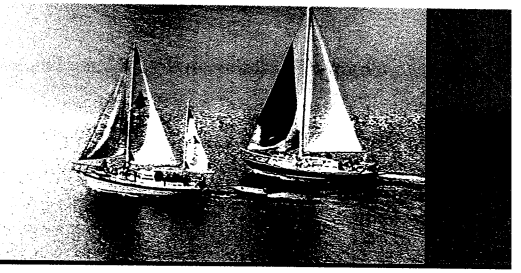
CHAPTER 8



Relational Database Design

In this chapter, we consider the problem of designing a schema for a relational database. Many of the issues in doing so are similar to design issues we considered in Chapter 7 using the E-R model.

In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate *normal form*. To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database. Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

In this chapter, we introduce a formal approach to relational database design based on the notion of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies. First, however, we view the problem of relational design from the standpoint of the schemas derived from a given entity-relationship design.

8.1 Features of Good Relational Designs

Our study of entity-relationship design in Chapter 7 provides an excellent starting point for creating a relational database design. We saw in Section 7.6 that it is possible to generate a set of relation schemas directly from the E-R design. Obviously, the goodness (or badness) of the resulting set of schemas depends on how good the E-R design was in the first place. Later in this chapter, we shall study precise ways of assessing the desirability of a collection of relation schemas. However, we can go a long way toward a good design using concepts we have already studied.

For ease of reference, we repeat the schemas for the university database in Figure 8.1.

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

Figure 8.1 Schema for the university database.

8.1.1 Design Alternative: Larger Schemas

Now, let us explore features of this relational database design as well as some alternatives. Suppose that instead of having the schemas *instructor* and *department*, we have the schema:

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

This represents the result of a natural join on the relations corresponding to *instructor* and *department*. This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design.

Let us consider the instance of the *inst_dept* relation shown in Figure 8.2. Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department. For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt.

It is important that all these tuples agree as to the budget amount since otherwise our database would be inconsistent. In our original design using *instructor* and *department*, we stored the amount of each budget exactly once. This suggests that using *inst_dept* is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency.

Even if we decided to live with the redundancy problem, there is still another problem with the *inst_dept* schema. Suppose we are creating a new department in the university. In the alternative design above, we cannot represent directly the information concerning a department (*dept_name*, *building*, *budget*) unless that department has at least one instructor at the university. This is because tuples in the *inst_dept* table require values for *ID*, *name*, and *salary*. This means that we cannot record information about the newly created department until the first instructor

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 8.2 The *inst_dept* table.

is hired for the new department. In the old design, the schema *department* can handle this, but under the revised design, we would have to create a tuple with a null value for *building* and *budget*. In some cases null values are troublesome, as we saw in our study of SQL. However, if we decide that this is not a problem to us in this case, then we can proceed to use the revised design.

8.1.2 Design Alternative: Smaller Schemas

Suppose again that, somehow, we had started out with the schema *inst_dept*. How would we recognize that it requires repetition of information and should be split into the two schemas *instructor* and *department*?

By observing the contents of actual relations on schema *inst_dept*, we could note the repetition of information resulting from having to list the building and budget once for each instructor associated with a department. However, this is an unreliable process. A real-world database has a large number of schemas and an even larger number of attributes. The number of tuples can be in the millions or higher. Discovering repetition would be costly. There is an even more fundamental problem with this approach. It does not allow us to determine whether the lack of repetition is just a "lucky" special case or whether it is a manifestation of a general rule. In our example, how would we *know* that in our university organization, each department (identified by its department name) *must* reside in a single building and must have a single budget amount? Is the fact that the budget amount for the Comp. Sci. department appears three times with the same budget amount just a coincidence? We cannot answer these questions without going back to the enterprise itself and understanding its rules. In particular, we would need to discover that the university requires that every department (identified by its department name) *must* have only one building and one budget value.

In the case of *inst_dept*, our process of creating an E-R design successfully avoided the creation of this schema. However, this fortuitous situation does not

always occur. Therefore, we need to allow the database designer to specify rules such as "each specific value for *dept_name* corresponds to at most one *budget*" even in cases where *dept_name* is not the primary key for the schema in question. In other words, we need to write a rule that says "if there were a schema (*dept_name*, *budget*), then *dept_name* is able to serve as the primary key." This rule is specified as a **functional dependency**

$$dept\_name \rightarrow budget$$

Given such a rule, we now have sufficient information to recognize the problem of the *inst_dept* schema. Because *dept_name* cannot be the primary key for *inst_dept* (because a department may need several tuples in the relation on schema *inst_dept*), the amount of a budget may have to be repeated.

Observations such as these and the rules (functional dependencies in particular) that result from them allow the database designer to recognize situations where a schema ought to be split, or *decomposed*, into two or more schemas. It is not hard to see that the right way to decompose *inst_dept* is into schemas *instructor* and *department* as in the original design. Finding the right decomposition is much harder for schemas with a large number of attributes and several functional dependencies. To deal with this, we shall rely on a formal methodology that we develop later in this chapter.

Not all decompositions of schemas are helpful. Consider an extreme case where all we had were schemas consisting of one attribute. No interesting relationships of any kind could be expressed. Now consider a less extreme case where we choose to decompose the *employee* schema (Section 7.8):

employee (*ID*, *name*, *street*, *city*, *salary*)

into the following two schemas:

employee1 (*ID*, *name*)
employee2 (*name*, *street*, *city*, *salary*)

The flaw in this decomposition arises from the possibility that the enterprise has two employees with the same name. This is not unlikely in practice, as many cultures have certain highly popular names. Of course each person would have a unique employee-id, which is why *ID* can serve as the primary key. As an example, let us assume two employees, both named Kim, work at the university and have the following tuples in the relation on schema *employee* in the original design:

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

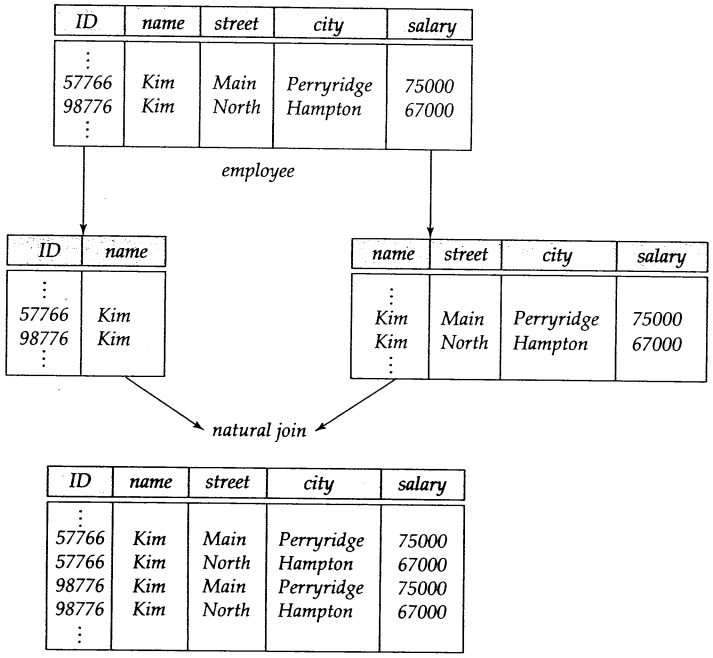


Figure 8.3 Loss of information via a bad decomposition.

Figure 8.3 shows these tuples, the resulting tuples using the schemas resulting from the decomposition, and the result if we attempted to regenerate the original tuples using a natural join. As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim. Although we have more tuples, we actually have less information in the following sense. We can indicate that a certain street, city, and salary pertain to someone named Kim, but we are unable to distinguish which of the Kims. Thus, our decomposition is unable to represent certain important facts about the university employees. Clearly, we would like to avoid such decompositions. We shall refer to such decompositions as being **lossy decompositions**, and, conversely, to those that are not as **lossless decompositions**.

8.2 Atomic Domains and First Normal Form

The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as

phone_number in Figure 7.11 and composite attributes (such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip*). When we create tables from E-R designs that contain these types of attributes, we eliminate this substructure. For composite attributes, we let each component be an attribute in its own right. For multivalued attributes, we create one tuple for each item in a multivalued set.

In the relational model, we formalize this idea that attributes do not have any substructure. A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema R is in **first normal form** (1NF) if the domains of all attributes of R are atomic.

A set of names is an example of a nonatomic value. For example, if the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.

Composite attributes, such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip* also have nonatomic domains.

Integers are assumed to be atomic, so the set of integers is an atomic domain; however, the set of all sets of integers is a nonatomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits.

As a practical illustration of the above point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be "CS001" and "EE1127". Such identification numbers can be divided into smaller units, and are therefore nonatomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes departments, the employee's identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong result.

From the above discussion, it may appear that our use of course identifiers such as "CS-101", where "CS" indicates the Computer Science department, means that the domain of course identifiers is not atomic. Such a domain is not atomic as far as humans using the system are concerned. However, the database application still treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation. The *course* schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead

of interpreting particular characters of the course identifier. Thus, our university schema can be considered to be in first normal form.

The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of having the relationship between instructors and sections being represented as a separate relation *teaches*, a database designer may be tempted to store a set of course section identifiers with each instructor and a set of instructor identifiers with each section. (The primary keys of *section* and *instructor* are used as identifiers.) Whenever data pertaining to which instructor teaches which section is changed, the update has to be performed at two places: in the set of instructors for the section, and the set of sections for the instructor. Failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets, that either the set of instructors of a section, or the set of sections of an instructor, would avoid repeated information; however keeping only one of these would complicate some queries, and it is unclear which of the two to retain.

Some types of nonatomic values can be useful, although they should be used with care. For example, composite-valued attributes are often useful, and set-valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also the runtime overhead of converting data back and forth from the atomic form. Support for nonatomic values can thus be very useful in such domains. In fact, modern database systems do support many types of nonatomic values, as we shall see in Chapter 22. However, in this chapter we restrict ourselves to relations in first normal form and, thus, all domains are atomic.

8.3 Decomposition Using Functional Dependencies

In Section 8.1, we noted that there is a formal methodology for evaluating whether a relational schema should be decomposed. This methodology is based upon the concepts of keys and functional dependencies.

In discussing algorithms for relational database design, we shall need to talk about arbitrary relations and their schema, rather than talking only about examples. Recalling our introduction to the relational model in Chapter 2, we summarize our notation here.

- In general, we use Greek letters for sets of attributes (for example, α). We use a lowercase Roman letter followed by an uppercase Roman letter in parentheses to refer to a relation schema (for example, $r(R)$). We use the notation $r(R)$ to show that the schema is for relation r, with R denoting the set of attributes, but at times simplify our notation to use just R when the relation name does not matter to us.

  Of course, a relation schema is a set of attributes, but not all sets of attributes are schemas. When we use a lowercase Greek letter, we are referring to a set

of attributes that may or may not be a schema. A Roman letter is used when we wish to indicate that the set of attributes is definitely a schema.

- When a set of attributes is a superkey, we denote it by K. A superkey pertains to a specific relation schema, so we use the terminology "K is a superkey of $r(R)$."
- We use a lowercase name for relations. In our examples, these names are intended to be realistic (for example, *instructor*), while in our definitions and algorithms, we use single letters, like r.
- A relation, of course, has a particular value at any given time; we refer to that as an instance and use the term "instance of r". When it is clear that we are talking about an instance, we may use simply the relation name (for example, r).

8.3.1 Keys and Functional Dependencies

A database models a set of entities and relationships in the real world. There are usually a variety of constraints (rules) on the data in the real world. For example, some of the constraints that are expected to hold in a university database are:

1. Students and instructors are uniquely identified by their ID.
2. Each student and instructor has only one name.
3. Each instructor and student is (primarily) associated with only one department.[1]
4. Each department has only one value for its budget, and only one associated building.

An instance of a relation that satisfies all such real-world constraints is called a legal instance of the relation; a legal instance of a database is one where all the relation instances are legal instances.

Some of the most commonly used types of real-world constraints can be represented formally as keys (superkeys, candidate keys and primary keys), or as functional dependencies, which we define below.

In Section 2.3, we defined the notion of a *superkey* as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation. We restate that definition here as follows: Let $r(R)$ be a relation schema. A subset K of R is a **superkey** of $r(R)$ if, in any legal instance of $r(R)$, for all pairs t_1 and t_2 of tuples in the instance of r if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal instance of relation $r(R)$ may have the same value on

[1] An instructor or a student can be associated with more than one department, for example as an adjunct faculty, or as a minor department. Our simplified university schema models only the primary department associated with each instructor or student. A real university schema would capture secondary associations in other relations.

attribute set K. Clearly, if no two tuples in r have the same value on K, then a K-value uniquely identifies a tuple in r.

Whereas a superkey is a set of attributes that uniquely identifies an entire tuple, a functional dependency allows us to express constraints that uniquely identify the values of certain attributes. Consider a relation schema $r(R)$, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of $r(R)$, we say that the instance satisfies the functional dependency $\alpha \rightarrow \beta$ if for all pairs of tuples t_1 and t_2 in the instance such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema $r(R)$ if, in every legal instance of $r(R)$ it satisfies the functional dependency.

Using the functional-dependency notation, we say that *K is a superkey of $r(R)$* if the functional dependency $K \rightarrow R$ holds on $r(R)$. In other words, K is a superkey if, for every legal instance of $r(R)$, for every pair of tuples t_1 and t_2 from the instance, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (that is, $t_1 = t_2$).[2]

Functional dependencies allow us to express constraints that we cannot express with superkeys. In Section 8.1.2, we considered the schema:

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

in which the functional dependency *dept_name* $\rightarrow$ *budget* holds because for each department (identified by *dept_name*) there is a unique budget amount.

We denote the fact that the pair of attributes (*ID*, *dept_name*) forms a superkey for *inst_dept* by writing:

ID, *dept_name* $\rightarrow$ *name*, *salary*, *building*, *budget*

We shall use functional dependencies in two ways:

1. To test instances of relations to see whether they satisfy a given set F of functional dependencies.
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with *only* those relation instances that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema $r(R)$ that satisfy a set F of functional dependencies, we say that F holds on $r(R)$.

Let us consider the instance of relation r of Figure 8.4, to see which functional dependencies are satisfied. Observe that $A \rightarrow C$ is satisfied. There are two tuples

[2] Note that we assume here that relations are sets. SQL deals with multisets, and a **primary key** declaration in SQL for a set of attributes K requires not only that $t_1 = t_2$ if $t_1[K] = t_2[K]$, but also that there be no duplicate tuples. SQL also requires that attributes in the set K cannot be assigned a *null* value.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure 8.4 Sample instance of relation r.

that have an A value of a_1. These tuples have the same C value—namely, c_1. Similarly, the two tuples with an A value of a_2 have the same C value, c_2. There are no other pairs of distinct tuples that have the same A value. The functional dependency $C \rightarrow A$ is not satisfied, however. To see that it is not, consider the tuples $t_1 = (a_2, b_3, c_2, d_3)$ and $t_2 = (a_3, b_3, c_2, d_4)$. These two tuples have the same C values, c_2, but they have different A values, a_2 and a_3, respectively. Thus, we have found a pair of tuples t_1 and t_2 such that $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples t_1 and t_2 such that $t_1[A] = t_2[A]$, it is the case that $t_1[A] = t_2[A]$. Similarly, $AB \rightarrow A$ is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is **trivial** if $\beta \subseteq \alpha$.

It is important to realize that an instance of a relation may satisfy some functional dependencies that are not required to hold on the relation's schema. In the instance of the *classroom* relation of Figure 8.5, we see that *room_number* $\rightarrow$ *capacity* is satisfied. However, we believe that, in the real world, two classrooms in different buildings can have the same room number but with different room capacity. Thus, it is possible, at some time, to have an instance of the *classroom* relation in which *room_number* $\rightarrow$ *capacity* is not satisfied. So, we would not include *room_number* $\rightarrow$ *capacity* in the set of functional dependencies that hold on the schema for the *classroom* relation. However, we would expect the functional dependency *building, room_number* $\rightarrow$ *capacity* to hold on the *classroom* schema.

Given that a set of functional dependencies F holds on a relation $r(R)$, it may be possible to infer that certain other functional dependencies must also hold on

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 8.5 An instance of the *classroom* relation.

the relation. For example, given a schema $r(A, B, C)$, if functional dependencies $A \rightarrow B$ and $B \rightarrow C$, hold on r, we can infer the functional dependency $A \rightarrow C$ must also hold on r. This is because, given any value of A there can be only one corresponding value for B, and for that value of B, there can only be one corresponding value for C. We study later, in Section 8.4.1, how to make such inferences.

We will use the notation F^+ to denote the **closure** of the set F, that is, the set of all functional dependencies that can be inferred given the set F. Clearly F^+ contains all of the functional dependencies in F.

8.3.2 Boyce–Codd Normal Form

One of the more desirable normal forms that we can obtain is **Boyce–Codd normal form** (BCNF). It eliminates all redundancy that can be discovered based on functional dependencies, though, as we shall see in Section 8.6, there may be other types of redundancy remaining. A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

We have already seen in Section 8.1 an example of a relational schema that is not in BCNF:

inst_dept (*ID, name, salary, dept_name, building, budget*)

The functional dependency *dept_name* $\rightarrow$ *budget* holds on *inst_dept*, but *dept_name* is not a superkey (because, a department may have a number of different instructors). In Section 8.1.2, we saw that the decomposition of *inst_dept* into *instructor* and *department* is a better design. The *instructor* schema is in BCNF. All of the nontrivial functional dependencies that hold, such as:

ID $\rightarrow$ *name, dept_name, salary*

include *ID* on the left side of the arrow, and *ID* is a superkey (actually, in this case, the primary key) for *instructor*. (In other words, there is no nontrivial functional dependency with any combination of *name*, *dept_name*, and *salary*, without *ID*, on the side.) Thus, *instructor* is in BCNF.

Similarly, the *department* schema is in BCNF because all of the nontrivial functional dependencies that hold, such as:

dept_name $\rightarrow$ *building, budget*

include *dept_name* on the left side of the arrow, and *dept_name* is a superkey (and the primary key) for *department*. Thus, *department* is in BCNF.

We now state a general rule for decomposing that are not in BCNF. Let R be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R. We replace R in our design with two schemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In the case of *inst_dept* above, α = *dept_name*, $\beta = \{building, budget\}$, and *inst_dept* is replaced by

- $(\alpha \cup \beta)$ = (*dept_name*, *building*,*budget*)
- $(R - (\beta - \alpha))$ = (*ID*, *name*, *dept_name*, *salary*)

In this example, it turns out that $\beta - \alpha = \beta$. We need to state the rule as we did so as to deal correctly with functional dependencies that have attributes that appear on both sides of the arrow. The technical reasons for this are covered later in Section 8.5.1.

When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.

8.3.3 BCNF and Dependency Preservation

We have seen several ways in which to express database consistency constraints: primary-key constraints, functional dependencies, **check** constraints, assertions, and triggers. Testing these constraints each time the database is updated can be costly and, therefore, it is useful to design the database in a way that constraints can be tested efficiently. In particular, if testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low. We shall see that, in some cases, decomposition into BCNF can prevent efficient testing of certain functional dependencies.

To illustrate this, suppose that we make a small change to our university organization. In the design of Figure 7.15, a student may have only one advisor. This follows from the relationship set *advisor* being many-to-one from *student* to *advisor*. The "small" change we shall make is that an instructor can be associated with only a single department and a student may have more than one advisor, but at most one from a given department.[3]

One way to implement this change using the E-R design is by replacing the *advisor* relationship set with a ternary relationship set, *dept_advisor*, involving entity sets *instructor*, *student*, and *department* that is many-to-one from the pair

[3]Such an arrangement makes sense for students with a double major.

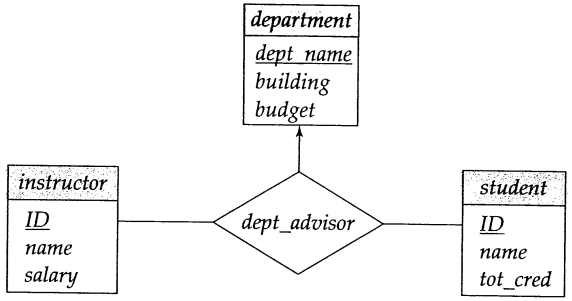


Figure 8.6 The *dept_advisor* relationship set.

{*student*, *instructor*} to *department* as shown in Figure 8.6. The E-R diagram specifies the constraint that "a student may have more than one advisor, but at most one corresponding to a given department".

With this new E-R diagram, the schemas for the *instructor*, *department*, and *student* are unchanged. However, the schema derived from *dept_advisor* is now:

dept_advisor (*s_ID*, *i_ID*, *dept_name*)

Although not specified in the E-R diagram, suppose we have the additional constraint that "an instructor can act as advisor for only a single department."

Then, the following functional dependencies hold on *dept_advisor*:

i_ID $\rightarrow$ *dept_name*
s_ID, *dept_name* $\rightarrow$ *i_ID*

The first functional dependency follows from our requirement that "an instructor can act as an advisor for only one department." The second functional dependency follows from our requirement that "a student may have at most one advisor for a given department."

Notice that with this design, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship. We see that *dept_advisor* is not in BCNF because *i_ID* is not a superkey. Following our rule for BCNF decomposition, we get:

(*s_ID*, *i_ID*)
(*i_ID*, *dept_name*)

Both the above schemas are BCNF. (In fact, you can verify that any schema with only two attributes is in BCNF by definition.) Note however, that in our BCNF design, there is no schema that includes all the attributes appearing in the functional dependency *s_ID*, *dept_name* $\rightarrow$ *i_ID*.

Because our design makes it computationally hard to enforce this functional dependency, we say our design is not **dependency preserving**.[4] Because dependency preservation is usually considered desirable, we consider another normal form, weaker than BCNF, that will allow us to preserve dependencies. That normal form is called third normal form.[5]

8.3.4 Third Normal Form

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a superkey. Third normal form (3NF) relaxes this constraint slightly by allowing certain nontrivial functional dependencies whose left side is not a superkey. Before we define 3NF, we recall that a candidate key is a minimal superkey—that is, a superkey no proper subset of which is also a superkey.

A relation schema R is in **third normal form** with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Note that the third condition above does not say that a single candidate key must contain all the attributes in $\beta - \alpha$; each attribute A in $\beta - \alpha$ may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative of the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive normal form than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\alpha \rightarrow \beta$ that satisfies only the third alternative of the 3NF definition is not allowed in BCNF, but is allowed in 3NF.[6]

Now, let us again consider the *dept_advisor* relationship set, which has the following functional dependencies:

[4]Technically, it is possible that a dependency whose attributes do not all appear in any one schema is still implicitly enforced, because of the presence of other dependencies that imply it logically. We address that case later, in Section 8.4.5.

[5]You may have noted that we skipped second normal form. It is of historical significance only and is not used in practice.

[6]These dependencies are examples of **transitive dependencies** (see Practice Exercise 8.16). The original definition of 3NF was in terms of transitive dependencies. The definition we use is equivalent but easier to understand.

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

In Section 8.3.3 we argued that the functional dependency "*i_ID* $\rightarrow$ *dept_name*" caused the *dept_advisor* schema not to be in BCNF. Note that here $\alpha = i\_ID$, $\beta = dept\_name$, and $\beta - \alpha = dept\_name$. Since the functional dependency *s_ID*, *dept_name* $\rightarrow$ *i_ID* holds on *dept_advisor*, the attribute *dept_name* is contained in a candidate key and, therefore, *dept_advisor* is in 3NF.

We have seen the trade-off that must be made between BCNF and 3NF when there is no dependency-preserving BCNF design. These trade-offs are described in more detail in Section 8.5.4.

8.3.5 Higher Normal Forms

Using functional dependencies to decompose schemas may not be sufficient to avoid unnecessary repetition of information in certain cases. Consider a slight variation in the *instructor* entity-set definition in which we record with each instructor a set of children's names and a set of phone numbers. The phone numbers may be shared by multiple people. Thus, *phone_number* and *child_name* would be multivalued attributes and, following our rules for generating schemas from an E-R design, we would have two schemas, one for each of the multivalued attributes, *phone_number* and *child_name*:

(*ID*, *child_name*)
(*ID*, *phone_number*)

If we were to combine these schemas to get

(*ID*, *child_name*, *phone_number*)

we would find the result to be in BCNF because only nontrivial functional dependencies hold. As a result we might think that such a combination is a good idea. However, such a combination is a bad idea, as we can see by considering the example of an instructor with two children and two phone numbers. For example, let the instructor with *ID* 99999 have two children named "David" and "William" and two phone numbers, 512-555-1234 and 512-555-4321. In the combined schema, we must repeat the phone numbers once for each dependent:

(99999, David, 512-555-1234)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)
(99999, William, 512-555-4321)

If we did not repeat the phone numbers, and stored only the first and last tuple, we would have recorded the dependent names and the phone numbers, but

the resultant tuples would imply that David corresponded to 512-555-1234, while William corresponded to 512-555-4321. As we know, this would be incorrect.

Because normal forms based on functional dependencies are not sufficient to deal with situations like this, other dependencies and normal forms have been defined. We cover these in Sections 8.6 and 8.7.

8.4 Functional-Dependency Theory

We have seen in our examples that it is useful to be able to reason systematically about functional dependencies as part of a process of testing schemas for BCNF or 3NF.

8.4.1 Closure of a Set of Functional Dependencies

We shall see that, given a set F of functional dependencies on a schema, we can prove that certain other functional dependencies also hold on the schema. We say that such functional dependencies are "logically implied" by F. When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to consider *all* functional dependencies that hold on the schema.

More formally, given a relational schema $r(R)$, a functional dependency f on R is logically implied by a set of functional dependencies F on r if every instance of $r(R)$ that satisfies F also satisfies f.

Suppose we are given a relation schema $r(A, B, C, G, H, I)$ and the set of functional dependencies:

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

The functional dependency:

$$A \rightarrow H$$

is logically implied. That is, we can show that, whenever a relation satisfies our given set of functional dependencies, $A \rightarrow H$ must also be satisfied by that relation. Suppose that t_1 and t_2 are tuples such that:

$$t_1[A] = t_2[A]$$

Since we are given that $A \rightarrow B$, it follows from the definition of functional dependency that:

$$t_1[B] = t_2[B]$$

Then, since we are given that $B \rightarrow H$, it follows from the definition of functional dependency that:

$$t_1[H] = t_2[H]$$

Therefore, we have shown that, whenever t_1 and t_2 are tuples such that $t_1[A] = t_2[A]$, it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of $A \rightarrow H$.

Let F be a set of functional dependencies. The closure of F, denoted by F^+, is the set of all functional dependencies logically implied by F. Given F, we can compute F^+ directly from the formal definition of functional dependency. If F were large, this process would be lengthy and difficult. Such a computation of F^+ requires arguments of the type just used to show that $A \rightarrow H$ is in the closure of our example set of dependencies.

Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters (α, β, γ, ...) for sets of attributes, and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use $\alpha\beta$ to denote $\alpha \cup \beta$.

We can use the following three rules to find logically implied functional dependencies. By applying these rules *repeatedly*, we can find all of F^+, given F. This collection of rules is called Armstrong's axioms in honor of the person who first proposed it.

- **Reflexivity rule.** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule.** If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule.** If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Armstrong's axioms are sound, because they do not generate any incorrect functional dependencies. They are complete, because, for a given set F of functional dependencies, they allow us to generate all F^+. The bibliographical notes provide references for proofs of soundness and completeness.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+. To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are sound (see Practice Exercises 8.4 and 8.5 and Exercise 8.26).

- **Union rule.** If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule.** If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule.** If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Let us apply our rules to the example of schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. We list several members of F^+ here:

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.
- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

  Another way of finding that $AG \rightarrow I$ holds is as follows: We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Figure 8.7 shows a procedure that demonstrates formally how to use Armstrong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+. We shall see an alternative way of computing F^+ in Section 8.4.2.

The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, there are a total of $2^n \times 2^n = 2^{2n}$ possible functional dependencies, where n is the number of attributes in R. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate.

8.4.2 Closure of Attribute Sets

We say that an attribute B is **functionally determined** by α if $\alpha \rightarrow B$. To test whether a set α is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by α. One way of doing this is to compute F^+, take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

An efficient algorithm for computing the set of attributes functionally determined by α is useful not only for testing whether α is a superkey, but also for several other tasks, as we shall see later in this section.

```
F+ = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F+
    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            add the resulting functional dependency to F+
until F+ does not change any further
```

Figure 8.7 A procedure to compute F^+.


```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

Figure 8.8 An algorithm to compute α^+, the closure of α under F.

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set F of functional dependencies the **closure** of α under F; we denote it by α^+. Figure 8.8 shows an algorithm, written in pseudocode, to compute α^+. The input is a set F of functional dependencies and the set α of attributes. The output is stored in the variable *result*.

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ with the functional dependencies defined in Section 8.4.1. We start with $result = AG$. The first time that we execute the **repeat** loop to test each functional dependency, we find that:

- $A \rightarrow B$ causes us to include B in *result*. To see this fact, we observe that $A \rightarrow B$ is in F, $A \subseteq result$ (which is AG), so $result := result \cup B$.
- $A \rightarrow C$ causes *result* to become $ABCG$.
- $CG \rightarrow H$ causes *result* to become $ABCGH$.
- $CG \rightarrow I$ causes *result* to become $ABCGHI$.

The second time that we execute the **repeat** loop, no new attributes are added to *result*, and the algorithm terminates.

Let us see why the algorithm of Figure 8.8 is correct. The first step is correct, since $\alpha \rightarrow \alpha$ always holds (by the reflexivity rule). We claim that, for any subset β of *result*, $\alpha \rightarrow \beta$. Since we start the **repeat** loop with $\alpha \rightarrow result$ being true, we can add γ to *result* only if $\beta \subseteq result$ and $\beta \rightarrow \gamma$. But then $result \rightarrow \beta$ by the reflexivity rule, so $\alpha \rightarrow \beta$ by transitivity. Another application of transitivity shows that $\alpha \rightarrow \gamma$ (using $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$). The union rule implies that $\alpha \rightarrow result \cup \gamma$, so α functionally determines any new result generated in the **repeat** loop. Thus, any attribute returned by the algorithm is in α^+.

It is easy to see that the algorithm finds all of α^+. If there is an attribute in α^+ that is not yet in *result* at any point during the execution, then there must be a functional dependency $\beta \rightarrow \gamma$ for which $\beta \subseteq result$, and at least one attribute in γ is not in *result*. When the algorithm terminates, all such functional dependencies have been processed, and the attributes in γ added to *result*; we can thus be sure that all attributes in α^+ are in *result*.

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of F. There is a faster (although slightly more complex) algorithm that runs in time linear in the size of F; that algorithm is presented as part of Practice Exercise 8.8.

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes in R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β. This test is particularly useful, as we shall see later in this chapter.
- It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

8.4.3 Canonical Cover

Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies, that is, all the functional dependencies in F are satisfied in the new database state.

The system must roll back the update if it violates any functional dependencies in the set F.

We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set. Any database that satisfies the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure. However, the simplified set is easier to test. We shall see how the simplified set can be constructed in a moment. First, we need some definitions.

An attribute of a functional dependency is said to be **extraneous** if we can remove it without changing the closure of the set of functional dependencies. The formal definition of **extraneous attributes** is as follows: Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

- Attribute A is extraneous in α if $A \in \alpha$, and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- Attribute A is extraneous in β if $A \in \beta$, and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

For example, suppose we have the functional dependencies $AB \rightarrow C$ and $A \rightarrow C$ in F. Then, B is extraneous in $AB \rightarrow C$. As another example, suppose we have the functional dependencies $AB \rightarrow CD$ and $A \rightarrow C$ in F. Then C would be extraneous in the right-hand side of $AB \rightarrow CD$.

Beware of the direction of the implications when using the definition of extraneous attributes: If you exchange the left-hand side with the right-hand side, the implication will *always* hold. That is, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ always logically implies F, and also F always logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.


```
F_c = F
repeat
    Use the union rule to replace any dependencies in F_c of the form
        α1 → β1 and α1 → β2 with α1 → β1 β2.
    Find a functional dependency α → β in F_c with an extraneous
        attribute either in α or in β.
        /* Note: the test for extraneous attributes is done using F_c, not F */
    If an extraneous attribute is found, delete it from α → β in F_c.
until (F_c does not change)
```

Figure 8.9 Computing canonical cover.

Here is how we can test efficiently if an attribute is extraneous. Let R be the relation schema, and let F be the given set of functional dependencies that hold on R. Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

- If $A \in \beta$, to check if A is extraneous, consider the set
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
and check if $\alpha \rightarrow A$ can be inferred from F'. To do so, compute α^+ (the closure of α) under F'; if α^+ includes A, then A is extraneous in β.
- If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F. To do so, compute γ^+ (the closure of γ) under F; if γ^+ includes all attributes in β, then A is extraneous in α.

For example, suppose F contains $AB \rightarrow CD$, $A \rightarrow E$, and $E \rightarrow C$. To check if C is extraneous in $AB \rightarrow CD$, we compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, \text{and } E \rightarrow C\}$. The closure is $ABCDE$, which includes CD, so we infer that C is extraneous.

A **canonical cover** F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

A canonical cover for a set of functional dependencies F can be computed as depicted in Figure 8.9. It is important to note that when checking if an attribute is extraneous, the check uses the dependencies in the current value of F_c, and **not** the dependencies in F. If a functional dependency contains only one attribute

in its right-hand side, for example $A \rightarrow C$, and that attribute is found to be extraneous, we would get a functional dependency with an empty right-hand side. Such functional dependencies should be deleted.

The canonical cover of F, F_c, can be shown to have the same closure as F; hence, testing whether F_c is satisfied is equivalent to testing whether F is satisfied. However, F_c is minimal in a certain sense—it does not contain extraneous attributes, and it combines functional dependencies with the same left side. It is cheaper to test F_c than it is to test F itself.

Consider the following set F of functional dependencies on schema (A, B, C):

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

Let us compute the canonical cover for F.

- There are two functional dependencies with the same set of attributes on the left side of the arrow:

$$A \rightarrow BC$$
$$A \rightarrow B$$

  We combine these functional dependencies into $A \rightarrow BC$.

- A is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. This assertion is true because $B \rightarrow C$ is already in our set of functional dependencies.

- C is extraneous in $A \rightarrow BC$, since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.

Thus, our canonical cover is:

$$A \rightarrow B$$
$$B \rightarrow C$$

Given a set F of functional dependencies, it may be that an entire functional dependency in the set is extraneous, in the sense that dropping it does not change the closure of F. We can show that a canonical cover F_c of F contains no such extraneous functional dependency. Suppose that, to the contrary, there were such an extraneous functional dependency in F_c. The right-side attributes of the dependency would then be extraneous, which is not possible by the definition of canonical covers.

A canonical cover might not be unique. For instance, consider the set of functional dependencies $F = \{A \rightarrow BC, B \rightarrow AC, \text{and } C \rightarrow AB\}$. If we apply

the extraneity test to $A \rightarrow BC$, we find that both B and C are extraneous under F. However, it is incorrect to delete both! The algorithm for finding the canonical cover picks one of the two, and deletes it. Then,

1. If C is deleted, we get the set $F' = \{A \rightarrow B, B \rightarrow AC, \text{and } C \rightarrow AB\}$. Now, B is not extraneous in the side of $A \rightarrow B$ under F'. Continuing the algorithm, we find A and B are extraneous in the right-hand side of $C \rightarrow AB$, leading to two canonical covers

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}.$$

2. If B is deleted, we get the set $\{A \rightarrow C, B \rightarrow AC, \text{and } C \rightarrow AB\}$. This case is symmetrical to the previous case, leading to the canonical covers

$$F_c = \{A \rightarrow C, C \rightarrow B, \text{and } B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, \text{and } C \rightarrow AB\}.$$

As an exercise, can you find one more canonical cover for F?

8.4.4 Lossless Decomposition

Let $r(R)$ be a relation schema, and let F be a set of functional dependencies on $r(R)$. Let R_1 and R_2 form a decomposition of R. We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing $r(R)$ with two relation schemas $r_1(R_1)$ and $r_2(R_2)$. More precisely, we say the decomposition is lossless if, for all legal database instances (that is, database instances that satisfy the specified functional dependencies and other constraints), relation r contains the same set of tuples as the result of the following SQL query:

```
select *
from (select R1 from r)
     natural join
     (select R2 from r)
```

This is stated more succinctly in the relational algebra as:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

In other words, if we project r onto R_1 and R_2, and compute the natural join of the projection results, we get back exactly r. A decomposition that is not a lossless decomposition is called a **lossy decomposition**. The terms **lossless-join decomposition** and **lossy-join decomposition** are sometimes used in place of lossless decomposition and lossy decomposition.

As an example of a lossy decomposition, recall the decomposition of the *employee* schema into:

employee1 (*ID*, *name*)
employee2 (*name*, *street*, *city*, *salary*)

that we saw earlier in Section 8.1.2. As we saw in Figure 8.3, the result of *employee1* ⋈ *employee2* is a superset of the original relation *employee*, but the decomposition is lossy since the join result has lost information about which employee identifiers correspond to which addresses and salaries, in the case where two or more employees have the same name.

We can use functional dependencies to show when certain decompositions are lossless. Let R, R_1, R_2, and F be as above. R_1 and R_2 form a lossless decomposition of R if at least one of the following functional dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey of either R_1 or R_2, the decomposition of R is a lossless decomposition. We can use attribute closure to test efficiently for superkeys, as we have seen earlier.

To illustrate this, consider the schema

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

that we decomposed in Section 8.1.2 into the *instructor* and *department* schemas:

instructor (*ID*, *name*, *dept_name*, *salary*)
department (*dept_name*, *building*, *budget*)

Consider the intersection of these two schemas, which is *dept_name*. We see that because *dept_name*→ *dept_name*, *building*, *budget*, the lossless-decomposition rule is satisfied.

For the general case of decomposition of a schema into multiple schemas at once, the test for lossless decomposition is more complicated. See the bibliographical notes for references on the topic.

While the test for binary decomposition is clearly a sufficient condition for lossless decomposition, it is a necessary condition only if all constraints are functional dependencies. We shall see other types of constraints later (in particular, a type of constraint called multivalued dependencies discussed in Section 8.6.1), that can ensure that a decomposition is lossless even if no functional dependencies are present.

8.4.5 Dependency Preservation

Using the theory of functional dependencies, it is easier to characterize dependency preservation than using the ad-hoc approach we took in Section 8.3.3.

Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. The **restriction** of F to R_i is the set F_i of all functional dependencies in F^+ that include *only* attributes of R_i. Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

Note that the definition of restriction uses all dependencies in F^+, not just those in F. For instance, suppose $F = \{A \rightarrow B, B \rightarrow C\}$, and we have a decomposition into AC and AB. The restriction of F to AC includes $A \rightarrow C$, since $A \rightarrow C$ is in F^+, even though it is not in F.

The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' is a set of functional dependencies on schema R, but, in general, $F' \neq F$. However, even if $F' \neq F$, it may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F', and, if we verify that F' is satisfied, we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a **dependency-preserving decomposition**.

Figure 8.10 shows an algorithm for testing dependency preservation. The input is a set $D = \{R_1, R_2, \ldots, R_n\}$ of decomposed relation schemas, and a set F of functional dependencies. This algorithm is expensive since it requires computation of F^+. Instead of applying the algorithm of Figure 8.10, we consider two alternatives.

First, note that if each member of F can be tested on one of the relations of the decomposition, then the decomposition is dependency preserving. This is an easy way to show dependency preservation; however, it does not always work. There are cases where, even though the decomposition is dependency preserving, there is a dependency in F that cannot be tested in any one relation in the decomposition. Thus, this alternative test can be used only as a sufficient condition that is easy

```
compute F+;
for each schema Ri in D do
   begin
      Fi := the restriction of F+ to Ri;
   end
F' := ∅
for each restriction Fi do
   begin
      F' = F' ∪ Fi
   end
compute F'+;
if (F'+ = F+) then return (true)
              else return (false);
```

Figure 8.10 Testing for dependency preservation.

to check; if it fails we cannot conclude that the decomposition is not dependency preserving; instead we will have to apply the general test.

We now give a second alternative test for dependency preservation that avoids computing F^+. We explain the intuition behind the test after presenting the test. The test applies the following procedure to each $\alpha \rightarrow \beta$ in F.

```
result = α
repeat
    for each Ri in the decomposition
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
until (result does not change)
```

The attribute closure here is under the set of functional dependencies F. If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved. The decomposition is dependency preserving if and only if the procedure shows that all the dependencies in F are preserved.

The two key ideas behind the above test are as follows:

- The first idea is to test each functional dependency $\alpha \rightarrow \beta$ in F to see if it is preserved in F' (where F' is as defined in Figure 8.10). To do so, we compute the closure of α under F'; the dependency is preserved exactly when the closure includes β. The decomposition is dependency preserving if (and only if) all the dependencies in F are found to be preserved.
- The second idea is to use a modified form of the attribute-closure algorithm to compute closure under F', without actually first computing F'. We wish to avoid computing F' since computing it is quite expensive. Note that F' is the union of F_i, where F_i is the restriction of F on R_i. The algorithm computes the attribute closure of $(result \cap R_i)$ with respect to F, intersects the closure with R_i, and adds the resultant set of attributes to *result*; this sequence of steps is equivalent to computing the closure of *result* under F_i. Repeating this step for each i inside the while loop gives the closure of *result* under F'.

  To understand why this modified attribute-closure approach works correctly, we note that for any $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ is a functional dependency in F^+, and $\gamma \rightarrow \gamma^+ \cap R_i$ is a functional dependency that is in F_i, the restriction of F^+ to R_i. Conversely, if $\gamma \rightarrow \delta$ were in F_i, then δ would be a subset of $\gamma^+ \cap R_i$.

This test takes polynomial time, instead of the exponential time required to compute F^+.

8.5 Algorithms for Decomposition

Real-world database schemas are much larger than the examples that fit in the pages of a book. For this reason, we need algorithms for the generation of designs

that are in appropriate normal form. In this section, we present algorithms for BCNF and 3NF.

8.5.1 BCNF Decomposition

The definition of BCNF can be used directly to test if a relation is in BCNF. However, computation of F^+ can be a tedious task. We first describe below simplified tests for verifying if a relation is in BCNF. If a relation is not in BCNF, it can be decomposed to create relations that are in BCNF. Later in this section, we describe an algorithm to create a lossless decomposition of a relation, such that the decomposition is in BCNF.

8.5.1.1 Testing for BCNF

Testing of a relation schema R to see if it satisfies BCNF can be simplified in some cases:

- To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute α^+ (the attribute closure of α), and verify that it includes all attributes of R; that is, it is a superkey of R.
- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F^+.

  We can show that if none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF, either.

Unfortunately, the latter procedure does not work when a relation is decomposed. That is, it *does not* suffice to use F when we test a relation R_i, in a decomposition of R, for violation of BCNF. For example, consider relation schema $R(A, B, C, D, E)$, with functional dependencies F containing $A \rightarrow B$ and $BC \rightarrow D$. Suppose this were decomposed into $R_1(A, B)$ and $R_2(A, C, D, E)$. Now, neither of the dependencies in F contains only attributes from (A, C, D, E) so we might be misled into thinking R_2 satisfies BCNF. In fact, there is a dependency $AC \rightarrow D$ in F^+ (which can be inferred using the pseudotransitivity rule from the two dependencies in F) that shows that R_2 is not in BCNF. Thus, we may need a dependency that is in F^+, but is not in F, to show that a decomposed relation is not in BCNF.

An alternative BCNF test is sometimes easier than computing every dependency in F^+. To check if a relation R_i in a decomposition of R is in BCNF, we apply this test:

- For every subset α of attributes in R_i, check that α^+ (the attribute closure of α under F) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.

```
result := {R};
done := false;
compute F+;
while (not done) do
   if (there is a schema Ri in result that is not in BCNF)
      then begin
            let α → β be a nontrivial functional dependency that holds
            on Ri such that α → Ri is not in F+, and α ∩ β = ∅;
            result := (result − Ri) ∪ (Ri − β) ∪ ( α, β);
         end
      else done := true;
```

Figure 8.11 BCNF decomposition algorithm.

If the condition is violated by some set of attributes α in R_i, consider the following functional dependency, which can be shown to be present in F^+:

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

The above dependency shows that R_i violates BCNF.

8.5.1.2 BCNF Decomposition Algorithm

We are now able to state a general method to decompose a relation schema so as to satisfy BCNF. Figure 8.11 shows an algorithm for this task. If R is not in BCNF, we can decompose R into a collection of BCNF schemas $R_1, R_2, \ldots, R_n$ by the algorithm. The algorithm uses dependencies that demonstrate violation of BCNF to perform the decomposition.

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless decomposition. To see why our algorithm generates only lossless decompositions, we note that, when we replace a schema R_i with $(R_i - \beta)$ and (α, β), the dependency $\alpha \rightarrow \beta$ holds, and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

If we did not require $\alpha \cap \beta = \emptyset$, then those attributes in $\alpha \cap \beta$ would not appear in the schema $(R_i - \beta)$ and the dependency $\alpha \rightarrow \beta$ would no longer hold.

It is easy to see that our decomposition of *inst_dept* in Section 8.3.2 would result from applying the algorithm. The functional dependency *dept_name* $\rightarrow$ *building, budget* satisfies the $\alpha \cap \beta = \emptyset$ condition and would therefore be chosen to decompose the schema.

The BCNF decomposition algorithm takes time exponential in the size of the initial schema, since the algorithm for checking if a relation in the decomposition satisfies BCNF can take exponential time. The bibliographical notes provide references to an algorithm that can compute a BCNF decomposition in polynomial time. However, the algorithm may "overnormalize," that is, decompose a relation unnecessarily.

As a longer example of the use of the BCNF decomposition algorithm, suppose we have a database design using the *class* schema below:

class (*course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id*)

The set of functional dependencies that we require to hold on *class* are:

course_id $\rightarrow$ *title, dept_name, credits*
building, room_number $\rightarrow$ *capacity*
course_id, sec_id, semester, year $\rightarrow$ *building, room_number, time_slot_id*

A candidate key for this schema is {*course_id, sec_id, semester, year*}.

We can apply the algorithm of Figure 8.11 to the *class* example as follows:

- The functional dependency:

  course_id $\rightarrow$ *title, dept_name, credits*

  holds, but *course_id* is not a superkey. Thus, *class* is not in BCNF. We replace *class* by:

  course(*course_id, title, dept_name, credits*)
  class-1 (*course_id, sec_id, semester, year, building, room_number capacity, time_slot_id*)

  The only nontrivial functional dependencies that hold on *course* include *course_id* on the left side of the arrow. Since *course_id* is a key for *course*, the relation *course* is in BCNF.

- A candidate key for *class-1* is {*course_id, sec_id, semester, year*}. The functional dependency:

  building, room_number $\rightarrow$ *capacity*

  holds on *class-1*, but {*building, room_number*} is not a superkey for *class-1*. We replace *class-1* by:

  classroom (*building, room_number, capacity*)
  section (*course_id, sec_id, semester, year, building, room_number, time_slot_id*)

  classroom and *section* are in BCNF.

Thus, the decomposition of *class* results in the three relation schemas *course*, *classroom*, and *section*, each of which is in BCNF. These correspond to the schemas that we have used in this, and previous, chapters. You can verify that the decomposition is lossless and dependency preserving.

```
let F_c be a canonical cover for F;
i := 0;
for each functional dependency α → β in F_c
    i := i + 1;
    R_i := α β;
if none of the schemas R_j, j = 1, 2, ..., i contains a candidate key for R
  then
    i := i + 1;
    R_i := any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        R_j := R_i;
        i := i - 1;
until no more R_j s can be deleted
return (R_1, R_2, ..., R_i)
```

Figure 8.12 Dependency-preserving, lossless decomposition into 3NF.

8.5.2 3NF Decomposition

Figure 8.12 shows an algorithm for finding a dependency-preserving, lossless decomposition into 3NF. The set of dependencies F_c used in the algorithm is a canonical cover for F. Note that the algorithm considers the set of schemas R_j, $j = 1, 2, \ldots, i$; initially $i = 0$, and in this case the set is empty.

Let us apply this algorithm to our example of Section 8.3.4, where we showed that:

$$dept\_advisor\ (s\_ID, i\_ID, dept\_name)$$

is in 3NF even though it is not in BCNF. The algorithm uses the following functional dependencies in F:

$$f_1\text{: } i\_ID \rightarrow dept\_name$$
$$f_2\text{: } s\_ID, dept\_name \rightarrow i\_ID$$

There are no extraneous attributes in any of the functional dependencies in F, so F_c contains f_1 and f_2. The algorithm then generates as R_1 the schema, (*i_ID dept_name*), and as R_2 the schema (*s_ID, dept_name, i_ID*). The algorithm then finds that R_2 contains a candidate key, so no further relation schema is created.

The resultant set of schemas can contain redundant schemas, with one schema R_k containing all the attributes of another schema R_j. For example, R_2 above contains all the attributes from R_1. The algorithm deletes all such schemas that are contained in another schema. Any dependencies that could be tested on an

R_j that is deleted can also be tested on the corresponding relation R_k, and the decomposition is lossless even if R_j is deleted.

Now let us consider again the *class* schema of Section 8.5.1.2 and apply the 3NF decomposition algorithm. The set of functional dependencies we listed there happen to be a canonical cover. As a result, the algorithm gives us the same three schemas *course, classroom,* and *section*.

The above example illustrates an interesting property of the 3NF algorithm. Sometimes, the result is not only in 3NF, but also in BCNF. This suggests an alternative method of generating a BCNF design. First use the 3NF algorithm. Then, for any schema in the 3NF design that is not in BCNF, decompose using the BCNF algorithm. If the result is not dependency-preserving, revert to the 3NF design.

8.5.3 Correctness of the 3NF Algorithm

The 3NF algorithm ensures the preservation of dependencies by explicitly building a schema for each dependency in a canonical cover. It ensures that the decomposition is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. Practice Exercise 8.14 provides some insight into the proof that this suffices to guarantee a lossless decomposition.

This algorithm is also called the **3NF synthesis algorithm**, since it takes a set of dependencies and adds one schema at a time, instead of decomposing the initial schema repeatedly. The result is not uniquely defined, since a set of functional dependencies can have more than one canonical cover, and, further, in some cases, the result of the algorithm depends on the order in which it considers the dependencies in F_c. The algorithm may decompose a relation even if it is already in 3NF; however, the decomposition is still guaranteed to be in 3NF.

If a relation R_i is in the decomposition generated by the synthesis algorithm, then R_i is in 3NF. Recall that when we test for 3NF it suffices to consider functional dependencies whose right-hand side is a single attribute. Therefore, to see that R_i is in 3NF you must convince yourself that any functional dependency $\gamma \rightarrow B$ that holds on R_i satisfies the definition of 3NF. Assume that the dependency that generated R_i in the synthesis algorithm is $\alpha \rightarrow \beta$. Now, B must be in α or β, since B is in R_i and $\alpha \rightarrow \beta$ generated R_i. Let us consider the three possible cases:

- B is in both α and β. In this case, the dependency $\alpha \rightarrow \beta$ would not have been in F_c since B would be extraneous in β. Thus, this case cannot hold.
- B is in β but not α. Consider two cases:
  - γ is a superkey. The second condition of 3NF is satisfied.
  - γ is not a superkey. Then α must contain some attribute not in γ. Now, since $\gamma \rightarrow B$ is in F^+, it must be derivable from F_c by using the attribute closure algorithm on γ. The derivation could not have used $\alpha \rightarrow \beta$—if it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey. Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha\beta$, and γ

cannot contain B because $\gamma \rightarrow B$ is nontrivial). This would imply that B is extraneous in the right-hand side of $\alpha \rightarrow \beta$, which is not possible since $\alpha \rightarrow \beta$ is in the canonical cover F_c. Thus, if B is in β, then γ must be a superkey, and the second condition of 3NF must be satisfied.

- B is in α but not β.
  Since α is a candidate key, the third alternative in the definition of 3NF is satisfied.

Interestingly, the algorithm we described for decomposition into 3NF can be implemented in polynomial time, even though testing a given relation to see if it satisfies 3NF is NP-hard (which means that it is very unlikely that a polynomial-time algorithm will ever be invented for this task).

8.5.4 Comparison of BCNF and 3NF

Of the two normal forms for relational database schemas, 3NF and BCNF there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation. Nevertheless, there are disadvantages to 3NF: We may have to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

Our goals of database design with functional dependencies are:

1. BCNF.
2. Losslessness.
3. Dependency preservation.

Since it is not always possible to satisfy all three, we may be forced to choose between BCNF and dependency preservation with 3NF.

It is worth noting that SQL does not provide a way of specifying functional dependencies, except for the special case of declaring superkeys by using the **primary key** or **unique** constraints. It is possible, although a little complicated, to write assertions that enforce a functional dependency (see Practice Exercise 8.9); unfortunately, currently no database system supports the complex assertions that are required to enforce a functional dependency, and the assertions would be expensive to test. Thus even if we had a dependency-preserving decomposition, if we use standard SQL we can test efficiently only those functional dependencies whose left-hand side is a key.

Although testing functional dependencies may involve a join if the decomposition is not dependency preserving, we could in principle reduce the cost by using materialized views, which many database systems support, provided the database system supports primary key constraints on materialized views. Given a BCNF decomposition that is not dependency preserving, we consider each dependency in a canonical cover F_c that is not preserved in the decomposition. For each such dependency $\alpha \rightarrow \beta$, we define a materialized view that computes a

join of all relations in the decomposition, and projects the result on $\alpha\beta$. The functional dependency can be tested easily on the materialized view, using one of the constraints **unique** (α) or **primary key** (α).

On the negative side, there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about writing code to keep redundant data consistent on updates; it is the job of the database system to maintain the materialized view, that is, keep it up to date when the database is updated. (Later in the book, in Section 13.5, we outline how a database system can perform materialized view maintenance efficiently.)

Unfortunately, most current database systems do not support constraints on materialized views. Although the Oracle database does support constraints on materialized views, by default it performs view maintenance when the view is accessed, not when the underlying relation is updated;[7] as a result a constraint violation may get detected well after the update has been performed, which makes the detection useless.

Thus, in case we are not able to get a dependency-preserving BCNF decomposition, it is generally preferable to opt for BCNF, since checking functional dependencies other than primary key constraints is difficult in SQL.

8.6 Decomposition Using Multivalued Dependencies

Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information. Consider a variation of the university organization where an instructor may be associated with multiple departments.

$$inst\ (ID, dept\_name, name, street, city)$$

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

$$ID \rightarrow name, street, city$$

and because ID is not a key for $inst$.

Further assume that an instructor may have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency "$ID \rightarrow street, city$", though, of course, we still want to enforce "$ID \rightarrow name$" (that is, the university is not dealing with instructors who operate under multiple aliases!). Following the BCNF decomposition algorithm, we obtain two schemas:

[7] At least as of Oracle version 10g.

8.6 } Good to know!
8.7 } not req on Syllabus

r_1 (*ID*, *name*)
r_2 (*ID*, *dept_name*, *street*, *city*)

Both of these are in BCNF (recall that an instructor can be associated with multiple departments and a department may have several instructors, and therefore, neither "*ID* → *dept_name*" nor "*dept_name* → *ID*" hold).

Despite r_2 being in BCNF, there is redundancy. We repeat the address information of each residence of an instructor once for each department with which the instructor is associated. We could solve this problem by decomposing r_2 further into:

r_{21}(*dept_name*, *ID*)
r_{22}(*ID*, *street*, *city*)

but there is no constraint that leads us to do this.

To deal with this problem, we must define a new form of constraint, called a *multivalued dependency*. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemas. This normal form, called **fourth normal form** (4NF), is more restrictive than BCNF. We shall see that every 4NF schema is also in BCNF but there are BCNF schemas that are not in 4NF.

8.6.1 Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, they *require* that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tuple-generating dependencies**.

Let $r(R)$ be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if, in any legal instance of relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figure 8.13 Tabular representation of $\alpha \twoheadrightarrow \beta$.

This definition is less complicated than it appears to be. Figure 8.13 gives a tabular picture of t_1, t_2, t_3, and t_4. Intuitively, the multivalued dependency $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R - \beta$. If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema R, then $\alpha \twoheadrightarrow \beta$ is a *trivial* multivalued dependency on schema R. Thus, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$.

To illustrate the difference between functional and multivalued dependencies, we consider the schema r_2 again, and an example relation on that schema shown in Figure 8.14. We must repeat the department name once for each address that an instructor has, and we must repeat the address for each department with which an instructor is associated. This repetition is unnecessary, since the relationship between an instructor and his address is independent of the relationship between that instructor and a department. If an instructor with *ID* 22222 is associated with the Physics department, we want that department to be associated with all of that instructor's addresses. Thus, the relation of Figure 8.15 is illegal. To make this relation legal, we need to add the tuples (Physics, 22222, Main, Manchester) and (Math, 22222, North, Rye) to the relation of Figure 8.15.

Comparing the preceding example with our definition of multivalued dependency, we see that we want the multivalued dependency:

$$ID \twoheadrightarrow street, city$$

to hold. (The multivalued dependency *ID* ↠ *dept_name* will do as well. We shall soon see that they are equivalent.)

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

Figure 8.14 An example of redundancy in a relation on a BCNF schema.

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Math	Main	Manchester

Figure 8.15 An illegal r_2 relation.

2. To specify constraints on the set of legal relations; we shall thus concern ourselves with *only* those relations that satisfy a given set of functional and multivalued dependencies

Note that, if a relation r fails to satisfy a given multivalued dependency, we can construct a relation r' that *does* satisfy the multivalued dependency by adding tuples to r.

Let D denote a set of functional and multivalued dependencies. The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D. As we did for functional dependencies, we can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies. We can manage with such reasoning for very simple multivalued dependencies. Luckily, multivalued dependencies that occur in practice appear to be quite simple. For complex dependencies, it is better to reason about sets of dependencies by using a system of inference rules.

From the definition of multivalued dependency, we can derive the following rules for $\alpha, \beta \subseteq R$:

- If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$. In other words, every functional dependency is also a multivalued dependency.
- If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \alpha - \beta$

Appendix C.1.1 outlines a system of inference rules for multivalued dependencies.

8.6.2 Fourth Normal Form

Consider again our example of the BCNF schema:

$$r_2 \text{ (}ID, dept\_name, street, city\text{)}$$

in which the multivalued dependency "$ID \twoheadrightarrow street, city$" holds. We saw in the opening paragraphs of Section 8.6 that, although this schema is in BCNF, the design is not ideal, since we must repeat an instructor's address information for each department. We shall see that we can use the given multivalued dependency to improve the database design, by decomposing this schema into a **fourth normal form** decomposition.

A relation schema $r(R)$ is in **fourth normal form** (4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies

in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for R.

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies. Every 4NF schema is in BCNF. To see this fact, we note that, if a schema $r(R)$ is not in BCNF, then there is a nontrivial functional dependency $\alpha \rightarrow \beta$ holding on R, where α is not a superkey. Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, $r(R)$ cannot be in 4NF.

Let $r(R)$ be a relation schema, and let $r_1(R_1), r_2(R_2), \ldots, r_n(R_n)$ be a decomposition of $r(R)$. To check if each relation schema r_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each r_i. Recall that, for a set F of functional dependencies, the restriction F_i of F to R_i is all functional dependencies in F^+ that include *only* attributes of R_i. Now consider a set D of both functional and multivalued dependencies. The restriction of D to R_i is the set D_i consisting of:

1. All functional dependencies in D^+ that include only attributes of R_i.
2. All multivalued dependencies of the form:

$$\alpha \twoheadrightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.

8.6.3 4NF Decomposition

The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema into 4NF. Figure 8.16 shows the 4NF decomposition algorithm. It is identical to the BCNF decomposition algorithm of Figure 8.11, except that it uses multivalued dependencies and uses the restriction of D^+ to R_i.

If we apply the algorithm of Figure 8.16 to (*ID, dept_name, street, city*), we find that $ID \twoheadrightarrow dept\_name$ is a nontrivial multivalued dependency, and *ID* is not a superkey for the schema. Following the algorithm, we replace it by two schemas:

$$r_{21} \text{ (}ID, dept\_name\text{)}$$
$$r_{22} \text{ (}ID, street, city\text{)}$$

This pair of schemas, which is in 4NF, eliminates the redundancy we encountered earlier.

As was the case when we were dealing solely with functional dependencies, we are interested in decompositions that are lossless and that preserve dependen-

```
result := {R};
done := false;
compute D+; Given schema Ri, let Di denote the restriction of D− to Ri
while (not done) do
    if (there is a schema Ri in result that is not in 4NF w.r.t. Di)
        then begin
            let α →→ β be a nontrivial multivalued dependency that holds
            on Ri such that α → Ri is not in Di, and α ∩ β = ∅;
            result := (result − Ri) ∪ (Ri − β) ∪ (α, β);
        end
    else done := true;
```

Figure 8.16 4NF decomposition algorithm.

cies. The following fact about multivalued dependencies and losslessness shows that the algorithm of Figure 8.16 generates only lossless decompositions:

- Let $r(R)$ be a relation schema, and let D be a set of functional and multivalued dependencies on R. Let $r_1(R_1)$ and $r_2(R_2)$ form a decomposition of R. This decomposition is lossless of R if and only if at least one of the following multivalued dependencies is in D^+:

$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

Recall that we stated in Section 8.4.4 that, if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$, then $r_1(R_1)$ and $r_2(R_2)$ are a lossless decomposition $r(R)$. The preceding fact about multivalued dependencies is a more general statement about losslessness. It says that, for *every* lossless decomposition of $r(R)$ into two schemas $r_1(R_1)$ and $r_2(R_2)$, one of the two dependencies $R_1 \cap R_2 \twoheadrightarrow R_1$ or $R_1 \cap R_2 \twoheadrightarrow R_2$ must hold.

The issue of dependency preservation when we decompose a relation schema becomes more complicated in the presence of multivalued dependencies. Appendix C.1.2 pursues this topic.

8.7 More Normal Forms

The fourth normal form is by no means the "ultimate" normal form. As we saw earlier, multivalued dependencies help us understand and eliminate some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called join dependencies that generalize multivalued dependencies, and lead to another normal form called project-join normal form (PJNF) (PJNF is called fifth normal form in some books). There is a class of even more general constraints that leads to a normal form called domain-key normal form (DKNF).

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints. Hence PJNF and DKNF are used quite rarely. Appendix C provides more details about these normal forms.

Conspicuous by its absence from our discussion of normal forms is **second normal form** (2NF). We have not discussed it, because it is of historical interest only. We simply define it, and let you experiment with it in Practice Exercise 8.17.

8.8 Database-Design Process

So far we have looked at detailed issues about normal forms and normalization. In this section, we study how normalization fits into the overall database-design process.

Earlier in the chapter, starting in Section 8.3, we assumed that a relation schema $r(R)$ is given, and proceeded to normalize it. There are several ways in which we could have come up with the schema $r(R)$:

1. $r(R)$ could have been generated in converting an E-R diagram to a set of relation schemas.
2. $r(R)$ could have been a single relation schema containing *all* attributes that are of interest. The normalization process then breaks up $r(R)$ into smaller schemas.
3. $r(R)$ could have been the result of an ad-hoc design of relations that we then test to verify that it satisfies a desired normal form.

In the rest of this section, we examine the implications of these approaches. We also examine some practical issues in database design, including denormalization for performance and examples of bad design that are not detected by normalization.

8.8.1 E-R Model and Normalization

When we define an E-R diagram carefully, identifying all entities correctly, the relation schemas generated from the E-R diagram should not need much further normalization. However, there can be functional dependencies between attributes of an entity. For instance, suppose an *instructor* entity set had attributes *dept_name* and *dept_address*, and there is a functional dependency *dept_name* → *dept_address*. We would then need to normalize the relation generated from *instructor*.

Most examples of such dependencies arise out of poor E-R diagram design. In the above example, if we had designed the E-R diagram correctly, we would have created a *department* entity set with attribute *dept_address* and a relationship set between *instructor* and *department*. Similarly, a relationship set involving more than two entity sets may result in a schema that may not be in a desirable normal form. Since most relationship sets are binary, such cases are relatively rare. (In

fact, some E-R-diagram variants actually make it difficult or impossible to specify nonbinary relationship sets.)

Functional dependencies can help us detect poor E-R design. If the generated relation schemas are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and can be done formally on the relation schemas generated from the E-R model.

A careful reader will have noted that in order for us to illustrate a need for multivalued dependencies and fourth normal form, we had to begin with schemas that were not derived from our E-R design. Indeed, the process of creating an E-R design tends to generate 4NF designs. If a multivalued dependency holds and is not implied by the corresponding functional dependency, it usually arises from one of the following sources:

- A many-to-many relationship set.
- A multivalued attribute of an entity set.

For a many-to-many relationship set each related entity set has its own schema and there is an additional schema for the relationship set. For a multivalued attribute, a separate schema is created consisting of that attribute and the primary key of the entity set (as in the case of the *phone_number* attribute of the entity set *instructor*).

The universal-relation approach to relational database design starts with an assumption that there is one single relation schema containing all attributes of interest. This single schema defines how users and applications interact with the database.

8.8.2 Naming of Attributes and Relationships

A desirable feature of a database design is the **unique-role assumption**, which means that each attribute name has a unique meaning in the database. This prevents us from using the same attribute to mean different things in different schemas. For example, we might otherwise consider using the attribute *number* for phone number in the *instructor* schema and for room number in the *classroom* schema. The join of a relation on schema *instructor* with one on *classroom* is meaningless. While users and application developers can work carefully to ensure use of the right *number* in each circumstance, having a different attribute name for phone number and for room number serves to reduce user errors.

While it is a good idea to keep names for incompatible attributes distinct, if attributes of different relations have the same meaning, it may be a good idea to use the same attribute name. For this reason we used the same attribute name "*name*" for both the *instructor* and the *student* entity sets. If this was not the case (that is, we used different naming conventions for the instructor and student names), then if we wished to generalize these entity sets by creating a *person* entity set, we would have to rename the attribute. Thus, even if we did not currently

have a generalization of *student* and *instructor*, if we foresee such a possibility it is best to use the same name in both entity sets (and relations).

Although technically, the order of attribute names in a schema does not matter, it is convention to list primary-key attributes first. This makes reading default output (as from **select** *) easier.

In large database schemas, relationship sets (and schemas derived therefrom) are often named via a concatenation of the names of related entity sets, perhaps with an intervening hyphen or underscore. We have used a few such names, for example *inst_sec* and *student_sec*. We used the names *teaches* and *takes* instead of using the longer concatenated names. This was acceptable since it is not hard for you to remember the associated entity sets for a few relationship sets. We cannot always create relationship-set names by simple concatenation; for example, a manager or works-for relationship between employees would not make much sense if it were called *employee_employee*! Similarly, if there are multiple relationship sets possible between a pair of entity sets, the relationship-set names must include extra parts to identify the relationship set.

Different organizations have different conventions for naming entity sets. For example, we may call an entity set of students *student* or *students*. We have chosen to use the singular form in our database designs. Using either singular or plural is acceptable, as long as the convention is used consistently across all entity sets.

As schemas grow larger, with increasing numbers of relationship sets, using consistent naming of attributes, relationships, and entities makes life much easier for the database designer and application programmers.

8.8.3 Denormalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose all course prerequisites have to be displayed along with a course information, every time a course is accessed. In our normalized schema, this requires a join of *course* with *prereq*.

One alternative to computing the join on the fly is to store a relation containing all the attributes of *course* and *prereq*. This makes displaying the "full" course information faster. However, the information for a course is repeated for every course prerequisite, and all copies must be updated by the application, whenever a course prerequisite is added or dropped. The process of taking a normalized schema and making it nonnormalized is called **denormalization**, and designers use it to tune performance of systems to support time-critical operations.

A better alternative, supported by many database systems today, is to use the normalized schema, and additionally store the join of *course* and *prereq* as a materialized view. (Recall that a materialized view is a view whose result is stored in the database and brought up to date when the relations used in the view are updated.) Like denormalization, using materialized views does have space and

time overhead; however, it has the advantage that keeping the view up to date is the job of the database system, not the application programmer.

8.8.4 Other Design Issues

There are some aspects of database design that are not addressed by normalization, and can thus lead to bad database design. Data pertaining to time or to ranges of time have several such issues. We give examples here; obviously, such designs should be avoided.

Consider a university database, where we want to store the total number of instructors in each department in different years. A relation *total_inst*(*dept_name*, *year*, *size*) could be used to store the desired information. The only functional dependency on this relation is *dept_name*, *year* $\rightarrow$ *size*, and the relation is in BCNF.

An alternative design is to use multiple relations, each storing the size information for a different year. Let us say the years of interest are 2007, 2008, and 2009; we would then have relations of the form *total_inst_2007*, *total_inst_2008*, *total_inst_2009*, all of which are on the schema (*dept_name*, *size*). The only functional dependency here on each relation would be *dept_name* $\rightarrow$ *size*, so these relations are also in BCNF.

However, this alternative design is clearly a bad idea—we would have to create a new relation every year, and we would also have to write new queries every year, to take each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

Yet another way of representing the same data is to have a single relation *dept_year*(*dept_name*, *total_inst_2007*, *total_inst_2008*, *total_inst_2009*). Here the only functional dependencies are from *dept_name* to the other attributes, and again the relation is in BCNF. This design is also a bad idea since it has problems similar to the previous design—namely we would have to modify the relation schema and write new queries every year. Queries would also be more complicated, since they may have to refer to many attributes.

Representations such as those in the *dept_year* relation, with one column for each value of an attribute, are called **crosstabs**; they are widely used in spreadsheets and reports and in data analysis tools. While such representations are useful for display to users, for the reasons just given, they are not desirable in a database design. SQL includes features to convert data from a normal relational representation to a crosstab, for display, as we discussed in Section 5.6.1.

8.9 Modeling Temporal Data

Suppose we retain data in our university organization showing not only the address of each instructor, but also all former addresses of which the university is aware. We may then ask queries such as "Find all instructors who lived in Princeton in 1981." In this case, we may have multiple addresses for instructors. Each address has an associated start and end date, indicating when the instructor was resident at that address. A special value for the end date, e.g., null, or a value well into the future such as 9999-12-31, can be used to indicate that the instructor is still resident at that address.

In general, **temporal data** are data that have an associated time interval during which they are **valid**.[8] We use the term **snapshot** of data to mean the value of the data at a particular point in time. Thus a snapshot of *course* data gives the values of all attributes, such as title and department, of all courses at a particular point in time.

Modeling temporal data is a challenging problem for several reasons. For example, suppose we have an *instructor* entity set with which we wish to associate a time-varying address. To add temporal information to an address, we would then have to create a multivalued attribute, each of whose values is a composite value containing an address and a time interval. In addition to time-varying attribute values, entities may themselves have an associated valid time. For example, a student entity may have a valid time from the date the student entered the university to the date the student graduated (or left the university). Relationships too may have associated valid times. For example, the *prereq* relationship may record when a course became a prerequisite for another course. We would thus have to add valid time intervals to attribute values, entity sets, and relationship sets. Adding such detail to an E-R diagram makes it very difficult to create and to comprehend. There have been several proposals to extend the E-R notation to specify in a simple manner that an attribute value or relationship is time varying, but there are no accepted standards.

When we track data values across time, functional dependencies that we assumed to hold, such as:

$$ID \rightarrow street, city$$

may no longer hold. The following constraint (expressed in English) would hold instead: "An instructor *ID* has only one *street* and *city* value for any given time *t*."

Functional dependencies that hold at a particular point in time are called temporal functional dependencies. Formally, a **temporal functional dependency** $X \xrightarrow{\tau} Y$ holds on a relation schema $r(R)$ if, for all legal instances of $r(R)$, all snapshots of r satisfy the functional dependency $X \rightarrow Y$.

We could extend the theory of relational database design to take temporal functional dependencies into account. However, reasoning with regular functional dependencies is difficult enough already, and few designers are prepared to deal with temporal functional dependencies.

In practice, database designers fall back to simpler approaches to designing temporal databases. One commonly used approach is to design the entire database (including E-R design and relational design) ignoring temporal changes (equivalently, taking only a snapshot into consideration). After this, the designer

[8] There are other models of temporal data that distinguish between **valid time** and **transaction time**, the latter recording when a fact was recorded in the database. We ignore such details for simplicity.

studies the various relations and decides which relations require temporal variation to be tracked.

The next step is to add valid time information to each such relation, by adding start and end time as attributes. For example, consider the *course* relation. The title of the course may change over time, which can be handled by adding a valid time range; the resultant schema would be

course (*course_id*, *title*, *dept_name*, *start*, *end*)

An instance of this relation might have two records (CS-101, "Introduction to Programming", 1985-01-01, 2000-12-31) and (CS-101, "Introduction to C", 2001-01-01, 9999-12-31). If the relation is updated by changing the course title to "Introduction to Java," the time "9999-12-31" would be updated to the time until which the old value ("Introduction to C") is valid, and a new tuple would be added containing the new title ("Introduction to Java"), with an appropriate start time.

If another relation had a foreign key referencing a temporal relation, the database designer has to decide if the reference is to the current version of the data or to the data as of a particular point in time. For example, we may extend the *department* relation to track changes in the building or budget of a department across time, but a reference from the *instructor* or *student* relation may not care about the history of the building or budget, but may instead implicitly refer to the temporally current record for the corresponding *dept_name*. On the other hand, a record in a student's transcript should refer to the course title at the time when the student took the course. In this latter case, the referencing relation must also record time information, to identify a particular record from the *course* relation. In our example, the *year* and *semester* when the course was taken can be mapped to a representative time/date value such as midnight of the start date of the semester; the resulting time/date value is used to identify a particular record from the temporal version of the *course* relation, from which the *title* is retrieved.

The original primary key for a temporal relation would no longer uniquely identify a tuple. To resolve this problem, we could add the start and end time attributes to the primary key. However, some problems remain:

- It is possible to store data with overlapping intervals, which the primary-key constraint would not catch. If the system supports a native *valid time* type, it can detect and prevent such overlapping time intervals.
- To specify a foreign key referencing such a relation, the referencing tuples would have to include the start- and end-time attributes as part of their foreign key, and the values must match that in the referenced tuple. Further, if the referenced tuple is updated (and the end time which was in the future is updated), the update must propagate to all the referencing tuples.

  If the system supports temporal data in a better fashion, we can allow the referencing tuple to specify a point in time, rather than a range, and rely on the system to ensure that there is a tuple in the referenced relation whose valid time interval contains the point in time. For example, a transcript record may specify a *course_id* and a time (say the start date of a semester), which is enough to identify the correct record in the *course* relation.

As a common special case, if all references to temporal data refer to only the current data, a simpler solution is to not add time information to the relation, but instead create a corresponding *history* relation that has temporal information, for past values. For example, in our bank database, we could use the design we have created, ignoring temporal changes, to store only the current information. All historical information is moved to historical relations. Thus, the *instructor* relation may store only the current address, while a relation *instructor_history* may contain all the attributes of *instructor*, with additional *start_time* and *end_time* attributes.

Although we have not provided any formal way to deal with temporal data, the issues we have discussed and the examples we have provided should help you in designing a database that records temporal data. Further issues in handling temporal data, including temporal queries, are covered later, in Section 25.2.

8.10 Summary

- We showed pitfalls in database design, and how to systematically design a database schema that avoids the pitfalls. The pitfalls included repeated information and inability to represent some information.
- We showed the development of a relational database design from an E-R design, when schemas may be combined safely, and when a schema should be decomposed. All valid decompositions must be lossless.
- We described the assumptions of atomic domains and first normal form.
- We introduced the concept of functional dependencies, and used it to present two normal forms, Boyce–Codd normal form (BCNF) and third normal form (3NF).
- If the decomposition is dependency preserving, given a database update, all functional dependencies can be verifiable from individual relations, without computing a join of relations in the decomposition.
- We showed how to reason with functional dependencies. We placed special emphasis on what dependencies are logically implied by a set of dependencies. We also defined the notion of a canonical cover, which is a minimal set of functional dependencies equivalent to a given set of functional dependencies.
- We outlined an algorithm for decomposing relations into BCNF. There are relations for which there is no dependency-preserving BCNF decomposition.
- We used the canonical covers to decompose a relation into 3NF, which is a small relaxation of the BCNF condition. Relations in 3NF may have some redundancy, but there is always a dependency-preserving decomposition into 3NF.

- We presented the notion of multivalued dependencies, which specify constraints that cannot be specified with functional dependencies alone. We defined fourth normal form (4NF) with multivalued dependencies. Appendix C.1.1 gives details on reasoning about multivalued dependencies.
- Other normal forms, such as PJNF and DKNF, eliminate more subtle forms of redundancy. However, these are hard to work with and are rarely used. Appendix C gives details on these normal forms.
- In reviewing the issues in this chapter, note that the reason we could define rigorous approaches to relational database design is that the relational data model rests on a firm mathematical foundation. That is one of the primary advantages of the relational model compared with the other data models that we have studied.

Review Terms

- E-R model and normalization
- Decomposition
- Functional dependencies
- Lossless decomposition
- Atomic domains
- First normal form (1NF)
- Legal relations
- Superkey
- R satisfies F
- F holds on R
- Boyce–Codd normal form (BCNF)
- Dependency preservation
- Third normal form (3NF)
- Trivial functional dependencies
- Closure of a set of functional dependencies
- Armstrong's axioms
- Closure of attribute sets
- Restriction of F to R_i
- Canonical cover
- Extraneous attributes
- BCNF decomposition algorithm
- 3NF decomposition algorithm
- Multivalued dependencies
- Fourth normal form (4NF)
- Restriction of a multivalued dependency
- Project-join normal form (PJNF)
- Domain-key normal form (DKNF)
- Universal relation
- Unique-role assumption
- Denormalization

Practice Exercises

8.1 Suppose that we decompose the schema $r(A, B, C, D, E)$ into

$$r_1(A, B, C)$$
$$r_2(A, D, E)$$

Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

8.2 List all functional dependencies satisfied by the relation of Figure 8.17.

8.3 Explain how functional dependencies can be used to indicate the following:

- A one-to-one relationship set exists between entity sets *student* and *instructor*.
- A many-to-one relationship set exists between entity sets *student* and *instructor*.

8.4 Use Armstrong's axioms to prove the soundness of the union rule. (*Hint*: Use the augmentation rule to show that, if $\alpha \rightarrow \beta$, then $\alpha \rightarrow \alpha\beta$. Apply the augmentation rule again, using $\alpha \rightarrow \gamma$, and then apply the transitivity rule.)

8.5 Use Armstrong's axioms to prove the soundness of the pseudotransitivity rule.

8.6 Compute the closure of the following set F of functional dependencies for relation schema r (A, B, C, D, E).

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

List the candidate keys for R.

8.7 Using the functional dependencies of Practice Exercise 8.6, compute the canonical cover F_c.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figure 8.17 Relation of Practice Exercise 8.2.

8.8 Consider the algorithm in Figure 8.18 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 8.8 (Section 8.4.2) and that it computes α^+ correctly.

8.9 Given the database schema $R(a, b, c)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $b \rightarrow c$ holds on relation r. Also write an SQL assertion that enforces the functional dependency; assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)

8.10 Our discussion of lossless-join decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?

8.11 In the BCNF decomposition algorithm, suppose you use a functional dependency $\alpha \rightarrow \beta$ to decompose a relation schema $r(\alpha, \beta, \gamma)$ into $r_1(\alpha, \beta)$ and $r_2(\alpha, \gamma)$.

a. What primary and foreign-key constraint do you expect to hold on the decomposed relations?

b. Give an example of an inconsistency that can arise due to an erroneous update, if the foreign-key constraint were not enforced on the decomposed relations above.

c. When a relation is decomposed into 3NF using the algorithm in Section 8.5.2, what primary and foreign key dependencies would you expect will hold on the decomposed schema?

8.12 Let $R_1, R_2, \ldots, R_n$ be a decomposition of schema U. Let $u(U)$ be a relation, and let $r_i = \Pi_{R_i}(u)$. Show that

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

8.13 Show that the decomposition in Practice Exercise 8.1 is not a dependency-preserving decomposition.

8.14 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

8.15 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless decompositions of R' into BCNF.


```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
  begin
    let β → γ denote the ith FD;
    fdcount [i] := |β|;
  end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
  begin
    appears [A] := NIL;
    for i := 1 to |F| do
      begin
        let β → γ denote the ith FD;
        if A ∈ β then add i to appears [A];
      end
  end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
  begin
    if A ∉ result then
      begin
        result := result ∪ {A};
        for each element i of appears[A] do
          begin
            fdcount [i] := fdcount [i] − 1;
            if fdcount [i] := 0 then
              begin
                let β → γ denote the ith FD;
                addin (γ);
              end
          end
      end
  end
```

Figure 8.18 An algorithm to compute α^+.

8.16 Let a prime attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that A is **transitively dependent** on α. We can restate our definition of 3NF as follows: *A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R.* Show that this new definition is equivalent to the original one.

8.17 A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$. We say that β is *partially dependent* on α. A relation schema R is in **second normal form** (2NF) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

8.18 Give an example of a relation schema R and a set of dependencies such that R is in BCNF but is not in 4NF.

Exercises

8.19 Give a lossless-join decomposition into BCNF of schema R of Practice Exercise 8.1.

8.20 Give a lossless-join, dependency-preserving decomposition into 3NF of schema R of Practice Exercise 8.1.

8.21 Normalize the following schema, with given constraints, to 4NF.

books(accessionno, isbn, title, author, publisher)
users(userid, name, deptid, deptname)
accessionno $\rightarrow$ *isbn*
isbn $\rightarrow$ *title*
isbn $\rightarrow$ *publisher*
isbn $\rightarrow\!\!\!\rightarrow$ *author*
userid $\rightarrow$ *name*
userid $\rightarrow$ *deptid*
deptid $\rightarrow$ *deptname*

8.22 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational database design.

8.23 Why are certain functional dependencies called *trivial* functional dependencies?

8.24 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

8.25 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

8.26 Use Armstrong's axioms to prove the soundness of the decomposition rule.

8.27 Using the functional dependencies of Practice Exercise 8.6, compute B^+.

8.28 Show that the following decomposition of the schema R of Practice Exercise 8.1 is not a lossless decomposition:

(A, B, C)
(C, D, E)

Hint: Give an example of a relation r on schema R such that

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

8.29 Consider the following set F of functional dependencies on the relation schema $r(A, B, C, D, E, F)$:

$A \rightarrow BCD$
$BC \rightarrow DE$
$B \rightarrow D$
$D \rightarrow A$

a. Compute B^+.
b. Prove (using Armstrong's axioms) that AF is a superkey.
c. Compute a canonical cover for the above set of functional dependencies F; give each step of your derivation with an explanation.
d. Give a 3NF decomposition of r based on the canonical cover.
e. Give a BCNF decomposition of r using the original set of functional dependencies.
f. Can you get the same BCNF decomposition of r as above, using the canonical cover?

8.30 List the three design goals for relational databases, and explain why each is desirable.

8.31 In designing a relational database, why might we choose a non-BCNF design?

8.32 Given the three goals of relational database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Practice Exercise 8.17 for the definition of 2NF.)

8.33 Given a relational schema $r(A, B, C, D)$, does $A \twoheadrightarrow BC$ logically imply $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$? If yes prove it, else give a counter example.

8.34 Explain why 4NF is a normal form more desirable than BCNF.

Bibliographical Notes

The first discussion of relational database design theory appeared in an early paper by Codd [1970]. In that paper, Codd also introduced functional dependencies and first, second, and third normal forms.

Armstrong's axioms were introduced in Armstrong [1974]. Significant development of relational database theory occurred in the late 1970s. These results are collected in several texts on database theory including Maier [1983], Atzeni and Antonellis [1993], and Abiteboul et al. [1995].

BCNF was introduced in Codd [1972]. Biskup et al. [1979] give the algorithm we used to find a lossless dependency-preserving decomposition into 3NF. Fundamental results on the lossless decomposition property appear in Aho et al. [1979a].

Beeri et al. [1977] gives a set of axioms for multivalued dependencies, and proves that the authors' axioms are sound and complete. The notions of 4NF, PJNF, and DKNF are from Fagin [1977], Fagin [1979], and Fagin [1981], respectively. See the bibliographical notes of Appendix C for further references to literature on normalization.

Jensen et al. [1994] presents a glossary of temporal-database concepts. A survey of extensions to E-R modeling to handle temporal data is presented by Gregersen and Jensen [1999]. Tansel et al. [1993] covers temporal database theory, design, and implementation. Jensen et al. [1996] describes extensions of dependency theory to temporal data.