chapter 16

Relational Database Design Algorithms and Further Dependencies

Chapter 15 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, 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 15.3 along with definitions of related normal forms, which are successively better states of design of individual relations. In Section 15.3 we assumed that primary keys were assigned to individual relations; in Section 15.4 a more general treatment of normalization was presented where all candidate keys are considered for each relation, and Section 15.5 discussed a further normal form called BCNF. Then in Sections 15.6 and 15.7 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 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 15, 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 15 and briefly point out other types of dependencies that have been identified.

In Section 16.1 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 16.2, 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 16.3 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 16.5 we discuss the multivalued dependency (MVD) concept further by applying the notions of inference, and equivalence to MVDs. Finally, in Section 16.6 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. Template dependencies are a way of representing any generalized constraint on attributes. 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 16.7 summarizes this chapter.

It is possible to skip some or all of Sections 16.3, 16.4, and 16.5 in an introductory database course.

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

We introduced the concept of functional dependencies (FDs) in Section 15.2, 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 15.3 and 15.4 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 15.3. In Sections 15.4 and 15.5 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.

16.1.1 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.

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 → Mgr_ssn), and a manager has a unique phone number called Mgr_phone (Mgr_ssn → Mgr_phone), then these two dependencies together imply that Dept_no → Mgr_phone. This is an inferred FD and need *not* be explicitly stated in addition to the two given FDs. Therefore, 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 15.3(a):

F = {Ssn → {Ename, Bdate, Address, Dnumber}, Dnumber → {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

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

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

IR1 (reflexive rule)[1]: If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule)[2]: $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

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

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

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

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

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

---

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

[2]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.

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

Proof of IR4 (Using IR1 through IR3).

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

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).

Proof of IR6 (using IR1 through IR3).

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

It has been shown by Armstrong (1974) that inference rules IR1 through IR3 are sound and complete. By **sound**, we mean that given a set of functional dependencies

F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R that *satisfies the dependencies* in F. By **complete**, we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of *all possible dependencies* that can be inferred from F. In other words, the set of dependencies F^+, which we called the **closure** of F, can be determined from F by using only inference rules IR1 through IR3. Inference rules IR1 through IR3 are known as **Armstrong's inference rules**.[3]

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 16.1 can be used to calculate X^+.

Algorithm 16.1. 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 16.1 starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the *repeat* loop) until no more attributes are added to X^+ *during a complete cycle* (of the *for* loop) through the dependencies in F. For example, consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PROJ:

$F = \{$Ssn $\rightarrow$ Ename,
Pnumber $\rightarrow$ {Pname, Plocation},
{Ssn, Pnumber} $\rightarrow$ Hours$\}$

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

Using Algorithm 16.1, we calculate the following closure sets with respect to F:

{Ssn} $^+$ = {Ssn, Ename}
{Pnumber} $^+$ = {Pnumber, Pname, Plocation}
{Ssn, Pnumber} $^+$ = {Ssn, Pnumber, Ename, Pname, Plocation, Hours}

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

16.1.2 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 $G = \{A \rightarrow CD, E \rightarrow AH\}$.

16.1.3 Minimal Sets of Functional Dependencies

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

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

We can think of a minimal set of dependencies as being a set of dependencies in a *standard* or *canonical form* and with *no redundancies.* Condition 1 just represents

every dependency in a canonical form with a single attribute on the right-hand side.[4] Conditions 2 and 3 ensure that there are no redundancies in the dependencies either by having redundant attributes on the left-hand side of a dependency (Condition 2) or by having a dependency that can be inferred from the remaining FDs in F (Condition 3).

> **Definition.** A minimal cover of a set of functional dependencies E is a minimal set of dependencies (in the standard canonical form 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 16.2.

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

> **Algorithm 16.2.** Finding a Minimal Cover F for a Set of Functional Dependencies E
>
> **Input:** A set of functional dependencies E.
>
> 1. Set $F := E$.
> 2. Replace each functional dependency $X \rightarrow \{A_1, A_2, ..., A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_n$.
> 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.
> 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.

We illustrate the above algorithm with the following:

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 16.2 and can proceed to step 2. In step 2 we need to determine if $AB \rightarrow D$ has any redundant attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?

[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.

- Since B → A, by augmenting with *B* on both sides (IR2), we have *BB* → *AB*, or *B* → *AB* (i). However, *AB* → *D* as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), *B* → *D*. Thus *AB* → *D* may be replaced by *B* → *D*.
- We now have a set equivalent to original *E*, say *E′*: {*B* → *A*, *D* → *A*, *B* → *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* → *D* and *D* → *A*, we derive *B* → *A*. Hence *B* → *A* is redundant in *E′* and can be eliminated.
- Therefore, the minimal cover of *E* is {*B* → *D*, *D* → *A*}.

In Section 16.3 we will see how relations can be synthesized 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 16.2(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 Algoritm 16.2(a), we start by setting *K* to all the attributes of *R*; we then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice, too, that Algorithm 16.2(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.

16.2 Properties of Relational Decompositions

We now turn our attention to the process of decomposition that we used throughout Chapter 15 to decompose relations in order to get rid of unwanted dependencies and achieve higher normal forms. In Section 16.2.1 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 16.2.2 and 16.2.3 we discuss two of these properties: the dependency preservation property and the nonadditive (or lossless) join property. Section 16.2.4 discusses binary decompositions and Section 16.2.5 discusses successive nonadditive join decompositions.

16.2.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 16.3 start from a single **universal relation schema** $R = \{A_1, A_2, ..., A_n\}$ 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 $D = \{R_1, R_2, ..., R_m\}$ 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 15.5, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF.[5] 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 result of the natural join in Figure 15.6). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted semantics by which Plocation gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(Pname, Pnumber, Plocation, Dnum) in Figure 15.2—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.

16.2.2 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

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

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 $\pi_{Ri}(F)$ 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 their left- and right-hand-side attributes are in R_i. We say that a decomposition $D = \{R_1, R_2, ..., R_m\}$ 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, $((\pi_{R_1}(F)) \cup ... \cup (\pi_{R_m}(F)))^+ = F^+$.

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 15.13(a), in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 15.12, however, are dependency-preserving. Similarly, for the example in Figure 15.14, 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 state a claim below related to this property without providing any proof.

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

In Section 16.3.1, we describe Algorithm 16.4, which creates a dependency-preserving decomposition $D = \{R_1, R_2, ..., R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF.

16.2.3 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 15.1.4 with the example in Figures 15.5

and 15.6. 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 $D = \{R_1, R_2, ..., R_m\}$ 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: $*(\pi_{R_1}(r), ..., \pi_{R_m}(r)) = r$.

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 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 (see example at the end of Algorithm 16.4).

The decomposition of EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation) in Figure 15.3 into EMP_LOCS(Ename, Plocation) and EMP_PROJ1(Ssn, Pnumber, Hours, Pname, Plocation) in Figure 15.5 obviously does not have the nonadditive join property, as illustrated by Figure 15.6. We will use 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 16.3 below. It is possible to apply a simpler test to check if the decomposition is nonadditive for binary decompositions; that test is described in Section 16.2.4.

Algorithm 16.3. Testing for Nonadditive Join Property

Input: A universal relation R, a decomposition $D = \{R_1, R_2, ..., R_m\}$ 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, ..., R_m, Algorithm 16.3 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 16.1(a) shows how we apply Algorithm 16.3 to the decomposition of the EMP_PROJ relation schema from Figure 15.3(b) into the two relation schemas EMP_PROJ1 and EMP_LOCS in Figure 15.5(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 16.1(b) shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the nonadditive join property, and Figure 16.1(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.

Figure 16.1
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)

16.2.4 Testing Binary Decompositions for the Nonadditive Join Property

Algorithm 16.3 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. We give an easier test to apply than Algorithm 16.3, but while it is very handy to use, it is *limited* to binary decompositions only.

> **Property NJB (Nonadditive Join Test for Binary Decompositions).** A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R *if and only if* either
>
> - The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+, or
> - The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+

You should verify that this property holds with respect to our informal successive normalization examples in Sections 15.3 and 15.4. In Section 15.5 we decomposed LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, and decomposed the TEACH relation in Figure 15.14 into the two relations {Instructor, Course} and {Instructor, Student}. These are valid decompositions because they are nonadditive per the above test.

16.2.5 Successive Nonadditive Join Decompositions

We saw the successive decomposition of relations during the process of second and third normalization in Sections 15.3 and 15.4. 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 $D = \{R_1, R_2, ..., R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $D_i = \{Q_1, Q_2, ..., Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, ..., R_{i-1}, Q_1, Q_2, ..., Q_k, R_{i+1}, ..., R_m\}$ of R has the nonadditive join property with respect to F.

16.3 Algorithms for Relational Database Schema Design

We now give three algorithms for creating a relational decomposition from a universal relation. Each algorithm has specific properties, as we discuss next.

arbitrary types of constraints. We 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

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

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

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

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

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

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

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

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

16.9. 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.

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

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

16.12. Discuss the NULL value and dangling tuple problems.

16.13. 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?

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

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

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

Exercises

16.17. Show that the relation schemas produced by Algorithm 16.4 are in 3NF.

16.18. Show that, if the matrix S resulting from Algorithm 16.3 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.

16.19. Show that the relation schemas produced by Algorithm 16.5 are in BCNF.

16.20. Show that the relation schemas produced by Algorithm 16.6 are in 3NF.

16.21. Specify a template dependency for join dependencies.

16.22. Specify all the inclusion dependencies for the relational schema in Figure 3.5.

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

16.24. Consider the example of normalizing the LOTS relation in Sections 15.4 and 15.5. Determine whether the decomposition of LOTS into {LOTS1AX, LOTS1AY, LOTS1B, LOTS2} has the lossless join property, by applying Algorithm 16.3 and also by using the test under Property NJB.

16.25. Show how the MVDs Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname in Figure 15.15(a) may arise during normalization into 1NF of a relation, where the attributes Pname and Dname are multivalued.

16.26. Apply Algorithm 16.2(a) to the relation in Exercise 15.24 to determine a key for R. Create a minimal set of dependencies G that is equivalent to F, and apply the synthesis algorithm (Algorithm 16.6) to decompose R into 3NF relations.

16.27. Repeat Exercise 16.26 for the functional dependencies in Exercise 15.25.

16.28. Apply the decomposition algorithm (Algorithm 16.5) to the relation R and the set of dependencies F in Exercise 15.24. Repeat for the dependencies G in Exercise 15.25.

16.29. Apply Algorithm 16.2(a) to the relations in Exercises 15.27 and 15.28 to determine a key for R. Apply the synthesis algorithm (Algorithm 16.6) to decompose R into 3NF relations and the decomposition algorithm (Algorithm 16.5) to decompose R into BCNF relations.

16.30. Write programs that implement Algorithms 16.5 and 16.6.

16.31. Consider the following decompositions for the relation schema R of Exercise 15.24. 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. $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C, D\}$, $R_2 = \{D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

16.32. 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, giving 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 16.2.4.)

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 15.24 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.

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

a. 16.24

b. 16.26

c. 16.27

d. 16.28

e. 16.29

f. 16.31 (a) and (b)

g. 16.32 (a) and (c)