15 SCHEMA REFINEMENT AND NORMAL FORMS

> It is a melancholy truth that even great men have their poor relations.
>
> —Charles Dickens

Conceptual database design gives us a set of relation schemas and integrity constraints (ICs) that can be regarded as a good starting point for the final database design. This initial design must be refined by taking the ICs into account more fully than is possible with just the ER model constructs and also by considering performance criteria and typical workloads. In this chapter we discuss how ICs can be used to refine the conceptual schema produced by translating an ER model design into a collection of relations. Workload and performance considerations are discussed in Chapter 16.

We concentrate on an important class of constraints called *functional dependencies*. Other kinds of ICs, for example *multivalued dependencies* and *join dependencies*, also provide useful information. They can sometimes reveal redundancies that cannot be detected using functional dependencies alone. We discuss these other constraints briefly.

This chapter is organized as follows. Section 15.1 is an overview of the schema refinement approach discussed in this chapter. We introduce functional dependencies in Section 15.2. In Section 15.3 we present several examples that highlight the problems caused by redundancy and illustrate how relational schemas obtained by translating an ER model design can nonetheless suffer from these problems. Thus, ER design is a good starting point, but we still need techniques to detect schemas with these problems and to refine such schemas to eliminate the problems. We lay the foundation for developing such schema refinement techniques in Section 15.4, where we show how to reason with functional dependency information to infer additional dependencies from a given set of dependencies.

We introduce normal forms for relations in Section 15.5; the normal form satisfied by a relation is a measure of the redundancy in the relation. A relation with redundancy can be refined by *decomposing it*, or replacing it with smaller relations that contain the same information, but without redundancy. We discuss decompositions and desirable properties of decompositions in Section 15.6. We show how relations can be decomposed into smaller relations that are in desirable normal forms in Section 15.7. Finally, we discuss the use of other kinds of dependencies for database design in Section 15.8.

15.1 INTRODUCTION TO SCHEMA REFINEMENT

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

15.1.1 Problems Caused by Redundancy

Storing the same information **redundantly**, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set from Chapter 2:

Hourly_Emps(<u>*ssn*</u>, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

In this chapter we will omit attribute type information for brevity, since our focus is on the grouping of attributes into relations. We will often abbreviate an attribute name to a single letter and refer to a relation schema by a string of letters, one per attribute. For example, we will refer to the Hourly_Emps schema as *SNLRWH* (*W* denotes the *hourly_wages* attribute).

The key for Hourly_Emps is *ssn*. In addition, suppose that the *hourly_wages* attribute is determined by the *rating* attribute. That is, for a given *rating* value, there is only one permissible *hourly_wages* value. This IC is an example of a *functional dependency.* It leads to possible redundancy in the relation Hourly_Emps, as illustrated in Figure 15.1.

If the same value appears in the *rating* column of two tuples, the IC tells us that the same value must appear in the *hourly_wages* column as well. This redundancy has several negative consequences:

ssn	*name*	*lot*	*rating*	*hourly_wages*	*hours_worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Figure 15.1 An Instance of the Hourly_Emps Relation

- Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times, redundancy leads to potential inconsistency. For example, the *hourly_wages* in the first tuple could be updated without making a similar change in the second tuple, which is an example of an *update anomaly.* Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value, which is an example of an *insertion anomaly.*

- If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that *rating* value and its *hourly_wage* value (a *deletion anomaly*).

Let us consider whether the use of *null* values can address some of these problems. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. For instance, to deal with the insertion anomaly example, we can insert an employee tuple with *null* values in the hourly wage field. However, *null* values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating, because we cannot store a null value in the *ssn* field, which is a primary key field. Similarly, to deal with the deletion anomaly example, we might consider storing a tuple with *null* values in all fields except *rating* and *hourly_wages* if the last tuple with a given *rating* would otherwise be deleted. However, this solution will not work because it requires the *ssn* value to be *null*, and primary key fields cannot be *null.* Thus, *null* values do not provide a general solution to the problems of redundancy, even though they can help in some cases. We will not discuss the use of *null* values further.

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

15.1.2 Use of Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other ICs) can be used to identify such situations and to suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of 'smaller' relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as *decomposition* of the larger relation into the smaller relations.

We can deal with the redundancy in Hourly_Emps by decomposing it into two relations:

Hourly_Emps2(*ssn*, *name*, *lot*, *rating*, *hours_worked*)
Wages(*rating*, *hourly_wages*)

The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 15.1 is shown in Figure 15.2.

ssn	*name*	*lot*	*rating*	*hours_worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	*hourly_wages*
8	10
5	7

Figure 15.2 Instances of Hourly_Emps2 and Wages

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it also eliminates the potential for inconsistency. Notice that the insertion and deletion anomalies have also been eliminated.

15.1.3 Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations. The *dependency-preservation* property enables us to enforce any constraint on the original relation by simply enforcing some contraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

A serious drawback of decompositions is that queries over the original relation may require us to join the decomposed relations. If such queries are common, the performance penalty of decomposing the relation may not be acceptable. In this case we may choose to live with some of the problems of redundancy and not decompose the relation. It is important to be aware of the potential problems caused by such residual redundancy in the design and to take steps to avoid them (e.g., by adding some checks to application code). We will not discuss the impact of decompositions on query performance in this chapter; this issue is covered in Section 16.8.

Our goal in this chapter is to explain some powerful concepts and design guidelines based on the theory of functional dependencies. A good database designer should have a firm grasp of normal forms and what problems they (do or do not) alleviate, the technique of decomposition, and potential problems with decompositions. For example, a designer will often ask questions such as these: Is a relation in a given normal form? Is a decomposition dependency-preserving? Our objective is to explain when to raise these questions and the significance of the answers.

15.2 FUNCTIONAL DEPENDENCIES

A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$ [1] if the following holds for every pair of tuples t_1 and t_2 in r:

If $t1.X = t2.X$, then $t1.Y = t2.Y$.

We use the notation $t1.X$ to refer to the projection of tuple t_1 onto the attributes in X, in a natural extension of our TRC notation (see Chapter 4) $t.a$ for referring to attribute a of tuple t. An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

Figure 15.3 illustrates the meaning of the FD $AB \rightarrow C$ by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, AB is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the A field or the B field, they can differ in the C field without violating the FD. On the other hand, if we add a tuple $\langle a1, b1, c2, d1 \rangle$ to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

Figure 15.3 An Instance that Satisfies $AB \rightarrow C$

Recall that a *legal* instance of a relation must satisfy all specified ICs, including all specified FDs. As noted in Section 3.2, ICs must be identified and specified based on the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does *not* hold. However, we can never deduce that an FD *does* hold by looking at one or more instances of the relation because an FD, like other ICs, is a statement about *all* possible legal instances of the relation.

[1] $X \rightarrow Y$ is read as X *functionally determines* Y, or simply as X *determines* Y.

A primary key constraint is a special case of an FD. The attributes in the key play the role of X, and the set of all attributes in the relation plays the role of Y. Note, however, that the definition of an FD does not require that the set X be minimal; the additional minimality condition must be met for X to be a key. If $X \rightarrow Y$ holds, where Y is the set of all attributes, and there is some subset V of X such that $V \rightarrow Y$ holds, then X is a *superkey*; if V is a strict subset of X, then X is not a key.

In the rest of this chapter, we will see several examples of FDs that are not key constraints.

15.3 EXAMPLES MOTIVATING SCHEMA REFINEMENT

It is natural to ask whether we need to decompose relations produced by translating an ER diagram. Shouldn't a good ER design lead to a collection of good relations? Unfortunately, ER design can generate some schemas with redundancy problems, because it is a complex, subjective process, and certain constraints are not expressible in terms of ER diagrams. The examples in this section are intended to illustrate why decomposition of relations produced through ER design might be necessary.

15.3.1 Constraints on an Entity Set

Consider the Hourly_Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:

$$\{ssn\} \rightarrow \{ssn,\ name,\ lot,\ rating,\ hourly\_wages,\ hours\_worked\}$$

For brevity, we will write this FD as $S \rightarrow SNLRWH$, using a single letter to denote each attribute and omitting the set braces, but the reader should remember that both sides of an FD contain sets of attributes. In addition, the constraint that the *hourly_wages* attribute is determined by the *rating* attribute is an FD: $R \rightarrow W$.

As we saw in Section 15.1.1, this FD led to redundant storage of rating–wage associations. *It cannot be expressed in terms of the ER model. Only FDs that determine all attributes of a relation (i.e., key constraints) can be expressed in the ER model.* Therefore, we could not detect it when we considered Hourly_Emps as an entity set during ER modeling.

We could argue that the problem with the original design was an artifact of a poor ER design, which could have been avoided by introducing an entity set called Wage_Table (with attributes *rating* and *hourly_wages*) and a relationship set Has_Wages associating Hourly_Emps and Wage_Table. The point, however, is that we could easily arrive at the original design given the subjective nature of ER modeling. Having formal techniques to identify the problem with this design, and to guide us to a better design, is very useful. The value of such techniques cannot be underestimated when designing large schemas—schemas with more than a hundred tables are not uncommon!

15.3.2 Constraints on a Relationship Set

The previous example illustrated how FDs can help to refine the subjective decisions made during ER design, but one could argue that the best possible ER diagram would have led to the same final set of relations. Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

We revisit an example from Chapter 2. Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them. We refer to the schema for Contracts as $CQPSD$. A contract with contract id C specifies that a supplier S will supply some quantity Q of a part P to a department D. (We have added the contract id field C to the version of the Contracts relation that was discussed in Chapter 2.)

We might have a policy that a department purchases at most one part from any given supplier. Thus, if there are several contracts between the same supplier and department, we know that the same part must be involved in all of them. This constraint is an FD, $DS \rightarrow P$.

Again we have redundancy and its associated problems. We can address this situation by decomposing Contracts into two relations with attributes $CQSD$ and SDP. Intuitively, the relation SDP records the part supplied to a department by a supplier, and the relation $CQSD$ records additional information about a contract. It is unlikely that we would arrive at such a design solely through ER modeling, since it is hard to formulate an entity or relationship that corresponds naturally to $CQSD$.

15.3.3 Identifying Attributes of Entities

This example illustrates how a careful examination of FDs can lead to a better understanding of the entities and relationships underlying the relational tables; in particular, it shows that attributes can easily be associated with the 'wrong' entity set during ER design. The ER diagram in Figure 15.4 shows a relationship set called Works_In that is similar to the Works_In relationship set of Chapter 2, but with an additional key constraint indicating that an employee can work in at most one department. (Observe the arrow connecting Employees to Works_In.)

Using the key constraint, we can translate this ER diagram into two relations:

Workers(<u>*ssn*</u>, *name*, *lot*, *did*, *since*)

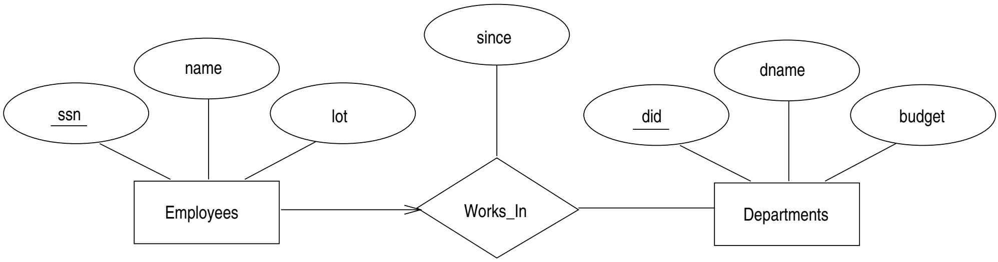

Figure 15.4 The Works_In Relationship Set

Departments(<u>*did*</u>, *dname*, *budget*)

The entity set Employees and the relationship set Works_In are mapped to a single relation, Workers. This translation is based on the second approach discussed in Section 2.4.1.

Now suppose that employees are assigned parking lots based on their department, and that all employees in a given department are assigned the same lot. This constraint is not expressible with respect to the ER diagram of Figure 15.4. It is another example of an FD: $did \rightarrow lot$. The redundancy in this design can be eliminated by decomposing the Workers relation into two relations:

Workers2(<u>*ssn*</u>, *name*, *did*, *since*)
Dept_Lots(<u>*did*</u>, *lot*)

The new design has much to recommend it. We can change the lots associated with a department by updating a single tuple in the second relation (i.e., no update anomalies). We can associate a lot with a department even if it currently has no employees, without using *null* values (i.e., no deletion anomalies). We can add an employee to a department by inserting a tuple to the first relation even if there is no lot associated with the employee's department (i.e., no insertion anomalies).

Examining the two relations Departments and Dept_Lots, which have the same key, we realize that a Departments tuple and a Dept_Lots tuple with the same key value describe the same entity. This observation is reflected in the ER diagram shown in Figure 15.5.

Translating this diagram into the relational model would yield:

Workers2(<u>*ssn*</u>, *name*, *did*, *since*)
Departments(<u>*did*</u>, *dname*, *budget*, *lot*)

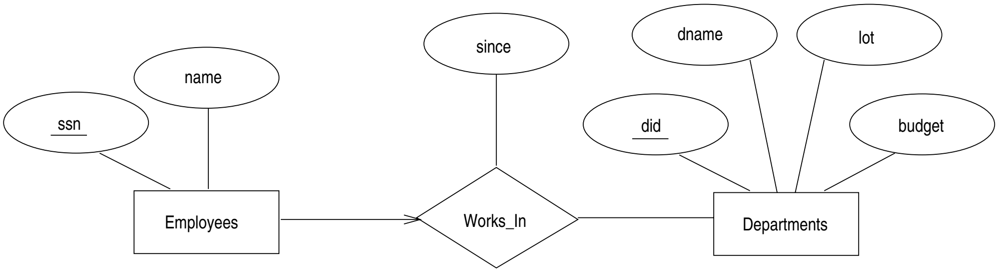

Figure 15.5 Refined Works_In Relationship Set

It seems intuitive to associate lots with employees; on the other hand, the ICs reveal that in this example lots are really associated with departments. The subjective process of ER modeling could miss this point. The rigorous process of normalization would not.

15.3.4 Identifying Entity Sets

Consider a variant of the Reserves schema used in earlier chapters. Let Reserves contain attributes S, B, and D as before, indicating that sailor S has a reservation for boat B on day D. In addition, let there be an attribute C denoting the credit card to which the reservation is charged. We use this example to illustrate how FD information can be used to refine an ER design. In particular, we discuss how FD information can help to decide whether a concept should be modeled as an entity or as an attribute.

Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD $S \rightarrow C$. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that sailor, and we have redundancy and potential update anomalies. A solution is to decompose Reserves into two relations with attributes SBD and SC. Intuitively, one holds information about reservations, and the other holds information about credit cards.

It is instructive to think about an ER design that would lead to these relations. One approach is to introduce an entity set called Credit_Cards, with the sole attribute *cardno*, and a relationship set Has_Card associating Sailors and Credit_Cards. By noting that each credit card belongs to a single sailor, we can map Has_Card and Credit_Cards to a single relation with attributes SC. We would probably not model credit card numbers as entities if our main interest in card numbers is to indicate how a reservation is to be paid for; it suffices to use an attribute to model card numbers in this situation.

A second approach is to make *cardno* an attribute of Sailors. But this approach is not very natural—a sailor may have several cards, and we are not interested in all of them. Our interest is in the one card that is used to pay for reservations, which is best modeled as an attribute of the relationship Reserves.

A helpful way to think about the design problem in this example is that we first make *cardno* an attribute of Reserves and then refine the resulting tables by taking into account the FD information. (Whether we refine the design by adding *cardno* to the table obtained from Sailors or by creating a new table with attributes *SC* is a separate issue.)

15.4 REASONING ABOUT FUNCTIONAL DEPENDENCIES

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further refine relations obtained through ER design (or, for that matter, through other approaches to conceptual design). Before proceeding with the main task at hand, which is the discussion of such schema refinement techniques, we digress to examine FDs in more detail because they play such a central role in schema analysis and refinement.

Given a set of FDs over a relation schema R, there are typically several additional FDs that hold over R whenever all of the given FDs hold. As an example, consider:

Workers($\underline{ssn}$, *name*, *lot*, *did*, *since*)

We know that $ssn \rightarrow did$ holds, since *ssn* is the key, and FD $did \rightarrow lot$ is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same *ssn* value, they must have the same *did* value (from the first FD), and because they have the same *did* value, they must also have the same *lot* value (from the second FD). Thus, the FD $ssn \rightarrow lot$ also holds on Workers.

We say that an FD f **is implied by** a given set F of FDs if f holds on every relation instance that satisfies all dependencies in F, that is, f holds whenever all FDs in F hold. Note that it is not sufficient for f to hold on some instance that satisfies all dependencies in F; rather, f must hold on *every* instance that satisfies all dependencies in F.

15.4.1 Closure of a Set of FDs

The set of all FDs implied by a given set F of FDs is called the **closure of F** and is denoted as F^+. An important question is how we can **infer**, or compute, the closure of a given set F of FDs. The answer is simple and elegant. The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by a set F of FDs. We use X, Y, and Z to denote *sets* of attributes over a relation schema R:

- **Reflexivity:** If $X \supseteq Y$, then $X \rightarrow Y$.
- **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Armstrong's Axioms are **sound** in that they generate only FDs in F^+ when applied to a set F of FDs. They are **complete** in that repeated application of these rules will generate all FDs in the closure F^+. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about F^+:

- **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

To illustrate the use of these inference rules for FDs, consider a relation schema ABC with FDs $A \rightarrow B$ and $B \rightarrow C$. A **trivial FD** is one in which the right side contains only attributes that also appear on the left side; such dependencies always hold due to reflexivity. Using reflexivity, we can generate all trivial dependencies, which are of the form:

$X \rightarrow Y$, where $Y \subseteq X$, $X \subseteq ABC$, and $Y \subseteq ABC$.

From transitivity we get $A \rightarrow C$. From augmentation we get the nontrivial dependencies:

$AC \rightarrow BC$, $AB \rightarrow AC$, $AB \rightarrow CB$.

As a second example, we use a more elaborate version of the Contracts relation:

Contracts(<u>*contractid*</u>, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *value*)

We denote the schema for Contracts as $CSJDPQV$. The meaning of a tuple in this relation is that the contract with *contractid* C is an agreement that supplier S (*supplierid*) will supply Q items of part P (*partid*) to project J (*projectid*) associated with department D (*deptid*); the value V of this contract is equal to *value*.

The following ICs are known to hold:

1. The contract id C is a key: $C \rightarrow CSJDPQV$.
2. A project purchases a given part using a single contract: $JP \rightarrow C$.
3. A department purchases at most one part from a supplier: $SD \rightarrow P$.

Several additional FDs hold in the closure of the set of given FDs:

From $JP \rightarrow C$, $C \rightarrow CSJDPQV$ and transitivity, we infer $JP \rightarrow CSJDPQV$.

From $SD \rightarrow P$ and augmentation, we infer $SDJ \rightarrow JP$.

From $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ and transitivity, we infer $SDJ \rightarrow CSJDPQV$. (Incidentally, while it may appear tempting to do so, we *cannot* conclude $SD \rightarrow CSDPQV$, canceling J on both sides. FD inference is not like arithmetic multiplication!)

We can infer several additional FDs that are in the closure by using augmentation or decomposition. For example, from $C \rightarrow CSJDPQV$, using decomposition we can infer:

$C \rightarrow C$, $C \rightarrow S$, $C \rightarrow J$, $C \rightarrow D$, etc.

Finally, we have a number of trivial FDs from the reflexivity rule.

15.4.2 Attribute Closure

If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set F of FDs, we can do so efficiently without computing F^+. We first compute the **attribute closure** X^+ with respect to F, which is the set of attributes A such that $X \rightarrow A$ can be inferred using the Armstrong Axioms. The algorithm for computing the attribute closure of a set X of attributes is shown in Figure 15.6.

closure $= X$;
repeat until there is no change: {
    if there is an FD $U \rightarrow V$ in F such that $U \subseteq$ *closure*,
        then set *closure* $=$ *closure* $\cup\ V$
    }

Figure 15.6 Computing the Attribute Closure of Attribute Set X

This algorithm can be modified to find keys by starting with set X containing a single attribute and stopping as soon as *closure* contains all attributes in the relation schema.

By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

15.5 NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several **normal forms** have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are *first normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in **first normal form** if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our definition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema R with attributes ABC. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD $A \rightarrow B$. Now if several tuples have the same A value, they must also have the same B value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

We will primarily discuss redundancy that is revealed by FD information. In Section 15.8, we discuss more sophisticated ICs called *multivalued dependencies* and *join dependencies* and normal forms based on them.

15.5.1 Boyce-Codd Normal Form

Let R be a relation schema, X be a subset of the attributes of R, and let A be an attribute of R. R is in **Boyce-Codd normal form** if for every FD $X \rightarrow A$ that holds over R, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey.

Note that if we are given a set F of FDs, according to this definition, we must consider each dependency $X \rightarrow A$ in the closure F^+ to determine whether R is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in F is a superkey (by computing the attribute closure and seeing if it includes all attributes of R).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: "Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure 15.7, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)

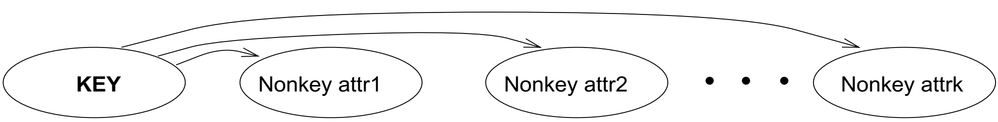

Figure 15.7 FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.

X	Y	A
x	y_1	a
x	y_2	?

Figure 15.8 Instance Illustrating BCNF

This figure shows (two tuples in) an instance of a relation with three attributes X, Y, and A. There are two tuples with the same value in the X column. Now suppose that we know that this instance satisfies an FD $X \rightarrow A$. We can see that one of the tuples has the value a in the A column. What can we infer about the value in the A column in the second tuple? Using the FD, we can conclude that the second tuple also has the value a in this column. (Note that this is really the only kind of inference we can make about values in the fields of tuples by using FDs.)

But isn't this situation an example of redundancy? We appear to have stored the value a twice. Can such a situation arise in a BCNF relation? No! If this relation is in BCNF, because A is distinct from X it follows that X must be a key. (Otherwise, the FD $X \rightarrow A$ would violate BCNF.) If X is a key, then $y_1 = y_2$, which means that the two tuples are identical. Since a relation is defined to be a *set* of tuples, we cannot have two copies of the same tuple and the situation shown in Figure 15.8 cannot arise.

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other fields in (all tuples of) the relation instance.

15.5.2 Third Normal Form

Let R be a relation schema, X be a subset of the attributes of R, and A be an attribute of R. R is in **third normal form** if for every FD $X \rightarrow A$ that holds over R, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey, or
- A is part of some key for R.

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determines all other attributes. A must be part of a key (any key, if there are several). It is not enough for A to be part of a superkey, because the latter condition is satisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

Suppose that a dependency $X \rightarrow A$ causes a violation of 3NF. There are two cases:

- X *is a proper subset of some key* K. Such a dependency is sometimes called a **partial dependency**. In this case we store *(X, A)* pairs redundantly. As an example, consider the Reserves relation with attributes *SBDC* from Section 15.3.4. The only key is *SBD*, and we have the FD $S \rightarrow C$. We store the credit card number for a sailor as many times as there are reservations for that sailor.
- X *is not a proper subset of any key.* Such a dependency is sometimes called a **transitive dependency** because it means we have a chain of dependencies $K \rightarrow X \rightarrow A$. The problem is that we cannot associate an X value with a K value unless we also associate an A value with an X value. As an example, consider the Hourly_Emps relation with attributes *SNLRWH* from Section 15.3.1. The only key is S, but there is an FD $R \rightarrow W$, which gives rise to the chain $S \rightarrow R$

$\rightarrow W$. The consequence is that we cannot record the fact that employee S has rating R without knowing the hourly wage for that rating. This condition leads to insertion, deletion, and update anomalies.

Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure 15.10. Note that in Figure 15.10, the set X of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that X is not a subset of KEY.

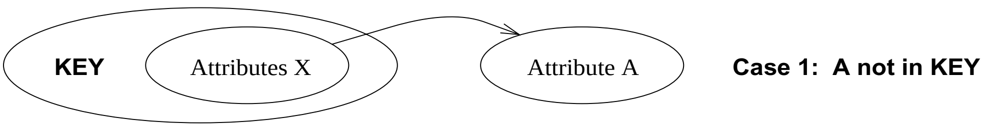

Figure 15.9 Partial Dependencies

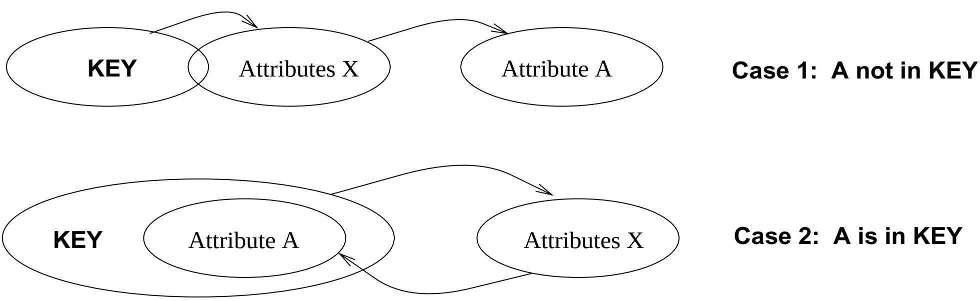

Figure 15.10 Transitive Dependencies

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties (Section 15.6). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise by settling for a 3NF design. As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.

Unlike BCNF, however, some redundancy is possible with 3NF. The problems associated with partial and transitive dependencies persist if there is a nontrivial dependency $X \rightarrow A$ and X is not a superkey, even if the relation is in 3NF because A is part of a key. To understand this point, let us revisit the Reserves relation with attributes *SBDC* and the FD $S \rightarrow C$, which states that a sailor uses a unique credit card to pay for reservations. S is not a key, and C is not part of a key. (In fact, the only key is *SBD*.) Thus, this relation is not in 3NF; *(S, C)* pairs are stored redundantly. However, if we also know that credit cards uniquely identify the owner, we have the FD $C \rightarrow$

S, which means that CBD is also a key for Reserves. Therefore, the dependency $S \rightarrow C$ does not violate 3NF, and Reserves is in 3NF. Nonetheless, in all tuples containing the same S value, the same *(S, C)* pair is redundantly recorded.

For completeness, we remark that the definition of **second normal form** is essentially that partial dependencies are not allowed. Thus, if a relation is in 3NF (which precludes both partial and transitive dependencies), it is also in 2NF.

15.6 DECOMPOSITIONS

As we have seen, a relation in BCNF is free of redundancy (to be precise, redundancy that can be detected using FD information), and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

A **decomposition of a relation schema** R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R. Intuitively, we want to store the information in any given instance of R by storing projections of the instance. This section examines the use of decompositions through several examples.

We begin with the Hourly_Emps example from Section 15.3.1. This relation has attributes $SNLRWH$ and two FDs: $S \rightarrow SNLRWH$ and $R \rightarrow W$. Since R is not a key and W is not part of any key, the second dependency causes a violation of 3NF.

The alternative design consisted of replacing Hourly_Emps with two relations having attributes $SNLRH$ and RW. $S \rightarrow SNLRH$ holds over $SNLRH$, and S is a key. $R \rightarrow W$ holds over RW, and R is a key for RW. The only other dependencies that hold over these schemas are those obtained by augmentation. Thus both schemas are in BCNF.

Our decision to decompose $SNLRWH$ into $SNLRH$ and RW, rather than, say, $SNLR$ and $LRWH$, was not just a good guess. It was guided by the observation that the dependency $R \rightarrow W$ caused the violation of 3NF; the most natural way to deal with this violation is to remove the attribute W from this schema. To compensate for removing W from the main schema, we can add a relation RW, because each R value is associated with at most one W value according to the FD $R \rightarrow W$.

A very important question must be asked at this point: If we replace a legal instance r of relation schema $SNLRWH$ with its projections on $SNLRH$ (r_1) and RW (r_2), can we recover r from r_1 and r_2? The decision to decompose $SNLRWH$ into $SNLRH$ and RW is equivalent to saying that we will store instances r_1 and r_2 instead of r. However, it is the instance r that captures the intended entities or relationships. If we cannot compute r from r_1 and r_2, our attempt to deal with redundancy has effectively thrown out the baby with the bathwater. We consider this issue in more detail below.

15.6.1 Lossless-Join Decomposition

Let R be a relation schema and let F be a set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a **lossless-join decomposition with respect to F** if for every instance r of R that satisfies the dependencies in F, $\pi_X(r) \bowtie \pi_Y(r) = r$.

This definition can easily be extended to cover a decomposition of R into more than two relations. It is easy to see that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$ always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3

Instance r

S	P
s1	p1
s2	p2
s3	p1

$\pi_{SP}(r)$

P	D
p1	d1
p2	d2
p1	d3

$\pi_{PD}(r)$

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

$\pi_{SP}(r) \bowtie \pi_{PD}(r)$

Figure 15.11 Instances Illustrating Lossy Decompositions

By replacing the instance r shown in Figure 15.11 with the instances $\pi_{SP}(r)$ and $\pi_{PD}(r)$, we lose some information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that the relationships (s_1, p_1, d_3) and (s_3, p_1, d_1) do not hold. The decomposition of schema *SPD* into *SP* and *PD* is therefore a 'lossy' decomposition if the instance r shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. (Observe the similarities between this example and the Contracts relationship set in Section 2.5.3.)

All decompositions used to eliminate redundancy **must** *be lossless.* The following simple test is very useful:

> Let R be a relation and F be a set of FDs that hold over R. The decomposition of R into relations with attribute sets R_1 and R_2 is lossless if and only if F^+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ or the FD $R_1 \cap R_2 \rightarrow R_2$.

In other words, the attributes common to R_1 and R_2 must contain a key for either R_1 or R_2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join.[2] If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the Hourly_Emps relation again. It has attributes *SNLRWH*, and the FD $R \rightarrow W$ causes a violation of 3NF. We dealt with this violation by decomposing the relation into *SNLRH* and *RW*. Since R is common to both decomposed relations, and $R \rightarrow W$ holds, this decomposition is lossless-join.

This example illustrates a general observation:

> If an FD $X \rightarrow Y$ holds over a relation R and $X \cap Y$ is empty, the decomposition of R into $R - Y$ and XY is lossless.

X appears in both $R-Y$ (since $X \cap Y$ is empty) and XY, and it is a key for XY. Thus, the above observation follows from the test for a lossless-join decomposition.

Another important observation has to do with repeated decompositions. Suppose that a relation R is decomposed into *R1* and *R2* through a lossless-join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition. Then the decomposition of R into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then joining *R1* and *R2*, we can recover R.

15.6.2 Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *CSJDPQV* from Section 15.4.1. The given FDs are $C \rightarrow CSJDPQV$, $JP \rightarrow C$, and $SD \rightarrow P$. Because *SD* is not a key the dependency $SD \rightarrow P$ causes a violation of BCNF.

We can decompose Contracts into two relations with schemas *CSJDQV* and *SDP* to address this violation; the decomposition is lossless-join. There is one subtle problem, however. We can enforce the integrity constraint $JP \rightarrow C$ easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same *JP* values (as the inserted tuple) but different C values. Once we decompose Contracts into *CSJDQV* and *SDP*, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into *CSJDQV*. We say that this decomposition is not dependency-preserving.

[2]It is necessary only if we assume that only functional dependencies can be specified as integrity constraints. See Exercise 15.8.

Intuitively, a *dependency-preserving decomposition* allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To define dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let R be a relation schema that is decomposed into two schemas with attribute sets X and Y, and let F be a set of FDs over R. The **projection of F on** X is the set of FDs in the closure F^+ (not just F!) that involve only attributes in X. We will denote the projection of F on attributes X as F_X. Note that a dependency $U \rightarrow V$ in F^+ is in F_X only if *all* the attributes in U and V are in X.

The decomposition of relation schema R with FDs F into schemas with attribute sets X and Y is **dependency-preserving** if $(F_X \cup F_Y)^+ = F^+$. That is, if we take the dependencies in F_X and F_Y and compute the closure of their union, we get back all dependencies in the closure of F. Therefore, we need to enforce only the dependencies in F_X and F_Y; all FDs in F^+ are then sure to be satisfied. To enforce F_X, we need to examine only relation X (on inserts to that relation). To enforce F_Y, we need to examine only relation Y.

To appreciate the need to consider the closure F^+ while computing the projection of F, suppose that a relation R with attributes ABC is decomposed into relations with attributes AB and BC. The set F of FDs over R includes $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$. Of these, $A \rightarrow B$ is in F_{AB} and $B \rightarrow C$ is in F_{BC}. But is this decomposition dependency-preserving? What about $C \rightarrow A$? This dependency is not implied by the dependencies listed (thus far) for F_{AB} and F_{BC}.

The closure of F contains all dependencies in F plus $A \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. Consequently, F_{AB} also contains $B \rightarrow A$, and F_{BC} contains $C \rightarrow B$. Thus, $F_{AB} \cup F_{BC}$ contains $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. The closure of the dependencies in F_{AB} and F_{BC} now includes $C \rightarrow A$ (which follows from $C \rightarrow B$, $B \rightarrow A$, and transitivity). Thus, the decomposition preserves the dependency $C \rightarrow A$.

A direct application of the definition gives us a straightforward algorithm for testing whether a decomposition is dependency-preserving. (This algorithm is exponential in the size of the dependency set; a polynomial algorithm is available, although we will not discuss it.)

We began this section with an example of a lossless-join decomposition that was not dependency-preserving. Other decompositions are dependency-preserving, but not lossless. A simple example consists of a relation ABC with FD $A \rightarrow B$ that is decomposed into AB and BC.

15.7 NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and decompositions in database design, we now consider algorithms for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas. Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas. However, there is always a dependency-preserving, lossless-join decomposition into a collection of 3NF relation schemas.

15.7.1 Decomposition into BCNF

We now present an algorithm for decomposing a relation schema R into a collection of BCNF relation schemas:

1. Suppose that R is not in BCNF. Let $X \subset R$, A be a single attribute in R, and $X \rightarrow A$ be an FD that causes a violation of BCNF. Decompose R into $R - A$ and XA.

2. If either $R - A$ or XA is not in BCNF, decompose them further by a recursive application of this algorithm.

$R - A$ denotes the set of attributes other than A in R, and XA denotes the union of attributes in X and A. Since $X \rightarrow A$ violates BCNF, it is not a trivial dependency; further, A is a single attribute. Therefore, A is not in X; that is, $X \cap A$ is empty. Thus, each decomposition carried out in Step (1) is lossless-join.

The set of dependencies associated with $R - A$ and XA is the projection of F onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step (2). Since a decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relation schemas that are all in BCNF. Further, joining instances of the (two or more) relations obtained through this algorithm will yield precisely the corresponding instance of the original relation (i.e., the decomposition into a collection of relations that are each in BCNF is a lossless-join decomposition).

Consider the Contracts relation with attributes $CSJDPQV$ and key C. We are given FDs $JP \rightarrow C$ and $SD \rightarrow P$. By using the dependency $SD \rightarrow P$ to guide the decomposition, we get the two schemas SDP and $CSJDQV$. SDP is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: $J \rightarrow S$. This means that the schema $CSJDQV$ is not in BCNF. So we decompose it further into JS and $CJDQV$. $C \rightarrow JDQV$ holds over $CJDQV$; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDs contain a key in the left side. Thus, each of the schemas *SDP*, *JS*, and *CJDQV* is in BCNF, and this collection of schemas also represents a lossless-join decomposition of *CSJDQV*.

The steps in this decomposition process can be visualized as a tree, as shown in Figure 15.12. The root is the original relation *CSJDPQV*, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, *SDP*, *JS*, and *CSDQV*. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.

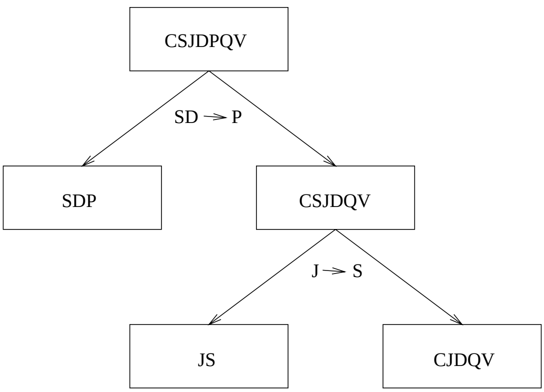

Figure 15.12 Decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV*

Redundancy in BCNF Revisited

The decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV* is not dependency-preserving. Intuitively, dependency $JP \rightarrow C$ cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes *CJP*. In effect, this solution amounts to storing some information redundantly in order to make the dependency enforcement cheaper.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation instances for *SDP* and *CJDQV* and project onto the attributes *CJP*, we must get exactly the instance stored in the relation with schema *CJP*. We saw in Section 15.5.1 that there is no such redundancy within a single BCNF relation. The current example shows that redundancy can still occur across relations, even though there is no redundancy within a relation.

Alternatives in Decomposing to BCNF

Suppose that several dependencies violate BCNF. Depending on which of these dependencies we choose to guide the next decomposition step, we may arrive at quite different collections of BCNF relations. Consider Contracts. We just decomposed it into SDP, JS, and $CJDQV$. Suppose that we choose to decompose the original relation $CSJDPQV$ into JS and $CJDPQV$, based on the FD $J \rightarrow S$. The only dependencies that hold over $CJDPQV$ are $JP \rightarrow C$ and the key dependency $C \rightarrow CJDPQV$. Since JP is a key, $CJDPQV$ is in BCNF. Thus, the schemas JS and $CJDPQV$ represent a lossless-join decomposition of Contracts into BCNF relations.

The lesson to be learned here is that the theory of dependencies can tell us when there is redundancy and give us clues about possible decompositions to address the problem, but it cannot discriminate between decomposition alternatives. A designer has to consider the alternatives and choose one based on the semantics of the application.

BCNF and Dependency-Preservation

Sometimes, there simply is no decomposition into BCNF that is dependency-preserving. As an example, consider the relation schema SBD, in which a tuple denotes that sailor S has reserved boat B on date D. If we have the FDs $SB \rightarrow D$ (a sailor can reserve a given boat for at most one day) and $D \rightarrow B$ (on any given day at most one boat can be reserved), SBD is not in BCNF because D is not a key. If we try to decompose it, however, we cannot preserve the dependency $SB \rightarrow D$.

15.7.2 Decomposition into 3NF *

Clearly, the approach that we outlined for lossless-join decomposition into BCNF will also give us a lossless-join decomposition into 3NF. (Typically, we can stop a little earlier if we are satisfied with a collection of 3NF relations.) But this approach does not ensure dependency-preservation.

A simple modification, however, yields a decomposition into 3NF relations that is lossless-join and dependency-preserving. Before we describe this modification, we need to introduce the concept of a minimal cover for a set of FDs.

Minimal Cover for a Set of FDs

A **minimal cover** for a set F of FDs is a set G of FDs such that:

1. Every dependency in G is of the form $X \rightarrow A$, where A is a single attribute.
2. The closure F^+ is equal to the closure G^+.

3. If we obtain a set H of dependencies from G by deleting one or more dependencies, or by deleting attributes from a dependency in G, then $F^+ \neq H^+$.

Intuitively, a minimal cover for a set F of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required in order for the closure to be equal to F^+.

As an example, let F be the set of dependencies:

$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow G$, $EF \rightarrow H$, and $ACDF \rightarrow EG$.

First, let us rewrite $ACDF \rightarrow EG$ so that every right side is a single attribute:

$ACDF \rightarrow E$ and $ACDF \rightarrow G$.

Next consider $ACDF \rightarrow G$. This dependency is implied by the following FDs:

$A \rightarrow B$, $ABCD \rightarrow E$, and $EF \rightarrow G$.

Therefore, we can delete it. Similarly, we can delete $ACDF \rightarrow E$. Next consider $ABCD \rightarrow E$. Since $A \rightarrow B$ holds, we can replace it with $ACD \rightarrow E$. (At this point, the reader should verify that each remaining FD is minimal and required.) Thus, a minimal cover for F is the set:

$A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, and $EF \rightarrow H$.

The preceding example suggests a general algorithm for obtaining a minimal cover of a set F of FDs:

1. **Put the FDs in a standard form:** Obtain a collection G of equivalent FDs with a single attribute on the right side (using the decomposition axiom).

2. **Minimize the left side of each FD:** For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to F^+.

3. **Delete redundant FDs:** Check each remaining FD in G to see if it can be deleted while preserving equivalence to F^+.

Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.

More important, it is necessary to minimize the left sides of FDs *before* checking for redundant FDs. If these two steps are reversed, the final set of FDs could still contain some redundant FDs (i.e., not be a minimal cover), as the following example illustrates. Let F be the set of dependencies, each of which is already in the standard form:

$ABCD \to E$, $E \to D$, $A \to B$, and $AC \to D$.

Observe that none of these FDs is redundant; if we checked for redundant FDs first, we would get the same set of FDs F. The left side of $ABCD \to E$ can be replaced by AC while preserving equivalence to F^+, and we would stop here if we checked for redundant FDs in F before minimizing the left sides. However, the set of FDs we have is not a minimal cover:

$AC \to E$, $E \to D$, $A \to B$, and $AC \to D$.

From transitivity, the first two FDs imply the last FD, which can therefore be deleted while preserving equivalence to F^+. The important point to note is that $AC \to D$ becomes redundant only after we replace $ABCD \to E$ with $AC \to E$. If we minimize left sides of FDs first and then check for redundant FDs, we are left with the first three FDs in the preceding list, which is indeed a minimal cover for F.

Dependency-Preserving Decomposition into 3NF

Returning to the problem of obtaining a lossless-join, dependency-preserving decomposition into 3NF relations, let R be a relation with a set F of FDs that is a minimal cover, and let $R_1, R_2, \ldots, R_n$ be a lossless-join decomposition of R. For $1 \leq i \leq n$, suppose that each R_i is in 3NF and let F_i denote the projection of F onto the attributes of R_i. Do the following:

- Identify the set N of dependencies in F that are not **preserved**, that is, not included in the closure of the union of F_is.
- For each FD $X \to A$ in N, create a relation schema XA and add it to the decomposition of R.

Obviously, every dependency in F is preserved if we replace R by the R_is plus the schemas of the form XA added in this step. The R_is are given to be in 3NF. We can show that each of the schemas XA is in 3NF as follows: Since $X \to A$ is in the minimal cover F, $Y \to A$ does not hold for any Y that is a strict subset of X. Therefore, X is a key for XA. Further, if any other dependencies hold over XA, the right side can involve only attributes in X because A is a single attribute (because $X \to A$ is an FD in a minimal cover). Since X is a key for XA, none of these additional dependencies causes a violation of 3NF (although they might cause a violation of BCNF).

As an optimization, if the set N contains several FDs with the same left side, say, $X \to A_1, X \to A_2, \ldots, X \to A_n$, we can replace them with a single equivalent FD $X \to A_1 \ldots A_n$. Therefore, we produce one relation schema $XA_1 \ldots A_n$, instead of several schemas $XA_1, \ldots, XA_n$, which is generally preferable.

Consider the Contracts relation with attributes $CSJDPQV$ and FDs $JP \rightarrow C$, $SD \rightarrow P$, and $J \rightarrow S$. If we decompose $CSJDPQV$ into SDP and $CSJDQV$, then SDP is in BCNF, but $CSJDQV$ is not even in 3NF. So we decompose it further into JS and $CJDQV$. The relation schemas SDP, JS, and $CJDQV$ are in 3NF (in fact, in BCNF), and the decomposition is lossless-join. However, the dependency $JP \rightarrow C$ is not preserved. This problem can be addressed by adding a relation schema CJP to the decomposition.

3NF Synthesis

We have assumed that the design process starts with an ER diagram, and that our use of FDs is primarily to guide decisions about decomposition. The algorithm for obtaining a lossless-join, dependency-preserving decomposition was presented in the previous section from this perspective—a lossless-join decomposition into 3NF is straightforward, and the algorithm addresses dependency-preservation by adding extra relation schemas.

An alternative approach, called **synthesis**, is to take all the attributes over the original relation R and a minimal cover F for the FDs that hold over it, and to add a relation schema XA to the decomposition of R for each FD $X \rightarrow A$ in F.

The resulting collection of relation schemas is in 3NF and preserves all FDs. If it is not a lossless-join decomposition of R, we can make it so by adding a relation schema that contains just those attributes that appear in some key. This algorithm gives us a lossless-join, dependency-preserving decomposition into 3NF, and has polynomial complexity—polynomial algorithms are available for computing minimal covers, and a key can be found in polynomial time (even though finding all keys is known to be NP-complete). The existence of a polynomial algorithm for obtaining a lossless-join, dependency-preserving decomposition into 3NF is surprising when we consider that testing whether a given schema is in 3NF is NP-complete.

As an example, consider a relation ABC with FDs $F = \{A \rightarrow B,\ C \rightarrow B\}$. The first step yields the relation schemas AB and BC. This is not a lossless-join decomposition of ABC; $AB \cap BC$ is B, and neither $B \rightarrow A$ nor $B \rightarrow C$ is in F^+. If we add a schema AC, we have the lossless-join property as well. Although the collection of relations AB, BC, and AC is a dependency-preserving, lossless-join decomposition of ABC, we obtained it through a process of *synthesis*, rather than through a process of repeated decomposition. We note that the decomposition produced by the synthesis approach is heavily dependent on the minimal cover that is used.

As another example of the synthesis approach, consider the Contracts relation with attributes $CSJDPQV$ and the following FDs:

$$C \rightarrow CSJDPQV,\ JP \rightarrow C,\ SD \rightarrow P, \text{ and } J \rightarrow S.$$

This set of FDs is not a minimal cover, and so we must find one. We first replace $C \rightarrow CSJDPQV$ with the FDs:

$$C \rightarrow S,\ C \rightarrow J,\ C \rightarrow D,\ C \rightarrow P,\ C \rightarrow Q, \text{ and } C \rightarrow V.$$

The FD $C \rightarrow P$ is implied by $C \rightarrow S$, $C \rightarrow D$, and $SD \rightarrow P$; so we can delete it. The FD $C \rightarrow S$ is implied by $C \rightarrow J$ and $J \rightarrow S$; so we can delete it. This leaves us with a minimal cover:

$$C \rightarrow J,\ C \rightarrow D,\ C \rightarrow Q,\ C \rightarrow V,\ JP \rightarrow C,\ SD \rightarrow P, \text{ and } J \rightarrow S.$$

Using the algorithm for ensuring dependency-preservation, we obtain the relational schema CJ, CD, CQ, CV, CJP, SDP, and JS. We can improve this schema by combining relations for which C is the key into $CDJPQV$. In addition, we have SDP and JS in our decomposition. Since one of these relations ($CDJPQV$) is a superkey, we are done.

Comparing this decomposition with the one that we obtained earlier in this section, we find that they are quite close, with the only difference being that one of them has $CDJPQV$ instead of CJP and $CJDQV$. In general, however, there could be significant differences.

Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

15.8 OTHER KINDS OF DEPENDENCIES *

FDs are probably the most common and important kind of constraint from the point of view of database design. However, there are several other kinds of dependencies. In particular, there is a well-developed theory for database design using *multivalued dependencies* and *join dependencies*. By taking such dependencies into account, we can identify potential redundancy problems that cannot be detected using FDs alone.

This section illustrates the kinds of redundancy that can be detected using multivalued dependencies. Our main observation, however, is that simple guidelines (which can be checked using only FD reasoning) can tell us whether we even need to worry about complex constraints such as multivalued and join dependencies. We also comment on the role of *inclusion dependencies* in database design.

15.8.1 Multivalued Dependencies

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*. The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course. There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor. The instance shown in Figure 15.13 illustrates this situation.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

Figure 15.13 BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be expressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively, the **multivalued dependency** $X \rightarrow\rightarrow Y$ is said to hold over R if, in every legal instance r of R, each X value is associated with a set of Y values and this set is independent of the values in the other attributes.

Formally, if the MVD $X \rightarrow\rightarrow Y$ holds over R and $Z = R - XY$, the following must be true for every legal instance r of R:

> If $t_1 \in r$, $t_2 \in r$ and $t_1.X = t_2.X$, then there must be some $t_3 \in r$ such that $t_1.XY = t_3.XY$ and $t_2.Z = t_3.Z$.

Figure 15.14 illustrates this definition. If we are given the first two tuples and told that the MVD $X \rightarrow\rightarrow Y$ holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the first two tuples—treating the first tuple as t_2 and the second tuple as t_1—we can deduce that the tuple t_4 must also be in the relation instance.

X	Y	Z	
a	b_1	c_1	— tuple t_1
a	b_2	c_2	— tuple t_2
a	b_1	c_2	— tuple t_3
a	b_2	c_1	— tuple t_4

Figure 15.14 Illustration of MVD Definition

This table suggests another way to think about MVDs: If $X \rightarrow\rightarrow Y$ holds over R, then $\pi_{YZ}(\sigma_{X=x}(R)) = \pi_Y(\sigma_{X=x}(R)) \times \pi_Z(\sigma_{X=x}(R))$ in every legal instance of R, for any value x that appears in the X column of R. In other words, consider groups of tuples in R with the same X-value, for each X-value. In each such group consider the projection onto the attributes YZ. This projection must be equal to the cross-product of the projections onto Y and Z. That is, for a given X-value, the Y-values and Z-values are independent. (From this definition it is easy to see that $X \rightarrow\rightarrow Y$ must hold whenever $X \rightarrow Y$ holds. If the FD $X \rightarrow Y$ holds, there is exactly one Y-value for a given X-value, and the conditions in the MVD definition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our *CTB* example, the constraint that course texts are independent of instructors can be expressed as $C \rightarrow\rightarrow T$. In terms of the definition of MVDs, this constraint can be read as follows:

> "If (there is a tuple showing that) C is taught by teacher T,
> and (there is a tuple showing that) C has book B as text,
> then (there is a tuple showing that) C is taught by T and has text B.

Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus five additional rules. Three of the additional rules involve only MVDs:

- **MVD Complementation:** If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$.
- **MVD Augmentation:** If $X \rightarrow\rightarrow Y$ and $W \supseteq Z$, then $WX \rightarrow\rightarrow YZ$.
- **MVD Transitivity:** If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow (Z - Y)$.

As an example of the use of these rules, since we have $C \rightarrow\rightarrow T$ over CTB, MVD complementation allows us to infer that $C \rightarrow\rightarrow CTB - CT$ as well, that is, $C \rightarrow\rightarrow B$. The remaining two rules relate FDs and MVDs:

- **Replication:** If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$.
- **Coalescence:** If $X \rightarrow\rightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

Observe that replication states that every FD is also an MVD.

15.8.2 Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let R be a relation schema, X and Y be nonempty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in **fourth normal form (4NF)** if for every MVD $X \rightarrow\rightarrow Y$ that holds over R, one of the following statements is true:

- $Y \subseteq X$ or $XY = R$, or
- X is a superkey.

In reading this definition, it is important to understand that the definition of a key has not changed—the key must uniquely determine all attributes through FDs alone. $X \rightarrow\rightarrow Y$ is a **trivial MVD** if $Y \subseteq X \subseteq R$ or $XY = R$; such MVDs always hold.

The relation CTB is not in 4NF because $C \rightarrow\rightarrow T$ is a nontrivial MVD and C is not a key. We can eliminate the resulting redundancy by decomposing CTB into CT and CB; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions—detected using only FD information!—under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

> If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation.* This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

We illustrate this point using an example. Consider a relation schema $ABCD$ and suppose that the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ are given. Considering only these dependencies, this relation schema appears to be a counter example to the result. The relation has a simple key, appears to be in BCNF, and yet is not in 4NF because $B \rightarrow\rightarrow C$ causes a violation of the 4NF conditions. But let's take a closer look.

Figure 15.15 shows three tuples from an instance of $ABCD$ that satisfies the given MVD $B \rightarrow\rightarrow C$. From the definition of an MVD, given tuples t_1 and t_2, it follows

B	C	A	D	
b	c_1	a_1	d_1	— tuple t_1
b	c_2	a_2	d_2	— tuple t_2
b	c_1	a_2	d_2	— tuple t_3

Figure 15.15 Three Tuples from a Legal Instance of $ABCD$

that tuple t_3 must also be included in the instance. Consider tuples t_2 and t_3. From the given FD $A \rightarrow BCD$ and the fact that these tuples have the same A-value, we can deduce that $c_1 = c_2$. Thus, we see that the FD $B \rightarrow C$ must hold over $ABCD$ whenever the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ hold. If $B \rightarrow C$ holds, the relation $ABCD$ is not in BCNF (unless additional FDs hold that make B a key)!

Thus, the apparent counter example is really not a counter example—rather, it illustrates the importance of correctly identifying all FDs that hold over a relation. In this example $A \rightarrow BCD$ is not the only FD; the FD $B \rightarrow C$ also holds but was not identified initially. Given a set of FDs and MVDs, the inference rules can be used to infer additional FDs (and MVDs); to apply the Date-Fagin result without first using the MVD inference rules, we must be certain that we have identified all the FDs.

In summary, the Date-Fagin result offers a convenient way to check that a relation is in 4NF (without reasoning about MVDs) if we are confident that we have identified all FDs. At this point the reader is invited to go over the examples we have discussed in this chapter and see if there is a relation that is not in 4NF.

15.8.3 Join Dependencies

A join dependency is a further generalization of MVDs. A **join dependency** (JD) $\bowtie \{R_1, \ldots, R_n\}$ is said to hold over a relation R if $R_1, \ldots, R_n$ is a lossless-join decomposition of R.

An MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie$ {XY, X(R−Y)}. As an example, in the *CTB* relation, the MVD $C \rightarrow\rightarrow T$ can be expressed as the join dependency $\bowtie$ {CT, CB}.

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

15.8.4 Fifth Normal Form

A relation schema R is said to be in **fifth normal form (5NF)** if for every JD $\bowtie \{R_1, \ldots, R_n\}$ that holds over R, one of the following statements is true:

- $R_i = R$ for some i, or
- The JD is implied by the set of those FDs over R in which the left side is a key for R.

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of R into $\{R_1, \ldots, R_n\}$ is lossless-join whenever the **key dependencies** (FDs in which the left side is a key for R) hold. $\bowtie \{R_1, \ldots, R_n\}$ is a **trivial JD** if $R_i = R$ for some i; such a JD always holds.

The following result, also due to Date and Fagin, identifies conditions—again, detected using only FD information—under which we can safely ignore JD information.

> If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF *without ever identifying the MVDs and JDs that may hold over the relation.*

15.8.5 Inclusion Dependencies

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about. In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design (beyond the ER design stage).

Informally, an inclusion dependency is a statement of the form that some columns of a relation are contained in other columns (usually of a second relation). A foreign key constraint is an example of an inclusion dependency; the referring column(s) in one relation must be contained in the primary key column(s) of the referenced relation. As another example, if R and S are two relations obtained by translating two entity sets such that every R entity is also an S entity, we would have an inclusion dependency; projecting R on its key attributes yields a relation that is contained in the relation obtained by projecting S on its key attributes.

The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency. For example, if we have an inclusion dependency $AB \subseteq CD$, while decomposing the relation schema containing AB, we should ensure that at least one of the schemas obtained in the decomposition contains both A and B. Otherwise, we cannot check the inclusion dependency $AB \subseteq CD$ without reconstructing the relation containing AB.

Most inclusion dependencies in practice are *key-based*, that is, involve only keys. Foreign key constraints are a good example of key-based inclusion dependencies. An ER diagram that involves ISA hierarchies also leads to key-based inclusion dependencies. If all inclusion dependencies are key-based, we rarely have to worry about splitting attribute groups that participate in inclusions, since decompositions usually do not split the primary key. Note, however, that going from 3NF to BCNF always involves splitting some key (hopefully not the primary key!), since the dependency guiding the split is of the form $X \rightarrow A$ where A is part of a key.

15.9 POINTS TO REVIEW

- *Redundancy*, storing the same information several times in a database, can result in *update anomalies* (all copies need to be updated), *insertion anomalies* (certain information cannot be stored unless other information is stored as well), and *deletion anomalies* (deleting some information means loss of other information as well). We can reduce redundancy by replacing a relation schema R with several smaller relation schemas. This process is called *decomposition*. **(Section 15.1)**

- A *functional dependency* $X \rightarrow Y$ is a type of IC. It says that if two tuples agree upon (i.e., have the same) values in the X attributes, then they also agree upon the values in the Y attributes. **(Section 15.2)**

- FDs can help to refine subjective decisions made during conceptual design. **(Section 15.3)**

- An FD f *is implied by* a set F of FDs if for all relation instances where F holds, f also holds. The closure of a set F of FDs is the set of all FDs F^+ implied by F. *Armstrong's Axioms* are a sound and complete set of rules to generate all FDs in the closure. An FD $X \rightarrow Y$ is *trivial* if X contains only attributes that also appear in Y. The *attribute closure* X^+ of a set of attributes X with respect to a set of FDs F is the set of attributes A such that $X \rightarrow A$ can be inferred using Armstrong's Axioms. **(Section 15.4)**

- A *normal form* is a property of a relation schema indicating the type of redundancy that the relation schema exhibits. If a relation schema is in *Boyce-Codd normal form* (BCNF), then the only nontrivial FDs are key constraints. If a relation is in third normal form (3NF), then all nontrivial FDs are key constraints or their right side is part of a candidate key. Thus, every relation that is in BCNF is also in 3NF, but not vice versa. **(Section 15.5)**

- A decomposition of a relation schema R into two relation schemas X and Y is a *lossless-join* decomposition with respect to a set of FDs F if for any instance r of R that satisfies the FDs in F, $\pi_X(r) \bowtie \pi_Y(r) = r$. The decomposition of R into X and Y is lossless-join if and only if F^+ contains either $X \cap Y \rightarrow X$ or the FD $X \cap Y \rightarrow Y$. The decomposition is *dependency-preserving* if we can enforce all FDs that are given to hold on R by simply enforcing FDs on X and FDs on Y independently (i.e., without joining X and Y). **(Section 15.6)**

- There is an algorithm to obtain a lossless-join decomposition of a relation into a collection of BCNF relation schemas, but sometimes there is no dependency-preserving decomposition into BCNF schemas. We also discussed an algorithm for decomposing a relation schema into a collection of 3NF relation schemas. There is always a lossless-join, dependency-preserving decomposition into a collection of 3NF relation schemas. A *minimal cover* of a set of FDs is an equivalent set of FDs that has certain minimality properties (intuitively, the set of FDs is as small as possible). Instead of decomposing a relation schema, we can also *synthesize* a corresponding collection of 3NF relation schemas. **(Section 15.7)**

- Other kinds of dependencies include *multivalued dependencies*, *join dependencies*, and *inclusion dependencies*. Fourth and fifth normal forms are more stringent than BCNF, and eliminate redundancy due to multivalued and join dependencies, respectively. **(Section 15.8)**

EXERCISES

Exercise 15.1 Briefly answer the following questions.

1. Define the term *functional dependency*.
2. Give a set of FDs for the relation schema *R(A,B,C,D)* with primary key AB under which R is in 1NF but not in 2NF.

3. Give a set of FDs for the relation schema *R(A,B,C,D)* with primary key AB under which R is in 2NF but not in 3NF.
4. Consider the relation schema *R(A,B,C)*, which has the FD $B \rightarrow C$. If A is a candidate key for R, is it possible for R to be in BCNF? If so, under what conditions? If not, explain why not.
5. Suppose that we have a relation schema *R(A,B,C)* representing a relationship between two entity sets with keys A and B, respectively, and suppose that R has (among others) the FDs $A \rightarrow B$ and $B \rightarrow A$. Explain what such a pair of dependencies means (i.e., what they imply about the relationship that the relation models).

Exercise 15.2 Consider a relation R with five attributes $ABCDE$. You are given the following dependencies: $A \rightarrow B$, $BC \rightarrow E$, and $ED \rightarrow A$.

1. List all keys for R.
2. Is R in 3NF?
3. Is R in BCNF?

Exercise 15.3 Consider the following collection of relations and dependencies. Assume that each relation is obtained through decomposition from a relation with attributes $ABCDEFGHI$ and that all the known dependencies over relation $ABCDEFGHI$ are listed for each question. (The questions are independent of each other, obviously, since the given dependencies over $ABCDEFGHI$ are different.) For each (sub) relation: (a) State the strongest normal form that the relation is in. (b) If it is not in BCNF, decompose it into a collection of BCNF relations.

1. *R1(A,C,B,D,E)*, $A \rightarrow B$, $C \rightarrow D$
2. *R2(A,B,F)*, $AC \rightarrow E$, $B \rightarrow F$
3. *R3(A,D)*, $D \rightarrow G$, $G \rightarrow H$
4. *R4(D,C,H,G)*, $A \rightarrow I$, $I \rightarrow A$
5. *R5(A,I,C,E)*

Exercise 15.4 Suppose that we have the following three tuples in a legal instance of a relation schema S with three attributes ABC (listed in order): (1,2,3), (4,2,3), and (5,3,3).

1. Which of the following dependencies can you infer does *not* hold over schema S?

   (a) $A \rightarrow B$ (b) $BC \rightarrow A$ (c) $B \rightarrow C$

2. Can you identify any dependencies that hold over S?

Exercise 15.5 Suppose you are given a relation R with four attributes, $ABCD$. For each of the following sets of FDs, assuming those are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) Identify the best normal form that R satisfies (1NF, 2NF, 3NF, or BCNF). (c) If R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

1. $C \rightarrow D$, $C \rightarrow A$, $B \rightarrow C$

2. $B \rightarrow C$, $D \rightarrow A$
3. $ABC \rightarrow D$, $D \rightarrow A$
4. $A \rightarrow B$, $BC \rightarrow D$, $A \rightarrow C$
5. $AB \rightarrow C$, $AB \rightarrow D$, $C \rightarrow A$, $D \rightarrow B$

Exercise 15.6 Consider the attribute set $R = ABCDEGH$ and the FD set $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow G\}$.

1. For each of the following attribute sets, do the following: (i) Compute the set of dependencies that hold over the set and write down a minimal cover. (ii) Name the strongest normal form that is not violated by the relation containing these attributes. (iii) Decompose it into a collection of BCNF relations if it is not in BCNF.

   (a) ABC (b) $ABCD$ (c) $ABCEG$ (d) $DCEGH$ (e) $ACEH$

2. Which of the following decompositions of $R = ABCDEG$, with the same set of dependencies F, is (a) dependency-preserving? (b) lossless-join?

   (a) $\{AB, BC, ABDE, EG\}$

   (b) $\{ABC, ACDE, ADG\}$

Exercise 15.7 Let R be decomposed into R_1, R_2, ..., R_n. Let F be a set of FDs on R.

1. Define what it means for F to *be preserved* in the set of decomposed relations.
2. Describe a polynomial-time algorithm to test dependency-preservation.
3. Projecting the FDs stated over a set of attributes X onto a subset of attributes Y requires that we consider the closure of the FDs. Give an example where considering the closure is important in testing dependency-preservation; that is, considering just the given FDs gives incorrect results.

Exercise 15.8 Consider a relation R that has three attributes ABC. It is decomposed into relations R_1 with attributes AB and R_2 with attributes BC.

1. State the definition of a lossless-join decomposition with respect to this example. Answer this question concisely by writing a relational algebra equation involving R, R_1, and R_2.
2. Suppose that $B \rightarrow\rightarrow C$. Is the decomposition of R into R_1 and R_2 lossless-join? Reconcile your answer with the observation that neither of the FDs $R_1 \cap R_2 \rightarrow R_1$ nor $R_1 \cap R_2 \rightarrow R_2$ hold, in light of the simple test offering a necessary and sufficient condition for lossless-join decomposition into two relations in Section 15.6.1.
3. If you are given the following instances of R_1 and R_2, what can you say about the instance of R from which these were obtained? Answer this question by listing tuples that are definitely in R and listing tuples that are possibly in R.

   Instance of $R_1 = \{(5,1), (6,1)\}$
   Instance of $R_2 = \{(1,8), (1,9)\}$

   Can you say that attribute B definitely *is* or *is not* a key for R?

Exercise 15.9 Suppose you are given a relation *R(A,B,C,D)*. For each of the following sets of FDs, assuming they are the only dependencies that hold for *R*, do the following: (a) Identify the candidate key(s) for *R*. (b) State whether or not the proposed decomposition of *R* into smaller relations is a good decomposition, and briefly explain why or why not.

1. $B \rightarrow C$, $D \rightarrow A$; decompose into BC and AD.
2. $AB \rightarrow C$, $C \rightarrow A$, $C \rightarrow D$; decompose into ACD and BC.
3. $A \rightarrow BC$, $C \rightarrow AD$; decompose into ABC and AD.
4. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB and ACD.
5. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB, AD and CD.

Exercise 15.10 Suppose that we have the following four tuples in a relation S with three attributes ABC: (1,2,3), (4,2,3), (5,3,3), (5,3,4). Which of the following functional ($\rightarrow$) and multivalued ($\rightarrow\rightarrow$) dependencies can you infer does *not* hold over relation S?

1. $A \rightarrow B$
2. $A \rightarrow\rightarrow B$
3. $BC \rightarrow A$
4. $BC \rightarrow\rightarrow A$
5. $B \rightarrow C$
6. $B \rightarrow\rightarrow C$

Exercise 15.11 Consider a relation R with five attributes $ABCDE$.

1. For each of the following instances of R, state whether (a) it violates the FD $BC \rightarrow D$, and (b) it violates the MVD $BC \rightarrow\rightarrow D$:
   (a) { } (i.e., empty relation)
   (b) {(a,2,3,4,5), (2,a,3,5,5)}
   (c) {(a,2,3,4,5), (2,a,3,5,5), (a,2,3,4,6)}
   (d) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5)}
   (e) {(a,2,3,4,5), (2,a,3,7,5), (a,2,3,4,6)}
   (f) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5), (a,2,3,6,6)}
   (g) {(a,2,3,4,5), (a,2,3,6,5), (a,2,3,6,6), (a,2,3,4,6)}
2. If each instance for R listed above is legal, what can you say about the FD $A \rightarrow B$?

Exercise 15.12 JDs are motivated by the fact that sometimes a relation that cannot be decomposed into two smaller relations in a lossless-join manner can be so decomposed into three or more relations. An example is a relation with attributes *supplier*, *part*, and *project*, denoted *SPJ*, with no FDs or MVDs. The JD $\bowtie \{SP, PJ, JS\}$ holds.

From the JD, the set of relation schemes *SP, PJ,* and *JS* is a lossless-join decomposition of *SPJ*. Construct an instance of *SPJ* to illustrate that no two of these schemes suffice.

Exercise 15.13 Consider a relation R with attributes $ABCDE$. Let the following FDs be given: $A \rightarrow BC$, $BC \rightarrow E$, and $E \rightarrow DA$. Similarly, let S be a relation with attributes $ABCDE$ and let the following FDs be given: $A \rightarrow BC$, $B \rightarrow E$, and $E \rightarrow DA$. (Only the second dependency differs from those that hold over R.) You do not know whether or which other (join) dependencies hold.

1. Is R in BCNF?
2. Is R in 4NF?
3. Is R in 5NF?
4. Is S in BCNF?
5. Is S in 4NF?
6. Is S in 5NF?

Exercise 15.14 Let us say that an FD $X \rightarrow Y$ is *simple* if Y is a single attribute.

1. Replace the FD $AB \rightarrow CD$ by the smallest equivalent collection of simple FDs.
2. Prove that every FD $X \rightarrow Y$ in a set of FDs F can be replaced by a set of simple FDs such that F^+ is equal to the closure of the new set of FDs.

Exercise 15.15 Prove that Armstrong's Axioms are sound and complete for FD inference. That is, show that repeated application of these axioms on a set F of FDs produces exactly the dependencies in F^+.

Exercise 15.16 Describe a linear-time (in the size of the set of FDs, where the size of each FD is the number of attributes involved) algorithm for finding the attribute closure of a set of attributes with respect to a set of FDs.

Exercise 15.17 Consider a scheme R with FDs F that is decomposed into schemes with attributes X and Y. Show that this is dependency-preserving if $F \subseteq (F_X \cup F_Y)^+$.

Exercise 15.18 Let R be a relation schema with a set F of FDs. Prove that the decomposition of R into R_1 and R_2 is lossless-join if and only if F^+ contains $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

Exercise 15.19 Prove that the optimization of the algorithm for lossless-join, dependency-preserving decomposition into 3NF relations (Section 15.7.2) is correct.

Exercise 15.20 Prove that the 3NF synthesis algorithm produces a lossless-join decomposition of the relation containing all the original attributes.

Exercise 15.21 Prove that an MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie \{XY,\ X(R-Y)\}$.

Exercise 15.22 Prove that if R has only one key, it is in BCNF if and only if it is in 3NF.

Exercise 15.23 Prove that if R is in 3NF and every key is simple, then R is in BCNF.

Exercise 15.24 Prove these statements:

1. If a relation scheme is in BCNF and at least one of its keys consists of a single attribute, it is also in 4NF.
2. If a relation scheme is in 3NF and each key has a single attribute, it is also in 5NF.

Exercise 15.25 Give an algorithm for testing whether a relation scheme is in BCNF. The algorithm should be polynomial in the size of the set of given FDs. (The *size* is the sum over all FDs of the number of attributes that appear in the FD.) Is there a polynomial algorithm for testing whether a relation scheme is in 3NF?

PROJECT-BASED EXERCISES

Exercise 15.26 Minibase provides a tool called Designview for doing database design using FDs. It lets you check whether a relation is in a particular normal form, test whether decompositions have nice properties, compute attribute closures, try several decomposition sequences and switch between them, generate SQL statements to create the final database schema, and so on.

1. Use Designview to check your answers to exercises that call for computing closures, testing normal forms, decomposing into a desired normal form, and so on.
2. (*Note to instructors: This exercise should be made more specific by providing additional details. See Appendix B.*) Apply Designview to a large, real-world design problem.

BIBLIOGRAPHIC NOTES

Textbook presentations of dependency theory and its use in database design include [3, 38, 436, 443, 656]. Good survey articles on the topic include [663, 355].

FDs were introduced in [156], along with the concept of 3NF, and axioms for inferring FDs were presented in [31]. BCNF was introduced in [157]. The concept of a legal relation instance and dependency satisfaction are studied formally in [279]. FDs were generalized to semantic data models in [674].

Finding a key is shown to be NP-complete in [432]. Lossless-join decompositions were studied in [24, 437, 546]. Dependency-preserving decompositions were studied in [61]. [68] introduced minimal covers. Decomposition into 3NF is studied by [68, 85] and decomposition into BCNF is addressed in [651]. [351] shows that testing whether a relation is in 3NF is NP-complete. [215] introduced 4NF and discussed decomposition into 4NF. Fagin introduced other normal forms in [216] (project-join normal form) and [217] (domain-key normal form). In contrast to the extensive study of vertical decompositions, there has been relatively little formal investigation of horizontal decompositions. [175] investigates horizontal decompositions.

MVDs were discovered independently by Delobel [177], Fagin [215], and Zaniolo [690]. Axioms for FDs and MVDs were presented in [60]. [516] shows that there is no axiomatization for JDs, although [575] provides an axiomatization for a more general class of dependencies. The sufficient conditions for 4NF and 5NF in terms of FDs that were discussed in Section 15.8 are from [171]. An approach to database design that uses dependency information to construct sample relation instances is described in [442, 443].