- 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 concepts of the "goodness" and the "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 the Boyce-Codd normal form (BCNF) for relation schemas. In Chapter 15 we give additional criteria for determining that a set of relation schemas together forms a good relational database schema. We also present algorithms that are a part of this theory to design relational databases and define additional normal forms beyond BCNF. The normal forms defined in this chapter are based on the concept of a functional dependency, which we describe next, whereas the normal forms discussed in Chapter 15 use additional types of data dependencies called multivalued dependencies and join dependencies.

14.2 Functional Dependencies

The single most important concept in relational schema design 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 , , . . . ; let us think of the whole database as being described by a single **universal** relation schema (Note 5). 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 (Note 6).

A **functional dependency,** denoted by *X* â *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 and in r that have [*X*] = [*X*], we must also have [*Y*] = [*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; or 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. Notice 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* â *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* â *Y* in *R*, this does not say whether or not *Y* â *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 extensions** (or **legal relation states**) of *R*, because they obey the functional dependency constraints. 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}` â 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 â 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 14.03(b); from the semantics of the attributes, we know that the following functional dependencies should hold:

a. SSN â ENAME
b. PNUMBER â {PNAME, PLOCATION}
c. {SSN, PNUMBER} â 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 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* (intension) *R*, not of a particular legal relation state (extension) *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 14.07 shows a particular state of the TEACH relation schema. Although at first glance we may think that TEXT â 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 14.03 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 Figure 14.03(a) and Figure 14.03(b).

14.2.2 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*. For reallife examples, it is practically impossible to specify all possible functional dependencies that may hold. The set of all such dependencies is called the **closure** of *F* and is denoted by . For example, suppose that we specify the following set *F* of obvious functional dependencies on the relation schema of Figure 14.03(a) :

F = {SSN â {ENAME, BDATE, ADDRESS, DNUMBER},

DNUMBER â {DNAME,DMGRSSN}}

We can *infer* the following additional functional dependencies from *F:*

SSN â {DNAME, DMGRSSN},

SSN â SSN,

DNUMBER â DNAME

An FD *X* â *Y* is **inferred from** a set of dependencies *F* specified on *R* if *X* â *Y* holds in *every* relation state *r* that is a legal extension of *R;* that is, whenever *r* satisfies all the dependencies in *F, X* â *Y* also holds in *r.* The closure 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 X* â *Y* to denote that the functional dependency *X* â *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*} â *Z* is abbreviated to *XY* â *Z,* and the FD {*X,Y,Z*} â {*U,V*} is abbreviated to *XYZ* â *UV.* The following six rules (IR1 through IR6) are well-known inference rules for functional dependencies:

IR1 (reflexive rule (Note 7)): If X Y, then *X* â *Y.*

IR2 (augmentation rule (Note 8)): {*X* â *Y* } *XZ* â *YZ.*

IR3 (transitive rule): {*X* â *Y, Y* â *Z*} *X* â *Z.*

IR4 (decomposition, or projective, rule): {*X* â *YZ*} *X* â *Y.*

IR5 (union, or additive, rule): {*X* â *Y, X* â *Z*} *X* â *YZ.*

IR6 (pseudotransitive rule): {*X* â *Y, WY* â *Z* } *WX* â *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* â *Y* is **trivial** if *X 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* â into the set of dependencies . The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies into the single FD *X* â .

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 Y* and that two tuples and exist in some relation instance *r* of *R* such that [*X*] = [*X*]. Then [*Y*] = [*Y*] because *X Y;* hence, *X* â *Y* must hold in *r.*

PROOF OF IR2 (BY CONTRADICTION)

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

PROOF OF IR3

Assume that (1) *X* â *Y* and (2) *Y* â *Z* both hold in a relation *r*. Then for any two tuples and in *r* such that [*X*] = [*X*], we must have (3) [*Y*] = [*Y*], from assumption (1); hence we must also have (4) [*Z*] = [*Z*], from (3) and assumption (2); hence *X* â *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* â *YZ* (given).
2. *YZ* â *Y* (using IR1 and knowing that *YZ Y*).
3. *X* â *Y* (using IR3 on 1 and 2).

PROOF OF IR5 (USING IR1 THROUGH IR3)

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

PROOF OF IR6 (USING IR1 THROUGH IR3)

1. *X* â *Y* (given).
2. *WY* â *Z* (given).
3. *WX* â *WY* (using IR2 on 1 by augmenting with W).
4. *WX* â *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 , 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** (Note 9).

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 of attributes that are functionally determined by *X* based on *F;* is called the **closure of *X* under *F.*** Algorithm 14.1 can be used to calculate .

Algorithm 14.1 Determining , the closure of *X* under *F*

```
:= X;

repeat

old:= ;

for each functional dependency Y â Z in F do

if Y then := D Z;

until ( = old);
```

Algorithm 14.1 starts by setting 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 , 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 *during a complete cycle* (the for loop) through the dependencies in *F*. For example, consider the relation schema EMP_PROJ in Figure 14.03(b); from the semantics of the attributes, we specify the following set *F* of functional dependencies that should hold on EMP_PROJ:

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

Using Algorithm 14.1, 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 }

14.2.3 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. A set of functional dependencies *E* is **covered by** a set of functional dependencies *F*—or alternatively, *F* is said to **cover** *E*—if every FD in *E* is also in ; that is, if every dependency in *E* can be inferred from *F.* Two sets of functional dependencies *E* and *F* are **equivalent** if = . 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 with respect to *F* for each FD *X* â *Y* in *E*, and then checking whether this 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.*

14.2.4 Minimal Sets of Functional Dependencies

A set of functional dependencies *F* is **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* â *A* in *F* with a dependency *Y* â *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 ensures that every dependency is in a canonical form with a single attribute on the right-hand side (Note 10). 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 *F* is a minimal set of dependencies that is equivalent to *F*. Unfortunately, there can be several minimal covers for a set of functional dependencies. We can always find *at least one* minimal cover *G* for any set of dependencies *F* using Algorithm 14.2.

Algorithm 14.2 Finding a minimal cover *G* for *F*

```
1. Set G := F.
2. Replace each functional dependency X  â  in G by the n functional dependencies X  â  , X
   â  , . . ., X  â  .
3. For each functional dependency X  â  A in G
     for each attribute B that is an element of X
       if ((G - {X  â  A}) D {(X - {B})  â  A}) is equivalent to G,
         then replace X  â  A with (X - {B})  â  A in G.
4. For each remaining functional dependency X  â  A in G
     if (G - {X  â  A}) is equivalent to G,
       then remove X  â  A from G.
```

14.3 Normal Forms Based on Primary Keys

Having studied functional dependencies and some of their properties, we are now ready to use them as information about the semantics of the 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. We will focus 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 14.4. In Section 14.5 we define Boyce-Codd normal form (BCNF), and in Chapter 15 we define further normal forms that are based on other types of data dependencies.

We start in Section 14.3.1 by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 7 that are needed here. We then discuss first normal form (1NF) in Section 14.3.2, and present the definitions of second normal form (2NF) and third normal form (3NF) that are based on primary keys in Section 14.3.3 and Section 14.3.4.

14.3.1 Introduction to Normalization

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 15. At the beginning of Chapter 15, 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 hence 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 14.1.2. 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:

- 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 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 relations 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 shall see in Section 15.1.2. We defer the presentation of the formal concepts and techniques that guarantee the above two properties to Chapter 15.

Additional normal forms may be defined to meet other desirable criteria, based on additional types of constraints, as we shall see in Chapter 15. However, the practical utility of 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 up to 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 for performance reasons, such as those discussed at the end of Section 14.1.2. 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.**

Before proceeding further, let us look again at the definitions of keys of a relation schema from Chapter 7. A **superkey** of a relation schema is a set of attributes *S* *R* with the property that no two tuples and in any legal relation state *r* of *R* will have [S] = [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 of *R,* then *K* - {} is *not a key* of *R* for any *i, 1* 1 *i* 1 *k.* In Figure 14.01 {SSN} is a key for EMPLOYEE, whereas {SSN}, {SSN, ENAME}, {SSN, ENAME, BDATE}, etc. 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 14.01 {SSN} is the only candidate key for EMPLOYEE, so it is also the primary key.

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 14.01 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.

14.3.2 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 (Note 11); 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 attributes of tuples." The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values.**

Consider the DEPARTMENT relation schema shown in Figure 14.01, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute as shown in Figure 14.08(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and an example extension are shown in Figure 14.08. As we can see, this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure 14.08(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 DNUMBER.
- The domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, DNUMBER â DLOCATIONS, because each set is considered a single member of the attribute domain (Note 12).

In either case, the DEPARTMENT relation of Figure 14.08 is not in 1NF; in fact, it does not even qualify as a relation, according to our definition of relation in Section 7.1. 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 14.02. 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 14.08(c). 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.

Of the three solutions above, the first is superior 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.

The 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 14.09 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 research into the relational model is attempting to allow and formalize nested relations (see Section 13.6), which were disallowed early on by 1NF.

Notice that SSN is the primary key of the EMP_PROJ relation in Figure 14.09(a) and Figure 14.09(b), while PNUMBER is the **partial** primary 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 14.09(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 unnormalized relation schema with many levels of nesting into 1NF relations, as we saw in Section 13.6. We also saw in that section that the unnest operator is a part of the nested relational model. Chapter 15 will show that restricting relations to 1NF leads to the problems associated with multivalued dependencies and 4NF.

14.3.3 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency.* A functional dependency *X* â *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 X*, (*X* - {*A*}) *does not* functionally determine *Y.* A functional dependency *X* â *Y* is a **partial dependency** if some attribute *A X* can be removed from *X* and the dependency still holds; that is, for some *A X*, (*X* - {*A*}) â *Y.* In Figure 14.03(b), {SSN, PNUMBER} â HOURS is a full dependency (neither SSN â HOURS nor PNUMBER â HOURS holds). However, the dependency {SSN, PNUMBER} â ENAME is partial because SSN â ENAME holds.

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. 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 EMP_PROJ relation in Figure 14.03(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. 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 14.03(b) hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 14.10(a), each of which is in 2NF.

14.3.4 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency.* A functional dependency *X* â *Y* in a relation schema *R* is a **transitive dependency** if there is a set of attributes *Z* that is neither a candidate key nor a subset of any key of *R* (Note 13), and both *X* â *Z* and *Z* â *Y* hold. The dependency SSN â DMGRSSN is transitive through DNUMBER in EMP_DEPT of Figure 14.03(a) because both the dependencies SSN â DNUMBER and DNUMBER â 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.

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 14.03(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 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 14.10(b). 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.

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 to achieve the normal form.

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 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).

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 discussed so far disallow partial and transitive dependencies on the *primary key.* These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF, since it is independent of keys and functional dependencies. As a general definition of **prime attribute,** an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be *with respect to all candidate keys* of a relation.

14.4.1 General Definition of Second Normal Form

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 (Note 14). Consider the relation schema LOTS shown in Figure 14.11(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: PROPERTY_ID# and {COUNTY_NAME, LOT#}; that is, lot numbers are unique only within each county but PROPERTY_ID numbers are unique across counties for the entire state.

Based on the two candidate keys PROPERTY_ID# and {COUNTY_NAME, LOT#}, we know that the functional dependencies FD1 and FD2 of Figure 14.11(a) hold. We choose PROPERTY_ID# as the primary key, so it is underlined in Figure 14.11(a); but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: COUNTRY_NAME â TAX_RATE

FD4: AREA â 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 14.11(b). We construct LOTS1 by removing the attribute TAX_RATE that violates 2NF from LOTS and placing it with COUNTY_NAME (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

14.4.2 General Definition of Third Normal Form

A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency *X* â *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 14.11b) 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 14.11(c). We construct LOTS1A by removing the attribute PRICE that violates 3NF from LOTS1 and placing it with AREA (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF. Two points are worth noting about 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 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.*

14.4.3 Interpreting the General Definition of 3NF

A relation schema *R* violates the general definition of 3NF if a functional dependency *X* â *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 terms:

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

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, because 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 14.11(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 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 â 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.

The formal definition of BCNF differs slightly from the definition of 3NF. A relation schema *R* is in **BCNF** if whenever a *nontrivial* functional dependency *X* â *A* holds in *R*, then *X* is a superkey of *R.* 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 14.12(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if *X* â *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 14.12(b) 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 status of just 1NF or 2NF is not considered adequate, as they were developed historically as stepping stones to 3NF and BCNF. Figure 14.13 shows a relation TEACH with the following dependencies:

FD1: {STUDENT, COURSE} â INSTRUCTOR

FD2 (Note 15): INSTRUCTOR â COURSE

Note that {STUDENT, COURSE} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.12(b). 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 in one of the three possible pairs:

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

All three decompositions "lose" the functional dependency FD1. The desirable decomposition out of the above three is the third one, because it will not generate spurious tuples after a join. A test to determine whether a decomposition is nonadditive (lossless) is discussed in Section 15.1.3 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 15.3 in the next chapter does that and could have been used above to give the same decomposition for TEACH.

14.6 Summary

In this chapter we discussed on an intuitive basis several pitfalls in relational database design, 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. The topics discussed in this chapter will be continued in Chapter 15, where we discuss more advanced concepts in relational design theory.

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 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 are the fundamental source of semantic information about 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 minimal cover of a set of dependencies, and we provided an algorithm to compute a minimal cover. We also showed how to check whether two sets of functional dependencies are equivalent.

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

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

Chapter 15 will present synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we will discuss the concepts of *lossless* (*nonadditive*) *join* and *dependency preservation*, which are enforced by some of these algorithms. Other topics in Chapter 15 include multivalued dependencies, join dependencies, and additional normal forms that take these dependencies into account.

Review Questions

14.1. Discuss the 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 are many nulls in a relation considered bad?

14.4. Discuss the problem of spurious tuples and how we may prevent it.

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

14.6. What is a functional dependency? Who specifies the functional dependencies that hold among the attributes of a relation schema?

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

14.8. Why are Armstrong's inference rules—the three inference rules IR1 through IR3—important?

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

14.10. What is meant by the closure of a set of functional dependencies?

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

14.12. What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set?

14.13. What does the term *unnormalized relation* refer to? How did the normal forms develop historically?

14.14. 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?

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

15.1.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present here start from a single **universal relation schema** 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 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 in the decomposition so that no attributes are "lost"; formally we have

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

Another goal is to have each individual relation 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 as a whole, in addition to looking at the individual relations. To illustrate this point, consider the EMP_LOCS(ENAME, PLOCATION) relation of Figure 14.05, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF (Note 1). Although EMP_LOCS is in BCNF, it still gives rise to spurious tuples when joined with EMP_PROJ1(SSN, PNUMBER, HOURS, PNAME, PLOCATION), which is not in BCNF (see the result of the natural join in Figure 14.06). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted semantics by which PLOCATION gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(PNAME, PNUMBER, PLOCATION, DNUM) of Figure 14.02—which *is* in BCNF—also gives rise to spurious tuples. We need other criteria that, together with the conditions of 3NF or BCNF, prevent such bad designs. In Section 15.1.2, Section 15.1.3 and Section 15.1.4 we discuss such additional conditions that should hold on a decomposition *D* as a whole.

15.1.2 Decomposition and Dependency Preservation

It would be useful if each functional dependency *X* â *Y* specified in *F* either appeared directly in one of the relation schemas in the decomposition *D* or could be inferred from the dependencies that appear in some . 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 of the decomposition, we cannot enforce this constraint by dealing with an individual relation; instead, we have to join two or more of the relations in the decomposition and then check that the functional dependency holds in the result of the join operation. This is clearly an inefficient and impractical procedure.

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.

First we need a preliminary definition. Given a set of dependencies *F* on *R*, the **projection** of *F* on , denoted by $p_{Ri}(F)$ where is a subset of *R* (Note 2), is the set of dependencies *X* â *Y* in such that the attributes in *X* D *Y* are all contained in . Hence, the projection of *F* on each relation schema in the decomposition *D* is the set of functional dependencies in , the closure of *F*, such that all their left- and right-hand-side attributes are in . We say that a decomposition of *R* is **dependency-preserving** with respect to *F* if the union of the projections of *F* on each in *D* is equivalent to *F;* that is

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

An example of a decomposition that does not preserve dependencies is shown in Figure 14.12(a), where the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 14.11, however, are dependency-preserving. Similarly, for the example in Figure 14.13, no matter what decomposition is chosen for the relation TEACH(STUDENT, COURSE, INSTRUCTOR) out of the three shown, one or both of the dependencies originally present are lost. We state a claim below related to this property without providing any proof.

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

Algorithm 15.1 creates a dependency-preserving decomposition of a universal relation *R* based on a set of functional dependencies *F,* such that each in *D* is in 3NF. It guarantees only the dependency-preserving property; it does *not* guarantee the lossless join property that will be discussed in the next section. The first step of Algorithm 15.1 is to find a minimal cover G for F; Algorithm 14.2 can be used for this step.

Algorithm 15.1 Relational synthesis algorithm with dependency preservation

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

1. Find a minimal cover *G* for *F* (use Algorithm 14.2);
2. For each left-hand-side *X* of a functional dependency that appears in *G*, create a relation schema in *D* with attributes , where *X* â , *X* â , ..., X â are the only dependencies in *G* with *X* as left-hand-side (*X* is the *key* of this relation);

3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

Claim 1A: Every relation schema created by Algorithm 15.1 is in 3NF. (We will not provide a formal proof here (Note 3); the proof depends on *G* being a minimal set of dependencies).

It is obvious that all the dependencies in *G* are preserved by the algorithm because each dependency appears in one of the relations in the decomposition *D.* Since *G* is equivalent to *F,* all the dependencies in *F* are either preserved directly in the decomposition or are derivable from those in the resulting relations, thus ensuring the dependency preservation property. Algorithm 15.1 is called the **relational synthesis algorithm,** because each relation schema in the decomposition is *synthesized* (constructed) from the set of functional dependencies in *G* with the same left-hand-side *X.*

15.1.3 Decomposition and Lossless (Nonadditive) Joins

Another property a decomposition *D* should possess is the lossless join or nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations in the decomposition. We already illustrated this problem in Section 14.1.4 with the example of Figure 14.05 and Figure 14.06. 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. Formally, a decomposition 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:*

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(p) and NATURAL JOIN(*) operations are applied; these additional tuples represent erroneous information. We prefer the term **nonadditive join** because it describes the situation more accurately; if the property holds on a decomposition, we are guaranteed that no spurious tuples bearing wrong information are *added* to the result after the PROJECT and NATURAL JOIN operations are applied.

The decomposition of EMP_PROJ(SSN, PNUMBER, HOURS, ENAME, PNAME, PLOCATION) from Figure 14.03 into EMP_LOCS(ENAME, PLOCATION) and EMP_PROJ1(SSN, PNUMBER, HOURS, PNAME, PLOCATION) in Figure 14.05 obviously does not have the lossless join property as illustrated in Figure 14.06. We can use Algorithm 15.2 to check whether a given decomposition *D* has the lossless join property with respect to a set of functional dependencies *F.*

Algorithm 15.2 Testing for the lossless (nonadditive) join property

Input: A universal relation *R*, a decomposition of *R*, and a set *F* of functional dependencies.

1. Create an initial matrix *S* with one row *i* for each relation in *D*, and one column *j* for each attribute in *R*.
2. Set *S*(*i*,*j*) := for all matrix entries.

(* each b_{ij} is a distinct symbol associated with indices (i,j) *)

3. For each row *i* representing relation schema

{for each column j representing attribute

{if (relation includes attribute) then set *S*(*i*,*j*):= ;};};

(* each 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* â *Y* in *F*

{for all rows in *S which 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 appear in one of the rows for the attribute and set the other rows to that same "*b*" symbol in the column ;};};};

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

Given a relation *R* that is decomposed into a number of relations Algorithm 15.2 begins by creating a relation state *r* in the matrix *S*. Row *i* in *S* represents a tuple (corresponding to relation) which has "*a*" symbols in the columns that correspond to the attributes of and "*b*" symbols in the remaining columns. The algorithm then transforms the rows of this matrix (during the loop of step 4) so that they represent tuples that satisfy all the functional dependencies in *F*. At the end of the loop of applying functional dependencies, 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* â *Y* in *F* will also agree in their values for the right-hand-side attributes *Y*. It can be shown that after applying the loop of Step 4, if any row in *S* ends up with all "a" symbols, then the decomposition *D* has the lossless join property with respect to *F*. If, on the other hand, no row ends up being all "a" symbols, *D* does not satisfy the lossless join property. In the latter case, the relation state *r* represented by *S* at the end of the algorithm will be an example of a relation state *r* of *R* that satisfies the dependencies in *F* but does not satisfy the lossless join condition; thus, this relation serves as a counterexample that proves that *D* does not have the lossless join property with respect to *F*. Note that the "a" and "b" symbols have no special meaning at the end of the algorithm.

Figure 15.01(a) shows how we apply Algorithm 15.2 to the decomposition of the EMP_PROJ relation schema from Figure 14.03(b) into the two relation schemas EMP_PROJ1 and EMP_LOCS of Figure 14.05(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 lossless join property.

Figure 15.01(b) shows another decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON that does have the lossless join property, and Figure 15.01(c) shows how we apply the algorithm to that decomposition. Once a row consists only of "a" symbols, we know that the decomposition has the lossless join property, and we can stop applying the functional dependencies (Step 4 of the algorithm) to the matrix *S*.

Algorithm 15.2 allows us to test whether a particular decomposition *D* obeys the lossless join property with respect to a set of functional dependencies *F*. The next question is whether there is an algorithm to decompose a universal relation schema into a decomposition such that each is in BCNF *and* the decomposition *D* has the lossless join property with respect to *F*. The answer is yes, but we need to present some properties of lossless join decompositions in general before describing the algorithm. The first property deals with **binary decompositions**—decomposition of a relation *R* into two relations. It gives an easier test to apply than Algorithm 15.2, but it is *limited* to binary decompositions only.

PROPERTY LJ1

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

You should verify that this property holds with respect to our informal successive normalization examples in Section 14.3 and Section 14.4. The second property deals with applying successive decompositions.

PROPERTY LJ2

If a decomposition of R has the lossless join property with respect to a set of functional dependencies *F* on *R,* and if a decomposition of has the lossless join property with respect to the projection of *F* on , then the decomposition of *R* has the lossless join property with respect to *F.*

Property LJ2 says that, *if* a decomposition *D* already has the lossless join property—with respect to *F*—*and* we further decompose one of the relation schemas in *D* into another decomposition that has the lossless join property—with respect to p_{Ri}(F)—*then* replacing in *D* by will result in a decomposition that also has the lossless join property—with respect to *F.* We implicitly assumed this property in the informal normalization examples of Section 14.3 and Section 14.4. For example, in Figure 14.11, as we normalized the LOTS relation into LOTS1 and LOTS2, this decomposition was assumed to be lossless. Decomposing LOTS1 further into LOTS1A and LOTS1B results in three relations: LOTS1A, LOTS1B, and LOTS2; this eventual decomposition maintains the losslessness by virtue of Property LJ2 above.

Algorithm 15.3 utilizes properties LJ1 and LJ2 to create a lossless join decomposition of a universal relation *R* based on a set of functional dependencies *F,* such that each in *D* is in BCNF.

Algorithm 15.3 Relational decomposition into BCNF relations with lossless 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 â Y in Q that violates BCNF;
replace Q in D by two relation schemas (Q – Y) and (X D Y);
};
```

Each time through the loop in Algorithm 15.3, we decompose one relation schema *Q* that is not in BCNF into two relation schemas. According to properties LJ1 and LJ2, the decomposition *D* has the lossless join property. At the end of the algorithm, all relation schemas in *D* will be in BCNF. The reader can check that the normalization example in Figure 14.11 and Figure 14.12 basically follows this algorithm. The functional dependencies FD3, FD4, and later FD5 violate BCNF, so the LOTS relation is decomposed appropriately into BCNF relations and the decomposition then satisfies the lossless join property. Similarly, if we apply the algorithm to the TEACH relation schema from Figure 14.13, it is decomposed into TEACH1(<u>INSTRUCTOR</u>, <u>STUDENT</u>) and TEACH2(<u>INSTRUCTOR</u>, COURSE) because the dependency FD2 : INSTRUCTOR â COURSE violates BCNF.

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

A related problem is that of **dangling tuples,** which may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation of Figure 15.02(a) further into EMPLOYEE_1 and EMPLOYEE_2, shown in Figure 15.03(a) and Figure 15.03(b) (Note 5). 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 15.03(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** because they are represented in only one of the two relations that represent employees and hence are lost if we apply an (inner) join operation.

15.1.5 Discussion of Normalization Algorithms

One of the problems with the normalization algorithms we described is that the database designer must first specify *all* the relevant functional dependencies among the database attributes. This is not a simple task for a large database with hundreds of attributes. Failure to specify one or two important dependencies may result in an undesirable design. Another problem is that these algorithms are not deterministic in general. For example, the *synthesis algorithms* (Algorithms 15.1 and 15.4) require the specification of a minimal cover *G* for the set of functional dependencies *F*. Because there may be in general many minimal covers corresponding to *F,* the algorithm can give different designs depending on the particular minimal cover used. Some of these designs may not be desirable. The *decomposition algorithm* (Algorithm 15.3) depends on the order in which the functional dependencies are supplied to the algorithm; again it is possible that many different designs may arise corresponding to the same set of functional dependencies, depending on the order in which such dependencies are considered for violation of BCNF. Again, some of the designs may be quite superior while others may be undesirable.

15.2 Multivalued Dependencies and Fourth Normal Form

So far we have discussed only functional dependency, which is by far the most important type of dependency in relational database design theory. However, in many cases relations have constraints that cannot be specified as functional dependencies. In this section, we discuss the concept of *multivalued dependency* (MVD) and define *fourth normal form*, which is based on this dependency. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 14.3.2), which disallowed an attribute in a tuple to have a *set of values.* If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure 15.04(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 (Note 6). To keep the relation state consistent, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multivalued dependency on the EMP relation. Informally, whenever two *independent* 1:N relationships *A:B* and *A:C* are mixed in the same relation, an MVD may arise.

15.2.1 Formal Definition of Multivalued Dependency

Formally, a **multivalued dependency (MVD)** *X Y* specified on relation schema *R,* where *X* and *Y* are both subsets of *R*, specifies the following constraint on any relation state *r* of *R*: If two tuples and exist in *r* such that [*X*] = [*X*], then two tuples and should also exist in *r* with the following properties (Note 7), where we use *Z* to denote (*R* - (*X* D *Y*)) (Note 8):

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

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

In Figure 15.04(a) the MVDs ENAME PNAME and ENAME DNAME (or ENAME PNAME | DNAME) hold in the EMP relation. The employee with ENAME 'Smith' works on projects with PNAME 'X' and 'Y' and has two dependents with DNAME 'John' and 'Anna'. If we stored only the first two tuples in EMP (<'Smith', 'X', 'John'> and <'Smith', 'Y', 'Anna'>), we would incorrectly show associations between project 'X' and 'John' and between project 'Y' and 'Anna'; these should not be conveyed, because no such meaning is intended in this relation. Hence, we must store the other two tuples (<'Smith', 'X', 'Anna'> and <'Smith', 'Y', 'John'>) to show that {'X', 'Y'} and {'John', 'Anna'} are associated only with 'Smith'; that is, there is no association between PNAME and DNAME—which means that the two attributes are independent.

An MVD *X Y* in *R* is called a **trivial MVD** if (a) *Y* is a subset of *X*, or (b) *X* D *Y* = *R.* For example, the relation EMP_PROJECTS in Figure 15.04(b) has the trivial MVD ENAME PNAME. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD.** A trivial MVD will hold in *any* relation state *r* of *R;* it is called trivial because it does not specify any significant or meaningful constraint on *R.*

If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 15.04(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. We first discuss some of the properties of MVDs and consider how they are related to functional dependencies.

15.2.2 Inference Rules for Functional and Multivalued Dependencies

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

IR1 (reflexive rule for FDs): If *X Y*, then *X* â *Y.*

IR2 (augmentation rule for FDs): {X â *Y*} *XZ* â *YZ.*

IR3 (transitive rule for FDs): {*X* â *Y*, *Y* â *Z*} *X* â *Z.*

IR4 (complementation rule for MVDs): {*X Y*} {*X* (*R* – (*X* D *Y*))}.

IR5 (augmentation rule for MVDs): If *X Y* and *W Z* then *WX YZ.*

IR6 (transitive rule for MVDs): {*X Y*, *Y Z*} *X* (*Z* – *Y*).

IR7 (replication rule for FD to MVD): {*X* â *Y*} *X Y.*

IR8 (coalescence rule for FDs and MVDs): If *X Y* and there exists *W* with the properties that (a) *W* C *Y* is empty, (b) *W* â *Z,* and (c) *Y Z*, then *X* â *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 MVD. Basically, an FD *X* â *Y* is an MVD *X Y* with the *additional restriction* that at most one value of *Y* is associated with each value of *X* (Note 9). Given a set *F* of functional and multivalued dependencies specified on , we can use IR1 through IR8 to infer the (complete) set of all dependencies (functional or multivalued) that will hold in every relation state *r* of *R* that satisfies *F*. We again call the **closure** of *F*.

15.2.3 Fourth Normal Form

We now present the definition of **fourth normal form (4NF),** which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations. A relation schema *R* is in 4NF with respect to a set of dependencies *F* (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency *X Y* in , *X* is a superkey for R.

The EMP relation of Figure 15.04(a) is not in 4NF because in the nontrivial MVDs ENAME PNAME and ENAME DNAME, ENAME is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 15.04(b). Both EMP_PROJECTS and EMP_ DEPENDENTS are in 4NF, because the MVDs ENAME PNAME in EMP_PROJECTS and ENAME DNAME in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

To illustrate the importance of 4NF, Figure 15.05(a) shows the EMP relation with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('*W*', '*X*', '*Y*', and '*Z*'). There are 16 tuples in EMP in Figure 15.05(a). If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 15.05(b), we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but also the update anomalies associated with multivalued dependencies are avoided. For example, if Brown starts working on another project, we must insert three tuples in EMP—one for each dependent. If we forget to insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent. However, only a single tuple need be inserted in the 4NF relation EMP_PROJECTS. Similar problems occur with deletion and modification anomalies if a relation is not in 4NF.

The EMP relation in Figure 15.04(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 between three entities that depends on all three participating entities, such as the SUPPLY relation shown in Figure 15.04(c). (Consider only the tuples in Figure 15.04(c) *above* the dotted line for now.) In this case a tuple represents a supplier supplying a specific part *to a particular project*, so there are *no* nontrivial MVDs. The SUPPLY relation is already in 4NF and should not be decomposed. Notice that relations containing nontrivial MVDs tend to be **all key relations**—that is, their key is all their attributes taken together.

15.2.4 Lossless Join Decomposition into 4NF Relations

Whenever we decompose a relation schema *R* into = (*X* D *Y*) and = (*R* – *Y*) based on an MVD *X Y* that holds in *R*, the decomposition has the lossless join property. It can be shown that this is a necessary and sufficient condition for decomposing a schema into two schemas that have the lossless join property, as given by property LJ1.

PROPERTY L J1

The relation schemas and form a lossless join decomposition of *R* if and only if (C) (-) (or by symmetry, if and only if (C) (-)).

This is similar to property LJ1 of Section 15.1.3, except that LJ1 dealt with FDs only, whereas LJ1' deals with both FDs and MVDs (recall that an FD is also an MVD). We can use a slight modification of Algorithm 15.3 to develop Algorithm 15.5, which creates a lossless join decomposition into relation schemas that are in 4NF (rather than in BCNF). As with Algorithm 15.3, Algorithm 15.5 does *not* necessarily produce a decomposition that preserves FDs.

Algorithm 15.5 Relational decomposition into 4NF relations with lossless 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 Y in Q that violates 4NF
replace Q in D by two relation schemas (Q – Y) and (X D Y);
};
```

15.3 Join Dependencies and Fifth Normal Form

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

A **join dependency** (**JD**), denoted by JD, 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 lossless join decomposition into ; that is, for every such *r* we have

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

For an example of a JD, consider once again the SUPPLY all-key relation of Figure 15.04(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(R1, R2, R3) among the three projections R1(SNAME, PARTNAME), R2(SNAME, PROJNAME), and R3(PARTNAME, PROJNAME) of SUPPLY. If this constraint holds, the tuples below the dotted line in Figure 15.04(c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dotted line. Figure 15.04(d) shows how the SUPPLY relation *with the join dependency* is decomposed into three relations R1, R2, and R3 that are each in 5NF. Notice that applying NATURAL JOIN to *any two* of these relations *produces spurious tuples,* but applying NATURAL JOIN to *all three together* does not. The reader should verify this on the example relation of Figure 15.04(c) and its projections in Figure 15.04(d). This is because only the JD exists, but no MVDs are specified. Notice, too, that the JD(R1, R2, R3) is specified on all legal relation states, not just on the one shown in Figure 15.04(c).

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

15.4 Inclusion Dependencies

Inclusion dependencies were defined in order to formalize certain interrelational constraints. For example, the foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations; but it can be specified as an inclusion dependency. Moreover, inclusion dependencies can also be used to represent the constraint between two relations that represent a class/subclass relationship (see Chapter 4). Formally, an **inclusion dependency** *R.X* < *S.Y* between two sets of attributes—*X* of relation schema *R,* and *Y* of relation schema *S*—specifies the constraint that, at any specific time when *r* is a relation state of *R* and *s* a relation state of *S,* we must have

$\mathrm{p}_X(r(R))\ \mathrm{p}_Y(s(S))$