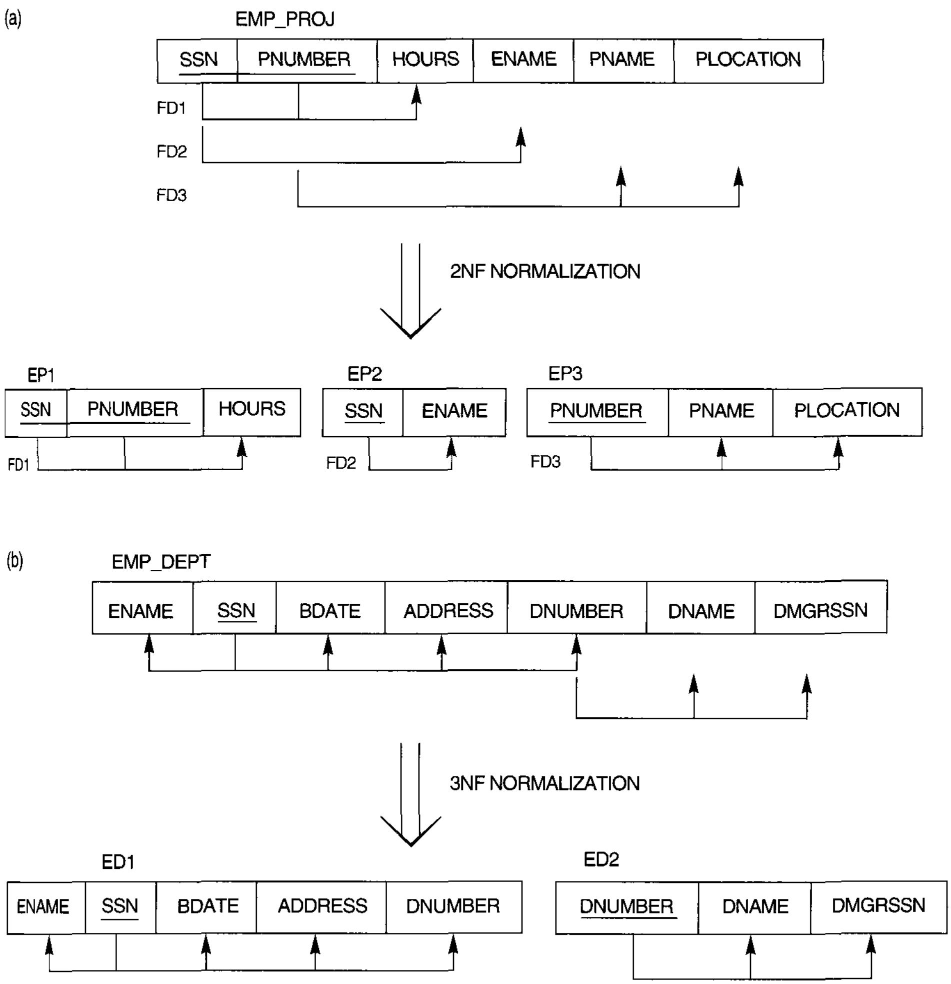


FIGURE 10.10 Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

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

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

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

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

10.4 General Definitions of Second and Third Normal Forms

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

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

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

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

Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

10.4.1 General Definition of Second Normal Form

Definition. A relation schema R is in **second normal form** (2NF) if every nonprime attribute A in R is not partially dependent on *any* key of R.[15]

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

Based on the two candidate keys PROPERTY_ID# and {COUNTY_NAME, LOT#}, we know that the functional dependencies FD1 and FD2 of Figure 10.11a hold. We choose PROPERTY_ID# as the primary key, so it is underlined in Figure 10.11a, but no special consideration will

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

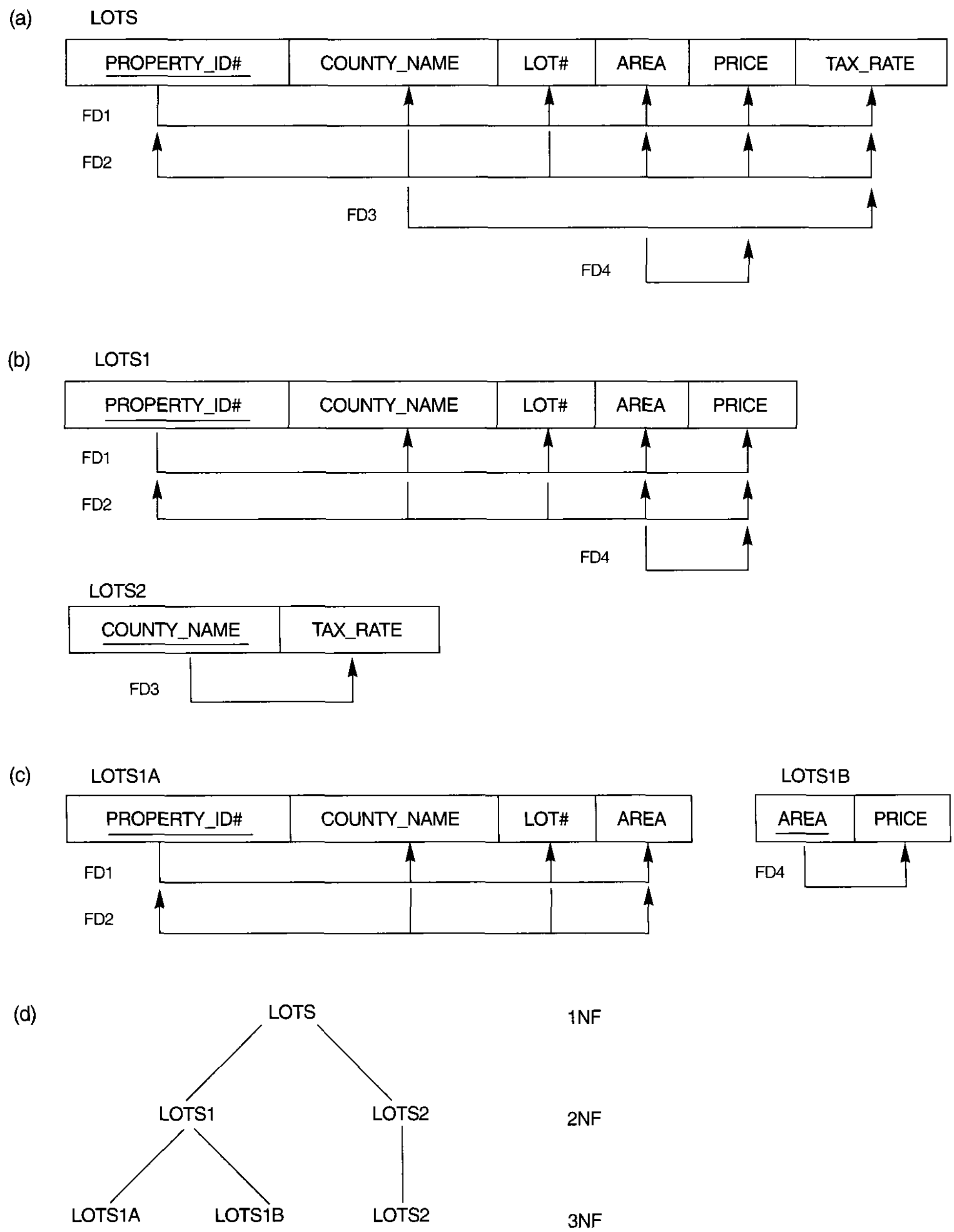


FIGURE 10.11 Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.

be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: COUNTY_NAME → TAX_RATE

FD4: AREA → PRICE

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

The LOTS relation schema violates the general definition of 2NF because TAX_RATE is partially dependent on the candidate key {COUNTY_NAME, LOT#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 10.11b. We construct LOTS1 by removing the attribute TAX_RATE that violates 2NF from LOTS and placing it with COUNTY_NAME (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

10.4.2 General Definition of Third Normal Form

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

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

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

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

10.4.3 Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that violates *both* conditions (a) and (b) of 3NF. Violating (b) means that

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

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

10.5 Boyce-Codd Normal Form

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

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

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

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

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

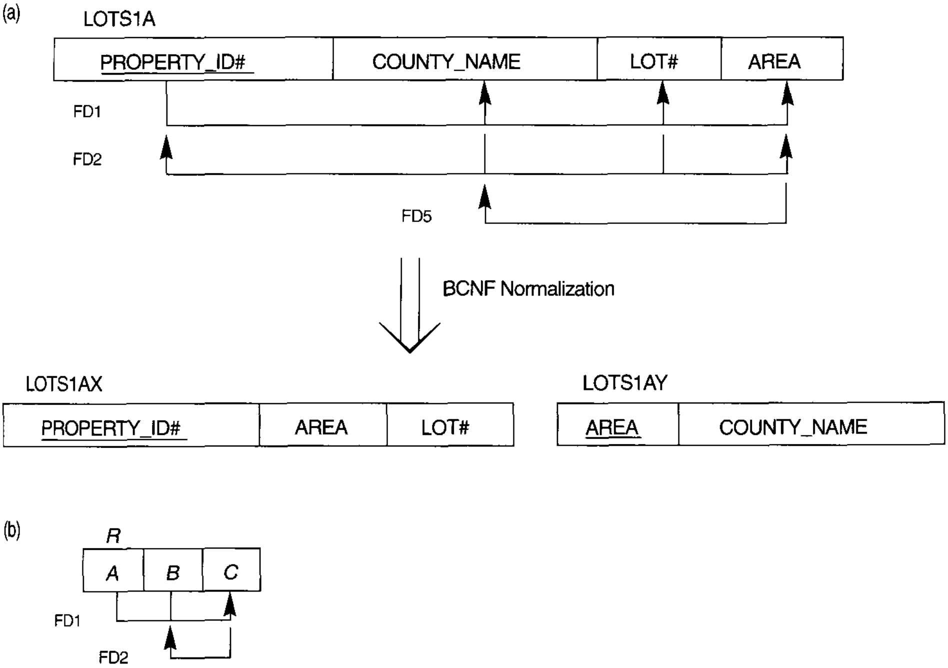


FIGURE 10.12 Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

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

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

FD1: { STUDENT, COURSE} → INSTRUCTOR

FD2:[16] INSTRUCTOR → COURSE

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

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

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

TEACH

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

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

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

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

10.6 Summary

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

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

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