Chapter 3

Design Theory for Relational Databases

There are many ways we could go about designing a relational database schema for an application. In Chapter 4 we shall see several high-level notations for describing the structure of data and the ways in which these high-level designs can be converted into relations. We can also examine the requirements for a database and define relations directly, without going through a high-level intermediate stage. Whatever approach we use, it is common for an initial relational schema to have room for improvement, especially by eliminating redundancy. Often, the problems with a schema involve trying to combine too much into one relation.

Fortunately, there is a well developed theory for relational databases: "dependencies," their implications for what makes a good relational database schema, and what we can do about a schema if it has flaws. In this chapter, we first identify the problems that are caused in some relation schemas by the presence of certain dependencies; these problems are referred to as "anomalies."

Our discussion starts with "functional dependencies," a generalization of the idea of a key for a relation. We then use the notion of functional dependencies to define normal forms for relation schemas. The impact of this theory, called "normalization," is that we decompose relations into two or more relations when that will remove anomalies. Next, we introduce "multivalued dependencies," which intuitively represent a condition where one or more attributes of a relation are independent from one or more other attributes. These dependencies also lead to normal forms and decomposition of relations to eliminate redundancy.

3.1 Functional Dependencies

There is a design theory for relations that lets us examine a design carefully and make improvements based on a few simple principles. The theory begins by having us state the constraints that apply to the relation. The most common constraint is the "functional dependency," a statement of a type that generalizes the idea of a key for a relation, which we introduced in Section 2.5.3. Later in this chapter, we shall see how this theory gives us simple tools to improve our designs by the process of "decomposition" of relations: the replacement of one relation by several, whose sets of attributes together include all the attributes of the original.

3.1.1 Definition of Functional Dependency

A *functional dependency* (FD) on a relation R is a statement of the form "If two tuples of R agree on all of the attributes $A_1, A_2, \ldots, A_n$ (i.e., the tuples have the same values in their respective components for each of these attributes), then they must also agree on all of another list of attributes $B_1, B_2, \ldots, B_m$. We write this FD formally as $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ and say that

"$A_1, A_2, \ldots, A_n$ functionally determine $B_1, B_2, \ldots, B_m$"

Figure 3.1 suggests what this FD tells us about any two tuples t and u in the relation R. However, the A's and B's can be anywhere; it is not necessary for the A's and B's to appear consecutively or for the A's to precede the B's.

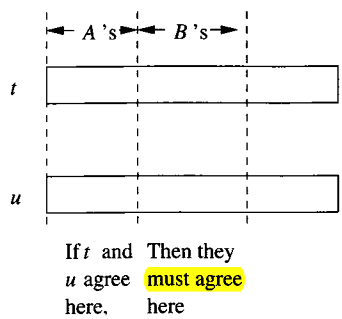

Figure 3.1: The effect of a functional dependency on two tuples.

If we can be sure every instance of a relation R will be one in which a given FD is true, then we say that *R satisfies* the FD. It is important to remember that when we say that R satisfies an FD f, we are asserting a constraint on R, not just saying something about one particular instance of R.

It is common for the right side of an FD to be a single attribute. In fact, we shall see that the one functional dependency $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ is equivalent to the set of FD's:

$$
\begin{array}{l}
A_1 A_2 \cdots A_n \rightarrow B_1 \\
A_1 A_2 \cdots A_n \rightarrow B_2 \\
\cdots \\
A_1 A_2 \cdots A_n \rightarrow B_m
\end{array}
$$

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 3.2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

Example 3.1: Let us consider the relation

`Movies1(title, year, length, genre, studioName, starName)`

an instance of which is shown in Fig. 3.2. While related to our running `Movies` relation, it has additional attributes, which is why we call it "`Movies1`" instead of "`Movies`." Notice that this relation tries to "do too much." It holds information that in our running database schema was attributed to three different relations: `Movies`, `Studio`, and `StarsIn`. As we shall see, the schema for `Movies1` is not a good design. But to see what is wrong with the design, we must first determine the functional dependencies that hold for the relation. We claim that the following FD holds:

`title year` → `length genre studioName`

Informally, this FD says that if two tuples have the same value in their `title` components, and they also have the same value in their `year` components, then these two tuples must also have the same values in their `length` components, the same values in their `genre` components, and the same values in their `studioName` components. This assertion makes sense, since we believe that it is not possible for there to be two movies released in the same year with the same title (although there could be movies of the same title released in different years). This point was discussed in Example 2.1. Thus, we expect that given a title and year, there is a unique movie. Therefore, there is a unique length for the movie, a unique genre, and a unique studio.

On the other hand, we observe that the statement

`title year` → `starName`

is false; it is not a functional dependency. Given a movie, it is entirely possible that there is more than one star for the movie listed in our database. Notice that even had we been lazy and only listed one star for *Star Wars* and one star for *Wayne's World* (just as we only listed one of the many stars for *Gone With the Wind*), this FD would not suddenly become true for the relation `Movies1`.

The reason is that the FD says something about all possible instances of the relation, not about one of its instances. The fact that we *could* have an instance with multiple stars for a movie rules out the possibility that title and year functionally determine starName. □

3.1.2 Keys of Relations

We say a set of one or more attributes $\{A_1, A_2, \ldots, A_n\}$ is a *key* for a relation R if:

1. Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of R to agree on all of $A_1, A_2, \ldots, A_n$.

2. No proper subset of $\{A_1, A_2, \ldots, A_n\}$ functionally determines all other attributes of R; i.e., a key must be *minimal*.

When a key consists of a single attribute A, we often say that A (rather than $\{A\}$) is a key.

Example 3.2: Attributes {`title`, `year`, `starName`} form a key for the relation `Movies1` of Fig. 3.2. First, we must show that they functionally determine all the other attributes. That is, suppose two tuples agree on these three attributes: `title`, `year`, and `starName`. Because they agree on `title` and `year`, they must agree on the other attributes — `length`, `genre`, and `studioName` — as we discussed in Example 3.1. Thus, two different tuples cannot agree on all of `title`, `year`, and `starName`; they would in fact be the same tuple.

Now, we must argue that no proper subset of {`title`, `year`, `starName`} functionally determines all other attributes. To see why, begin by observing that `title` and `year` do not determine `starName`, because many movies have more than one star. Thus, {`title`, `year`} is not a key.

{`year`, `starName`} is not a key because we could have a star in two movies in the same year; therefore

$$\texttt{year starName} \rightarrow \texttt{title}$$

is not an FD. Also, we claim that {`title`, `starName`} is not a key, because two movies with the same title, made in different years, occasionally have a star in common.[1] □

Sometimes a relation has more than one key. If so, it is common to designate one of the keys as the *primary key*. In commercial database systems, the choice of primary key can influence some implementation issues such as how the relation is stored on disk. However, the theory of FD's gives no special role to "primary keys."

[1] Since we asserted in an earlier book that there were no known examples of this phenomenon, several people have shown us we were wrong. It's an interesting challenge to discover stars that appeared in two versions of the same movie.

What Is "Functional" About Functional Dependencies?

$A_1 A_2 \cdots A_n \rightarrow B$ is called a "functional" dependency because in principle there is a function that takes a list of values, one for each of attributes $A_1, A_2, \ldots, A_n$ and produces a unique value (or no value at all) for B. For instance, in the `Movies1` relation, we can imagine a function that takes a string like `"Star Wars"` and an integer like 1977 and produces the unique value of `length`, namely 124, that appears in the relation `Movies1`. However, this function is not the usual sort of function that we meet in mathematics, because there is no way to compute it from first principles. That is, we cannot perform some operations on strings like `"Star Wars"` and integers like 1977 and come up with the correct length. Rather, the function is only computed by lookup in the relation. We look for a tuple with the given `title` and `year` values and see what value that tuple has for `length`.

3.1.3 Superkeys

A set of attributes that contains a key is called a *superkey*, short for "superset of a key." Thus, every key is a superkey. However, some superkeys are not (minimal) keys. Note that every superkey satisfies the first condition of a key: it functionally determines all other attributes of the relation. However, a superkey need not satisfy the second condition: minimality.

Example 3.3: In the relation of Example 3.2, there are many superkeys. Not only is the key

{`title`, `year`, `starName`}

a superkey, but any superset of this set of attributes, such as

{`title`, `year`, `starName`, `length`, `studioName`}

is a superkey. □

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Consider a relation about people in the United States, including their name, Social Security number, street address, city, state, ZIP code, area code, and phone number (7 digits). What FD's would you expect to hold? What are the keys for the relation? To answer this question, you need to know something about the way these numbers are assigned. For instance, can an area

Other Key Terminology

In some books and articles one finds different terminology regarding keys. One can find the term "key" used the way we have used the term "superkey," that is, a set of attributes that functionally determine all the attributes, with no requirement of minimality. These sources typically use the term "candidate key" for a key that is minimal — that is, a "key" in the sense we use the term.

code straddle two states? Can a ZIP code straddle two area codes? Can two people have the same Social Security number? Can they have the same address or phone number?

Exercise 3.1.2: Consider a relation representing the present position of molecules in a closed container. The attributes are an ID for the molecule, the x, y, and z coordinates of the molecule, and its velocity in the x, y, and z dimensions. What FD's would you expect to hold? What are the keys?

!! **Exercise 3.1.3:** Suppose R is a relation with attributes $A_1, A_2, \ldots, A_n$. As a function of n, tell how many superkeys R has, if:

a) The only key is A_1.

b) The only keys are A_1 and A_2.

c) The only keys are $\{A_1, A_2\}$ and $\{A_3, A_4\}$.

d) The only keys are $\{A_1, A_2\}$ and $\{A_1, A_3\}$.

3.2 Rules About Functional Dependencies

In this section, we shall learn how to *reason* about FD's. That is, suppose we are told of a set of FD's that a relation satisfies. Often, we can deduce that the relation must satisfy certain other FD's. This ability to discover additional FD's is essential when we discuss the design of good relation schemas in Section 3.3.

3.2.1 Reasoning About Functional Dependencies

Let us begin with a motivating example that will show us how we can infer a functional dependency from other given FD's.

Example 3.4: If we are told that a relation $R(A, B, C)$ satisfies the FD's $A \to B$ and $B \to C$, then we can deduce that R also satisfies the FD $A \to C$. How does that reasoning go? To prove that $A \to C$, we must consider two tuples of R that agree on A and prove they also agree on C.

Let the tuples agreeing on attribute A be (a, b_1, c_1) and (a, b_2, c_2). Since R satisfies $A \to B$, and these tuples agree on A, they must also agree on B. That is, $b_1 = b_2$, and the tuples are really (a, b, c_1) and (a, b, c_2), where b is both b_1 and b_2. Similarly, since R satisfies $B \to C$, and the tuples agree on B, they agree on C. Thus, $c_1 = c_2$; i.e., the tuples *do* agree on C. We have proved that any two tuples of R that agree on A also agree on C, and that is the FD $A \to C$. □

FD's often can be presented in several different ways, without changing the set of legal instances of the relation. We say:

- Two sets of FD's S and T are *equivalent* if the set of relation instances satisfying S is exactly the same as the set of relation instances satisfying T.

- More generally, a set of FD's S *follows* from a set of FD's T if every relation instance that satisfies all the FD's in T also satisfies all the FD's in S.

Note then that two sets of FD's S and T are equivalent if and only if S follows from T, and T follows from S.

In this section we shall see several useful rules about FD's. In general, these rules let us replace one set of FD's by an equivalent set, or to add to a set of FD's others that follow from the original set. An example is the *transitive rule* that lets us follow chains of FD's, as in Example 3.4. We shall also give an algorithm for answering the general question of whether one FD follows from one or more other FD's.

3.2.2 The Splitting/Combining Rule

Recall that in Section 3.1.1 we commented that the FD:

$$A_1 A_2 \cdots A_n \to B_1 B_2 \cdots B_m$$

was equivalent to the set of FD's:

$$A_1 A_2 \cdots A_n \to B_1, \quad A_1 A_2 \cdots A_n \to B_2, \ldots, A_1 A_2 \cdots A_n \to B_m$$

That is, we may split attributes on the right side so that only one attribute appears on the right of each FD. Likewise, we can replace a collection of FD's having a common left side by a single FD with the same left side and all the right sides combined into one set of attributes. In either event, the new set of FD's is equivalent to the old. The equivalence noted above can be used in two ways.

- We can replace an FD $A_1 A_2 \cdots A_n \to B_1 B_2 \cdots B_m$ by a set of FD's $A_1 A_2 \cdots A_n \to B_i$ for $i = 1, 2, \ldots, m$. This transformation we call the *splitting rule.*

- We can replace a set of FD's $A_1A_2 \cdots A_n \rightarrow B_i$ for $i = 1, 2, \ldots, m$ by the single FD $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$. We call this transformation the *combining rule.*

Example 3.5: In Example 3.1 the set of FD's:

```
title year → length
title year → genre
title year → studioName
```

is equivalent to the single FD:

```
title year → length genre studioName
```

that we asserted there. □

The reason the splitting and combining rules are true should be obvious. Suppose we have two tuples that agree in $A_1, A_2, \ldots, A_n$. As a single FD, we would assert "then the tuples must agree in all of $B_1, B_2, \ldots, B_m$." As individual FD's, we assert "then the tuples agree in B_1, and they agree in B_2, and,..., and they agree in B_m." These two conclusions say exactly the same thing.

One might imagine that splitting could be applied to the left sides of FD's as well as to right sides. However, there is no splitting rule for left sides, as the following example shows.

Example 3.6: Consider one of the FD's such as:

```
title year → length
```

for the relation `Movies1` in Example 3.1. If we try to split the left side into

```
title → length
year → length
```

then we get two false FD's. That is, `title` does not functionally determine `length`, since there can be several movies with the same title (e.g., *King Kong*) but of different lengths. Similarly, `year` does not functionally determine `length`, because there are certainly movies of different lengths made in any one year. □

3.2.3 Trivial Functional Dependencies

A constraint of any kind on a relation is said to be *trivial* if it holds for every instance of the relation, regardless of what other constraints are assumed. When the constraints are FD's, it is easy to tell whether an FD is trivial. They are the FD's $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ such that

$$\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$$

That is, a trivial FD has a right side that is a subset of its left side. For example,

$$\texttt{title year} \rightarrow \texttt{title}$$

is a trivial FD, as is

$$\texttt{title} \rightarrow \texttt{title}$$

Every trivial FD holds in every relation, since it says that "two tuples that agree in all of $A_1, A_2, \ldots, A_n$ agree in a subset of them." Thus, we may assume any trivial FD, without having to justify it on the basis of what FD's are asserted for the relation.

There is an intermediate situation in which some, but not all, of the attributes on the right side of an FD are also on the left. This FD is not trivial, but it can be simplifed by removing from the right side of an FD those attributes that appear on the left. That is:

- The FD $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ is equivalent to

  $$A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$$

  where the C's are all those B's that are not also A's.

We call this rule, illustrated in Fig. 3.3, the *trivial-dependency rule*.

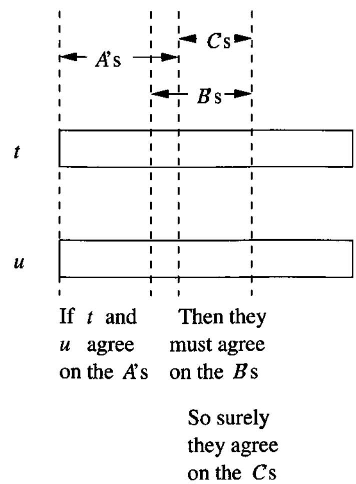

Figure 3.3: The trivial-dependency rule

3.2.4 Computing the Closure of Attributes

Before proceeding to other rules, we shall give a general principle from which all true rules follow. Suppose $\{A_1, A_2, \ldots, A_n\}$ is a set of attributes and S

is a set of FD's. The *closure* of $\{A_1, A_2, \ldots, A_n\}$ under the FD's in S is the set of attributes B such that every relation that satisfies all the FD's in set S also satisfies $A_1A_2\cdots A_n \rightarrow B$. That is, $A_1A_2\cdots A_n \rightarrow B$ follows from the FD's of S. We denote the closure of a set of attributes $A_1A_2\cdots A_n$ by $\{A_1, A_2, \ldots, A_n\}^+$. Note that $A_1, A_2, \ldots, A_n$ are always in $\{A_1, A_2, \ldots, A_n\}^+$ because the FD $A_1A_2\cdots A_n \rightarrow A_i$ is trivial when i is one of $1, 2, \ldots, n$.

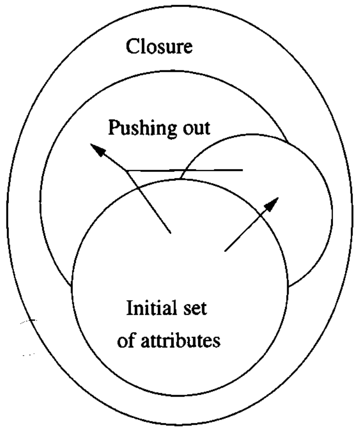

Figure 3.4: Computing the closure of a set of attributes

Figure 3.4 illustrates the closure process. Starting with the given set of attributes, we repeatedly expand the set by adding the right sides of FD's as soon as we have included their left sides. Eventually, we cannot expand the set any further, and the resulting set is the closure. More precisely:

Algorithm 3.7: Closure of a Set of Attributes.

INPUT: A set of attributes $\{A_1, A_2, \ldots, A_n\}$ and a set of FD's S.

OUTPUT: The closure $\{A_1, A_2, \ldots, A_n\}^+$.

1. If necessary, split the FD's of S, so each FD in S has a single attribute on the right.

2. Let X be a set of attributes that eventually will become the closure. Initialize X to be $\{A_1, A_2, \ldots, A_n\}$.

3. Repeatedly search for some FD

$$B_1B_2\cdots B_m \rightarrow C$$

such that all of $B_1, B_2, \ldots, B_m$ are in the set of attributes X, but C is not. Add C to the set X and repeat the search. Since X can only grow, and the number of attributes of any relation schema must be finite, eventually nothing more can be added to X, and this step ends.

4. The set X, after no more attributes can be added to it, is the correct value of $\{A_1, A_2, \ldots, A_n\}^+$.

□

Example 3.8: Let us consider a relation with attributes A, B, C, D, E, and F. Suppose that this relation has the FD's $AB \to C$, $BC \to AD$, $D \to E$, and $CF \to B$. What is the closure of $\{A, B\}$, that is, $\{A, B\}^+$?

First, split $BC \to AD$ into $BC \to A$ and $BC \to D$. Then, start with $X = \{A, B\}$. First, notice that both attributes on the left side of FD $AB \to C$ are in X, so we may add the attribute C, which is on the right side of that FD. Thus, after one iteration of Step 3, X becomes $\{A, B, C\}$.

Next, we see that the left sides of $BC \to A$ and $BC \to D$ are now contained in X, so we may add to X the attributes A and D. A is already there, but D is not, so X next becomes $\{A, B, C, D\}$. At this point, we may use the FD $D \to E$ to add E to X, which is now $\{A, B, C, D, E\}$. No more changes to X are possible. In particular, the FD $CF \to B$ can not be used, because its left side never becomes contained in X. Thus, $\{A, B\}^+ = \{A, B, C, D, E\}$. □

By computing the closure of any set of attributes, we can test whether any given FD $A_1A_2 \cdots A_n \to B$ follows from a set of FD's S. First compute $\{A_1, A_2, \ldots, A_n\}^+$ using the set of FD's S. If B is in $\{A_1, A_2, \ldots, A_n\}^+$, then $A_1A_2 \cdots A_n \to B$ does follow from S, and if B is not in $\{A_1, A_2, \ldots, A_n\}^+$, then this FD does not follow from S. More generally, $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$ follows from set of FD's S if and only if all of $B_1, B_2, \ldots, B_m$ are in

$$\{A_1, A_2, \ldots, A_n\}^+$$

Example 3.9: Consider the relation and FD's of Example 3.8. Suppose we wish to test whether $AB \to D$ follows from these FD's. We compute $\{A, B\}^+$, which is $\{A, B, C, D, E\}$, as we saw in that example. Since D is a member of the closure, we conclude that $AB \to D$ does follow.

On the other hand, consider the FD $D \to A$. To test whether this FD follows from the given FD's, first compute $\{D\}^+$. To do so, we start with $X = \{D\}$. We can use the FD $D \to E$ to add E to the set X. However, then we are stuck. We cannot find any other FD whose left side is contained in $X = \{D, E\}$, so $\{D\}^+ = \{D, E\}$. Since A is not a member of $\{D, E\}$, we conclude that $D \to A$ does not follow. □

3.2.5 Why the Closure Algorithm Works

In this section, we shall show why Algorithm 3.7 correctly decides whether or not an FD $A_1A_2 \cdots A_n \to B$ follows from a given set of FD's S. There are two parts to the proof:

1. We must prove that Algorithm 3.7 does not claim too much. That is, we must show that if $A_1A_2 \cdots A_n \to B$ is asserted by the closure test (i.e.,

B is in $\{A_1, A_2, \ldots, A_n\}^+$), then $A_1A_2 \cdots A_n \rightarrow B$ holds in any relation that satisfies all the FD's in S.

2. We must prove that Algorithm 3.7 does not fail to discover a FD that truly follows from the set of FD's S.

Why the Closure Algorithm Claims only True FD's

We can prove by induction on the number of times that we apply the growing operation of Step 3 that for every attribute D in X, the FD $A_1A_2 \cdots A_n \rightarrow D$ holds. That is, every relation R satisfying all of the FD's in S also satisfies $A_1A_2 \cdots A_n \rightarrow D$.

BASIS: The basis case is when there are zero steps. Then D must be one of $A_1, A_2, \ldots, A_n$, and surely $A_1A_2 \cdots A_n \rightarrow D$ holds in any relation, because it is a trivial FD.

INDUCTION: For the induction, suppose D was added when we used the FD $B_1B_2 \cdots B_m \rightarrow D$ of S. We know by the inductive hypothesis that R satisfies $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$. Now, suppose two tuples of R agree on all of $A_1, A_2, \ldots, A_n$. Then since R satisfies $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$, the two tuples must agree on all of $B_1, B_2, \ldots, B_m$. Since R satisfies $B_1B_2 \cdots B_m \rightarrow D$, we also know these two tuples agree on D. Thus, R satisfies $A_1A_2 \cdots A_n \rightarrow D$.

Why the Closure Algorithm Discovers All True FD's

Suppose $A_1A_2 \cdots A_n \rightarrow B$ were an FD that Algorithm 3.7 says does not follow from set S. That is, the closure of $\{A_1, A_2, \ldots, A_n\}$ using set of FD's S does not include B. We must show that FD $A_1A_2 \cdots A_n \rightarrow B$ really doesn't follow from S. That is, we must show that there is at least one relation instance that satisfies all the FD's in S, and yet does not satisfy $A_1A_2 \cdots A_n \rightarrow B$.

This instance I is actually quite simple to construct; it is shown in Fig. 3.5. I has only two tuples: t and s. The two tuples agree in all the attributes of $\{A_1, A_2, \ldots, A_n\}^+$, and they disagree in all the other attributes. We must show first that I satisfies all the FD's of S, and then that it does not satisfy $A_1A_2 \cdots A_n \rightarrow B$.

	$\{A_1, A_2, \ldots, A_n\}^+$	Other Attributes
t:	$1\,1\,1 \cdots 1\,1$	$0\,0\,0 \cdots 0\,0$
s:	$1\,1\,1 \cdots 1\,1$	$1\,1\,1 \cdots 1\,1$

Figure 3.5: An instance I satisfying S but not $A_1A_2 \cdots A_n \rightarrow B$

Suppose there were some FD $C_1C_2 \cdots C_k \rightarrow D$ in set S (after splitting right sides) that instance I does not satisfy. Since I has only two tuples, t and s, those must be the two tuples that violate $C_1C_2 \cdots C_k \rightarrow D$. That is, t and s agree in all the attributes of $\{C_1, C_2, \ldots, C_k\}$, yet disagree on D. If we examine Fig. 3.5 we see that all of $C_1, C_2, \ldots, C_k$ must be among the attributes of $\{A_1, A_2, \ldots, A_n\}^+$, because those are the only attributes on which t and s agree. Likewise, D must be among the other attributes, because only on those attributes do t and s disagree.

But then we did not compute the closure correctly. $C_1C_2 \cdots C_k \rightarrow D$ should have been applied when X was $\{A_1, A_2, \ldots, A_n\}$ to add D to X. We conclude that $C_1C_2 \cdots C_k \rightarrow D$ cannot exist; i.e., instance I satisfies S.

Second, we must show that I does not satisfy $A_1A_2 \cdots A_n \rightarrow B$. However, this part is easy. Surely, $A_1, A_2, \ldots, A_n$ are among the attributes on which t and s agree. Also, we know that B is not in $\{A_1, A_2, \ldots, A_n\}^+$, so B is one of the attributes on which t and s disagree. Thus, I does not satisfy $A_1A_2 \cdots A_n \rightarrow B$. We conclude that Algorithm 3.7 asserts neither too few nor too many FD's; it asserts exactly those FD's that do follow from S.

3.2.6 The Transitive Rule

The transitive rule lets us cascade two FD's, and generalizes the observation of Example 3.4.

- If $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ and $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$ hold in relation R, then $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$ also holds in R.

If some of the C's are among the A's, we may eliminate them from the right side by the trivial-dependencies rule.

To see why the transitive rule holds, apply the test of Section 3.2.4. To test whether $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$ holds, we need to compute the closure $\{A_1, A_2, \ldots, A_n\}^+$ with respect to the two given FD's.

The FD $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ tells us that all of $B_1, B_2, \ldots, B_m$ are in $\{A_1, A_2, \ldots, A_n\}^+$. Then, we can use the FD $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$ to add $C_1, C_2, \ldots, C_k$ to $\{A_1, A_2, \ldots, A_n\}^+$. Since all the C's are in

$$\{A_1, A_2, \ldots, A_n\}^+$$

we conclude that $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$ holds for any relation that satisfies both $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ and $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$.

Example 3.10: Here is another version of the `Movies` relation that includes both the studio of the movie and some information about that studio.

title	*year*	*length*	*genre*	*studioName*	*studioAddr*
`Star Wars`	`1977`	`124`	`sciFi`	`Fox`	`Hollywood`
`Eight Below`	`2005`	`120`	`drama`	`Disney`	`Buena Vista`
`Wayne's World`	`1992`	`95`	`comedy`	`Paramount`	`Hollywood`

Two of the FD's that we might reasonably claim to hold are:

```
title year → studioName
studioName → studioAddr
```

> **Closures and Keys**
>
> Notice that $\{A_1, A_2, \ldots, A_n\}^+$ is the set of all attributes of a relation if and only if $A_1, A_2, \ldots, A_n$ is a superkey for the relation. For only then does $A_1, A_2, \ldots, A_n$ functionally determine all the other attributes. We can test if $A_1, A_2, \ldots, A_n$ is a key for a relation by checking first that $\{A_1, A_2, \ldots, A_n\}^+$ is all attributes, and then checking that, for no set X formed by removing one attribute from $\{A_1, A_2, \ldots, A_n\}$, is X^+ the set of all attributes.

The first is justified because there can be only one movie with a given title and year, and there is only one studio that owns a given movie. The second is justified because studios have unique addresses.

The transitive rule allows us to combine the two FD's above to get a new FD:

$$\texttt{title year} \rightarrow \texttt{studioAddr}$$

This FD says that a title and year (i.e., a movie) determines an address — the address of the studio owning the movie. □

3.2.7 Closing Sets of Functional Dependencies

Sometimes we have a choice of which FD's we use to represent the full set of FD's for a relation. If we are given a set of FD's S (such as the FD's that hold in a given relation), then any set of FD's equivalent to S is said to be a ***basis*** for S. To avoid some of the explosion of possible bases, we shall limit ourselves to considering only bases whose FD's have singleton right sides. If we have any basis, we can apply the splitting rule to make the right sides be singletons. A ***minimal basis*** for a relation is a basis B that satisfies three conditions:

1. All the FD's in B have singleton right sides.

2. If any FD is removed from B, the result is no longer a basis.

3. If for any FD in B we remove one or more attributes from the left side of F, the result is no longer a basis.

Notice that no trivial FD can be in a minimal basis, because it could be removed by rule (2).

Example 3.11: Consider a relation $R(A, B, C)$ such that each attribute functionally determines the other two attributes. The full set of derived FD's thus includes six FD's with one attribute on the left and one on the right; $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$, $C \rightarrow A$, and $C \rightarrow B$. It also includes the three

A Complete Set of Inference Rules

If we want to know whether one FD follows from some given FD's, the closure computation of Section 3.2.4 will always serve. However, it is interesting to know that there is a set of rules, called *Armstrong's axioms*, from which it is possible to derive any FD that follows from a given set. These axioms are:

1. *Reflexivity*. If $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$, then $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$. These are what we have called trivial FD's.

2. *Augmentation*. If $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$, then

$$A_1A_2 \cdots A_nC_1C_2 \cdots C_k \rightarrow B_1B_2 \cdots B_mC_1C_2 \cdots C_k$$

for any set of attributes $C_1, C_2, \ldots, C_k$. Since some of the C's may also be A's or B's or both, we should eliminate from the left side duplicate attributes and do the same for the right side.

3. *Transitivity*. If

$$A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m \text{ and } B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$$

then $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$.

nontrivial FD's with two attributes on the left: $AB \rightarrow C$, $AC \rightarrow B$, and $BC \rightarrow A$. There are also FD's with more than one attribute on the right, such as $A \rightarrow BC$, and trivial FD's such as $A \rightarrow A$.

Relation R and its FD's have several minimal bases. One is

$$\{A \rightarrow B,\ B \rightarrow A,\ B \rightarrow C,\ C \rightarrow B\}$$

Another is $\{A \rightarrow B,\ B \rightarrow C,\ C \rightarrow A\}$. There are several other minimal bases for R, and we leave their discovery as an exercise. □

3.2.8 Projecting Functional Dependencies

When we study design of relation schemas, we shall also have need to answer the following question about FD's. Suppose we have a relation R with set of FD's S, and we project R by computing $R_1 = \pi_L(R)$, for some list of attributes R. What FD's hold in R_1?

The answer is obtained in principle by computing the *projection of functional dependencies S*, which is all FD's that:

a) Follow from S, and

b) Involve only attributes of R_1.

Since there may be a large number of such FD's, and many of them may be redundant (i.e., they follow from other such FD's), we are free to simplify that set of FD's if we wish. However, in general, the calculation of the FD's for R_1 is exponential in the number of attributes of R_1. The simple algorithm is summarized below.

Algorithm 3.12: Projecting a Set of Functional Dependencies.

INPUT: A relation R and a second relation R_1 computed by the projection $R_1 = \pi_L(R)$. Also, a set of FD's S that hold in R.

OUTPUT: The set of FD's that hold in R_1.

METHOD:

1. Let T be the eventual output set of FD's. Initially, T is empty.
2. For each set of attributes X that is a subset of the attributes of R_1, compute X^+. This computation is performed with respect to the set of FD's S, and may involve attributes that are in the schema of R but not R_1. Add to T all nontrivial FD's $X \rightarrow A$ such that A is both in X^+ and an attribute of R_1.
3. Now, T is a basis for the FD's that hold in R_1, but may not be a minimal basis. We may construct a minimal basis by modifying T as follows:
   (a) If there is an FD F in T that follows from the other FD's in T, remove F from T.
   (b) Let $Y \rightarrow B$ be an FD in T, with at least two attributes in Y, and let Z be Y with one of its attributes removed. If $Z \rightarrow B$ follows from the FD's in T (including $Y \rightarrow B$), then replace $Y \rightarrow B$ by $Z \rightarrow B$.
   (c) Repeat the above steps in all possible ways until no more changes to T can be made.

□

Example 3.13: Suppose $R(A, B, C, D)$ has FD's $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow D$. Suppose also that we wish to project out the attribute B, leaving a relation $R_1(A, C, D)$. In principle, to find the FD's for R_1, we need to take the closure of all eight subsets of $\{A, C, D\}$, using the full set of FD's, including those involving B. However, there are some obvious simplifications we can make.

- Closing the empty set and the set of all attributes cannot yield a nontrivial FD.

- If we already know that the closure of some set X is all attributes, then we cannot discover any new FD's by closing supersets of X.

Thus, we may start with the closures of the singleton sets, and then move on to the doubleton sets if necessary. For each closure of a set X, we add the FD $X \to E$ for each attribute E that is in X^+ and in the schema of R_1, but not in X.

First, $\{A\}^+ = \{A,B,C,D\}$. Thus, $A \to C$ and $A \to D$ hold in R_1. Note that $A \to B$ is true in R, but makes no sense in R_1 because B is not an attribute of R_1.

Next, we consider $\{C\}^+ = \{C,D\}$, from which we get the additional FD $C \to D$ for R_1. Since $\{D\}^+ = \{D\}$, we can add no more FD's, and are done with the singletons.

Since $\{A\}^+$ includes all attributes of R_1, there is no point in considering any superset of $\{A\}$. The reason is that whatever FD we could discover, for instance $AC \to D$, follows from an FD with only A on the left side: $A \to D$ in this case. Thus, the only doubleton whose closure we need to take is $\{C,D\}^+ = \{C,D\}$. This observation allows us to add nothing. We are done with the closures, and the FD's we have discovered are $A \to C$, $A \to D$, and $C \to D$.

If we wish, we can observe that $A \to D$ follows from the other two by transitivity. Therefore a simpler, equivalent set of FD's for R_1 is $A \to C$ and $C \to D$. This set is, in fact, a minimal basis for the FD's of R_1. □

3.2.9 Exercises for Section 3.2

Exercise 3.2.1: Consider a relation with schema $R(A,B,C,D)$ and FD's $AB \to C$, $C \to D$, and $D \to A$.

a) What are all the nontrivial FD's that follow from the given FD's? You should restrict yourself to FD's with single attributes on the right side.

b) What are all the keys of R?

c) What are all the superkeys for R that are not keys?

Exercise 3.2.2: Repeat Exercise 3.2.1 for the following schemas and sets of FD's:

i) $S(A,B,C,D)$ with FD's $A \to B$, $B \to C$, and $B \to D$.

ii) $T(A,B,C,D)$ with FD's $AB \to C$, $BC \to D$, $CD \to A$, and $AD \to B$.

iii) $U(A,B,C,D)$ with FD's $A \to B$, $B \to C$, $C \to D$, and $D \to A$.

Exercise 3.2.3: Show that the following rules hold, by using the closure test of Section 3.2.4.

a) *Augmenting left sides.* If $A_1A_2 \cdots A_n \to B$ is an FD, and C is another attribute, then $A_1A_2 \cdots A_nC \to B$ follows.

b) *Full augmentation.* If $A_1A_2\cdots A_n \to B$ is an FD, and C is another attribute, then $A_1A_2\cdots A_nC \to BC$ follows. Note: from this rule, the "augmentation" rule mentioned in the box of Section 3.2.7 on "A Complete Set of Inference Rules" can easily be proved.

c) *Pseudotransitivity.* Suppose FD's $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ and

$$C_1C_2\cdots C_k \to D$$

hold, and the B's are each among the C's. Then

$$A_1A_2\cdots A_nE_1E_2\cdots E_j \to D$$

holds, where the E's are all those of the C's that are not found among the B's.

d) *Addition.* If FD's $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ and

$$C_1C_2\cdots C_k \to D_1D_2\cdots D_j$$

hold, then FD $A_1A_2\cdots A_nC_1C_2\cdots C_k \to B_1B_2\cdots B_mD_1D_2\cdots D_j$ also holds. In the above, we should remove one copy of any attribute that appears among both the A's and C's or among both the B's and D's.

! **Exercise 3.2.4:** Show that each of the following are *not* valid rules about FD's by giving example relations that satisfy the given FD's (following the "if") but not the FD that allegedly follows (after the "then").

a) If $A \to B$ then $B \to A$.

b) If $AB \to C$ and $A \to C$, then $B \to C$.

c) If $AB \to C$, then $A \to C$ or $B \to C$.

! **Exercise 3.2.5:** Show that if a relation has no attribute that is functionally determined by all the other attributes, then the relation has no nontrivial FD's at all.

! **Exercise 3.2.6:** Let X and Y be sets of attributes. Show that if $X \subseteq Y$, then $X^+ \subseteq Y^+$, where the closures are taken with respect to the same set of FD's.

! **Exercise 3.2.7:** Prove that $(X^+)^+ = X^+$.

!! **Exercise 3.2.8:** We say a set of attributes X is *closed* (with respect to a given set of FD's) if $X^+ = X$. Consider a relation with schema $R(A, B, C, D)$ and an unknown set of FD's. If we are told which sets of attributes are closed, we can discover the FD's. What are the FD's if:

a) All sets of the four attributes are closed.

b) The only closed sets are $\emptyset$ and $\{A, B, C, D\}$.

c) The closed sets are $\emptyset$, {A,B}, and $\{A, B, C, D\}$.

! **Exercise 3.2.9:** Find all the minimal bases for the FD's and relation of Example 3.11.

! **Exercise 3.2.10:** Suppose we have relation $R(A, B, C, D, E)$, with some set of FD's, and we wish to project those FD's onto relation $S(A, B, C)$. Give the FD's that hold in S if the FD's for R are:

a) $AB \to DE$, $C \to E$, $D \to C$, and $E \to A$.

b) $A \to D$, $BD \to E$, $AC \to E$, and $DE \to B$.

c) $AB \to D$, $AC \to E$, $BC \to D$, $D \to A$, and $E \to B$.

d) $A \to B$, $B \to C$, $C \to D$, $D \to E$, and $E \to A$.

In each case, it is sufficient to give a minimal basis for the full set of FD's of S.

!! **Exercise 3.2.11:** Show that if an FD F follows from some given FD's, then we can prove F from the given FD's using Armstrong's axioms (defined in the box "A Complete Set of Inference Rules" in Section 3.2.7). *Hint*: Examine Algorithm 3.7 and show how each step of that algorithm can be mimicked by inferring some FD's by Armstrong's axioms.

3.3 Design of Relational Database Schemas

Careless selection of a relational database schema can lead to redundancy and related anomalies. For instance, consider the relation in Fig. 3.2, which we reproduce here as Fig. 3.6. Notice that the length and genre for *Star Wars* and *Wayne's World* are each repeated, once for each star of the movie. The repetition of this information is redundant. It also introduces the potential for several kinds of errors, as we shall see.

In this section, we shall tackle the problem of design of good relation schemas in the following stages:

1. We first explore in more detail the problems that arise when our schema is poorly designed.

2. Then, we introduce the idea of "decomposition," breaking a relation schema (set of attributes) into two smaller schemas.

3. Next, we introduce "Boyce-Codd normal form," or "BCNF," a condition on a relation schema that eliminates these problems.

4. These points are tied together when we explain how to assure the BCNF condition by decomposing relation schemas.

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 3.6: The relation `Movies1` exhibiting anomalies

3.3.1 Anomalies

Problems such as redundancy that occur when we try to cram too much into a single relation are called *anomalies.* The principal kinds of anomalies that we encounter are:

1. *Redundancy.* Information may be repeated unnecessarily in several tuples. Examples are the length and genre for movies in Fig. 3.6.

2. *Update Anomalies*. We may change information in one tuple but leave the same information unchanged in another. For example, if we found that *Star Wars* is really 125 minutes long, we might carelessly change the length in the first tuple of Fig. 3.6 but not in the second or third tuples. You might argue that one should never be so careless, but it is possible to redesign relation `Movies1` so that the risk of such mistakes does not exist.

3. *Deletion Anomalies*. If a set of values becomes empty, we may lose other information as a side effect. For example, should we delete Vivien Leigh from the set of stars of *Gone With the Wind*, then we have no more stars for that movie in the database. The last tuple for *Gone With the Wind* in the relation `Movies1` would disappear, and with it information that it is 231 minutes long and a drama.

3.3.2 Decomposing Relations

The accepted way to eliminate these anomalies is to *decompose* relations. Decomposition of R involves splitting the attributes of R to make the schemas of two new relations. After describing the decomposition process, we shall show how to pick a decomposition that eliminates anomalies.

Given a relation $R(A_1, A_2, \ldots, A_n)$, we may *decompose* R into two relations $S(B_1, B_2, \ldots, B_m)$ and $T(C_1, C_2, \ldots, C_k)$ such that:

1. $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$.

2. $S = \pi_{B_1, B_2, \ldots, B_m}(R)$.

3. $T = \pi_{C_1,C_2,\ldots,C_k}(R)$.

Example 3.14: Let us decompose the Movies1 relation of Fig. 3.6. Our choice, whose merit will be seen in Section 3.3.3, is to use:

1. A relation called Movies2, whose schema is all the attributes except for starName.

2. A relation called Movies3, whose schema consists of the attributes title, year, and starName.

The projection of Movies1 onto these two new schemas is shown in Fig, 3.7. □

title	*year*	*length*	*genre*	*studioName*
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	231	drama	MGM
Wayne's World	1992	95	comedy	Paramount

(b) The relation Movies2.

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

(b) The relation Movies3.

Figure 3.7: Projections of relation Movies1

Notice how this decomposition eliminates the anomalies we mentioned in Section 3.3.1. The redundancy has been eliminated; for example, the length of each film appears only once, in relation Movies2. The risk of an update anomaly is gone. For instance, since we only have to change the length of *Star Wars* in one tuple of Movies2, we cannot wind up with two different lengths for that movie.

Finally, the risk of a deletion anomaly is gone. If we delete all the stars for *Gone With the Wind*, say, that deletion makes the movie disappear from Movies3. But all the other information about the movie can still be found in Movies2.

It might appear that `Movies3` still has redundancy, since the title and year of a movie can appear several times. However, these two attributes form a key for movies, and there is no more succinct way to represent a movie. Moreover, `Movies3` does not offer an opportunity for an update anomaly. For instance, one might suppose that if we changed to 2008 the year in the Carrie Fisher tuple, but not the other two tuples for *Star Wars*, then there would be an update anomaly. However, there is nothing in our assumed FD's that prevents there being a different movie named *Star Wars* in 2008, and Carrie Fisher may star in that one as well. Thus, we do not want to prevent changing the year in one *Star Wars* tuple, nor is such a change necessarily incorrect.

3.3.3 Boyce-Codd Normal Form

The goal of decomposition is to replace a relation by several that do not exhibit anomalies. There is, it turns out, a simple condition under which the anomalies discussed above can be guaranteed not to exist. This condition is called *Boyce-Codd normal form*, or *BCNF*.

- A relation R is in BCNF if and only if: whenever there is a nontrivial FD $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ for R, it is the case that $\{A_1, A_2, \ldots, A_n\}$ is a superkey for R.

That is, the left side of every nontrivial FD must be a superkey. Recall that a superkey need not be minimal. Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.

Example 3.15: Relation `Movies1`, as in Fig. 3.6, is not in BCNF. To see why, we first need to determine what sets of attributes are keys. We argued in Example 3.2 why {`title`, `year`, `starName`} is a key. Thus, any set of attributes containing these three is a superkey. The same arguments we followed in Example 3.2 can be used to explain why no set of attributes that does not include all three of `title`, `year`, and `starName` could be a superkey. Thus, we assert that {`title`, `year`, `starName`} is the only key for `Movies1`.

However, consider the FD

$$\texttt{title year} \rightarrow \texttt{length genre studioName}$$

which holds in `Movies1` according to our discussion in Example 3.2.

Unfortunately, the left side of the above FD is not a superkey. In particular, we know that `title` and `year` do not functionally determine the sixth attribute, `starName`. Thus, the existence of this FD violates the BCNF condition and tells us `Movies1` is not in BCNF. □

Example 3.16: On the other hand, `Movies2` of Fig. 3.7 is in BCNF. Since

$$\texttt{title year} \rightarrow \texttt{length genre studioName}$$

holds in this relation, and we have argued that neither `title` nor `year` by itself functionally determines any of the other attributes, the only key for `Movies2` is {`title`, `year`}. Moreover, the only nontrivial FD's must have at least `title` and `year` on the left side, and therefore their left sides must be superkeys. Thus, `Movies2` is in BCNF. □

Example 3.17: We claim that any two-attribute relation is in BCNF. We need to examine the possible nontrivial FD's with a single attribute on the right. There are not too many cases to consider, so let us consider them in turn. In what follows, suppose that the attributes are A and B.

1. There are no nontrivial FD's. Then surely the BCNF condition must hold, because only a nontrivial FD can violate this condition. Incidentally, note that $\{A, B\}$ is the only key in this case.

2. $A \to B$ holds, but $B \to A$ does not hold. In this case, A is the only key, and each nontrivial FD contains A on the left (in fact the left can only be A). Thus there is no violation of the BCNF condition.

3. $B \to A$ holds, but $A \to B$ does not hold. This case is symmetric to case (2).

4. Both $A \to B$ and $B \to A$ hold. Then both A and B are keys. Surely any FD has at least one of these on the left, so there can be no BCNF violation.

It is worth noticing from case (4) above that there may be more than one key for a relation. Further, the BCNF condition only requires that *some* key be contained in the left side of any nontrivial FD, not that all keys are contained in the left side. Also observe that a relation with two attributes, each functionally determining the other, is not completely implausible. For example, a company may assign its employees unique employee ID's and also record their Social Security numbers. A relation with attributes `empID` and `ssNo` would have each attribute functionally determining the other. Put another way, each attribute is a key, since we don't expect to find two tuples that agree on either attribute. □

3.3.4 Decomposition into BCNF

By repeatedly choosing suitable decompositions, we can break any relation schema into a collection of subsets of its attributes with the following important properties:

1. These subsets are the schemas of relations in BCNF.

2. The data in the original relation is represented faithfully by the data in the relations that are the result of the decomposition, in a sense to be made precise in Section 3.4.1. Roughly, we need to be able to reconstruct the original relation instance exactly from the decomposed relation instances.

Example 3.17 suggests that perhaps all we have to do is break a relation schema into two-attribute subsets, and the result is surely in BCNF. However, such an arbitrary decomposition will not satisfy condition (2), as we shall see in Section 3.4.1. In fact, we must be more careful and use the violating FD's to guide our decomposition.

The decomposition strategy we shall follow is to look for a nontrivial FD $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ that violates BCNF; i.e., $\{A_1, A_2, \ldots, A_n\}$ is not a superkey. We shall add to the right side as many attributes as are functionally determined by $\{A_1, A_2, \ldots, A_n\}$. This step is not mandatory, but it often reduces the total amount of work done, and we shall include it in our algorithm. Figure 3.8 illustrates how the attributes are broken into two overlapping relation schemas. One is all the attributes involved in the violating FD, and the other is the left side of the FD plus all the attributes *not* involved in the FD, i.e., all the attributes except those B's that are not A's.

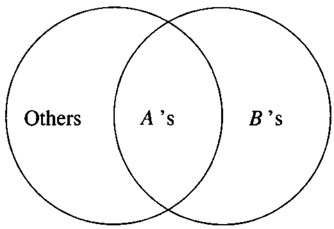

Figure 3.8: Relation schema decomposition based on a BCNF violation

Example 3.18: Consider our running example, the `Movies1` relation of Fig. 3.6. We saw in Example 3.15 that

$$\texttt{title year} \rightarrow \texttt{length genre studioName}$$

is a BCNF violation. In this case, the right side already includes all the attributes functionally determined by `title` and `year`, so we shall use this BCNF violation to decompose `Movies1` into:

1. The schema {`title`, `year`, `length`, `genre`, `studioName`} consisting of all the attributes on either side of the FD.

2. The schema {`title`, `year`, `starName`} consisting of the left side of the FD plus all attributes of `Movies1` that do not appear in either side of the FD (only `starName`, in this case).

Notice that these schemas are the ones selected for relations `Movies2` and `Movies3` in Example 3.14. We observed in Example 3.16 that `Movies2` is in BCNF. `Movies3` is also in BCNF; it has no nontrivial FD's. □

In Example 3.18, one judicious application of the decomposition rule is enough to produce a collection of relations that are in BCNF. In general, that is not the case, as the next example shows.

Example 3.19: Consider a relation with schema

{`title`, `year`, `studioName`, `president`, `presAddr`}

That is, each tuple of this relation tells about a movie, its studio, the president of the studio, and the address of the president of the studio. Three FD's that we would assume in this relation are

`title year` $\rightarrow$ `studioName`
`studioName` $\rightarrow$ `president`
`president` $\rightarrow$ `presAddr`

By closing sets of these five attributes, we discover that {`title`, `year`} is the only key for this relation. Thus the last two FD's above violate BCNF. Suppose we choose to decompose starting with

`studioName` $\rightarrow$ `president`

First, we add to the right side of this functional dependency any other attributes in the closure of `studioName`. That closure includes `presAddr`, so our final choice of FD for the decomposition is:

`studioName` $\rightarrow$ `president presAddr`

The decomposition based on this FD yields the following two relation schemas.

{`title`, `year`, `studioName`}
{`studioName`, `president`, `presAddr`}

If we use Algorithm 3.12 to project FD's, we determine that the FD's for the first relation has a basis:

`title year` $\rightarrow$ `studioName`

while the second has:

`studioName` $\rightarrow$ `president`
`president` $\rightarrow$ `presAddr`

The sole key for the first relation is {`title`, `year`}, and it is therefore in BCNF. However, the second has {`studioName`} for its only key but also has the FD:

`president` $\rightarrow$ `presAddr`

which is a BCNF violation. Thus, we must decompose again, this time using the above FD. The resulting three relation schemas, all in BCNF, are:

{title, year, studioName}
{studioName, president}
{president, presAddr}

□

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in BCNF. We can be sure of ultimate success, because every time we apply the decomposition rule to a relation R, the two resulting schemas each have fewer attributes than that of R. As we saw in Example 3.17, when we get down to two attributes, the relation is sure to be in BCNF; often relations with larger sets of attributes are also in BCNF. The strategy is summarized below.

Algorithm 3.20: BCNF Decomposition Algorithm.

INPUT: A relation R_0 with a set of functional dependencies S_0.

OUTPUT: A decomposition of R_0 into a collection of relations, all of which are in BCNF.

METHOD: The following steps can be applied recursively to any relation R and set of FD's S. Initially, apply them with $R = R_0$ and $S = S_0$.

1. Check whether R is in BCNF. If so, nothing more needs to be done. Return $\{R\}$ as the answer.

2. If there are BCNF violations, let one be $X \rightarrow Y$. Use Algorithm 3.7 to compute X^+. Choose $R_1 = X^+$ as one relation schema and let R_2 have attributes X and those attributes of R that are not in X^+.

3. Use Algorithm 3.12 to compute the sets of FD's for R_1 and R_2; let these be S_1 and S_2, respectively.

4. Recursively decompose R_1 and R_2 using this algorithm. Return the union of the results of these decompositions.

□

3.3.5 Exercises for Section 3.3

Exercise 3.3.1: For each of the following relation schemas and sets of FD's:

a) $R(A,B,C,D)$ with FD's $AB \rightarrow C$, $C \rightarrow D$, and $D \rightarrow A$.

b) $R(A,B,C,D)$ with FD's $B \rightarrow C$ and $B \rightarrow D$.

c) $R(A,B,C,D)$ with FD's $AB \rightarrow C$, $BC \rightarrow D$, $CD \rightarrow A$, and $AD \rightarrow B$.

d) $R(A,B,C,D)$ with FD's $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, and $D \rightarrow A$.

e) $R(A, B, C, D, E)$ with FD's $AB \rightarrow C$, $DE \rightarrow C$, and $B \rightarrow D$.

f) $R(A, B, C, D, E)$ with FD's $AB \rightarrow C$, $C \rightarrow D$, $D \rightarrow B$, and $D \rightarrow E$.

do the following:

i) Indicate all the BCNF violations. Do not forget to consider FD's that are not in the given set, but follow from them. However, it is not necessary to give violations that have more than one attribute on the right side.

ii) Decompose the relations, as necessary, into collections of relations that are in BCNF.

Exercise 3.3.2: We mentioned in Section 3.3.4 that we would exercise our option to expand the right side of an FD that is a BCNF violation if possible. Consider a relation R whose schema is the set of attributes $\{A, B, C, D\}$ with FD's $A \rightarrow B$ and $A \rightarrow C$. Either is a BCNF violation, because the only key for R is $\{A, D\}$. Suppose we begin by decomposing R according to $A \rightarrow B$. Do we ultimately get the same result as if we first expand the BCNF violation to $A \rightarrow BC$? Why or why not?

! **Exercise 3.3.3:** Let R be as in Exercise 3.3.2, but let the FD's be $A \rightarrow B$ and $B \rightarrow C$. Again compare decomposing using $A \rightarrow B$ first against decomposing by $A \rightarrow BC$ first.

! **Exercise 3.3.4:** Suppose we have a relation schema $R(A, B, C)$ with FD $A \rightarrow B$. Suppose also that we decide to decompose this schema into $S(A, B)$ and $T(B, C)$. Give an example of an instance of relation R whose projection onto S and T and subsequent rejoining as in Section 3.4.1 does not yield the same relation instance. That is, $\pi_{A,B}(R) \bowtie \pi_{B,C}(R) \neq R$.

3.4 Decomposition: The Good, Bad, and Ugly

So far, we observed that before we decompose a relation schema into BCNF, it can exhibit anomalies; after we decompose, the resulting relations do not exhibit anomalies. That's the "good." But decomposition can also have some bad, if not downright ugly, consequences. In this section, we shall consider three distinct properties we would like a decomposition to have.

1. *Elimination of Anomalies* by decomposition as in Section 3.3.

2. *Recoverability of Information.* Can we recover the original relation from the tuples in its decomposition?

3. *Preservation of Dependencies.* If we check the projected FD's in the relations of the decomposition, can we can be sure that when we reconstruct the original relation from the decomposition by joining, the result will satisfy the original FD's?

It turns out that the BCNF decomposition of Algorithm 3.20 gives us (1) and (2), but does not necessarily give us all three. In Section 3.5 we shall see another way to pick a decomposition that gives us (2) and (3) but does not necessarily give us (1). In fact, there is no way to get all three at once.

3.4.1 Recovering Information from a Decomposition

Since we learned that every two-attribute relation is in BCNF, why did we have to go through the trouble of Algorithm 3.20? Why not just take any relation R and decompose it into relations, each of whose schemas is a pair of R's attributes? The answer is that the data in the decomposed relations, even if their tuples were each the projection of a relation instance of R, might not allow us to join the relations of the decomposition and get the instance of R back. If we do get R back, then we say the decomposition has a *lossless join*.

However, if we decompose using Algorithm 3.20, where all decompositions are motivated by a BCNF-violating FD, then the projections of the original tuples can be joined again to produce all and only the original tuples. We shall consider why here. Then, in Section 3.4.2 we shall give an algorithm called the "chase," for testing whether the projection of a relation onto any decomposition allows us to recover the relation by rejoining.

To simplify the situation, consider a relation $R(A, B, C)$ and an FD $B \to C$ that is a BCNF violation. The decomposition based on the FD $B \to C$ separates the attributes into relations $R_1(A, B)$ and $R_2(B, C)$.

Let t be a tuple of R. We may write $t = (a, b, c)$, where a, b, and c are the components of t for attributes A, B, and C, respectively. Tuple t projects as (a, b) in $R_1(A, B) = \pi_{A,B}(R)$ and as (b, c) in $R_2(B, C) = \pi_{B,C}(R)$. When we compute the natural join $R_1 \bowtie R_2$, these two projected tuples join, because they agree on the common B component (they both have b there). They give us $t = (a, b, c)$, the tuple we started with, in the join. That is, regardless of what tuple t we started with, we can always join its projections to get t back.

However, getting back those tuples we started with is not enough to assure that the original relation R is truly represented by the decomposition. Consider what happens if there are two tuples of R, say $t = (a, b, c)$ and $v = (d, b, e)$. When we project t onto $R_1(A, B)$ we get $u = (a, b)$, and when we project v onto $R_2(B, C)$ we get $w = (b, e)$. These tuples also match in the natural join, and the resulting tuple is $x = (a, b, e)$. Is it possible that x is a bogus tuple? That is, could (a, b, e) not be a tuple of R?

Since we assume the FD $B \to C$ for relation R, the answer is "no." Recall that this FD says any two tuples of R that agree in their B components must also agree in their C components. Since t and v agree in their B components, they also agree on their C components. That means $c = e$; i.e., the two values we supposed were different are really the same. Thus, tuple (a, b, e) of R is really (a, b, c); that is, $x = t$.

Since t is in R, it must be that x is in R. Put another way, as long as FD $B \to C$ holds, the joining of two projected tuples cannot produce a bogus tuple.

Rather, every tuple produced by the natural join is guaranteed to be a tuple of R.

This argument works in general. We assumed A, B, and C were each single attributes, but the same argument would apply if they were any sets of attributes X, Y and Z. That is, if $Y \rightarrow Z$ holds in R, whose attributes are $X \cup Y \cup Z$, then $R = \pi_{X \cup Y}(R) \bowtie \pi_{Y \cup Z}(R)$.

We may conclude:

- If we decompose a relation according to Algorithm 3.20, then the original relation can be recovered exactly by the natural join.

To see why, we argued above that at any one step of the recursive decomposition, a relation is equal to the join of its projections onto the two components. If those components are decomposed further, they can also be recovered by the natural join from their decomposed relations. Thus, an easy induction on the number of binary decomposition steps says that the original relation is always the natural join of whatever relations it is decomposed into. We can also prove that the natural join is associative and commutative, so the order in which we perform the natural join of the decomposition components does not matter.

The FD $Y \rightarrow Z$, or its symmetric FD $Y \rightarrow X$, is essential. Without one of these FD's, we might not be able to recover the original relation. Here is an example.

Example 3.21: Suppose we have the relation $R(A, B, C)$ as above, but neither of the FD's $B \rightarrow A$ nor $B \rightarrow C$ holds. Then R might consist of the two tuples

A	B	C
1	2	3
4	2	5

The projections of R onto the relations with schemas $\{A, B\}$ and $\{B, C\}$ are $R_1 = \pi_{AB}(R) =$

A	B
1	2
4	2

and $R_2 = \pi_{BC}(R) =$

B	C
2	3
2	5

respectively. Since all four tuples share the same B-value, 2, each tuple of one relation joins with both tuples of the other relation. When we try to reconstruct R by the natural join of the projected relations, we get $R_3 = R_1 \bowtie R_2 =$

Is Join the Only Way to Recover?

We have assumed that the only possible way we could reconstruct a relation from its projections is to use the natural join. However, might there be some other algorithm to reconstruct the original relation that would work even in cases where the natural join fails? There is in fact no such other way. In Example 3.21, the relations R and R_3 are different instances, yet have exactly the same projections onto $\{A, B\}$ and $\{B, C\}$, namely the instances we called R_1 and R_2, respectively. Thus, given R_1 and R_2, no algorithm whatsoever can tell whether the original instance was R or R_3.

Moreover, this example is not unusual. Given any decomposition of a relation with attributes $X \cup Y \cup Z$ into relations with schemas $X \cup Y$ and $Y \cup Z$, where neither $Y \to X$ nor $Y \to Z$ holds, we can construct an example similar to Example 3.21 where the original instance cannot be determined from its projections.

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5

That is, we get "too much"; we get two bogus tuples, $(1, 2, 5)$ and $(4, 2, 3)$, that were not in the original relation R. $\square$

3.4.2 The Chase Test for Lossless Join

In Section 3.4.1 we argued why a particular decomposition, that of $R(A, B, C)$ into $\{A, B\}$ and $\{B, C\}$, with a particular FD, $B \to C$, had a lossless join. Now, consider a more general situation. We have decomposed relation R into relations with sets of attributes $S_1, S_2, \ldots, S_k$. We have a given set of FD's F that hold in R. Is it true that if we project R onto the relations of the decomposition, then we can recover R by taking the natural join of all these relations? That is, is it true that $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples t such that for all $i = 1, 2, \ldots, k$, t projected onto the set of attributes S_i is a tuple in $\pi_{S_i}(R)$.

- Any tuple t in R is surely in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$. The reason is that the projection of t onto S_i is surely in $\pi_{S_i}(R)$ for each i, and therefore by our first point above, t is in the result of the join.

- As a consequence, $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$ when the FD's in F hold for R if and only if every tuple in the join is also in R. That is, the membership test is all we need to verify that the decomposition has a lossless join.

The *chase* test for a lossless join is just an organized way to see whether a tuple t in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ can be proved, using the FD's in F, also to be a tuple in R. If t is in the join, then there must be tuples in R, say $t_1, t_2, \ldots, t_k$, such that t is the join of the projections of each t_i onto the set of attributes S_i, for $i = 1, 2, \ldots, k$. We therefore know that t_i agrees with t on the attributes of S_i, but t_i has unknown values in its components not in S_i.

We draw a picture of what we know, called a *tableau*. Assuming R has attributes $A, B, \ldots$ we use $a, b, \ldots$ for the components of t. For t_i, we use the same letter as t in the components that are in S_i, but we subscript the letter with i if the component is not in i. In that way, t_i will agree with t for the attributes of S_i, but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

Example 3.22: Suppose we have relation $R(A, B, C, D)$, which we have decomposed into relations with sets of attributes $S_1 = \{A, D\}$, $S_2 = \{A, C\}$, and $S_3 = \{B, C, D\}$. Then the tableau for this decomposition is shown in Fig. 3.9.

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

Figure 3.9: Tableau for the decomposition of R into $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$

The first row corresponds to set of attributes A and D. Notice that the components for attributes A and D are the unsubscripted letters a and d. However, for the other attributes, b and c, we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple (a, b_1, c_1, d) represents a tuple of R that contributes to $t = (a, b, c, d)$ by being projected onto $\{A, D\}$ and then joined with other tuples. Since the B- and C-components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.

Similarly, the second row has the unsubscripted letters in attributes A and C, while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for $\{B, C, D\}$ and subscript 3 on a. Since each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters. □

Remember that our goal is to use the given set of FD's F to prove that t is really in R. In order to do so, we "chase" the tableau by applying the FD's in F to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as t (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple t in the join of the projections was actually a tuple of R.

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurences.

Example 3.23: Let us continue with the decomposition of Example 3.22, and suppose the given FD's are $A \to B$, $B \to C$, and $CD \to A$. Start with the tableau of Fig. 3.9. Since the first two rows agree in their A-components, the FD $A \to B$ tells us they must also agree in their B-components. That is, $b_1 = b_2$. We can replace either one with the other, since they are both subscripted. Let us replace b_2 by b_1. Then the resulting tableau is:

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Now, we see that the first two rows have equal B-values, and so we may use the FD $B \to C$ to deduce that their C-components, c_1 and c, are the same. Since c is unsubscripted, we replace c_1 by c, leaving:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

Next, we observe that the first and third rows agree in both columns C and D. Thus, we may apply the FD $CD \to A$ to deduce that these rows also have the same A-value; that is, $a = a_3$. We replace a_3 by a, giving us:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

At this point, we see that the last row has become equal to t, that is, (a, b, c, d). We have proved that if R satisfies the FD's $A \to B$, $B \to C$, and $CD \to A$, then whenever we project onto $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$ and rejoin, what we get must have been in R. In particular, what we get is the same as the tuple of R that we projected onto $\{B, C, D\}$. $\square$

3.4.3 Why the Chase Works

There are two issues to address:

1. When the chase results in a row that matches the tuple t (i.e., the tableau is shown to have a row with all unsubscripted variables), why must the join be lossless?

2. When, after applying FD's whenever we can, we still find no row of all unsubscripted variables, why must the join not be lossless?

Question (1) is easy to answer. The chase process itself is a proof that one of the projected tuples from R must in fact be the tuple t that is produced by the join. We also know that every tuple in R is sure to come back if we project and join. Thus, the chase has proved that the result of projection and join is exactly R.

For the second question, suppose that we eventually derive a tableau without an unsubscripted row, and that this tableau does not allow us to apply any of the FD's to equate any symbols. Then think of the tableau as an instance of the relation R. It obviously satisfies the given FD's, because none can be applied to equate symbols. We know that the ith row has unsubscripted symbols in the attributes of S_i, the ith relation of the decomposition. Thus, when we project this relation onto the S_i's and take the natural join, we get the tuple with all unsubscripted variables. This tuple is not in R, so we conclude that the join is not lossless.

Example 3.24: Consider the relation $R(A, B, C, D)$ with the FD $B \to AD$ and the proposed decomposition $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$. Here is the initial tableau:

A	B	C	D
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

When we apply the lone FD, we deduce that $a = a_2$ and $d_1 = d_2$. Thus, the final tableau is:

A	B	C	D
a	b	c_1	d_1
a	b	c	d_1
a_3	b_3	c	d

No more changes can be made because of the given FD's, and there is no row that is fully unsubscripted. Thus, this decomposition does not have a lossless join. We can verify that fact by treating the above tableau as a relation with three tuples. When we project onto $\{A, B\}$, we get $\{(a, b)\}$, $(a_3, b_3)\}$. The projection onto $\{B, C\}$ is $\{(b, c_1)$, (b, c), $(b_3, c)\}$, and the projection onto $\{C, D\}$ is (c_1, d_1), (c, d_1), $(c, d)\}$. If we join the first two projections, we get $\{(a, b, c_1)$, (a, b, c), $(a_3, b_3, c)\}$. Joining this relation with the third projection gives $\{(a, b, c_1, d_1)$, (a, b, c, d_1), (a, b, c, d), (a_3, b_3, c, d_1), $(a_3, b_3, c, d)\}$. Notice that this join has two more tuples than R, and in particular it has the tuple (a, b, c, d), as it must. □

3.4.4 Dependency Preservation

We mentioned that it is not possible, in some cases, to decompose a relation into BCNF relations that have both the lossless-join and dependency-preservation properties. Below is an example where we need to make a tradeoff between preserving dependencies and BNCF.

Example 3.25: Suppose we have a relation `Bookings` with attributes:

1. `title`, the name of a movie.
2. `theater`, the name of a theater where the movie is being shown.
3. `city`, the city where the theater is located.

The intent behind a tuple (m, t, c) is that the movie with title m is currently being shown at theater t in city c.

We might reasonably assert the following FD's:

$$\texttt{theater} \rightarrow \texttt{city}$$
$$\texttt{title city} \rightarrow \texttt{theater}$$

The first says that a theater is located in one city. The second is not obvious but is based on the common practice of not booking a movie into two theaters in the same city. We shall assert this FD if only for the sake of the example.

Let us first find the keys. No single attribute is a key. For example, `title` is not a key because a movie can play in several theaters at once and in several cities at once.[2] Also, `theater` is not a key, because although `theater` functionally determines `city`, there are multiscreen theaters that show many movies at once. Thus, `theater` does not determine `title`. Finally, `city` is not a key because cities usually have more than one theater and more than one movie playing.

[2] In this example we assume that there are not two "current" movies with the same title, even though we have previously recognized that there could be two movies with the same title made in different years.

On the other hand, two of the three sets of two attributes are keys. Clearly {title, city} is a key because of the given FD that says these attributes functionally determine theater.

It is also true that {theater, title} is a key, because its closure includes city due to the given FD theater → city. The remaining pair of attributes, city and theater, do not functionally determine title, because of multiscreen theaters, and are therefore not a key. We conclude that the only two keys are

{title, city}
{theater, title}

Now we immediately see a BCNF violation. We were given functional dependency theater → city, but its left side, theater, is not a superkey. We are therefore tempted to decompose, using this BCNF-violating FD, into the two relation schemas:

{theater, city}
{theater, title}

There is a problem with this decomposition, concerning the FD

title city→theater

There could be current relations for the decomposed schemas that satisfy the FD theater → city (which can be checked in the relation {theater, city}) but that, when joined, yield a relation not satisfying title city→theater. For instance, the two relations

theater	*city*
Guild	Menlo Park
Park	Menlo Park

and

theater	*title*
Guild	Antz
Park	Antz

are permissible according to the FD's that apply to each of the above relations, but when we join them we get two tuples

theater	*city*	*title*
Guild	Menlo Park	Antz
Park	Menlo Park	Antz

that violate the FD title city → theater. □

3.4.5 Exercises for Section 3.4

Exercise 3.4.1: Let $R(A, B, C, D, E)$ be decomposed into relations with the following three sets of attributes: $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, C, E\}$. For each of the following sets of FD's, use the chase test to tell whether the decomposition of R is lossless. For those that are not lossless, give an example of an instance of R that returns more than R when projected onto the decomposed relations and rejoined.

a) $B \to E$ and $CE \to A$.

b) $AC \to E$ and $BC \to D$.

c) $A \to D$, $D \to E$, and $B \to D$.

d) $A \to D$, $CD \to E$, and $E \to D$.

! **Exercise 3.4.2:** For each of the sets of FD's in Exercise 3.4.1, are dependencies preserved by the decomposition?

3.5 Third Normal Form

The solution to the problem illustrated by Example 3.25 is to relax our BCNF requirement slightly, in order to allow the occasional relation schema that cannot be decomposed into BCNF relations without our losing the ability to check the FD's. This relaxed condition is called "third normal form." In this section we shall give the requirements for third normal form, and then show how to do a decomposition in a manner quite different from Algorithm 3.20, in order to obtain relations in third normal form that have both the lossless-join and dependency-preservation properties.

3.5.1 Definition of Third Normal Form

A relation R is in *third normal form* (3NF) if:

- Whenever $A_1A_2\cdots A_n \to B_1B_2\cdots B_m$ is a nontrivial FD, either

$$\{A_1, A_2, \ldots, A_n\}$$

  is a superkey, or those of $B_1, B_2, \ldots, B_m$ that are not among the A's, are each a member of some key (not necessarily the same key).

An attribute that is a member of some key is often said to be *prime*. Thus, the 3NF condition can be stated as "for each nontrivial FD, either the left side is a superkey, or the right side consists of prime attributes only."

Note that the difference between this 3NF condition and the BCNF condition is the clause "is a member of some key (i.e., prime)." This clause "excuses" an FD like `theater` $\to$ `city` in Example 3.25, because the right side, `city`, is prime.

Other Normal Forms

If there is a "third normal form," what happened to the first two "normal forms"? They indeed were defined, but today there is little use for them. *First normal form* is simply the condition that every component of every tuple is an atomic value. *Second normal form* is a less restrictive verison of 3NF. There is also a "fourth normal form" that we shall meet in Section 3.6.

3.5.2 The Synthesis Algorithm for 3NF Schemas

We can now explain and justify how we decompose a relation R into a set of relations such that:

a) The relations of the decomposition are all in 3NF.

b) The decomposition has a lossless join.

c) The decomposition has the dependency-preservation property.

Algorithm 3.26: Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.

INPUT: A relation R and a set F of functional dependencies that hold for R.

OUTPUT: A decomposition of R into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.

METHOD: Perform the following steps:

1. Find a minimal basis for F, say G.

2. For each functional dependency $X \to A$ in G, use XA as the schema of one of the relations in the decomposition.

3. If none of the relation schemas from Step 2 is a superkey for R, add another relation whose schema is a key for R.

□

Example 3.27: Consider the relation $R(A, B, C, D, E)$ with FD's $AB \to C$, $C \to B$, and $A \to D$. To start, notice that the given FD's are their own minimal basis. To check, we need to do a bit of work. First, we need to verify that we cannot eliminate any of the given dependencies. That is, we show, using Algorithm 3.7, that no two of the FD's imply the third. For example, we must take the closure of $\{A, B\}$, the left side of the first FD, using only the second and third FD's, $C \to B$ and $A \to D$. This closure includes D but not C, so we conclude that the first FD $AB \to C$ is not implied by the second and third FD's. We get a similar conclusion if we try to drop the second or third FD.

We must also verify that we cannot eliminate any attributes from a left side. In this simple case, the only possibility is that we could eliminate A or B from the first FD. For example, if we eliminate A, we would be left with $B \to C$. We must show that $B \to C$ is not implied by the three original FD's, $AB \to C$, $C \to B$, and $A \to D$. With these FD's, the closure of $\{B\}$ is just B, so $B \to C$ does not follow. A similar conclusion is drawn if we try to drop B from $AB \to C$. Thus, we have our minimal basis.

We start the 3NF synthesis by taking the attributes of each FD as a relation schema. That is, we get relations $S_1(A, B, C)$, $S_2(B, C)$, and $S_3(A, D)$. It is never necessary to use a relation whose schema is a proper subset of another relation's schema, so we can drop S_2.

We must also consider whether we need to add a relation whose schema is a key. In this example, R has two keys: $\{A, B, E\}$ and $\{A, C, E\}$, as you can verify. Neither of these keys is a subset of the schemas chosen so far. Thus, we must add one of them, say $S_4(A, B, E)$. The final decomposition of R is thus $S_1(A, B, C)$, $S_3(A, D)$, and $S_4(A, B, E)$. □

3.5.3 Why the 3NF Synthesis Algorithm Works

We need to show three things: that the lossless-join and dependency-preservation properties hold, and that all the relations of the decomposition are in 3NF.

1. *Lossless Join*. Start with a relation of the decomposition whose set of attributes K is a superkey. Consider the sequence of FD's that are used in Algorithm 3.7 to expand K to become K^+. Since K is a superkey, we know K^+ is all the attributes. The same sequence of FD applications on the tableau cause the subscripted symbols in the row corresponding to K to be equated to unsubscripted symbols in the same order as the attributes were added to the closure. Thus, the chase test concludes that the decomposition is lossless.

2. *Dependency Preservation*. Each FD of the minimal basis has all its attributes in some relation of the decomposition. Thus, each dependency can be checked in the decomposed relations.

3. *Third Normal Form*. If we have to add a relation whose schema is a key, then this relation is surely in 3NF. The reason is that all attributes of this relation are prime, and thus no violation of 3NF could be present in this relation. For the relations whose schemas are derived from the FD's of a minimal basis, the proof that they are in 3NF is beyond the scope of this book. The argument involves showing that a 3NF violation implies that the basis is not minimal.

3.5.4 Exercises for Section 3.5

Exercise 3.5.1: For each of the relation schemas and sets of FD's of Exercise 3.3.1:

i) Indicate all the 3NF violations.

ii) Decompose the relations, as necessary, into collections of relations that are in 3NF.

Exercise 3.5.2: Consider the relation `Courses`(C, T, H, R, S, G), whose attributes may be thought of informally as course, teacher, hour, room, student, and grade. Let the set of FD's for `Courses` be $C \rightarrow T$, $HR \rightarrow C$, $HT \rightarrow R$, $HS \rightarrow R$, and $CS \rightarrow G$. Intuitively, the first says that a course has a unique teacher, and the second says that only one course can meet in a given room at a given hour. The third says that a teacher can be in only one room at a given hour, and the fourth says the same about students. The last says that students get only one grade in a course.

a) What are all the keys for `Courses`?

b) Verify that the given FD's are their own minimal basis.

c) Use the 3NF synthesis algorithm to find a lossless-join, dependency-preserving decomposition of R into 3NF relations. Are any of the relations not in BCNF?

Exercise 3.5.3: Consider a relation `Stocks`(B, O, I, S, Q, D), whose attributes may be thought of informally as broker, office (of the broker), investor, stock, quantity (of the stock owned by the investor), and dividend (of the stock). Let the set of FD's for `Stocks` be $S \rightarrow D$, $I \rightarrow B$, $IS \rightarrow Q$, and $B \rightarrow O$. Repeat Exercise 3.5.2 for the relation `Stocks`.

Exercise 3.5.4: Verify, using the chase, that the decomposition of Example 3.27 has a lossless join.

!! **Exercise 3.5.5:** Suppose we modified Algorithm 3.20 (BNCF decomposition) so that instead of decomposing a relation R whenever R was not in BCNF, we only decomposed R if it was not in 3NF. Provide a counterexample to show that this modified algorithm would not necessarily produce a 3NF decomposition with dependency preservation.

3.6 Multivalued Dependencies

A "multivalued dependency" is an assertion that two attributes or sets of attributes are independent of one another. This condition is, as we shall see, a generalization of the notion of a functional dependency, in the sense that

3.8 Summary of Chapter 3

- ✦ *Functional Dependencies*: A functional dependency is a statement that two tuples of a relation that agree on some particular set of attributes must also agree on some other particular set of attributes.

- ✦ *Keys of a Relation*: A superkey for a relation is a set of attributes that functionally determines all the attributes of the relation. A key is a superkey, no proper subset of which is also a superkey.

- ✦ *Reasoning About Functional Dependencies*: There are many rules that let us infer that one FD $X \rightarrow A$ holds in any relation instance that satisfies some other given set of FD's. To verify that $X \rightarrow A$ holds, compute the closure of X, using the given FD's to expand X until it includes A.

- ✦ *Minimal Basis for a set of FD's*: For any set of FD's, there is at least one minimal basis, which is a set of FD's equivalent to the original (each set implies the other set), with singleton right sides, no FD that can be eliminated while preserving equivalence, and no attribute in a left side that can be eliminated while preserving equivalence.

- ✦ *Boyce-Codd Normal Form*: A relation is in BCNF if the only nontrivial FD's say that some superkey functionally determines one or more of the other attributes. A major benefit of BCNF is that it eliminates redundancy caused by the existence of FD's.

- ✦ *Lossless-Join Decomposition*: A useful property of a decomposition is that the original relation can be recovered exactly by taking the natural join of the relations in the decomposition. Any decomposition gives us back at least the tuples with which we start, but a carelessly chosen decomposition can give tuples in the join that were not in the original relation.

- ✦ *Dependency-Preserving Decomposition*: Another desirable property of a decomposition is that we can check all the functional dependencies that hold in the original relation by checking FD's in the decomposed relations.

- ✦ *Third Normal Form*: Sometimes decomposition into BCNF can lose the dependency-preservation property. A relaxed form of BCNF, called 3NF, allows an FD $X \rightarrow A$ even if X is not a superkey, provided A is a member of some key. 3NF does not guarantee to eliminate all redundancy due to FD's, but often does so.

- ✦ *The Chase*: We can test whether a decomposition has the lossless-join property by setting up a tableau — a set of rows that represent tuples of the original relation. We chase a tableau by applying the given functional dependencies to infer that certain pairs of symbols must be the same. The decomposition is lossless with respect to a given set of FD's if and only if the chase leads to a row identical to the tuple whose membership in the join of the projected relations we assumed.

- *Synthesis Algorithm for 3NF*: If we take a minimal basis for a given set of FD's, turn each of these FD's into a relation, and add a key for the relation, if necessary, the result is a decomposition into 3NF that has the lossless-join and dependency-preservation properties.

- *Multivalued Dependencies*: A multivalued dependency is a statement that two sets of attributes in a relation have sets of values that appear in all possible combinations.

- *Fourth Normal Form*: MVD's can also cause redundancy in a relation. 4NF is like BCNF, but also forbids nontrivial MVD's whose left side is not a superkey. It is possible to decompose a relation into 4NF without losing information.

- *Reasoning About MVD's*: We can infer MVD's and FD's from a given set of MVD's and FD's by a chase process. We start with a two-row tableau that represent the dependency we are trying to prove. FD's are applied by equating symbols, and MVD's are applied by adding rows to the tableau that have the appropriate components interchanged.

3.9 References for Chapter 3

Third normal form was described in [6]. This paper introduces the idea of functional dependencies, as well as the basic relational concept. Boyce-Codd normal form is in a later paper [7].

Multivalued dependencies and fourth normal form were defined by Fagin in [9]. However, the idea of multivalued dependencies also appears independently in [8] and [11].

Armstrong was the first to study rules for inferring FD's [2]. The rules for FD's that we have covered here (including what we call "Armstrong's axioms") and rules for inferring MVD's as well, come from [3].

The technique for testing an FD by computing the closure for a set of attributes is from [4], as is the fact that a minimal basis provides a 3NF decomposition. The fact that this decomposition provides the lossless-join and dependency-preservation propoerties is from [5].

The tableau test for the lossless-join property and the chase are from [1]. More information and the history of the idea is found in [10].

1. A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Transactions on Database Systems* 4:3, pp. 297-314, 1979.

2. W. W. Armstrong, "Dependency structures of database relationships," *Proceedings of the 1974 IFIP Congress*, pp. 580–583.