chapter 15

Relational Database Design Algorithms and Further Dependencies

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

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

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

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

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

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

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

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

15.1.1 Inference Rules for Functional Dependencies

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof of IR5 (using IR1 through IR3).

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

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

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

Algorithm 15.1 can be used to calculate X^+.

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

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

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

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

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

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

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

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

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

15.1.2 Equivalence of Sets of Functional Dependencies

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

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

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

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

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

15.1.3 Minimal Sets of Functional Dependencies

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

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

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

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

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

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

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

---

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

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

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

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

Input: A set of functional dependencies E.

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

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

We illustrate the above algorithm with the following examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

15.2 Properties of Relational Decompositions

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

15.2.1 Relation Decomposition and Insufficiency of Normal Forms

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

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

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

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

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

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

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

15.2.2 Dependency Preservation Property of a Decomposition

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

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

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

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

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

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

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

15.2.3 Nonadditive (Lossless) Join Property of a Decomposition

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

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

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

Algorithm 15.3. Testing for Nonadditive Join Property

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

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

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

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

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

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

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

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

15.2.4 Testing Binary Decompositions for the Nonadditive Join Property

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

15.2.5 Successive Nonadditive Join Decompositions

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

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

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

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

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

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

(No changes to matrix after applying functional dependencies)

(b)

EMP

Ssn	Ename

PROJECT

Pnumber	Pname	Plocation

WORKS_ON

Ssn	Pnumber	Hours

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

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

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

(Original matrix S at start of algorithm)

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

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

15.3 Algorithms for Relational Database Schema Design

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

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

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

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

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

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

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

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

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

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

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

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

The following dependencies are present:

FD1: Emp_ssn → {Esal, Ephone, Dno}

FD2: Pno → { Pname, Plocation}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FD1: Property_id → Lot#, County, Area

FD2: Lot#, County → Area, Property_id

FD3: Area → County

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

F: { P → LCA, LC → AP, A → C }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

15.3.2 Nonadditive Join Decomposition into BCNF Schemas

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

Algorithm 15.5. Relational Decomposition into BCNF with Nonadditive Join Property

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

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

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

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

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

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

15.4 About Nulls, Dangling Tuples, and Alternative Relational Designs

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

15.4.1 Problems with NULL Values and Dangling Tuples

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

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

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

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

15.4.2 Discussion of Normalization Algorithms and Alternative Relational Designs

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

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

(a)

EMPLOYEE

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

DEPARTMENT

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

(b)

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

(c)

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

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

(a) EMPLOYEE_1

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

(b) EMPLOYEE_2

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

(c) EMPLOYEE_3

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

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

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

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

Table 15.1 Summary of the Algorithms Discussed in This Chapter

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

15.5 Further Discussion of Multivalued Dependencies and 4NF

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

15.5.1 Inference Rules for Functional and Multivalued Dependencies

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

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

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

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

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

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

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

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

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

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

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

15.5.2 Fourth Normal Form Revisited

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

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

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

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

(a) EMP

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

(b) EMP_PROJECTS

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

EMP_DEPENDENTS

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

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

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

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

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

15.5.3 Nonadditive Join Decomposition into 4NF Relations

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

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

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

or, by symmetry, if and only if

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

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

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

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

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

15.6 Other Dependencies and Normal Forms

15.6.1 Join Dependencies and the Fifth Normal Form

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

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

15.6.2 Inclusion Dependencies

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

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

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

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

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

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

DEPARTMENT.Dmgr_ssn < EMPLOYEE.Ssn

WORKS_ON.Ssn < EMPLOYEE.Ssn

EMPLOYEE.Dnumber < DEPARTMENT.Dnumber

PROJECT.Dnum < DEPARTMENT.Dnumber

WORKS_ON.Pnumber < PROJECT.Pnumber

DEPT_LOCATIONS.Dnumber < DEPARTMENT.Dnumber

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

EMPLOYEE.Ssn < PERSON.Ssn

ALUMNUS.Ssn < PERSON.Ssn

STUDENT.Ssn < PERSON.Ssn

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

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

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

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

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

15.6.3 Functional Dependencies Based on Arithmetic Functions and Procedures

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

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

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

Extended_price = Unit_price * Quantity

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

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

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

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

15.6.4 Domain-Key Normal Form

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

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

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

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

15.7 Summary

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

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

Review Questions

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

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

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

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

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

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

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

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

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

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

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

15.12. Discuss the NULL value and dangling tuple problems.

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

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

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

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

Exercises

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

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

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

15.20. Write programs that implement Algorithms 15.4 and 15.5.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Laboratory Exercises

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

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

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

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

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

Selected Bibliography

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

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

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