CHAPTER 11

Functional Dependencies

11.1 INTRODUCTION

In this chapter, we examine a concept that has been characterized (by Hugh Darwen, in a private communication) as "not quite fundamental, but very nearly so"—*viz.*, the concept of **functional dependence.** This concept turns out to be crucially important to a number of issues to be discussed in later chapters, including in particular the database design theory described in Chapter 12. Please note immediately, however, that its usefulness is not limited to that purpose alone; indeed, this chapter could well have been included in Part II of this book instead of Part III.

A functional dependency (FD for short) is basically *a many-to-one relationship* from one set of attributes to another within a given relvar. In the case of the shipments relvar SP, for example, there is a functional dependency from the set of attributes {S#,P#} to the set of attributes {QTY}. What this means is that within any relation that is a legal value for that relvar:

1. For any given value for the pair of attributes S# and P#, there is just one corresponding value of attribute QTY.[1]
2. However, any number of distinct values of the pair of attributes S# and P# can have the same corresponding value for attribute QTY (in general).

Observe that our usual sample SP value (see Fig. 3.8 on the inside back cover) does satisfy both of these properties; observe too that once again we have a concept whose definition relies on the concept of tuple equality.

In Section 11.2, we define the notion of functional dependence more precisely, distinguishing carefully between those FDs that happen to be satisfied by a given relvar at some particular time and those that are satisfied by that relvar at *all* times. As already mentioned, it turns out that FDs provide a basis for a scientific attack on a number of practical problems. And the reason they do so is because they possess a rich set of interesting formal properties, which make it possible to treat the problems in question in a formal and rigorous manner. Sections 11.3–11.6 explore some of those formal properties in detail and explain some of their practical consequences. Finally, Section 11.7 presents a brief summary.

Note: This is the most formal chapter in the book, and you might like to skip portions of it on a first reading. Indeed, most of what you need in order to understand the material of the next three chapters is covered in Sections 11.2 and 11.3; you might therefore prefer to give the remaining sections a "once over lightly" reading for now, and come back to them later when you have assimilated the material of the next three chapters.

A small point regarding terminology: The terms *functional dependence* and *functional dependency* are used interchangeably in the literature. Normal English usage would suggest that the term *dependence* be used for the FD concept *per se* and would reserve the term *dependency* for "the thing that depends." But we very frequently need to refer to FDs in the plural, and "dependencies" seems to trip off the tongue more readily than "dependences"; hence our use of both terms.

11.2 BASIC DEFINITIONS

In order to illustrate the ideas of the present section, we make use of a slightly revised version of the shipments relvar, one that includes, in addition to the usual attributes S#, P#, and QTY, an attribute CITY, representing the city for the relevant supplier. We will refer to this revised relvar as SCP to avoid confusion. A sample value is shown in Fig. 11.1.

Now, it is very important in this area—as in so many others!—to distinguish clearly between (a) the value of a given relvar at a given point in time and (b) the set of all possible values that the given relvar might assume at different times. In what follows, we first define

[1] Note that this rather formal statement is true precisely because a certain "business rule" is in effect (see Chapter 9)—namely, that there is at most one shipment at any given time for a given supplier and a given part. Loosely speaking, in other words, FDs are a matter of *semantics* (what the data means), not a fluke arising from the particular values that happen to appear in the database at some particular point in time.

SCP

S#	CITY	P#	QTY
S1	London	P1	100
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

Fig. 11.1 Sample value for relvar SCP

the concept of functional dependence as it applies to Case *a*, and then extend it to apply to Case *b*. Here then is the definition for Case *a:*

- **Functional dependence, Case** *a:* Let *r* be a relation, and let *X* and *Y* be arbitrary subsets of the set of attributes of *r*. Then we say that *Y* is **functionally dependent** on *X*—in symbols,

  ```
  X → Y
  ```

  (read "*X* **functionally determines** *Y*," or simply "*X* arrow *Y*")—if and only if each *X* value in *r* has associated with it precisely one *Y* value in *r*. In other words, whenever two tuples of *r* agree on their *X* value, they also agree on their *Y* value.

  For example, the relation shown in Fig. 11.1 satisfies the FD

  ```
  { S# } → { CITY }
  ```

because all tuples of that relation with a given S# value also have the same CITY value. Indeed, it also satisfies several more FDs, the following among them:

```
{ S#, P# } → { QTY }
{ S#, P# } → { CITY }
{ S#, P# } → { CITY, QTY }
{ S#, P# } → { S# }
{ S#, P# } → { S#, P#, CITY, QTY }
{ S# }     → { QTY }
{ QTY }    → { S# }
```

(*Exercise:* Check these.)

The left and right sides of an FD are called the **determinant** and the **dependent**, respectively. As the definition indicates, the determinant and dependent are both *sets* of attributes. When such a set contains just one attribute, however—that is, when it is a *singleton set*—we often drop the set braces and write just, for example:

```
S# → CITY
```

As already explained, the foregoing definitions apply to Case *a*—that is, to individual relation values. However, when we consider relation variables (i.e., relvars)—in particular, when we consider *base* relvars—we are usually interested not so much in the FDs that happen to hold for the particular value that the relvar happens to have at some particular time, but rather in those FDs that hold for *all possible values* of that relvar. In the case of SCP, for example, the FD

```
S# → CITY
```

holds for all possible values of SCP, because at any given time a given supplier has precisely one corresponding city, and so any two tuples appearing in SCP at the same time with the same supplier number must necessarily have the same city as well. Indeed, the fact that this FD holds "for all time" (i.e., for all possible values of SCP) is an *integrity constraint* on relvar SCP—it places limits on the values that relvar SCP can legitimately have. Here is a formulation of that constraint using the calculus-based Tutorial D syntax from Chapter 9:

```
CONSTRAINT S#_CITY_FD
    FORALL SCPX FORALL SCPY
       ( IF SCPX.S# = SCPY.S#
         THEN SCPX.CITY = SCPY.CITY END IF ) ;
```

(SCPX and SCPY are range variables ranging over SCP.) The syntax S# → CITY can be regarded as shorthand for this longer formulation. *Exercise:* Give an algebraic version of this constraint.

Here then is the Case *b* definition of functional dependence (the extensions over the Case *a* definition are shown in boldface):

- Functional dependence, Case *b:* Let *R* be a **relation variable**, and let *X* and *Y* be arbitrary subsets of the set of attributes of *R*. Then we say that *Y* is functionally dependent on—in symbols,

  ```
  X → Y
  ```

  (read "*X* functionally determines *Y*" or simply "*X* arrow *Y*")—if and only if, **in every possible legal value of R**, each *X* value has associated with it precisely one *Y* value. In other words, **in every possible legal value of R**, whenever two tuples agree on their *X* value, they also agree on their *Y* value.

Henceforth, we will use the term *functional dependence* in this latter, more demanding, and *time-independent* sense, barring explicit statements to the contrary.

Here then are some (time-independent) FDs that apply to relvar SCP:

```
{ S#, P# } → QTY
{ S#, P# } → CITY
{ S#, P# } → { CITY, QTY }
{ S#, P# } → S#
{ S#, P# } → { S#, P#, CITY, QTY }
{ S# }     → CITY
```

Note in particular that the following FDs, which do happen to hold for the relation value shown in Fig. 11.1, do not hold "for all time" for relvar SCP:

```
S#  → QTY
QTY → S#
```

In other words, the statement that (e.g.) "every shipment for a given supplier involves the same quantity" does happen to be true for the particular SCP relation value shown in Fig. 11.1, but it is not true for all possible legal values of the SCP relvar.

We now observe that if X is a candidate key for relvar R, then all attributes Y of relvar R must be functionally dependent on X. (We mentioned this important fact in passing in Chapter 9, Section 9.10. It follows from the definition of candidate key.) For the parts relvar P, for example, we necessarily have:

```
P# → { P#, PNAME, COLOR, WEIGHT, CITY }
```

In fact, if relvar R satisfies the FD $A \rightarrow B$ and A is *not* a candidate key,[2] then R will necessarily involve some redundancy. In the case of relvar SCP, for example, the FD S# → CITY implies that the fact that a given supplier is located in a given city will appear many times in that relvar, in general (see Fig. 11.1 for an illustration of this point). We will take up this question of redundancy and discuss it in detail in the next chapter.

Now, even if we restrict our attention to FDs that hold for all time, the complete set of FDs for a given relvar can still be very large, as the SCP example suggests. (*Exercise:* State the complete set of FDs satisfied by relvar SCP.) What we would like is to find some way of reducing that set to a manageable size—and, indeed, most of the remainder of this chapter is concerned with exactly this issue.

Why is this objective desirable? One reason is that (as already stated) FDs represent certain integrity constraints, and we would thus like the DBMS to enforce them. Given a particular set S of FDs, therefore, it is desirable to find some other set T that is (ideally) much smaller than S and has the property that every FD in S is implied by the FDs in T. If such a set T can be found, it is sufficient that the DBMS enforce just the FDs in T, and the FDs in S will then be enforced automatically. The problem of finding such a set T is thus of considerable practical interest.

11.3 TRIVIAL AND NONTRIVIAL DEPENDENCIES

Note: In the remainder of this chapter, we will occasionally abbreviate "functional dependency" to just "dependency." Similarly for "functionally dependent on," "functionally determines," and so on.

One obvious way to reduce the size of the set of FDs we need to deal with is to eliminate the *trivial* dependencies. A dependency is trivial if it cannot possibly fail to be satisfied. Just one of the FDs shown for relvar SCP in the previous section was trivial in this sense—*viz.*, the FD:

```
{ S#, P# } → S#
```

In fact, an FD is trivial if and only if the right side is a subset (not necessarily a proper subset) of the left side.

[2] And if the FD is not trivial (see Section 11.3) and A is not a superkey (see Section 11.5) and R contains at least two tuples (!).

As the name implies, trivial dependencies are not very interesting in practice; we are usually more interested in practice in **nontrivial** dependencies (which are, of course, precisely the ones that are not trivial), because they are the ones that correspond to "genuine" integrity constraints. When we are dealing with the formal theory, however, we have to account for *all* dependencies, trivial ones as well as nontrivial.

.4 CLOSURE OF A SET OF DEPENDENCIES

We have already suggested that some FDs might imply others. As a simple example, the FD

```
{ S#, P# } → { CITY, QTY }
```

implies both of the following:

```
{ S#, P# } → CITY
{ S#, P# } → QTY
```

As a more complex example, suppose we have a relvar R with attributes A, B, and C, such that the FDs $A \to B$ and $B \to C$ both hold for R. Then it is easy to see that the FD $A \to C$ also holds for R. The FD $A \to C$ here is an example of a **transitive** FD—C is said to depend on A *transitively*, via B.

The set of all FDs that are implied by a given set S of FDs is called the **closure** of S, written S^+ (nothing to do with closure in the relational algebra sense, please note). Clearly we need an algorithm that will allow us to compute S^+ from S. The first attack on this problem appeared in a paper by Armstrong [11.2], which gave a set of **inference rules** (more usually called **Armstrong's axioms**) by which new FDs can be inferred from given ones. Those rules can be stated in a variety of equivalent ways, of which one of the simplest is as follows. Let A, B, and C be arbitrary subsets of the set of attributes of the given relvar R, and let us agree to write (e.g.) AB to mean the union of A and B. Then we have:

1. **Reflexivity:** If B is a subset of A, then $A \to B$.
2. **Augmentation:** If $A \to B$, then $AC \to BC$.
3. **Transitivity:** If $A \to B$ and $B \to C$, then $A \to C$.

Each of these three rules can be directly proved from the definition of functional dependency (the first is just the definition of a trivial dependency, of course). Moreover, the rules are **complete**, in the sense that, given a set S of FDs, all FDs implied by S can be derived from S using the rules. They are also **sound**, in the sense that no additional FDs (i.e., FDs not implied by S) can be so derived. In other words, the rules can be used to derive precisely the closure S^+.

Several further rules can be derived from the three already given, the following among them. These additional rules can be used to simplify the practical task of computing S^+ from S. (In what follows, D is another arbitrary subset of the set of attributes of R.)

4. **Self-determination:** $A \to A$.

5. **Decomposition:** If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$.
6. **Union:** If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.
7. **Composition:** If $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$.

And in reference [11.7], Darwen proves the following rule, which he calls the *General Unification Theorem:*

8. If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup (C - B) \rightarrow BD$

(where "$\cup$" is union and "$-$" is set difference). The name "General Unification Theorem" refers to the fact that several of the earlier rules can be seen as special cases [11.7].

Example: Suppose we are given a relvar R with attributes A, B, C, D, E, F, and the FDs:

```
A  → BC
B  → E
CD → EF
```

Observe that we are extending our notation slightly, though not incompatibly, by writing, for example, BC for the set consisting of attributes B and C (previously BC would have meant the *union* of B and C, where B and C were *sets* of attributes). *Note:* If you would prefer a more concrete example, take A as employee number, B as department number, C as manager's employee number, D as project number for a project directed by that manager (unique within manager), E as department name, and F as percentage of time spent by the specified manager on the specified project.

We now show that the FD $AD \rightarrow F$ holds for R and is thus a member of the closure of the given set:

1.	A → BC	(given)
2.	A → C	(1, decomposition)
3.	AD → CD	(2, augmentation)
4.	CD → EF	(given)
5.	AD → EF	(3 and 4, transitivity)
6.	AD → F	(5, decomposition)

11.5 CLOSURE OF A SET OF ATTRIBUTES

In principle, we can compute the closure S^+ of a given set S of FDs by means of an algorithm that says "Repeatedly apply the rules from the previous section until they stop producing new FDs." In practice, there is little need to compute the closure *per se* (which is just as well, because the algorithm just mentioned is hardly very efficient). In this section, however, we will show how to compute a certain subset of the closure: namely, that subset consisting of all FDs with a certain (specified) set Z of attributes as the left side. More precisely, we will show how, given a relvar R, a set Z of attributes of R, and a set S of FDs that hold for R, we can determine the set of all attributes of R that are functionally dependent

```
CLOSURE[Z,S] := Z ;
do "forever" ;
   for each FD X → Y in S
      do ;
         if X ⊆ CLOSURE[Z,S]
         then CLOSURE[Z,S] := CLOSURE[Z,S] ∪ Y ;
      end
   if CLOSURE[Z,S] did not change on this iteration
   then leave the loop ;            /* computation complete */
end ;
```

Fig. 11.2 Computing the closure Z^+ of Z under *S*

on *Z*—the closure Z^+ of *Z* under *S*.[3] A simple algorithm for computing that closure is given in pseudocode in Fig. 11.2. *Exercise:* Prove that algorithm is correct.

Example: Suppose we are given a relvar *R* with attributes *A, B, C, D, E, F*, and FDs:

```
A  → BC
E  → CF
B  → E
CD → EF
```

We now compute the closure $\{A,B\}^+$ of the set of attributes $\{A,B\}$ under this set of FDs.

1. We initialize the result CLOSURE[Z,S] to $\{A,B\}$.
2. We now go round the inner loop four times, once for each of the given FDs. On the first iteration (for the FD $A \rightarrow BC$), we find that the left side is indeed a subset of CLOSURE[Z,S] as computed so far, so we add attributes (*B* and) *C* to the result. CLOSURE[Z,S] is now the set $\{A,B,C\}$.
3. On the second iteration (for the FD $E \rightarrow CF$), we find that the left side is *not* a subset of the result as computed so far, which thus remains unchanged.
4. On the third iteration (for the FD $B \rightarrow E$), we add *E* to CLOSURE[Z,S], which now has the value $\{A,B,C,E\}$.
5. On the fourth iteration (for the FD $CD \rightarrow EF$), CLOSURE[Z,S] remains unchanged.
6. Now we go round the inner loop four times again. On the first iteration, the result does not change; on the second, it expands to $\{A,B,C,E,F\}$; on the third and fourth, it does not change.
7. Now we go round the inner loop four times again. CLOSURE[Z,S] does not change, and so the whole process terminates, with $\{A,B\}^+ = \{A,B,C,E,F\}$.

Note that if (as stated) Z is a set of attributes of relvar *R* and *S* is a set of FDs that hold for *R*, then the set of FDs that hold for *R* with *Z* as the left side is the set consisting of all FDs of the form $Z \rightarrow Z'$, where Z' is some subset of the closure Z^+ of *Z* under *S*. The

[3] Note that we now have two kinds of closure: closure of a set of FDs, and closure of a set of attributes under a set of FDs. Note too that we use the same "superscript plus" notation for both of them. We trust that this dual usage on our part will not prove confusing.

closure S^+ of the original set S of FDs is then the union of all such sets of FDs, taken over all possible attribute sets Z.

An important corollary of all of the foregoing is as follows: Given a set S of FDs, we can easily tell whether a specific FD $X \rightarrow Y$ follows from S, because that FD will follow if and only if Y is a subset of the closure X^+ of X under S. In other words, we now have a simple way of determining whether a given FD $X \rightarrow Y$ is in the closure S^+ of S, without actually having to compute that closure S^+.

Another important corollary is the following. Recall from Chapter 9 that a superkey for a relvar R is a set of attributes of R that includes some candidate key of R as a subset (not necessarily a proper subset). Now, it follows immediately from the definition that the superkeys for a given relvar R are precisely those subsets K of the attributes of R such that the FD

$$K \rightarrow A$$

holds true for every attribute A of R. In other words, K is a superkey if and only if the closure K^+ of K—under the given set of FDs—is precisely the set of all attributes of R (and K is a *candidate* key if and only if it is an irreducible superkey).

11.6 IRREDUCIBLE SETS OF DEPENDENCIES

Let $S1$ and $S2$ be two sets of FDs. If every FD implied by $S1$ is implied by $S2$—that is, if $S1^+$ is a subset of $S2^+$—we say that $S2$ is a cover for $S1$.[4] What this means is that if the DBMS enforces the FDs in $S2$, then it will automatically be enforcing the FDs in $S1$.

Next, if $S2$ is a cover for $S1$ and $S1$ is a cover for $S2$—that is, if $S1^+ = S2^+$—we say that $S1$ and $S2$ are equivalent. Clearly, if $S1$ and $S2$ are equivalent, then if the DBMS enforces the FDs in $S2$ it will automatically be enforcing the FDs in $S1$, and *vice versa.*

Now we define a set S of FDs to be irreducible[5] if and only if it satisfies the following three properties:

1. The right side (the dependent) of every FD in S involves just one attribute (i.e., it is a singleton set).
2. The left side (the determinant) of every FD in S is irreducible in turn—meaning that no attribute can be discarded from the determinant without changing the closure S^+ (i.e., without converting S into some set not equivalent to S). We call such an FD left-irreducible.
3. No FD in S can be discarded from S without changing the closure S^+ (i.e., without converting S into some set not equivalent to S).

With regard to points 2 and 3 here, by the way, note carefully that it is not necessary to know exactly what the closure S^+ is in order to tell whether it will be changed *if* something is discarded. For example, consider the familiar parts relvar P. The following FDs (among others) hold for that relvar:

[4] Some writers use the term *cover* to mean what we will be calling (in just a moment) an *equivalent* set.

[5] Usually called *minimal* in the literature.

```
P# → PNAME
P# → COLOR
P# → WEIGHT
P# → CITY
```

This set of FDs is easily seen to be irreducible: The right side is a single attribute in each case, the left side is obviously irreducible in turn, and none of the FDs can be discarded without changing the closure (i.e., without *losing some information*). By contrast, the following sets of FDs are not irreducible:

1.	P# → { PNAME, COLOR } P# → WEIGHT P# → CITY	*The right side of the first FD here is not a singleton set*
2.	{ P#, PNAME } → COLOR P# → PNAME P# → WEIGHT P# → CITY	*The first FD here can be simplified by dropping PNAME from the left side without changing the closure (i.e., it is not left-irreducible)*
3.	P# → P# P# → PNAME P# → COLOR P# → WEIGHT P# → CITY	*The first FD here can be discarded without changing the closure*

We now claim that for every set of FDs there exists at least one equivalent set that is irreducible. In fact, this is easy to see. Let the original set of FDs be S. Thanks to the decomposition rule, we can assume without loss of generality that every FD in S has a singleton right side. Next, for each FD f in S we examine each attribute A in the left side of f; if deleting A from the left side of f has no effect on the closure S^+, we delete A from the left side of f. Then, for each FD f remaining in S, if deleting f from S has no effect on the closure S^+, we delete f from S. The final set S is irreducible and is equivalent to the original set S.

Example: Suppose we are given a relvar R with attributes A, B, C, D, and FDs:

$$\begin{aligned}
A &\rightarrow BC\\
B &\rightarrow C\\
A &\rightarrow B\\
AB &\rightarrow C\\
AC &\rightarrow D
\end{aligned}$$

We now compute an irreducible set of FDs that is equivalent to this given set:

1. The first step is to rewrite the FDs such that each has a singleton right side:

$$\begin{aligned}
A &\rightarrow B\\
A &\rightarrow C\\
B &\rightarrow C\\
A &\rightarrow B\\
AB &\rightarrow C\\
AC &\rightarrow D
\end{aligned}$$

We observe immediately that the FD $A \rightarrow B$ occurs twice, so one occurrence can be eliminated.

2. Next, attribute C can be eliminated from the left side of the FD $AC \rightarrow D$, because we have $A \rightarrow C$, so $A \rightarrow AC$ by augmentation, and we are given $AC \rightarrow D$, so $A \rightarrow D$ by transitivity; thus the C on the left side of $AC \rightarrow D$ is redundant.
3. Next, we observe that the FD $AB \rightarrow C$ can be eliminated, because again we have $A \rightarrow C$, so $AB \rightarrow CB$ by augmentation, so $AB \rightarrow C$ by decomposition.
4. Finally, the FD $A \rightarrow C$ is implied by the FDs $A \rightarrow B$ and $B \rightarrow C$, so it can also be eliminated. We are left with:

   ```
   A → B
   B → C
   A → D
   ```

   This set is irreducible.

A set I of FDs that is irreducible and equivalent to some other set S of FDs is said to be an **irreducible equivalent** of S. Thus, given some particular set S of FDs that need to be enforced, it is sufficient for the system to find and enforce the FDs in an irreducible equivalent I instead (and, to repeat, there is no need to compute the closure S^+ in order to compute an irreducible equivalent I). We should make it clear, however, that a given set of FDs does not necessarily have a *unique* irreducible equivalent (see Exercise 11.12).

11.7 SUMMARY

A **functional dependency** (FD) is a many-to-one relationship between two sets of attributes of a given relvar (it is a particularly common and important kind of integrity constraint). More precisely, given a relvar R, the FD $A \rightarrow B$ (where A and B are subsets of the set of attributes of R) is said to hold for R if and only if, whenever two tuples of R have the same value for A, they also have the same value for B. Every relvar necessarily satisfies certain **trivial** FDs; an FD is trivial if and only if the right side (the **dependent**) is a subset of the left side (the **determinant**).

Certain FDs imply others. Given a set S of FDs, the **closure** S^+ of that set is the set of all FDs implied by the FDs in S. S^+ is necessarily a superset of S. **Armstrong's inference rules** provide a sound and complete basis for computing S^+ from S (usually, however, we do not actually need to perform that computation). Several other useful rules can easily be derived from Armstrong's rules.

Given a subset Z of the set of attributes of relvar R and a set S of FDs that hold for R, the **closure** Z^+ of Z under S is the set of all attributes A of R such that the FD $Z \rightarrow A$ is a member of S^+. If Z^+ consists of all attributes of R, Z is said to be a **superkey** for R (and a **candidate key** is an irreducible superkey). We gave a simple algorithm for computing Z^+ from Z and S, and hence a simple way of determining whether a given FD $X \rightarrow Y$ is a member of S^+ ($X \rightarrow Y$ is a member of S^+ if and only if Y is a subset of X^+).

Two sets of FDs $S1$ and $S2$ are **equivalent** if and only if they are **covers** for each other, implying that $S1^+ = S2^+$. Every set of FDs is equivalent to at least one **irreducible** set. A set of FDs is irreducible if (a) every FD in the set has a singleton right side, (b) no FD in the set can be discarded without changing the closure of the set, and (c) no attribute can be discarded from the left side of any FD in the set without changing the closure of the set. If *I* is an irreducible set equivalent to *S*, enforcing the FDs in *I* will automatically enforce the FDs in *S*.

In conclusion, we note that many of the foregoing ideas can be extended to apply to integrity constraints in general, not just to FDs. For example, it is true in general that:

1. Certain constraints are trivial.
2. Certain constraints imply others.
3. The set of all constraints implied by a given set can be regarded as the closure of the given set.
4. The question of whether a specific constraint is in a certain closure—that is, whether the specific constraint is implied by certain given constraints—is an interesting practical problem.
5. The question of finding an irreducible equivalent for a given set of constraints is an interesting practical problem.

What makes FDs in particular much more tractable than integrity constraints in general is the existence of a sound and complete set of inference rules for FDs. The "References and Bibliography" sections in this chapter and Chapter 13 give references to publications describing several other specific kinds of constraints—*MVDs*, *JDs*, and *INDs*—for which such sets of inference rules also exist. In this book, however, we choose not to give those other kinds of constraints so extensive and so formal a treatment as the one we have just given FDs.

XERCISES

11.1 (a) Let *R* be a relvar of degree *n*. What is the maximum number of functional dependencies *R* can possibly satisfy (trivial as well as nontrivial)? (b) Given that *A* and *B* in the FD $A \rightarrow B$ are both *sets* of attributes, what happens if either is the *empty* set?

11.2 What does it mean to say that Armstrong's inference rules are sound? Complete?

11.3 Prove the ***reflexivity, augmentation,*** and ***transitivity*** rules, assuming only the basic definition of functional dependence.

11.4 Prove that the three rules of the previous exercise imply the ***self-determination, decomposition, union,*** and ***composition*** rules.

11.5 Prove Darwen's "General Unification Theorem." Which of the rules of the previous two exercises did you use? Which rules can be derived as special cases of the theorem?

11.6 Define (a) the closure of a set of FDs; (b) the closure of a set of attributes under a set of FDs.

11.7 List the FDs satisfied by the shipments relvar SP.

11.8 Relvar *R*{*A,B,C,D,E,F,G*} satisfies the following FDs:

```
A   → B
BC  → DE
AEF → G
```

Compute the closure $\{A,C\}^+$ under this set of FDs. Is the FD $ACF \rightarrow DG$ implied by this set?

11.9 What does it mean to say that two sets *S1* and *S2* of FDs are equivalent?

11.10 What does it mean to say that a set of FDs is irreducible?

11.11 Here are two sets of FDs for a relvar $R\{A,B,C,D,E\}$. Are they equivalent?

1. $A \rightarrow B \quad AB \rightarrow C \quad D \rightarrow AC \quad D \rightarrow E$
2. $A \rightarrow BC \quad D \rightarrow AE$

11.12 Relvar $R\{A,B,C,D,E,F\}$ satisfies the following FDs:

```
AB  → C
C   → A
BC  → D
ACD → B
BE  → C
CE  → FA
CF  → BD
D   → EF
```

Find an irreducible equivalent for this set of FDs.

11.13 A relvar TIMETABLE is defined with the following attributes:

D	Day of the week (1 to 5)
P	Period within day (1 to 6)
C	Classroom number
T	Teacher name
L	Lesson name

Tuple (*d,p,c,t,l*) appears in this relvar if and only if at time (*d,p*) lesson *l* is taught by teacher *t* in classroom *c* (using the simplified notation for tuples introduced in Section 10.4). You can assume that lessons are one period in duration and that every lesson has a name that is unique with respect to all lessons taught in the week. What functional dependencies hold in this relvar? What are the candidate keys?

11.14 A relvar NADDR is defined with attributes NAME (unique), STREET, CITY, STATE, and ZIP. Assume that (a) for any given zip code, there is just one city and state; (b) for any given street, city, and state, there is just one zip code. Give an irreducible set of FDs for this relvar. What are the candidate keys?

11.15 Do the assumptions of the previous exercise hold in practice?

11.16 Let relvar *R* have attributes *A, B, C, D, E, F, G, H, I,* and *J*, and suppose it satisfies the following FDs:

```
ABD → E
AB  → G
B   → F
C   → J
CJ  → I
G   → H
```

Is this an irreducible set? What are the candidate keys?

REFERENCES AND BIBLIOGRAPHY

As noted in Section 11.1, this is the most formal chapter in the book; it thus seemed appropriate to include references [11.1], [11.3], and [11.10] here, since each is a book that offers a formal treatment of a variety of aspects of database theory (not just functional dependencies as such).

11.1 Serge Abiteboul, Richard Hull, and Victor Vianu: *Foundations of Databases*. Reading, Mass.: Addison-Wesley (1995).

11.2 W. W. Armstrong: "Dependency Structures of Data Base Relationships," Proc. IFIP Congress, Stockholm, Sweden (1974).

The paper that first formalized the theory of FDs (it is the source of Armstrong's axioms). The paper also gives a precise characterization of candidate keys.

11.3 Paolo Atzeni and Valeria De Antonellis: *Relational Database Theory.* Redwood City, Calif.: Benjamin/Cummings (1993).

11.4 Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou: "Inclusion Dependencies and Their Interaction with Functional Dependencies," Proc. 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, Calif. (March 1982).

Inclusion dependencies (INDs) can be regarded as a generalization of referential constraints. For example, the IND

```
SP.S# —— S.S#
```

(not the notation used in the paper) states that the set of values appearing in attribute S# of relvar SP must be a subset of the set of values appearing in attribute S# of relvar S. This particular example in fact *is* a referential constraint; in general, however, there is no requirement for an IND that the left side be a foreign key or the right side a candidate key. *Note:* INDs do have some points in common with FDs, since both represent many-to-one relationships; however, INDs usually span relvars, while FDs do not.

The paper provides a sound and complete set of inference rules for INDs, which we may state (a little loosely) as follows:

1. $A \longrightarrow A$.
2. If $AB \longrightarrow CD$, then $A \longrightarrow C$ and $B \longrightarrow D$.
3. If $A \longrightarrow B$ and $B \longrightarrow C$, then $A \longrightarrow C$.

11.5 R. G. Casey and C. Delobel: "Decomposition of a Data Base and the Theory of Boolean Switching Funcrions," *IBM J. R&D 17*, No. 5 (September 1973).

This paper shows that for any given relvar, the set of FDs (called *functional relations* in this paper) satisfied by that relvar can be represented by a "boolean switching function." Moreover, that function is unique in the following sense: The original FDs can be specified in many superficially different (but actually equivalent) ways, each giving rise to a superficially different boolean function—but all such functions can be reduced by the laws of boolean algebra to the same "canonical form" (see Chapter 18). The problem of *nonloss-decomposing* the original relvar (see Chapter 12) is then shown to be logically equivalent to the well-understood boolean algebra problem of finding "a covering set of prime implicants" for the boolean function corresponding to that relvar together with its FDs. Hence the original problem can be transformed into an equivalent problem in boolean algebra, and well-known techniques can be brought to bear on it.

This paper was the first of several to draw parallels between dependency theory and other disciplines. See, for example, reference [11.8] and several of the references in Chapter 13.

11.6 E. F. Codd: "Further Normalization of the Data Base Relational Model," in Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6.* Englewood Cliffs, N.J.: Prentice-Hall (1972).

The paper was the first to describe the concept of functional dependence (apart from an early IBM internal memo, also by Codd). The "further normalization" of the title refers to the specific database design discipline discussed in Chapter 12; the purpose of the paper was, very

specifically, to show the applicability of the ideas of functional dependence to the database design problem. (Indeed, FDs represented the first scientific attack on that problem.) As noted in Section 11.1, however, the functional dependency idea has since shown itself to be of much wider applicability (see, e.g., reference [11.7]).

11.7 Hugh Darwen: "The Role of Functional Dependence in Query Decomposition," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

This paper gives a set of inference rules by which FDs holding for an arbitrary derived relvar can be inferred from those holding for the relvar(s) from which the relvar in question is derived. The set of FDs thus inferred can then be inspected to determine candidate keys for the derived relvar, thus providing the *candidate key* inference rules mentioned in passing (very briefly) in Chapters 9 and 10. The paper shows how these various rules can be used to provide significant improvements in DBMS performance, functionality, and usability. *Note:* Such rules are used in the SQL:1999 standard (a) to extend, slightly, the range of views the standard regards as updatable—see Chapter 10—and (b) to extend also the standard's understanding of what it means for an expression (e.g., in the SELECT clause) to be "single-valued per group." As an illustration of this latter point, consider the following query:

```
SELECT S.S#, S.CITY, SUM ( SP.QTY ) AS TQ
FROM   S, SP
WHERE  S.S# = SP.S#
GROUP  BY S.S# ;
```

This query was invalid in SQL:1992 because S.CITY is mentioned in the SELECT clause and not in the GROUP BY clause, but is valid in SQL:1999 because SQL now understands that relvar S satisfies the FD S# → CITY.

11.8 R. Fagin: "Functional Dependencies in a Relational Database and Propositional Logic," *IBM J. R&D 21*, No. 6 (November 1977).

Shows that Armstrong's axioms [11.2] are strictly equivalent to the system of implicational statements in propositional logic. In other words, the paper defines a mapping between FDs and propositional statements, and then shows that a given FD *f* is a consequence of a given set *S* of FDs if and only if the proposition corresponding to *f* is a logical consequence of the set of propositions corresponding to *S*.

11.9 Claudio L. Lucchesi and Sylvia L. Osborn: "Candidate Keys for Relations," *J. Comp. and Sys. Sciences 17*, No. 2 (1978).

Presents an algorithm for finding all candidate keys for a given relvar, given the set of FDs that hold in that relvar.

11.10 David Maier: *The Theory of Relational Databases*. Rockville, Md.: Computer Science Press (1983).

CHAPTER 12

Further Normalization I: 1NF, 2NF, 3NF, BCNF

12.1 INTRODUCTION

Throughout this book so far we have made use of the suppliers-and-parts database as a running example, with logical design as follows (in outline):

```
S  { S#, SNAME, STATUS, CITY }
   PRIMARY KEY { S# }

P  { P#, PNAME, COLOR, WEIGHT, CITY }
   PRIMARY KEY { P# }

SP { S#, P#, QTY }
   PRIMARY KEY { S#, P# }
   FOREIGN KEY { S# } REFERENCES S
   FOREIGN KEY { P# } REFERENCES P
```

Note: As these definitions suggest, we assume in this chapter (until further notice) that relvars always have a primary key specifically.

Now, this design does have a feeling of rightness about it: It is "obvious" that three relvars S, P, and SP are necessary, and it is also "obvious" that the supplier CITY attribute belongs in relvar S, the part COLOR attribute belongs in relvar P, the shipment QTY attribute belongs in relvar SP, and so ou. But what is it that tells us these things are so? Some insight into this questiou can be gained by seeing what happens if we change the design in some way. Suppose, for example, that we move the supplier CITY attribute out of the suppliers relvar and into the shipmeuts relvar (intuitively the wrong place for it, since "supplier city" obviously concerns suppliers, not shipments). Fig. 12.1, a variation on Fig. 11.1 from Chapter 11, shows a sample value for this revised shipments relvar. *Note:* In order to avoid confusion with our usual shipments relvar SP, we will refer to this revised version as SCP, as we did in Chapter 11.

A glance at Fig. 12.1 is sufficient to show what is wrong with this design: redundancy. To be specific, every SCP tuple for supplier S1 tells us S1 is located in London, every SCP tuple for supplier S2 tells us S2 is located in Paris, and so on. More generally, the fact that a given supplier is located in a given city is stated as many times as there are shipments for that supplier. This redundancy in turn leads to several further problems. For example, after an update, supplier S1 might be shown as being located in London according to one tuple and in Amsterdam according to another.[1] So perhaps a good design principle is "one fact in one place" (i.e., avoid redundancy). *The subject of further normalization is essentially just a formalization of simple ideas like this one*—a formalization, however, that does have very practical application to the problem of database design.

Of course, relation values are always normalized as far as the relational model is concerned, as we saw in Chapter 6. As for relation variables (relvars), we can say that

SCP

S#	CITY	P#	QTY
S1	London	P1	300
S1	London	P2	200
S1	London	P3	400
S1	London	P4	200
S1	London	P5	100
S1	London	P6	100
S2	Paris	P1	300
S2	Paris	P2	400
S3	Paris	P2	200
S4	London	P2	200
S4	London	P4	300
S4	London	P5	400

Fig. 12.1 Sample value for relvar SCP

[1] Throughout this chapter and the next, it is necessary to assume (realistically enough!) that relvar predicates are not being fully enforced—for if they were, problems such as this one could not possibly arise (it would not be possible to update the city for supplier S1 in some tuples and not in others). In fact, one way to think about the normalization discipline is as follows: It helps us structure the database in such a way as to make more single-tuple updates logically acceptable than would otherwise be the case (i.e., if the design were not fully normalized). This goal is achieved because the predicates are simpler if the design is fully normalized than they would be otherwise.

they are normalized too as long as their legal values are normalized relations; thus, relvars are always normalized too as far as the relational model is concerned. Equivalently, we can say that relvars (and relations) are always in **first normal form** (abbreviated 1NF). In other words, "normalized" and "1NF" mean *exactly the same thing*—though you should be aware that the term *normalized* is often used to mean one of the higher levels of normalization (typically *third* normal form, 3NF); this latter usage is sloppy but very common.

Now, a given relvar might be normalized in the foregoing sense and yet still possess certain undesirable properties. Relvar SCP is a case in point (see Fig. 12.1). The principles of further normalization allow us to recognize such cases and to replace the relvars in question by ones that are more desirable in some way. In the case of relvar SCP, for example, those principles would tell us precisely what is wrong with that relvar, and they would tell us how to replace it by two "more desirable" relvars, one with attributes S# and CITY and the other with attributes S#, P#, and QTY.

Normal Forms

The process of further normalization—hereinafter abbreviated to just *normalization*—is built around the concept of **normal forms**. A relvar is said to be in a particular normal form if it satisfies a certain prescribed set of conditions. For example, a relvar is said to be in second normal form (2NF) if and only if it is in 1NF and also satisfies another condition, to be discussed in Section 12.3.

Many normal forms have been defined (see Fig. 12.2). The first three (1NF, 2NF, 3NF) were defined by Codd in reference [11.6]. As Fig. 12.2 suggests, all normalized relvars are in 1NF; some 1NF relvars are also in 2NF; and some 2NF relvars are also in 3NF. The motivation behind Codd's definitions was that 2NF was more desirable (in a sense to be explained) than 1NF, and 3NF in turn was more desirable than 2NF. Thus, the database designer should generally aim for a design involving relvars in 3NF, not ones that are merely in 2NF or 1NF.

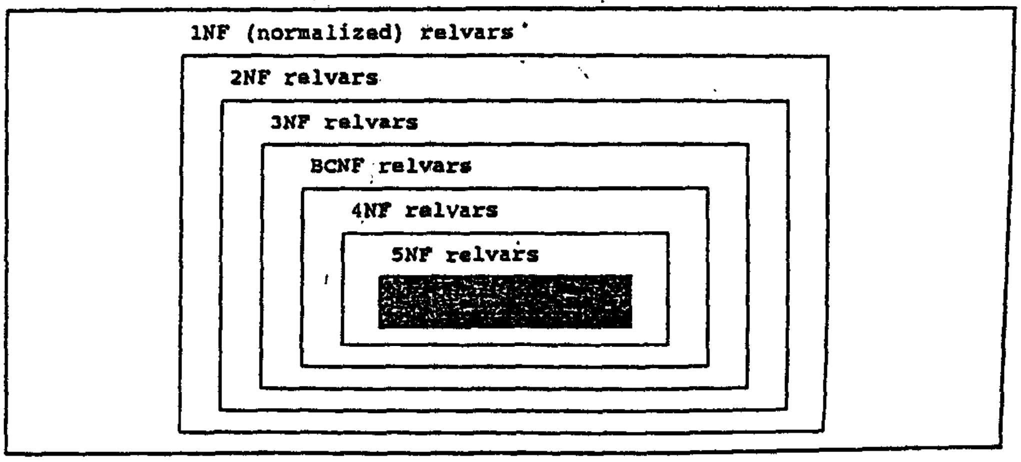


Fig. 12.2 Levels of normalization

Reference [11.6] also introduced the idea of a procedure, the **normalization procedure**, by which a relvar that happens to be in some given normal form, say 2NF, can be replaced by a set of relvars in some more desirable form, say 3NF. (The procedure as originally defined only went as far as 3NF, but it was subsequently extended all the way to 5NF, as we will see in the next chapter.) We can characterize that procedure as *the successive reduction of a given collection of relvars to some more desirable form*. Note that the procedure is **reversible**; that is, it is always possible to take the output from the procedure (say the set of 3NF relvars) and map it back to the input (say the original 2NF relvar). Reversibility is important, because it means the normalization process is **nonloss** or **information-preserving**.

To return to the topic of normal forms *per se:* Codd's original definition of 3NF as given in reference [11.6] turned out to suffer from certain inadequacies, as we will see in Section 12.5. A revised and stronger definition, due to Boyce and Codd, was given in reference [12.2]—stronger, in the sense that any relvar that was in 3NF by the new definition was certainly in 3NF by the old, but a relvar could be in 3NF by the old definition and not by the new. The new 3NF is now usually referred to as **Boyce/Codd** normal form (BCNF) in order to distinguish it from the old form.

Subsequently, Fagin [12.8] defined a new **"fourth"** normal form (4NF—"fourth" because at that time BCNF was still being called "third"). And in reference [12.9] Fagin defined yet another normal form, which he called **projection-join** normal form (PJ/NF, also known as "fifth" normal form or 5NF). As Fig. 12.2 shows, some BCNF relvars are also in 4NF, and some 4NF relvars are also in 5NF.

By now you might well be wondering whether there is any end to this progression, and whether there might be a 6NF, a 7NF, and so on *ad infinitum*. Although this is a good question to ask, we are obviously not yet in a position to give it detailed consideration. We content ourselves with the rather equivocal statement that there are indeed additional normal forms not shown in Fig. 12.2, but 5NF is the "final" normal form in a special (but important) sense. We will return to this question in Chapter 13.

Structure of the Chapter

The aim of this chapter is to examine the concepts of further normalization, up to and including Boyce/Codd normal form (we leave the others to Chapter 13). The plan of the chapter is as follows. Following this lengthy introduction, Section 12.2 discusses the basic concept of *nonloss decomposition*, and demonstrates the crucial importance of functional dependence to this concept (indeed, functional dependence forms the basis for Codd's original three normal forms, as well as for BCNF). Section 12.3 then describes the original three normal forms, showing by example how a given relvar can be carried through the normalization procedure as far as 3NF. Section 12.4 digresses slightly to consider the question of *alternative decompositions*—that is, the question of choosing the "best" decomposition of a given relvar, when there is a choice. Next, Section 12.5 discusses BCNF. Finally, Section 12.6 provides a summary and offers a few concluding remarks.

Caveat: You are warned that we make little attempt at rigor in what follows; rather, we rely to a considerable extent on plain intuition. Indeed, part of the point is that concepts such as nonloss decomposition, BCNF, and so on, despite the somewhat esoteric terminology, are essentially very simple and commonsense ideas. Most of the references treat the material in a much more formal and rigorous manner. A good tutorial can be found in reference [12.5].

Two final introductory remarks:

1. As already suggested, the general idea of normalization is that the database designer should aim for relvars in the "ultimate" normal form (5NF). However, this recommendation should not be construed as law; occasionally—*very* occasionally!—there might be good reasons for flouting the principles of normalization (see, e.g., Exercise 12.7 at the end of the chapter). Indeed, this is as good a place as any to make the point that database design can be an extremely complex task. Normalization is a useful aid in the process, but it is not a panacea; thus, anyone designing a database is certainly advised to be familiar with normalization principles, but we do not claim that the design process should be based on those principles alone. Chapter 14 discusses a number of other aspects of design that have little or nothing to do with normalization as such.
2. As already indicated, we will be using the normalization procedure as a basis for introducing and discussing the various normal forms. However, we do not mean to suggest that database design is actually done by applying that procedure in practice: in fact, it probably is not—it is much more likely that some top-down scheme will be used instead (see Chapter 14). The ideas of normalization can then be used to *verify* that the resulting design does not unintentionally violate any of the normalization principles. Nevertheless, the normalization procedure does provide a convenient framework in which to describe those principles. For the purposes of this chapter, therefore, we adopt the useful fiction that we are indeed carrying out the design process by applying that procedure.

12.2 NONLOSS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES

Before we can get into the specifics of the normalization procedure, we need to examine one crucial aspect of that procedure more closely: namely, the concept of nonloss (also called **lossless**) **decomposition**. We have seen that the normalization procedure involves decomposing a given relvar into other relvars, and moreover that the decomposition is required to be reversible, so that no information is lost in the process; in other words, the only decompositions we are interested in are ones that are indeed nonloss. As we will see, the question of whether a given decomposition is nonloss is intimately bound up with the concept of functional dependence.

S

S#	STATUS	CITY
S3	30	Paris
S5	30	Athens

(a) SST

S#	STATUS
S3	30
S5	30

SC

S#	CITY
S3	Paris
S5	Athens

(b) SST

S#	STATUS
S3	30
S5	30

STC

STATUS	CITY
30	Paris
30	Athens

Fig. 12.3 Sample value for relvar S and two corresponding decompositions

By way of example, consider the familiar suppliers relvar S, with attributes S#, STATUS, and CITY (for simplicity, we ignore attribute SNAME until further notice). Fig. 12.3 shows a sample value for this relvar and—in the parts of the figure labeled *a* and *b*—two possible decompositions corresponding to that sample value.

Examining the two decompositions, we observe that:

1. In Case *a*, no information is lost; the SST and SC values still tell us that supplier S3 has status 30 and city Paris, and supplier S5 has status 30 and city Athens. In other words, this first decomposition is indeed nonloss.
2. In Case *b*, by contrast, information definitely is lost; we can still tell that both suppliers have status 30, but we cannot tell which supplier has which city. In other words, the second decomposition is not nonloss but lossy.

What exactly is it here that makes the first decomposition nonloss and the other lossy? Well, observe first that the process we have been referring to as "decomposition" is actually a process of *projection;* SST, SC, and STC in the figure are each projections of the original relvar S. So the decomposition operator in the normalization procedure is projection. *Note:* As in Part II of this book, we often say things like "SST is a projection of relvar S" when what we should more correctly be saying is "SST is a relvar whose value at any given time is a projection of the relation that is the value of relvar S at that time." We hope these shorthands will not cause any confusion.

Observe next that when we say in Case *a* that no information is lost, what we mean, more precisely, is that *if we join SST and SC back together again, we get back to the original S.* In Case *b*, by contrast, if we join SST and SC together again, we do *not* get back the original S, and so we have lost information.[2] In other words, "reversibility" means, pre-

[2] To be more specific, we get back all of the tuples in the original S, together with some additional "spurious" tuples; we can never get back anything *less* than the original S. (*Exercise:* Prove this statement.) Since we have no way in general of knowing which tuples in the result are spurious and which genuine, we have indeed lost information.

cisely, that *the original relvar is equal to the join of its projections.* Thus, just as the decomposition operator for normalization purposes is projection, so the *re*composition operator is join.

So the interesting question is this: If *R1* and *R2* are both projections of some relvar *R*, and *R1* and *R2* between them include all of the attributes of *R*, what conditions must be satisfied in order to guarantee that joining *R1* and *R2* back together takes us back to the original *R?* And this is where functional dependencies come in. Returning to our example, observe that relvar S satisfies the irreducible set of FDs:

```
S# → STATUS
S# → CITY
```

Given the fact that it satisfies these FDs, it surely cannot be coincidence that relvar S is equal to the join of its projections on {S#,STATUS} and {S#,CITY}—and of course it is not. In fact, we have the following theorem (due to Heath [12.4]):

- **Heath's theorem:** Let *R*{*A*,*B*,*C*} be a relvar, where *A*, *B*, and *C* are sets of attributes. If *R* satisfies the FD *A* → *B*, then *R* is equal to the join of its projections on {*A*,*B*} and {*A*,*C*}.

Taking *A* as S#, *B* as STATUS, and *C* as CITY, this theorem confirms what we have already observed: namely, that relvar S can be nonloss-decomposed into its projections on {S#,STATUS} and {S#,CITY}. At the same time, we also know that relvar S *cannot* be nonloss-decomposed into its projections on {S#,STATUS} and {STATUS,CITY}. Heath's theorem does not explain why this is so;[3] intuitively, however, we can see that the problem is that *one of the FDs is lost in this latter decomposition.* Specifically, the FD S# → STATUS is still represented (by the projection on {S#,STATUS}), but the FD S# → CITY has been lost.

To sum up, then, we can say that the decomposition of relvar *R* into projections *R1*, *R2*, ..., *Rn* is nonloss if *R* is equal to the join of *R1*, *R2*, ..., *Rn*. *Note:* In practice we would probably want to impose the additional requirement that *R1*, *R2*, ..., *Rn* are all needed in the join, in order to guarantee that certain redundancies that might otherwise occur are avoided. For example, we probably would not want to consider the decomposition of relvar S into its projections on (say) {S#}, {S#,STATUS}, and {S#,CITY} as a nonloss decomposition, yet S is certainly equal to the join of those three projections. For simplicity, let us agree from this point forward that this additional requirement is always in force, barring explicit statements to the contrary.

More on Functional Dependencies

We conclude this section with a few additional remarks concerning FDs.

1. **Irreducibility:** Recall from Chapter 11 that an FD is said to be left-irreducible if its left side is "not too big." For example, consider relvar SCP once again from Section 12.1. That relvar satisfies the FD

[3] It does not do so because it is of the form "if . . . then . . . ," not "if *and only if* . . . then . . ." (see Exercise 12.1 at the end of the chapter). We will be discussing a stronger form of Heath's theorem in Chapter 13, Section 13.2.

```
{ S#, P# } → CITY
```

However, attribute P# on the left side here is redundant for functional dependency purposes; that is, we also have the FD

```
S# → CITY
```

(CITY is also functionally dependent on S# alone). This latter FD is left-irreducible, but the previous one is not; equivalently, CITY is **irreducibly dependent** on S#, but not irreducibly dependent on {S#,P#}.[4] Left-irreducible FDs and irreducible dependencies will turn out to be important in the definition of the second and third normal forms (see Section 12.3).

2. **FD diagrams:** Let *R* be a relvar and let *I* be some irreducible set of FDs that apply to *R* (again, refer to Chapter 11 if you need to refresh your memory regarding irreducible sets of FDs). It is convenient to represent the set *I* by means of a *functional dependency diagram* (FD diagram). FD diagrams for relvars S, SP, and P—which should be self-explanatory—are given in Fig. 12.4. We will make frequent use of such diagrams throughout the rest of this chapter.

   Now, you will observe that every arrow in Fig. 12.4 is an *arrow out of a candidate key* (actually the primary key) of the relevant relvar. By definition, there will always be arrows out of each candidate key,[5] because, for one value of each candidate key, there is always one value of everything else; those arrows can never be eliminated. *It is if there are any other arrows that difficulties arise.* Thus, the normalization procedure can be characterized, very informally, as a procedure for eliminating arrows that are not arrows out of candidate keys.

3. **FDs are a semantic notion:** As noted in Chapter 11, FDs are really a special kind of integrity constraint. As such, they are definitely a *semantic* notion (in fact, they are

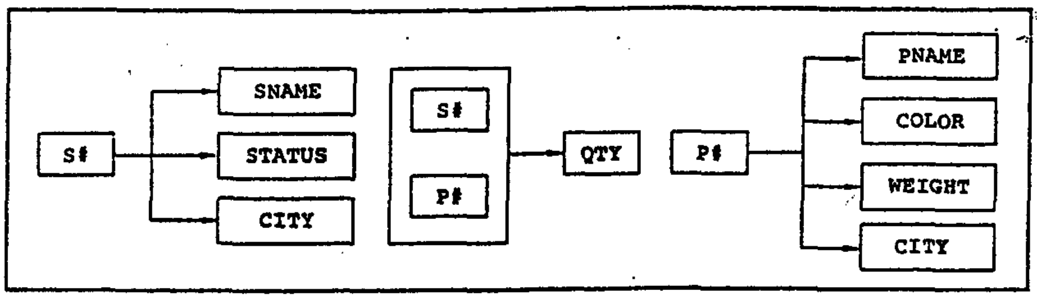


Fig. 12.4 FD diagrams for relvars S, SP, and P

[4] *Left-irreducible FD* and *irreducibly dependent* are our preferred terms for what are more usually called "full FD" and "fully dependent" in the literature (and were so called in the first few editions of this book). These latter terms have the merit of brevity but are less descriptive and less apt.

[5] More precisely, there will always be arrows out of *superkeys*. If the set *I* of FDs is irreducible as stated, however, all FDs (or "arrows") in *I* will be left-irreducible.

part of the relvar predicate). Recognizing the FDs is part of the process of understanding what the data *means;* the fact that relvar S satisfies the FD S# → CITY, for example, means that each supplier is located in precisely one city. To look at this another way:

- There is a constraint in the real world that the database represents: namely, that each supplier is located in precisely one city.
- Since it is part of the semantics of the situation, that constraint must somehow be observed in the database.
- The way to ensure that it is so observed is to specify it in the database definition, so that the DBMS can enforce it.
- The way to specify it in the database definition is to declare the FD.

And we will see later that the concepts of normalization lead to a very simple means of declaring FDs.

12.3 FIRST, SECOND, AND THIRD NORMAL FORMS

Caveat: Throughout this section, we assume for simplicity that each relvar has exactly one candidate key, which we further assume is the primary key. These assumptions are reflected in our definitions, which (we repeat) are not very rigorous. The case of a relvar having more than one candidate key is discussed in Section 12.5.

We are now in a position to describe Codd's original three normal forms. We present a preliminary, very informal, definition of 3NF first in order to give some idea of what we are aiming for. We then consider the process of reducing an arbitrary relvar to an equivalent collection of 3NF relvars, giving somewhat more precise definitions as we go. However, we note at the outset that 1NF, 2NF, and 3NF are not very significant in themselves except as stepping-stones to BCNF (and beyond).

Here then is our preliminary 3NF definition:

- **Third normal form** *(very informal definition):* A relvar is in 3NF if and only if the nonkey attributes (if any) are both:

  a. Mutually independent

  b. Irreducibly dependent on the primary key

We explain the terms *nonkey attribute* and *mutually independent* (loosely) as follows:

- A *nonkey attribute* is any attribute that does not participate in the primary key of the relvar concerned.
- Two or more attributes are *mutually independent* if none of them is functionally dependent on any combination of the others. Such independence implies that each attribute can be updated independently of the rest.

By way of example, the parts relvar P is in 3NF according to the foregoing definition: Attributes PNAME, COLOR, WEIGHT, and CITY are all independent of one another (it is possible to change, e.g., the color of a part without simultaneously having to change its weight), and they are all irreducibly dependent on the primary key {P#}.

The foregoing informal definition of 3NF can be interpreted, even more informally, as follows:

- **Third normal form** *(even more informal definition):* A relvar is in third normal form (3NF) if and only if, for all time, each tuple consists of a primary key value that identifies some entity, together with a set of zero or more mutually independent attribute values that describe that entity in some way.

Again, relvar P fits the definition: Each tuple of P consists of a primary key value (a part number) that identifies some part in the real world, together with four additional values (part name, part color, part weight, part city), each of which serves to describe that part, and each of which is independent of all of the others.

Now we turn to the normalization procedure. We begin with a definition of first normal form:

- **First normal form:** A relvar is in 1NF if and only if, in every legal value of that relvar, every tuple contains exactly one value for each attribute.

This definition merely states that relvars are always in 1NF, which is of course correct. However, a relvar that is *only* in first normal form—that is, a 1NF relvar that is not also in 2NF, and therefore not in 3NF either—has a structure that is undesirable for a number of reasons. To illustrate the point, let us suppose that information concerning suppliers and shipments, rather than being split into the two relvars S and SP, is lumped together into a single relvar as follows:

```
FIRST { S#, STATUS, CITY, P#, QTY }
      PRIMARY KEY { S#, P# }
```

This relvar is an extended version of SCP from Section 12.1. The attributes have their usual meanings, except that for the sake of the example we introduce an additional constraint:

```
CITY → STATUS
```

(STATUS is functionally dependent on CITY; the meaning of this constraint is that a supplier's status is determined by the location of that supplier—e.g., all London suppliers *must* have a status of 20). Also, we ignore SNAME again, for simplicity. The primary key of FIRST is the combination {S#,P#}; the FD diagram is shown in Fig. 12.5.

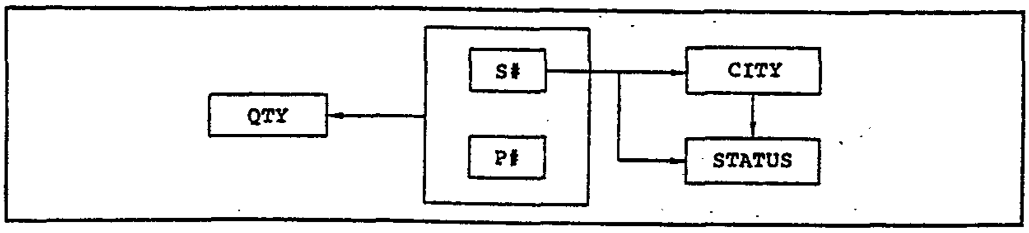


Fig. 12.5 FDs for relvar FIRST

Observe that this FD diagram is, informally, "more complex" than the FD diagram for a 3NF relvar. As indicated in the previous section, a 3NF diagram has arrows out of candidate keys only, whereas a non3NF diagram such as that for FIRST has arrows out of candidate keys *together with certain additional arrows*—and it is those additional arrows that cause the trouble. In fact, relvar FIRST violates both conditions *a* and *b* in our preliminary 3NF definition: The nonkey attributes are not all mutually independent, because STATUS depends on CITY (one additional arrow), and they are not all irreducibly dependent on the primary key, because STATUS and CITY are each dependent on S# alone (two more additional arrows).

As a basis for illustrating some of the difficulties that arise from those additional arrows, Fig. 12.6 shows a sample value for relvar FIRST. The attribute values are basically as usual, except that the status of supplier S3 has been changed from 30 to 10 to be consistent with the new constraint that CITY determines STATUS. The redundancies are obvious. For example, every tuple for supplier S1 shows the city as London; likewise, every tuple for city London shows the status as 20.

The redundancies in relvar FIRST lead to a variety of what (for historical reasons) are usually called **update anomalies**—that is, difficulties with the update operations INSERT, DELETE, and UPDATE. To fix our ideas, we concentrate first on the supplier-city redundancy, corresponding to the FD S# → CITY. Problems occur with each of the three update operations:

- **INSERT:** We cannot insert the fact that a particular supplier is located in a particular city until that supplier supplies at least one part. Indeed, Fig. 12.6 does not show that supplier S5 is located in Athens. The reason is that, uutil S5 supplies some part, we have no appropriate primary key value. (As in Chapter 10, Section 10.4, we assume throughout this chapter—reasonably enough—that primary key attributes have no default value. See Chapter 19 for further discussion.)
- **DELETE:** If we delete the sole FIRST tuple for a particular supplier, we delete not only the shipment connecting that supplier to a particular part but also the information that the supplier is located in a particular city. For example, if we delete the

FIRST

S#	STATUS	CITY	P#	QTY
S1	20	London	P1	300
S1	20	London	P2	200
S1	20	London	P3	400
S1	20	London	P4	200
S1	20	London	P5	100
S1	20	London	P6	100
S2	10	Paris	P1	300
S2	10	Paris	P2	400
S3	10	Paris	P2	200
S4	20	London	P2	200
S4	20	London	P4	300
S4	20	London	P5	400

Fig. 12.6 Sample value for relvar FIRST

FIRST tuple with S# value S3 and P# value P2, we lose the information that S3 is located in Paris. (The INSERT and DELETE problems are really two sides of the same coin.)

Note: The real problem here is that relvar FIRST contains too much information all bundled together; hence, when we delete a tuple, *we delete too much.* To be more precise, relvar FIRST contains information regarding shipments *and* information regarding suppliers, and so deleting a shipment causes supplier information to be deleted as well. The solution to this problem is to "unbundle"—that is, to place shipment information in one relvar and supplier information in another (and this is exactly what we will do in just a moment). Thus, another informal way of characterizing the normalization procedure is as an *unbundling* procedure: Place logically separate information into separate relvars.

- UPDATE: The city value for a given supplier appears in FIRST many times, in general. This redundancy causes update problems. For example, if supplier S1 moves from London to Amsterdam, we are faced with *either* the problem of searching FIRST to find every tuple connecting S1 and London (and changing it) *or* the possibility of producing an inconsistent result (the city for S1 might be given as Amsterdam in one tuple and London in another).

The solution to these problems is, as already indicated, to replace relvar FIRST by the two relvars:

```
SECOND { S#, STATUS, CITY }
SP     { S#, P#, QTY }
```

The FD diagrams for these two relvars are given in Fig. 12.7; sample values are given in Fig. 12.8. Observe that information for supplier S5 has now been included (in relvar SECOND but not in relvar SP). In fact, relvar SP is now exactly our usual shipments relvar.

It should be clear that this revised structure overcomes all of the problems with update operations sketched earlier:

- INSERT: We can insert the information that S5 is located in Athens, even though S5 does not currently supply any parts, by simply inserting the appropriate tuple into SECOND.

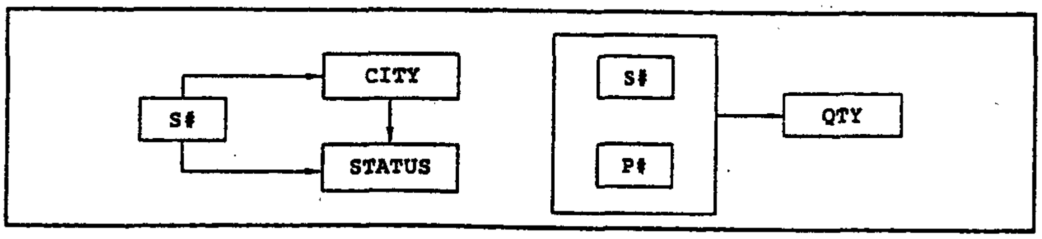


Fig. 12.7 FDs for relvars SECOND and SP

SECOND

S#	STATUS	CITY
S1	20	London
S2	10	Paris
S3	10	Paris
S4	20	London
S5	30	Athens

No counterpart in Fig. 12.6

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig. 12.8 Sample values for relvars SECOND and SP

- **DELETE:** We can delete the shipment connecting S3 and P2 by deleting the appropriate tuple from SP; we do not lose the information that S3 is located in Paris.
- **UPDATE:** In the revised structure, the city for a given supplier appears just once, not many times, because there is precisely one tuple for a given supplier in relvar SECOND (the primary key for that relvar is {S#}); in other words, the S#-CITY redundancy has been eliminated. Thus, we can change the city for S1 from London to Amsterdam by changing it once and for all in the relevant SECOND tuple.

Comparing Figs. 12.5 and 12.7, we see that the effect of the decomposition of FIRST into SECOND and SP has been to eliminate the dependencies that were not irreducible, and it is that elimination that has resolved the difficulties. Intuitively, we can say that in relvar FIRST the attribute CITY did not describe the entity identified by the primary key (i.e., a shipment); instead, it described the supplier involved in that shipment (and likewise for attribute STATUS). Mixing the two kinds of information in the same relvar was what caused the problems in the first place.

We now give a definition of second normal form:[6]

- **Second normal form** *(definition assuming only one candidate key, which we assume is the primary key):* A relvar is in 2NF if and only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key.

Relvars SECOND and SP are both in 2NF (the primary keys are {S#} and the combination {S#,P#}, respectively). Relvar FIRST is not in 2NF. A relvar that is in first normal form and not in second can always be reduced to an equivalent collection of 2NF relvars. The reduction process consists of replacing the 1NF relvar by suitable projections; the collection of projections so obtained is equivalent to the original relvar, in the sense that the original relvar can always be recovered by joining those projections back together

[6] Strictly speaking, 2NF can be defined only *with respect to a specified set of dependencies*, but it is usual to ignore this point in informal discussion. Analogous remarks apply to all of the normal forms other than first.

again. In our example, SECOND and SP are projections of FIRST,[7] and FIRST is the join of SECOND and SP over S#.

To summarize, the first step in the normalization procedure is to take projections to eliminate "nonirreducible" functional dependencies. Thus, given relvar *R* as follows—

```
R { A, B, C, D }
  PRIMARY KEY { A, B }
  /* assume A → D holds */
```

—the normalization discipline recommends replacing *R* by its two projections *R1* and *R2*, as follows:

```
R1 { A, D }
   PRIMARY KEY { A }

R2 { A, B, C }
   PRIMARY KEY { A, B }
   FOREIGN KEY { A } REFERENCES R1
```

R can be recovered by taking the foreign-to-matching-primary-key join of *R2* and *R1*.

To return to the example: The SECOND-SP structure still causes problems, however. Relvar SP is satisfactory; as a matter of fact, relvar SP is now in 3NF, and we will ignore it for the rest of this section. Relvar SECOND, on the other hand, still suffers from a lack of mutual independence among its nonkey attributes. The FD diagram for SECOND is still "more complex" than a 3NF diagram. To be specific, the dependency of STATUS on S#, though it is functional, and indeed irreducible, is transitive (via CITY): Each S# value determines a CITY value, and that CITY value in turn determines the STATUS value. More generally, whenever the FDs $A \rightarrow B$ and $B \rightarrow C$ both hold, then it is a logical consequence that the transitive FD $A \rightarrow C$ holds also, as explained in Chapter 11. And transitive dependencies lead, once again, to update anomalies (we concentrate now on the city-status redundancy, corresponding to the FD CITY → STATUS):

- INSERT: We cannot insert the fact that a particular city has a particular status—for example, we cannot state that any supplier in Rome must have a status of 50—until we have some supplier actually located in that city.
- DELETE: If we delete the sole SECOND tuple for a particular city, we delete not only the information for the supplier concerned but also the information that that city has that particular status. For example, if we delete the SECOND tuple for S5, we lose the information that the status for Athens is 30. (Again, the INSERT and DELETE problems are really two sides of the same coin.)

  Note: The problem is bundling again, of course: Relvar SECOND contains information regarding suppliers *and* information regarding cities. And again the solution is to "unbundle"—that is, to place supplier information in one relvar and city information in another.

[7] Except for the fact that SECOND can include tuples, such as the tuple for supplier S5 in Fig. 12.8, that have no counterpart in FIRST; in other words, the new structure can represent information that could not be represented in the previous one. In this sense, the new structure can be regarded as a slightly more faithful representation of the real world.

- **UPDATE:** The status for a given city appears in SECOND many times, in general (the relvar still contains some redundancy). Thus, if we need to change the status for London from 20 to 30, we are faced with *either* the problem of searching SECOND to find every tuple for London (and changing it) *or* the possibility of producing an inconsistent result (the status for London might be given as 20 in one tuple and 30 in another).

Again the solution to the problems is to replace the original relvar (SECOND, in this case) by two projections—namely, the projections:

```
SC { S#, CITY }
CS { CITY, STATUS }
```

The FD diagrams for these two relvars are given in Fig. 12.9; sample values are given in Fig. 12.10. Observe that status information for Rome has been included in relvar CS. The reduction is reversible once again, since SECOND is the join of SC and CS over CITY.

It should be clear, again, that this revised structure overcomes all of the problems with update operations sketched earlier. Detailed consideration of those problems is left as an exercise. Comparing Figs. 12.7 and 12.9, we see that the effect of the further decomposition is to eliminate the transitive dependence of STATUS on S#, and again it is that elimination that has resolved the difficulties. Intuitively, we can say that in relvar SECOND the attribute STATUS did not describe the entity identified by the primary key (i.e., a supplier); instead, it described the city in which that supplier happened to be located. Once again, mixing the two kinds of information in the same relvar was what caused the problems.

We now give a definition of third normal form:

- **Third normal form** *(definition assuming only one candidate key, which we further assume is the primary key):* A relvar is in 3NF if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.

$$\begin{tikzcd}\boxed{\text{S\#}} \arrow[r] & \boxed{\text{CITY}} & & \boxed{\text{CITY}} \arrow[r] & \boxed{\text{STATUS}}\end{tikzcd}$$

Fig. 12.9 FDs for relvars SC and CS

SC

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

CS

CITY	STATUS
Athens	30
London	20
Paris	10
Rome	50

No counterpart in Fig. 12.8

Fig. 12.10 Sample values for relvars SC and CS

(Note that "no transitive dependencies" implies no *mutual* dependencies, in the sense of that term explained near the beginning of this section.)

Relvars SC and CS are both in 3NF (the primary keys are {S#} and {CITY}, respectively). Relvar SECOND is not in 3NF. A relvar that is in second normal form and not in third can always be reduced to an equivalent collection of 3NF relvars. We have already indicated that the process is reversible, and hence that no information is lost in the reduction; however, the 3NF collection can contain information, such as the fact that the status for Rome is 50, that could not be represented in the original 2NF relvar.[8]

To summarize, the second step in the normalization procedure is to take projections to eliminate transitive dependencies. In other words, given relvar *R* as follows—

```
R { A, B, C }
  PRIMARY KEY { A }
  /* assume B → C holds */
```

—the normalization discipline recommends replacing *R* by its two projections *R1* and *R2*, as follows:

```
R1 { B, C }
   PRIMARY KEY { B }

R2 { A, B }
   PRIMARY KEY { A }
   FOREIGN KEY { B } REFERENCES R1
```

R can be recovered by taking the foreign-to-matching-primary-key join of *R2* and *R1*.

We conclude this section by stressing the point that the level of normalization of a given relvar is a matter of semantics, not merely a matter of the value that relvar happens to have at some particular time. In other words, it is not possible just to look at the value at a given time and to say whether the relvar is in (say) 3NF—it is also necessary to know the dependencies before such a judgment can be made. Note too that even knowing the dependencies, it is never possible to *prove* by examining a given value that the relvar is in 3NF. The best that can be done is to show that the value in question does not violate any of the dependencies; assuming it does not, then the value is *consistent with the hypothesis* that the relvar is in 3NF, but that fact does not guarantee that the hypothesis is valid.

.4 DEPENDENCY PRESERVATION

It is frequently the case that a given relvar can be nonloss-decomposed in a variety of different ways. Consider the relvar SECOND from Section 12.3 once again, with FDs S# → CITY and CITY → STATUS and therefore also, by transitivity, S# → STATUS (refer to Fig. 12.11, in which the transitive FD is shown as a broken arrow). We showed in Section 12.3 that the update anomalies encountered with SECOND could be overcome by replacing it by its decomposition into the two 3NF projections:

```
SC { S#, CITY }
```

[8] It follows that, just as the SECOND-SP combination was a slightly better representation of the real world than the 1NF relvar FIRST, so the SC-CS combination is a slightly better representation than the 2NF relvar SECOND.

```
CS { CITY, STATUS }
```

Let us refer to this decomposition as Decomposition *A*. Here by contrast is an alternative decomposition (Decomposition *B*):

```
SC { S#, CITY }
SS { S#, STATUS }
```

(projection SC is the same in both cases). Decomposition *B* is also nonloss, and the two projections are again both in 3NF. But Decomposition *B* is less satisfactory than Decomposition *A* for a number of reasons. For example, it is still not possible in *B* to insert the information that a particular city has a particular status unless some supplier is located in that city.

Let us examine this example a little more closely. First, note that the projections in Decomposition *A* correspond to the *solid* arrows in Fig. 12.11, whereas one of the projections in Decomposition *B* corresponds to the *broken* arrow. In Decomposition *A*, in fact, the two projections are independent of one another, in the following sense: Updates can be made to either one without regard for the other.[9] Provided only that such an update is legal within the context of the projection concerned—which means only that it must not violate the primary key uniqueness constraint for that projection—then *the join of the two projections after the update will always be a valid SECOND* (i.e., the join cannot possibly violate the FD constraints on SECOND). In Decomposition *B*, by contrast, updates to either of the two projections must be monitored to ensure that the FD CITY → STATUS is not violated (if two suppliers have the same city, then they must have the same status; consider, for example, what is involved in Decomposition *B* in moving supplier S1 from London to Paris). In other words, the two projections are not independent of each other in Decomposition *B*.

The basic problem is that, in Decomposition *B*, the FD CITY → STATUS has become—to use the terminology of Chapter 9—a *database constraint* that spans two relvars (implying, incidentally, that in many of today's products it will have to be maintained by procedural code). In Decomposition *A*, by contrast, it is the *transitive* FD S# → STATUS that has become the database constraint, and that constraint will be enforced automatically if the two *relvar* constraints S# → CITY and CITY → STATUS are enforced. And enforcing these two latter constraints is very simple, of course, involving as it does nothing more than enforcing the corresponding primary key uniqueness constraints.

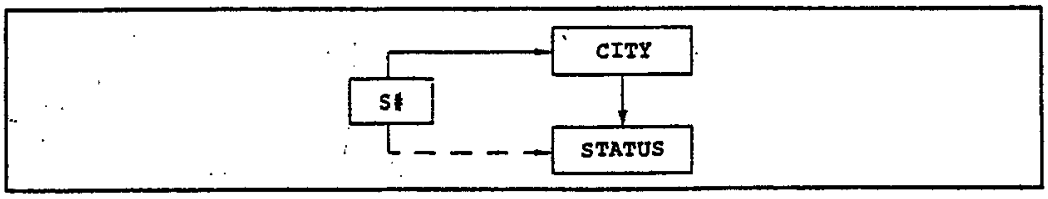


Fig. 12.11 FDs for relvar SECOND

[9] Except for the referential constraint from SC to CS.

The concept of independent projections thus provides a guideline for choosing a particular decomposition when there is more than one possibility. Specifically, a decomposition in which the projections are independent in the foregoing sense is generally preferable to one in which they are not. Rissanen shows in reference [12.6] that projections *R1* and *R2* of a relvar *R* are independent in the foregoing sense if and only if both of the following are true:

- Every FD in *R* is a logical consequence of those in *R1* and *R2*.
- The common attributes of *R1* and *R2* form a candidate key for at least one of the pair.

Consider Decompositions *A* and *B* as defined earlier. In *A* the two projections are independent, because their common attribute CITY constitutes the primary key for CS, and every FD in SECOND either appears in one of the two projections or is a logical consequence of those that do. In *B*, by contrast, the two projections are not independent, because the FD CITY → STATUS cannot be deduced from the FDs for those projections—although it is true that their common attribute, S#, does constitute a candidate key for both. *Note:* The third possibility, replacing SECOND by its two projections on {S#,STATUS} and {CITY,STATUS}, is not a valid decomposition, because it is not nonloss. *Exercise:* Prove this statement.

Reference [12.6] calls a relvar that cannot be decomposed into independent projections atomic (not a terribly good term). Note carefully, however, that the fact that some given relvar is not atomic in this sense should not necessarily be taken to mean that it should be decomposed into atomic components; for example, relvars S and P of the suppliers-and-parts database are not atomic, but there seems little point in decomposing them further. (Relvar SP, by contrast, *is* atomic.)

The idea that the normalization procedure should decompose relvars into projections that are independent in Rissanen's sense has come to be known as dependency preservation. We close this section by explaining this concept more precisely:

1. Suppose we are given some relvar *R*, which (after we have applied all steps of the normalization procedure) we replace by a set of projections *R1, R2, ..., Rn*.
2. Let the set of given FDs for the original relvar *R* be *S*, and let the sets of FDs that apply to relvars *R1, R2, ..., Rn* be *S1, S2, ..., Sn*, respectively.
3. Each FD in the set *Si* will refer to attributes of *Ri* only ($i = 1, 2, ..., n$). Enforcing the constraints (FDs) in any given set *Si* is thus a simple matter. But what we need to do is to enforce the constraints in the original set *S*. We would therefore like the decomposition into *R1, R2, ..., Rn* to be such that enforcing the constraints in *S1, S2, ..., Sn* individually is together equivalent to enforcing the constraints in the original set *S*—in other words, we would like the decomposition to be *dependency-preserving*.
4. Let *S'* be the union of *S1, S2, ..., Sn*. Note that it is *not* the case that *S'* = *S*, in general; in order for the decomposition to be dependency-preserving, however, it is sufficient that the *closures* of *S* and *S'* be equal (refer back to Chapter 11, Section 11.4, if you need to refresh your memory regarding the notion of the closure of a set of FDs).
5. There is no efficient way of computing the closure S^+ of a set of FDs, in general, so actually computing the two closures and testing them for equality is infeasible.

Nevertheless, there *is* an efficient way of testing whether a given decomposition is dependency-preserving. Details of the algorithm are beyond the scope of this chapter; see, for example, reference [8.13] for the specifics.

Here for purposes of future reference is a nine-step algorithm by which an arbitrary relvar *R* can be nonloss-decomposed, in a dependency-preserving way, into a set *D* of 3NF projections. Let the set of FDs satisfied by *R* be *S*. Then:

1. Initialize *D* to the empty set.
2. Let *I* be an irreducible cover for *S*.
3. Let *X* be a set of attributes appearing on the left side of some FD $X \rightarrow Y$ in *I*.
4. Let the complete set of FDs in *I* with left side *X* be $X \rightarrow Y1$, $X \rightarrow Y2$, ..., $X \rightarrow Yn$.
5. Let the union of *Y1*, *Y2*, ..., *Yn* be *Z*.
6. Replace *D* by the union of *D* and the projection of *R* over *X* and *Z*.
7. Repeat Steps 4 through 6 for each distinct *X*.
8. Let *A1*, *A2*, ..., *An* be those attributes of *R* (if any) still unaccounted for (i.e., still not included in any relvar in *D*); replace *D* by the union of *D* and the projection of *R* over *A1*, *A2*, ..., *An*.
9. If no relvar in *D* includes a candidate key of *R*, replace *D* by the union of *D* and the projection of *R* over some candidate key of *R*.

12.5 BOYCE/CODD NORMAL FORM

In this section we drop our assumption that every relvar has just one candidate key and consider what happens in the general case. The fact is, Codd's original definition of 3NF in reference [11.6] did not treat the general case satisfactorily. To be precise, it did not adequately deal with the case of a relvar that

1. Had two or more candidate keys, such that
2. The candidate keys were composite, and
3. They overlapped (i.e., had at least one attribute in common).

The original definition of 3NF was therefore subsequently replaced by a stronger definition, due to Boyce and Codd, that catered for this case also [12.2]. However, since that new definition actually defines a normal form that is strictly stronger than the old 3NF, it is better to use a different name for it, instead of just continuing to call it 3NF; hence the name *Boyce/Codd normal form* (BCNF).[10] *Note:* The combination of conditions 1, 2, and 3 might not occur very often in practice. For a relvar where it does not, 3NF and BCNF are equivalent.

In order to explain BCNF, we first remind you of the term ***determinant***, introduced in Chapter 11 to refer to the left side of an FD. We also remind you of the term ***trivial FD***,

[10] A definition of "third" normal form that was in fact equivalent to the BCNF definition was first given by Heath in 1971 [12.4]; "Heath normal form" might thus have been a more appropriate name.

which is an FD in which the left side is a superset of the right side. Now we can define BCNF:

- **Boyce/Codd normal form:** A relvar is in BCNF if and only if every nontrivial, left-irreducible FD has a candidate key as its determinant.

Or less formally:

- **Boyce/Codd normal form** *(informal definition):* A relvar is in BCNF if and only if every determinant is a candidate key.

In other words, the only arrows in the FD diagram are arrows out of candidate keys. We have already explained that there will always be arrows out of candidate keys; the BCNF definition says *there are no other arrows,* meaning there are no arrows that can be eliminated by the normalization procedure. *Note:* The difference between the two BCNF definitions is that we tacitly assume in the informal case (a) that determinants are "not too big" and (b) that all FDs are nontrivial. In the interest of simplicity, we will continue to make these same assumptions throughout the rest of this chapter, barring explicit statements to the contrary.

It is worth pointing out that the BCNF definition is conceptually simpler than the old 3NF definition, in that it makes no explicit reference to first and second normal forms as such, nor to the concept of transitive dependence. Furthermore, although (as already stated) BCNF is strictly stronger than 3NF, it is still the case that any given relvar can be nonloss-decomposed into an equivalent collection of BCNF relvars.

Before considering some examples involving more than one candidate key, let us convince ourselves that relvars FIRST and SECOND, which were not in 3NF, are not in BCNF either; also that relvars SP, SC, and CS, which were in 3NF, are also in BCNF. Relvar FIRST contains three determinants, {S#}, {CITY}, and {S#,P#}; of these, only {S#,P#} is a candidate key, so FIRST is not in BCNF. Similarly, SECOND is not in BCNF either, because the determinant {CITY} is not a candidate key. Relvars SP, SC, and CS, on the other hand, are each in BCNF, because in each case the sole candidate key is the only determinant in the relvar.

We now consider an example involving two disjoint—that is, nonoverlapping—candidate keys. Suppose in the usual suppliers relvar S{S#,SNAME,STATUS,CITY} that {S#} and {SNAME} are both candidate keys (i.e., for all time, it is the case that every supplier has a unique supplier number and also a unique supplier name). Assume, however (as elsewhere in this book), that attributes STATUS and CITY are mutually independent—that is, the FD CITY → STATUS, which we introduced purely for the purposes of Section 12.3, no longer holds. Then the FD diagram is as shown in Fig. 12.12.

Relvar S is in BCNF. Although the FD diagram does look "more complex" than a 3NF diagram, it is nevertheless still the case that the only determinants are candidate keys; that is, the only arrows are arrows out of candidate keys. So the message of this first example is just that having more than one candidate key is not necessarily bad.

Now we present some examples in which the candidate keys overlap. Two candidate keys overlap if they involve two or more attributes each and have at least one attribute in common. *Note:* In accordance with our discussions on this subject in Chapter 9, we will

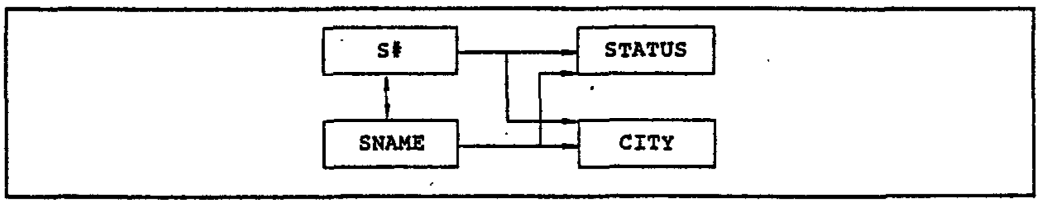


Fig. 12.12 FDs for relvar S if {SNAME} is a candidate key (and CITY → STATUS does not hold)

not attempt to choose one of the candidate keys as the primary key in any of the examples that follow. We will therefore also not mark any columns with double underlining in our figures in this section.

For our first example, we suppose again that supplier names are unique, and we consider the relvar:

```
SSP { S#, SNAME, P#, QTY }
```

The candidate keys are {S#,P#} and {SNAME,P#}. Is this relvar in BCNF? The answer is no, because it contains two determinants, S# and SNAME, that are not candidate keys for the relvar ({S#} and {SNAME} are both determinants because each determines the other). A sample value for this relvar is shown in Fig. 12.13.

As the figure shows, relvar SSP involves the same kind of redundancies as relvars FIRST and SECOND of Section 12.3 (and relvar SCP of Section 12.1) did, and hence suffers from the same kind of problems as those relvars did. For example, changing the name of supplier S1 from Smith to Robinson leads, once again, either to search problems or to possibly inconsistent results. Yet SSP *is* in 3NF by the old definition, because that definition did not require an attribute to be irreducibly dependent on each candidate key if it was itself a component of some candidate key of the relvar; thus, the fact that SNAME is not irreducibly dependent on {S#,P#} was ignored. *Note:* By "3NF" here we mean 3NF as originally defined by Codd in reference [11.6], not the simplified version as defined in Section 12.3.

The solution to the SSP problems is to break the relvar down into two projections, in this case the projections:

```
SS { S#, SNAME }
SP { S#, P#, QTY }
```

SSP

S#	SNAME	P#	QTY
S1	Smith	P1	300
S1	Smith	P2	200
S1	Smith	P3	400
S1	Smith	P4	200
..	.....	..	...

Fig. 12.13 Sample value (partial) for relvar SSP

—or alternatively the projections

```
SS { S#, SNAME }
SP { SNAME, P#, QTY }
```

(there are two equally valid decompositions in this example). All of these projections are in BCNF.

At this point, we should probably stop for a moment and reflect on what is "really" going on here. The original design, consisting of the single relvar SSP, is *clearly* bad; the problems with it are intuitively obvious, and it is unlikely that any competent database designer would ever seriously propose it, even if he or she had no exposure to the ideas of BCNF and so on at all. Common sense would tell the designer that the SS-SP design is better. But what do we mean by "common sense"? What are the *principles* (inside the designer's brain) that the designer is applying when he or she chooses the SS-SP design over the SSP design?

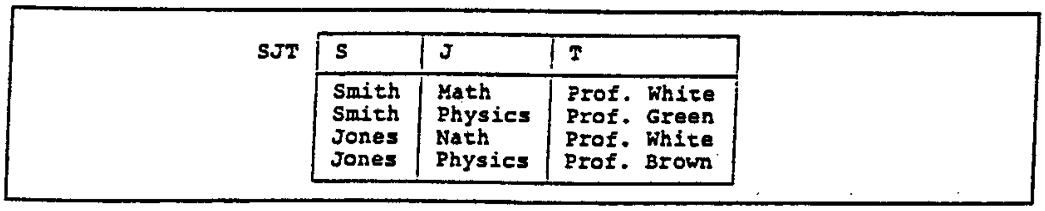
The answer is, of course, that they are exactly the principles of functional dependency and Boyce/Codd normal form. In other words, those concepts (FD, BCNF, and all of the other formal ideas discussed in this chapter and the next) are nothing more nor less than *formalized common sense*. The whole point of normalization theory is to try to identify such commonsense principles and formalize them—which is not an easy thing to do! But if it can be done, then we can *mechanize* those principles; in other words, we can write a program and get the machine to do the work. Critics of normalization usually miss this point; they claim, quite rightly, that the ideas are all basically common sense, but they typically do not realize that it is a significant achievement to state what "common sense" means in a precise and formal way.

To return to the main thread of our discussion: As a second example of overlapping candidate keys—an example that we should warn you some people might consider pathological—we consider a relvar SJT with attributes S, J, and T, standing for student, subject, and teacher, respectively. The meaning of an SJT tuple (*s,j,t*)—simplified notation—is that student *s* is taught subject *j* by teacher *t*. The following constraints apply:

- For each subject, each student of that subject is taught by only one teacher.
- Each teacher teaches only one subject (but each subject is taught by several teachers).

A sample SJT value is given in Fig. 12.14.

What are the FDs for relvar SJT? From the first constraint, we have the FD {S,J} → T. From the second constraint, we have the FD T → J. Finally, the fact that each subject is


SJT

S	J	T
Smith	Math	Prof. White
Smith	Physics	Prof. Green
Jones	Nath	Prof. White
Jones	Physics	Prof. Brown

Fig. 12.14 Sample value for relvar SJT

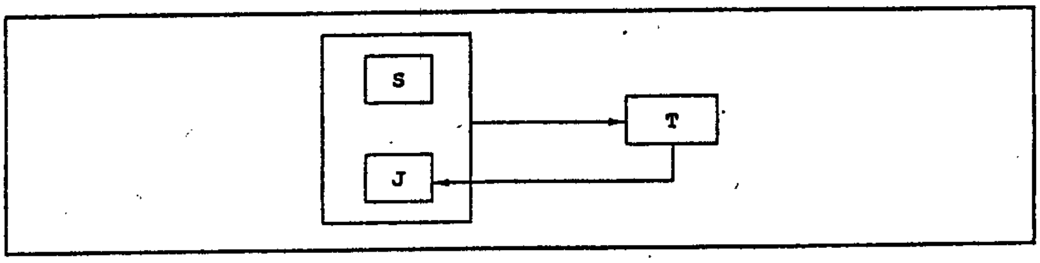


Fig. 12.15 FDs for relvar SJT

taught by several teachers tells us that the FD J → T does *not* hold. So the FD diagram is as shown in Fig. 12.15.

Again we have two overlapping candidate keys, {S,J} and {S,T}. Once again the relvar is in 3NF and not BCNF, and once again the relvar suffers from certain update anomalies; for example, if we wish to delete the information that Jones is studying physics, we cannot do so without at the same time losing the information that Professor Brown teaches physics. Such difficulties are caused by the fact that attribute T is a determinant but not a candidate key. Again we can get over the problems by replacing the original relvar by two BCNF projections, in this case the projections:

```
ST { S, T }
TJ { T, J }
```

It is left as an exercise to show the values of these two relvars corresponding to the data of Fig. 12.14, to draw a corresponding FD diagram, to prove that the two projections are indeed in BCNF (what are the candidate keys?), and to check that the decomposition does in fact avoid the anomalies.

There is another problem, however. The fact is, although the decomposition into ST and TJ does avoid certain anomalies, it unfortunately introduces others! The trouble is, the two projections are not *independent*, in Rissanen's sense (see Section 12.4). To be specific, the FD

```
{ S, J } → T
```

cannot be deduced from the FD

```
T → J
```

(which is the only FD represented in the two projections). As a result, the two projections cannot be independently updated. For example, an attempt to insert a tuple for Smith and Prof. Brown into relvar ST must be rejected, because Prof. Browu teaches physics and Smith is already being taught physics by Prof. Green; yet the system cannot detect this fact without examining relvar TJ. We are forced to the unpleasant conclusion that the twin objectives of (a) decomposing a relvar into *BCNF* components, and (b) decomposing it into *independent* components, can occasionally be in conflict—that is, it is not always possible to satisfy both of them at the same time.

To elaborate on the example for a moment longer: In fact, relvar SJT is *atomic* in Rissanen's sense (see Section 12.4), even though it is not in BCNF. Observe, therefore, that the fact that an atomic relvar cannot be decomposed into independent components does not mean it cannot be decomposed at all (where by "decomposed" we mean decomposed in a nonloss way). Intuitively speaking, therefore, *atomicity* is not a very good term, since it is neither necessary nor sufficient for good database design.

Our third and final example of overlapping candidate keys concerns a relvar EXAM with attributes S (student), J (subject), and P (position). The meaning of an EXAM tuple (*s,j,p*) is that student *s* was examined in subject *j* and achieved position *p* in the class list. For the purposes of the example, we assume that the following constraint holds:

- There are no ties—that is, no two students obtained the same position in the same subject.

Then the FDs are as shown in Fig. 12.16.

Again we have two overlapping candidate keys, {S,J} and {J,P}, because (a) if we are given a student and a subject, then there is exactly one corresponding position, and equally (b) if we are given a subject and a position, there is exactly one corresponding student. However, the relvar is in BCNF, because those candidate keys are the only determinants, and update anomalies of the kind we have been discussing in this chapter do not occur with this relvar. (*Exercise:* Check this claim.) Thus, overlapping candidate keys do not *necessarily* lead to problems of the kind we have been discussing.

In conclusion, we see that the concept of BCNF eliminates certain additional problem cases that could occur under the old definition of 3NF. Also, BCNF is conceptually simpler than 3NF, in that it makes no overt reference to the concepts of 1NF, 2NF, primary key, or transitive dependence. What is more, the reference it does make to candidate keys could be replaced by a reference to the more fundamental notion of functional dependence (the definition given in reference [12.2] in fact makes this replacement). On the other hand, the concepts of primary key, transitive dependence, and so on are useful in practice, since they give some idea of the actual step-by-step process the designer might have to go through in order to reduce an arbitrary relvar to an equivalent collection of BCNF relvars.

For purposes of future reference, we close this section with a four-step algorithm by which an arbitrary relvar *R* can be nonloss-decomposed into a set *D* of BCNF projections (without necessarily preserving all dependencies, however):

1. Initialize *D* to contain just *R*.
2. For each nonBCNF relvar *T* in *D*, execute Steps 3 and 4.

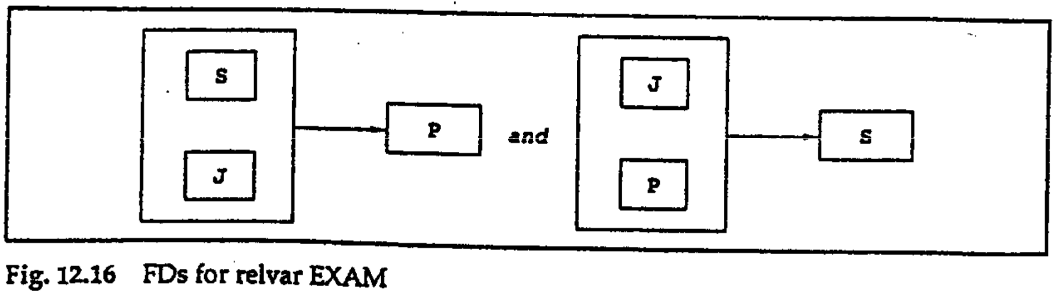


Fig. 12.16 FDs for relvar EXAM

3. Let $X \rightarrow Y$ be an FD for T that violates the requirements for BCNF.
4. Replace T in D by two of its projections: that over X and Y and that over all attributes except those in Y.

12.6 A NOTE ON RELATION-VALUED ATTRIBUTES

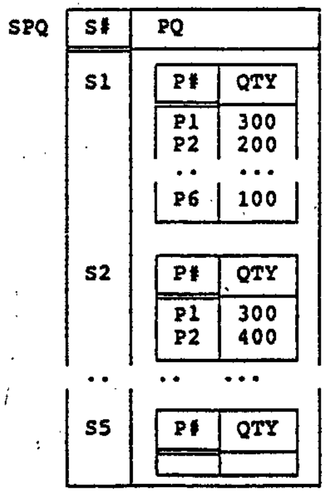
In Chapter 6, we saw that it is possible for a relation to include an attribute whose values are relations in turn (an example is shown in Fig. 12.17). As a result, relvars can have relation-valued attributes too. From the point of view of database design, however, such relvars are usually contraindicated, because they tend to be *asymmetric*[11]—not to mention the fact that their predicates tend to be rather complicated!—and such asymmetry can lead to various practical problems. In the case of Fig. 12.17, for example, suppliers and parts are treated asymmetrically. As a consequence, the (symmetric) queries

1. Get S# for suppliers who supply part P1
2. Get P# for parts supplied by supplier S1

have very different formulations:

```
1. ( SPQ WHERE TUPLE { P# P# ('P1') } ∈ PQ { P# } ) { S# }
2. ( ( SPQ WHERE S# = S# ('S1') ) UNGROUP PQ ) { P# }
```

(SPQ here is assumed to be a relvar whose values are relations of the form indicated by Fig. 12.17.) Note, incidentally, that not only do these two formulations differ considerably, but they are both much more complicated than their SP counterparts.



```
SPQ  S#    PQ
     S1    P#   QTY
           P1   300
           P2   200
           ..   ...
           P6   100
     S2    P#   QTY
           P1   300
           P2   400
     ..    ..   ...
     S5    P#   QTY
```

Fig. 12.17 A relation with a relation-valued attribute

[11] Historically, in fact, such relvars were not even legal—they were said to be *unnormalized*, meaning they were not even regarded as being in 1NF (see Chapter 6).

Matters are even worse for update operations. For example, consider the following two updates:

1.. Create a new shipment for supplier S6, part P5, quantity 500.
2. Create a new shipment for supplier S2, part P5, quantity 500.

With our usual shipments relvar SP, there is no qualitative difference between these two updates—both involve the insertion of a single tuple into the relvar. With relvar SPQ, by contrast, the two updates differ in kind significantly (not to mention the fact that, again, they are both much more complicated than their SP counterpart):

```
1. INSERT SPQ RELATION
       { TUPLE { S# S# ('S6'),
                 PQ RELATION { TUPLE { P#  P# ('P5'),
                                       QTY QTY ( 500 ) } } } } ;
2. UPDATE SPQ WHERE S# = S# ('S2')
       { INSERT PQ RELATION { TUPLE { P#  P# ('P5'),
                                      QTY QTY ( 500 ) } } } ;
```

Relvars—at least, base relvars—without relation-valued attributes are thus usually to be preferred, because the fact that they have a simpler logical structure leads to corresponding simplifications in the operations we need to perform on them. Please understand, however, that this position should be seen as a guideline only, not as an inviolable law. In practice, there might well be cases where a relation-valued attribute does make sense, even for a base relvar. For example, Fig. 12.18 shows (part of) a possible value for

RVK

RVNAME	CK
S	ATTRNAME S#
SP	ATTRNAME S# P#
MARRIAGE	ATTRNAME HUSBAND DATE
MARRIAGE	ATTRNAME DATE WIFE
MARRIAGE	ATTRNAME WIFE HUSBAND

Fig. 12.18 Sample value for catalog relvar RVK

a *catalog* relvar RVK that lists the relvars in the database and their candidate keys. Attribute CK in that relvar is relation-valued. It is also a component of the sole candidate key for RVK! A Tutorial D definition for RVK might thus look something like this:

```
VAR RVK BASE RELATION
      { RVNAME NAME, CK RELATION { ATTRNAME NAME } }
        KEY { RVNAME, CK } ;
```

Note: Exercise 12.3 at the end of the chapter asks you to consider what is involved in eliminating relation-valued attributes, if (as is usually the case) such elimination is considered desirable.[12]

12.7 SUMMARY

This brings us to the end of the first of our two chapters on further normalization. We have discussed the concepts of **first, second, third,** and **Boyce/Codd normal form.** The various normal forms (including those to be discussed in the next chapter) constitute *a total ordering,* in the sense that every relvar at a given level of normalization is automatically at all lower levels also, whereas the converse is not true—there exist relvars at each level that are not at any higher level. Furthermore, BCNF (and indeed 5NF) is always achievable; that is, any given relvar can always be replaced by an equivalent set of relvars in BCNF (or 5NF). And the purpose of such reduction is to **avoid redundancy,** and hence to avoid certain **update anomalies.**

The normalization process consists of replacing the given relvar by certain **projections,** in such a way that **joining** those projections back together again gives us back the original relvar; in other words, the process is **reversible** (equivalently, the decomposition is nonloss). We also saw the crucial role that **functional dependencies** play in the process; in fact, **Heath's theorem** tells us that if a certain FD is satisfied, then a certain decomposition is nonloss. This state of affairs can be seen as further confirmation of the claim made in Chapter 11 to the effect that FDs are "not quite fundamental, but very nearly so."

We also discussed Rissanen's concept of **independent projections,** and suggested that it is better to decompose into such projections rather than into projections that are not independent, when there is a choice. A decomposition into such independent projections is said to be **dependency-preserving.** Unfortunately, we also saw that the objectives of nonloss decomposition to BCNF and dependency preservation can sometimes be in conflict with one another.

[12] And is possible! Note that it is *not* possible in the case of RVK, at least not directly (i.e., without the introduction of some kind of CKNAME—"candidate key name"—attribute).

We conclude this chapter with a very elegant (and fully accurate) pair of definitions, due to Zaniolo [12.7], of the concepts of 3NF and BCNF. First, 3NF:

- **Third normal form** (*Zaniolo's definition*): Let *R* be a relvar, let *X* be any subset of the attributes of *R*, and let *A* be any single attribute of *R*. Then *R* is in 3NF if and only if, for every FD $X \rightarrow A$ in *R*, at least one of the following is true:

1. *X* contains *A* (so the FD is trivial).
2. *X* is a superkey.
3. *A* is contained in a candidate key of *R*.

The definition of **Boyce/Codd normal form** is obtained from the 3NF definition by simply dropping possibility 3 (a fact that shows clearly that BCNF is strictly stronger than 3NF). In fact, possibility 3 is precisely the cause of the "inadequacy" in Codd's original 3NF definition that eventually led to the introduction of BCNF.

:XERCISES

12.1 Prove Heath's theorem. Is the converse of that theorem valid?

12.2 It is sometimes claimed that every binary relvar is necessarily in BCNF. Is this claim valid?

12.3 Fig. 12.19 shows the information to be recorded in a company personnel database, represented as it would be in a *hierarchic* system such as IMS. The figure is to be understood as follows:

- The company has a set of departments.
- Each department has a set of employees, a set of projects, and a set of offices.
- Each employee has a job history (set of jobs the employee has held).
- For each such job, the employee also has a salary history (set of salaries received while employed on that job).
- Each office has a set of phones.

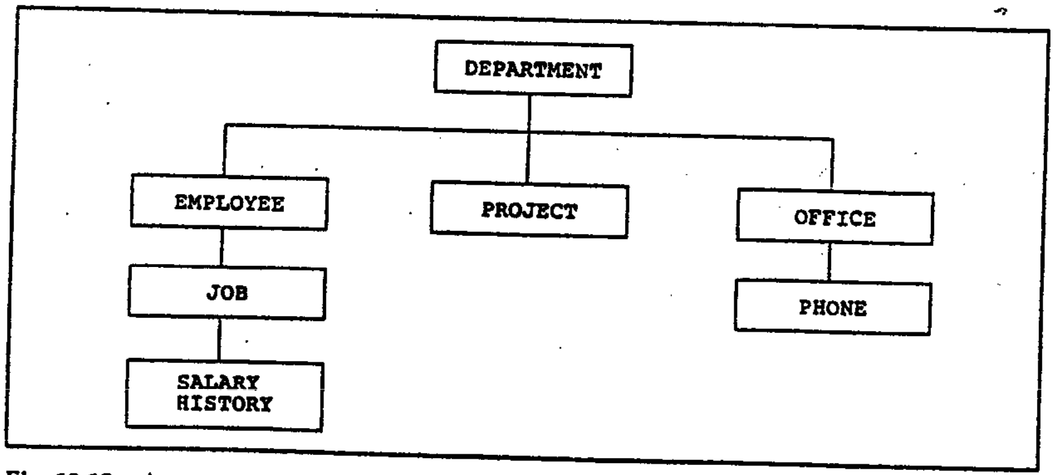


Fig. 12.19 A company database (hierarchic view)

The database is to contain the following information:

- For each department: department number (unique), budget, and the department manager's employee number (unique)
- For each employee: employee number (unique), current project number, office number, and phone number; also, title of each job the employee has held, plus date and salary for each distinct salary received in that job
- For each project: project number (unique) and budget
- For each office: office number (unique), floor area, and phone number (unique) for all phones in that office

Design an appropriate set of relvars to represent this information. State any assumptions you make regarding functional dependencies.

12.4 A database used in an order-entry system is to contain information about customers, items, and orders. The following information is to be included:

- For each customer:
  - Customer number (unique)
  - "Ship-to" addresses (several per customer)
  - Balance
  - Credit limit
  - Discount
- For each order:
  - Heading information:
    Customer number
    Ship-to address
    Date of order
  - Detail lines (several per order):
    Item number
    Quantity ordered
- For each item:
  - Item number (unique)
  - Manufacturing plants
  - Quantity on hand at each plant
  - Stock danger level for each plant
  - Item description

For internal processing reasons, a "quantity outstanding" value is associated with each detail line of each order; this value is initially set equal to the quantity of the item ordered and is progressively reduced to zero as partial shipments are made. Again, design a database for this data. As in the previous exercise, state any assumptions you make regarding dependencies.

12.5 Suppose that in Exercise 12.4 only a very small number of customers, say one percent or less, actually have more than one ship-to address. (This is typical of real-life situations, in which it is frequently the case that just a few exceptions—often rather important ones—fail to conform to some general pattern.) Can you see any drawbacks to your solution to Exercise 12.4? Can you think of any improvements?

12.6 *(Modified version of Exercise 11.13)* Relvar TIMETABLE has the following attributes:

D	Day of the week (1 to 5)
P	Period within day (1 to 6)
C	Classroom number
T	Teacher name
S	Student name
L	Lesson name

Tuple (*d,p,c,t,s,l*) appears in this relvar if and only if at time (*d,p*) student *s* is attending lesson *l*, which is being taught by teacher *t* in classroom *c*. You can assume that lessons are one period in duration and that every lesson has a name that is unique with respect to all lessons taught in the week. Reduce TIMETABLE to a more desirable structure.

12.7 *(Modified version of Exercise 11.14)* Relvar NADDR has attributes NAME (unique), STREET, CITY, STATE, and ZIP. Assume that (a) for any given zip code, there is just one city and state; (b) for any given street, city, and state, there is just one zip code. Is NADDR in BCNF? 3NF? 2NF? Can you think of a better design?

12.8 Let SPQ be a relvar whose values are relations of the form indicated by Fig. 12.17. State the external predicate for SPQ.

REFERENCES AND BIBLIOGRAPHY

In addition to the following, see also the references in Chapter 11, especially Codd's original paper on the first three normal forms [11.6].

12.1 Philip A. Bernstein: "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM TODS 1*, No. 4 (December 1976).

In this chapter we have discussed techniques for decomposing "large" relvars into "smaller" ones (i.e., ones of lower degree). In this paper, Bernstein considers the inverse problem of combining "small" relvars into "larger" ones (i.e., ones of higher degree). The problem is not actually characterized in this way in the paper; rather, it is described as the problem of *synthesizing* relvars given a set of attributes and a set of corresponding FDs, with the constraint that the synthesized relvars must be in 3NF. However, since attributes and FDs have no meaning outside the context of some containing relvar, it would be more accurate to regard the fundamental construct as a binary relvar (plus an FD), rather than as a pair of attributes (plus an FD).

Note: It would equally well be possible to regard the given set of attributes and FDs as defining a universal relvar—see, for example, reference [13.20]—that satisfies a given set of FDs, in which case the "synthesis" process can alternatively be perceived as a process of *decomposing* that universal relvar into 3NF projections. But we stay with the original "synthesis" interpretation for the purposes of the present discussion.

The synthesis process, then, is one of constructing *n*-ary relvars from binary relvars, given a set of FDs that apply to those binary relvars, and given the objective that all constructed relvars be in 3NF (BCNF had not been defined when this work was done). Algorithms are presented for performing this task.

One objection to the approach (recognized by Bernstein) is that the manipulations performed by the synthesis algorithm are purely syntactic in nature and take no account of semantics. For instance, given the FDs

```
A → B (for relvar R{A,B})
B → C (for relvar S{B,C})
A → C (for relvar T{A,C})
```

the third might or might not be redundant (i.e., implied by the first and second), depending on the meanings of *R*, *S*, and *T*. As an example of where it is not so implied, take *A* as employee number, *B* as office number, *C* as department number; take *R* as "office of employee," *S* as "department owning office," *T* as "department of employee"; and consider the case of an employee working in an office belonging to a department not the employee's own. The synthesis algorithm simply assumes that, for example, the two *C* attributes are one and the same (in fact, it does not recognize relvar names at all); it thus relies on the existence of some external mechanism—in other words, human intervention—for avoiding semantically invalid manipulations. In the case at hand, it would be the responsibility of the person defining the original FDs to use distinct attribute names *C1* and *C2* (say) in the two relvars *S* and *T*.

12.2 E. F. Codd: "Recent Investigations into Relational Data Base Systems," Proc. IFIP Congress, Stockholm, Sweden (1974), and elsewhere.

This paper covers a mixed bag of topics. In particular, it gives "an improved definition of third normal form," where "third normal form" in fact refers to what is now known as BCNF. Other topics discussed include *views and view updating*, *data sublanguages*, *data exchange*, and *needed investigations* (all as of 1974).

12.3 C. J. Date: "A Normalization Problem," in *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

To quote the abstract, this paper "examines a simple problem of normalization and uses it to make some observations on the subject of database design and explicit integrity constraint declaration." The problem involves a simple airline application and the following FDs:

```
{ FLIGHT } → DESTINATION
{ FLIGHT } → HOUR
{ DAY, FLIGHT } → GATE
{ DAY, FLIGHT } → PILOT
{ DAY, HOUR, GATE } → DESTINATION
{ DAY, HOUR, GATE } → FLIGHT
{ DAY, HOUR, GATE } → PILOT
{ DAY, HOUR, PILOT } → DESTINATION
{ DAY, HOUR, PILOT } → FLIGHT
{ DAY, HOUR, PILOT } → GATE
```

Among other things, this example serves as a good illustration of the point that the "right" database design can rarely be decided on the basis of normalization principles alone.

12.4 I. J. Heath: "Unacceptable File Operations in a Relational Database," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (November 1971).

This paper gives a definition of "3NF" that was in fact the first published definition of *BCNF*. It also includes a proof of what we referred to in Section 12.2 as *Heath's theorem*. Note that the three steps in the normalization procedure as discussed in the body of this chapter are all applications of that theorem.

12.5 William Kent: "A Simple Guide to Five Normal Forms in Relational Database Theory," *CACM* *26*, No. 2 (February 1983).

The source of the following intuitively attractive characterization of "3NF" (more accurately, BCNF): *Each attribute must represent a fact about the key, the whole key, and nothing but the key* (slightly paraphrased).

12.6 Jorma Rissanen: "Independent Components of Relations," *ACM TODS 2*, No. 4 (December 1977).

12.7 Carlo Zaniolo: "A New Normal Form for the Design of Relational Database Schemata," *ACM TODS 7*, No. 3 (September 1982).

The source of the elegant definitions of 3NF and BCNF discussed in Section 12.7. However, the principal purpose of the paper is to define another normal form, *elementary key normal form* (EKNF), which lies between 3NF and BCNF and "captures the salient qualities of both" while avoiding the problems of both (namely, that 3NF is "too forgiving" and BCNF is "prone to computational complexity"). The paper also shows that Bernstein's algorithm [12.1] in fact generates relvars that are in EKNF, not just 3NF.

CHAPTER 13

Further Normalization II: Higher Normal Forms

13.1 INTRODUCTION

In the previous chapter we discussed the ideas of further normalization up to and including Boyce/Codd normal form, BCNF (which is as far as the functional dependency concept can carry us). Now we complete our discussions by examining *fourth* and *fifth* normal forms (4NF and 5NF). As we will see, the definition of fourth normal form makes use of a new kind of dependency, called a *multi-valued* dependency (MVD); MVDs are a generalization of FDs. Likewise, the definition of fifth normal form makes use of another new kind of dependency, called a *join* dependency (JD); JDs in turn are a generalization of MVDs. Section 13.2 discusses MVDs and 4NF, Section 13.3 discusses JDs and 5NF (and explains why 5NF is, in a certain special sense, "the final normal form"). Note, however, that our discussions of MVDs and JDs are deliberately less formal and complete than our discussions of FDs in Chapter 11—we leave the formal treatment to the research papers (see the "References and Bibliography" section).

Section 13.4 then reviews the entire normalization procedure and makes some additional comments on it. Next, Section 13.5 briefly discusses the notion of *de*normalization. Section 13.6 then describes another important, and related, design principle called *orthogonal design*. Finally, Section 13.7 briefly examines some recent developments and possible directions for future research in the normalization field, and Section 13.8 presents a summary.

13.2 MULTI-VALUED DEPENDENCIES AND FOURTH NORMAL FORM

Suppose we are given a relvar HCTX—H for hierarchic—containing information about courses, teachers, and texts, in which the attributes corresponding to teachers and texts are *relation-valued* (see Fig. 13.1 for a sample HCTX value). As you can see, each HCTX tuple consists of a course name, plus a relation containing teacher names, plus a relation containing text names (two such tuples are shown in the figure). The intended meaning of such a tuple is that the specified course can be taught by any of the specified teachers and uses all of the specified texts as references. We assume that, for a given course *c*, there can be any number *m* of corresponding teachers and any number *n* of corresponding texts ($m > 0$, $n > 0$). Moreover, we also assume—perhaps not very realistically!—that teachers and texts are quite independent of one another; that is, no matter who actually teaches any particular offering of a given course, the same texts are used. Finally, we also assume that a given teacher or a given text can be associated with any number of courses.

Now suppose that (as in Chapter 12, Section 12.6) we want to eliminate the relation-valued attributes. One way to do this—probably not the best way, though, a point we will come back to at the end of this section—is simply to replace relvar HCTX by a relvar CTX with three *scalar* attributes COURSE, TEACHER, and TEXT, as indicated in Fig. 13.2. As you can see from the figure, each HCTX tuple gives rise to $m * n$ CTX tuples, where *m* and *n* are the cardinalities of the TEACHERS and TEXTS relations in that HCTX tuple. Note that the resulting relvar CTX is "all key" (the sole candidate key for

HCTX

COURSE	TEACHERS	TEXTS
Physics	TEACHER Prof. Green Prof. Brown	TEXT *Basic Mechanics* *Principles of Optics*
Math	TEACHER Prof. Green	TEXT *Basic Mechanics* *Vector Analysis* *Trigonometry*

Fig. 13.1 Sample value for relvar HCTX

CTX

COURSE	TEACHER	TEXT
Physics	Prof. Green	*Basic Mechanics*
Physics	Prof. Green	*Principles of Optics*
Physics	Prof. Brown	*Basic Mechanics*
Physics	Prof. Brown	*Principles of Optics*
Math	Prof. Green	*Basic Mechanics*
Math	Prof. Green	*Vector Analysis*
Math	Prof. Green	*Trigonometry*

Fig. 13.2 Value for relvar CTX corresponding to the HCTX value in Fig. 13.1

HCTX, by contrast, was just {COURSE}). *Exercise:* Give a relational expression by which CTX can be derived from HCTX.

The meaning of relvar CTX is basically as follows: A tuple (*c*,*t*,*x*)—simplified notation—appears in CTX if and only if course *c* can be taught by teacher *t* and uses text *x* as a reference. Observe that, for a given course, all possible combinations of teacher and text appear; that is, CTX satisfies the (relvar) constraint

if tuples (*c*,*t1*,*x1*) and (*c*,*t2*,*x2*) both appear
then tuples (*c*,*t1*,*x2*) and (*c*,*t2*,*x1*) both appear also

Now, it should be apparent that relvar CTX involves a good deal of *redundancy*, leading as usual to certain *update anomalies*. For example, to add the information that the physics course can be taught by a new teacher, it is necessary to insert two separate tuples, one for each of the two texts. Can we avoid such problems? Well, it is easy to see that the problems in question are caused by the fact that teachers and texts are *completely independent of one another*. It is also easy to see that matters would be improved if CTX were decomposed into its two projections—let us call them CT and CX—on {COURSE, TEACHER} and {COURSE,TEXT}, respectively (see Fig. 13.3).

To add the information that the physics course can be taught by a new teacher, all we have to do given the design of Fig. 13.3 is insert a single tuple into relvar CT. (Note too that relvar CTX can be recovered by joining CT and CX back together again, so the decomposition is nonloss.) Thus, it does seem reasonable to suggest that there should be a way of further normalizing a relvar like CTX.

CT

COURSE	TEACHER
Physics	Prof. Green
Physics	Prof. Brown
Math	Prof. Green

CX

COURSE	TEXT
Physics	*Basic Mechanics*
Physics	*Principles of Optics*
Math	*Basic Mechanics*
Math	*Vector Analysis*
Math	*Trigonometry*

Fig. 13.3 Values for relvars CT and CX corresponding to the CTX value in Fig. 13.2

Now, you might be thinking that the redundancy in CTX was unnecessary in the first place, and hence that the corresponding update anomalies were unnecessary too. More specifically, you might suggest that CTX need not include all possible teacher/text combinations for a given course; for example, two tuples are obviously sufficient to show that the physics course has two teachers and two texts. The problem is, *which* two tuples? Any particular choice leads to a relvar having a very unobvious interpretation and very strange update behavior. (Try stating the predicate for such a relvar!—i.e., try stating the criteria for deciding whether or not some given update is acceptable on that relvar.)

Informally, therefore, it is obvious that the design of CTX is bad and the decomposition into CT and CX is better. The trouble is, however, these facts are not *formally* obvious. Observe in particular that CTX satisfies no functional dependencies at all (apart from trivial ones such as COURSE → COURSE); in fact, CTX is in BCNF,[1] since as already noted it is all key—any "all key" relvar must necessarily be in BCNF. (Note, incidentally, that the two projections CT and CX are also all key and hence in BCNF.) The ideas of the previous chapter are therefore of no help with the problem at hand.

The existence of "problem" BCNF relvars like CTX was recognized very early on, and the way to deal with them was also understood, at least intuitively. However, it was not until 1977 that these intuitive ideas were put on a sound theoretical footing by Fagin's introduction of the notion of *multi-valued dependencies*, MVDs [13,14]. Multi-valued dependencies are a generalization of functional dependencies—meaning every FD is an MVD, but the converse is not true (i.e., there exist MVDs that are not FDs). In the case of relvar CTX, two MVDs hold:

```
COURSE →→ TEACHER
COURSE →→ TEXT
```

(note the double arrows; *A* →→ *B* is read as "*B* is **multi-dependent** on *A*," or, equivalently, "*A* **multi-determines** *B*"). Let us concentrate on the MVD COURSE →→ TEACHER. Intuitively, what this MVD means is that, although a course does not have a *unique* corresponding teacher (i.e., the *functional* dependence COURSE → TEACHER does not hold), each course nevertheless does have a well-defined set of corresponding teachers. By "well-defined" here we mean, more precisely, that for a given course *c* and a given text *x*, the set of teachers *t* matching the pair (*c*,*x*) in CTX depends on the value *c* alone—it makes no difference which particular value of *x* we choose. The MVD COURSE →→ TEXT is interpreted analogously.

Here then is the formal definition:

- **Multi-valued dependence:** Let *R* be a relvar, and let *A*, *B*, and *C* be subsets of the attributes of *R*. Then we say that *B* is **multi-dependent** on *A*—in symbols,

  ```
  A →→ B
  ```

  (read "*A* multi-determines *B*," or simply "*A* double arrow *B*")—if and only if, in every legal value of *R*, the set of *B* values matching a given *AC* value pair depends only on the *A* value and is independent of the *C* value.

[1] HCTX is in BCNF, too; as a matter of fact, it is also in 4NF and 5NF (see the definitions later in this chapter).

It is easy to show (see Fagin [13.14]) that, given the relvar $R\{A,B,C\}$, the MVD $A \twoheadrightarrow B$ holds if and only if the MVD $A \twoheadrightarrow C$ also holds. MVDs always go together in pairs in this way. For this reason it is usual to represent them both in one statement, thus:

```
A →→ B | C
```

For example:

```
COURSE →→ TEACHER | TEXT
```

Now, we stated earlier that multi-valued dependencies are a generalization of functional dependencies, in the sense that every FD is an MVD. More precisely, an FD is an MVD in which the set of dependent values matching a given determinant value is always a singleton set. Thus, if $A \rightarrow B$, then certainly $A \twoheadrightarrow B$.

Returning to our original CTX problem, we can now see that the trouble with relvars such as CTX is that they involve MVDs that are not FDs. (In case it is not obvious, we point out that it is precisely the existence of those MVDs that leads to, e.g., the need to insert two tuples to add one new physics teacher; both tuples are needed in order to maintain the integrity constraint that is represented by the MVD.) The two projections CT and CX do not involve any such MVDs, which is why they represent an improvement over the original design. We would therefore like to replace CTX by those two projections, and a theorem proved by Fagin in reference [13.14] allows us to make exactly that replacement:

- **Theorem** (Fagin): Let $R\{A,B,C\}$ be a relvar, where A, B, and C are sets of attributes. Then R is equal to the join of its projections on $\{A,B\}$ and $\{A,C\}$ if and only if R satisfies the MVDs $A \twoheadrightarrow B \mid C$.

(Notice that this is a stronger version of Heath's theorem [12.4] as stated in Chapter 12.) Following Fagin [13.14], we can now define *fourth normal form* (so called because—as noted in Chapter 12—BCNF was still being called *third* normal form at the time):

- **Fourth normal form:** Relvar R is in 4NF if and only if, whenever there exist subsets A and B of the attributes of R such that the nontrivial MVD $A \twoheadrightarrow B$ is satisfied, then all attributes of R are also *functionally* dependent on A. *Note:* The MVD $A \twoheadrightarrow B$ is trivial if either A is a superset of B or the union AB of A and B is the entire heading.

In other words, the only nontrivial dependencies (FDs or MVDs) in R are of the form $K \rightarrow X$ (i.e., a *functional* dependency from some superkey K to some other attribute X). Equivalently: R is in 4NF if and only if it is in BCNF and all MVDs in R are in fact "FDs out of keys." Note in particular, therefore, that 4NF implies BCNF.

Relvar CTX is not in 4NF, since it involves an MVD that is not an FD at all, let alone an FD "out of a key." The two projections CT and CX are both in 4NF, however. Thus, 4NF is an improvement over BCNF, in that it eliminates another form of undesirable dependency. What is more, Fagin shows in reference [13.14] that 4NF is always achievable; that is, any relvar can be nonloss-decomposed into an equivalent collection of 4NF relvars—though our discussion of the SJT example in Chapter 12, Section 12.5, shows that in some cases it might not be desirable to carry the decomposition that far (or even as far as BCNF).

As an aside, we remark that Rissanen's work on independent projections [12.6], though couched in terms of FDs, is applicable to MVDs also. Recall that a relvar $R\{A,B,C\}$ satisfying the FDs $A \rightarrow B$ and $B \rightarrow C$ is better decomposed into its projections on $\{A,B\}$ and $\{B,C\}$ rather than into those on $\{A,B\}$ and $\{A,C\}$. The same holds true if we replace the FDs $A \rightarrow B$ and $B \rightarrow C$ by the MVDs $A \rightarrow\!\!\!\rightarrow B$ and $B \rightarrow\!\!\!\rightarrow C$, respectively.

We conclude this section by returning, as promised, to the question of eliminating relation-valued attributes (RVAs for short). The point is this: If we start with a relvar like HCTX that involves two or more independent RVAs, then, instead of simply replacing those RVAs by scalar attributes (as we did earlier in this section) and then performing nonloss decomposition on the result, *it is better to separate the RVAs first.* In the case of HCTX, for example, it is better to replace the relvar by its two projections HCT {COURSE,TEACHERS} and HCX {COURSE,TEXTS} (where TEACHERS and TEXTS are still RVAs). The RVAs in those two projections can then be replaced by scalar attributes and the results reduced to BCNF (actually 4NF) in the usual way if necessary, and the "problem" BCNF relvar CTX will simply never arise. But the theory of MVDs and 4NF gives us a formal basis for what would otherwise be a mere rule of thumb.

13.3 JOIN DEPENDENCIES AND FIFTH NORMAL FORM

So far in this chapter (and throughout the previous chapter) we have tacitly assumed that the sole operation necessary or available in the normalization process is the replacement of a relvar in a nonloss way by *exactly two* of its projections. This assumption has successfully carried us as far as 4NF. It comes perhaps as a surprise, therefore, to discover that there exist relvars that cannot be nonloss-decomposed into two projections but *can* be nonloss-decomposed into three or more. To coin an ugly but convenient term, we will say a relvar (or a relation) is "n-decomposable" if it can be nonloss-decomposed into n projections but not into m, where $1 < m$ and $m < n$. *Note:* The phenomenon of n-decomposability for $n > 2$ was first noted by Aho, Beeri, and Ullman [13.1]. The particular case $n = 3$ was also studied by Nicolas [13.26].

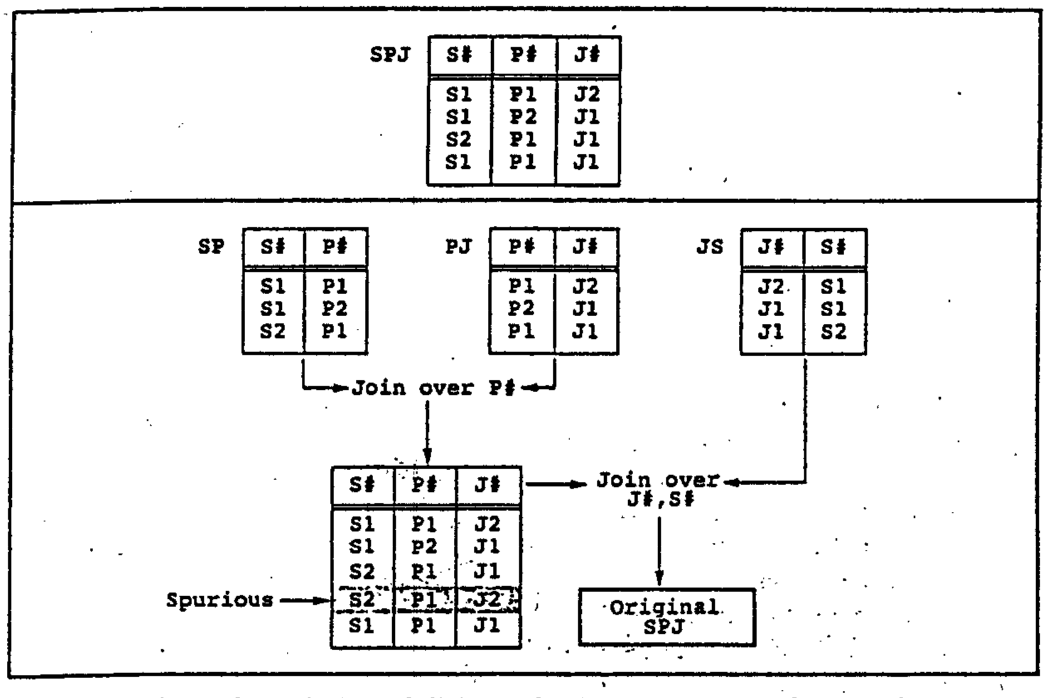
Consider relvar SPJ from the suppliers-parts-projects database (but ignore attribute QTY for simplicity); a sample value is shown at the top of Fig. 13.4. That relvar is all key and involves no nontrivial FDs or MVDs at all, and is therefore in 4NF. Note too that Fig. 13.4 also shows:

a. The three binary projections SP, PJ, and JS corresponding to the SPJ relation value shown at the top of the figure

b. The effect of joining the SP and PJ projections (over P#)

c. The effect of joining that result and the JS projection (over J# and S#)

Observe that the result of the first join is to produce a copy of the original SPJ relation plus one additional ("spurious") tuple, and the effect of the second join is then to eliminate that additional tuple, thereby getting us back to the original SPJ relation. In other words, that original SPJ relation is 3-decomposable. *Note:* The net result is the same whatever pair of projections we choose for the first join, though the intermediate result is different in each case. *Exercise:* Check this claim.



SPJ

S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

SP

S#	P#
S1	P1
S1	P2
S2	P1

PJ

P#	J#
P1	J2
P2	J1
P1	J1

JS

J#	S#
J2	S1
J1	S1
J1	S2

S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1
S2	P1	J2
S1	P1	J1

Fig. 13.4 Relation SPJ is the join of all three of its binary projections but not of any two

Now, the example of Fig. 13.4 is expressed in terms of relations, not relvars. However, the 3-decomposability of SPJ could be a more fundamental, time-independent property—that is, a property satisfied by all legal values of the relvar—*if* the relvar satisfies a certain time-independent integrity constraint. To understand what that constraint must be, observe first that the statement "SPJ is equal to the join of its three projections SP, PJ, and JS" is precisely equivalent to the following statement:

if the pair	(*s1,p1*)	appears in SP
and the pair	(*p1,j1*)	appears in PJ
and the pair	(*j1,s1*)	appears in JS
then the triple	(*s1,p1,j1*)	appears in SPJ

because the triple (*s1,p1,j1*) obviously appears in the join of SP, PJ, and JS. (*Note:* The converse of this statement—that if (*s1,p1,j1*) appears in SPJ then (*s1,p1*) appears in projection SP, etc.—is clearly true for any degree-3 relation SPJ.) Since (*s1,p1*) appears in SP if and only if (*s1,p1,j2*) appears in SPJ for some *j2*, and similarly for (*p1,j1*) and (*j1,s1*), we can rewrite the foregoing statement as a constraint on SPJ:

if	(*s1,p1,j2*), (*s2,p1,j1*), and (*s1,p2,j1*) appear in SPJ
then	(*s1,p1,j1*) appears in SPJ also

And if *this* statement is true for all time—that is, for all possible values of relvar SPJ—then we do have a time-independent constraint on the relvar (albeit a rather bizarre one). Notice the cyclic nature of that constraint ("if *s1* is linked to *p1* and *p1* is linked to *j1* and *j1* is linked back to *s1* again, then *s1* and *p1* and *j1* must all coexist in the same tuple"). *A relvar will be n-decomposable for some n > 2 if and only if it satisfies some such (n-way) cyclic constraint.*

Suppose then for the remainder of this section that relvar SPJ does in fact satisfy that time-independent constraint (the sample values in Fig. 13.4 are consistent with this hypothesis). For brevity, let us agree to refer to that constraint as *Constraint 3D* (3D for 3-decomposable). What does Constraint 3D mean in real-world terms? Let us try to make it a little more concrete by giving an example. The constraint says that, in the portion of the real world that relvar SPJ is supposed to represent, it is a fact that *if* (for example)

a. Smith supplies monkey wrenches, and

b. Monkey wrenches are used in the Manhattan project, and

c. The Manhattan project is supplied by Smith,

then

d. Smith supplies monkey wrenches to the Manhattan project.

Note that (as pointed out in Chapter 1, Section 1.3), *a, b,* and *c* together normally do *not* imply *d:* indeed, exactly this example was held up in Chapter 1 as an illustration of "the connection trap." In the case at hand, however, we are saying *there is no trap*—because there is an additional real-world constraint in effect (Constraint 3D) that makes the inference of *d* from *a, b,* and *c* valid in this particular case.

To return to the main topic of discussion: Because Constraint 3D is satisfied if and only if the relvar concerned is equal to the join of certain of its projections, we refer to that constraint as a *join dependency* (JD). A JD is a constraint on the relvar concerned, just as an MVD or an FD is a constraint on the relvar concerned. Here is the definition:

- **Join dependency:** Let *R* be a relvar, and let *A, B, ..., Z* be subsets of the attributes of *R*. Then we say that *R* satisfies the JD

  `* { A, B, ..., Z }`

  (pronounced "star *A, B, ..., Z*") if and only if every legal value of *R* is equal to the join of its projections on *A, B, ..., Z*.

For example, if we agree to use SP to mean the subset {S#,P#} of the set of attributes of SPJ, and similarly for PJ and JS, then we can say that—given Constraint 3D—relvar SPJ satisfies the JD * {SP,PJ,JS}.

By way of another example, consider the usual suppliers relvar S. If we agree to use SN to refer to the subset {S#,SNAME} of the set of attributes of S, and similarly for ST and SC, then we can say that relvar S satisfies the join dependency * {SN,ST,SC}.

We have seen, then, that relvar SPJ, with its JD * {SP,PJ,JS}, can be 3-decomposed. The question is, *should* it be? And the answer is "Probably yes." Relvar SPJ (with its JD) suffers from a number of update anomalies, anomalies that disappear when it is 3-decomposed. Examples of such anomalies are illustrated in Fig. 13.5. Consideration of what happens after 3-decomposition is left as an exercise.

SPJ

S#	P#	J#
S1	P1	J2
S1	P2	J1

- If (S2,P1,J1) is inserted, (S1,P1,J1) must also be inserted.
- Yet converse is not true.

SPJ

S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

- Can delete (S2,P1,J1) without side effects.
- If (S1,P1,J1) is deleted, another tuple must also be deleted (which?).

Fig. 13.5 Sample update anomalies in SPJ

Fagin's theorem, discussed in Section 13.2, to the effect that $R\{A,B,C\}$ can be nonloss-decomposed into its projections on $\{A,B\}$ and $\{A,C\}$ if and only if the MVDs $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ hold in R, can now be restated as follows:

- $R\{A,B,C\}$ satisfies the JD $*\ \{AB,AC\}$ if and only if it satisfies the MVDs $A \twoheadrightarrow B \mid C$.

Since this theorem can be taken as a *definition* of multi-valued dependency, it follows that an MVD is just a special case of a JD, or (equivalently) that JDs are a generalization of MVDs. Formally, we have:

$A \twoheadrightarrow B \mid C \equiv *\ \{ AB,\ AC \}$

Note: It is immediate from the definition of join dependency that JDs are the most general form of dependency possible (using, of course, the term *dependency* in a very special sense!). That is, there does not exist a still higher form of dependency such that JDs are merely a special case of that higher form—as long as we restrict our attention to dependencies that deal with a relvar being decomposed via projection and recomposed via join. (However, if we permit other decomposition and recomposition operators, then other types of dependencies might come into play. We consider this possibility briefly in Section 13.7.)

Returning now to our example, we can see that the problem with relvar SPJ is that it involves a JD that is not an MVD, and hence not an FD either. (*Exercise: Why* is this a problem, exactly?) We have also seen that it is possible, and probably desirable, to decompose such a relvar into smaller components: namely, into the projections specified by the join dependency. That decomposition process can be repeated until all resulting relvars are in *fifth normal form*, which we now define:

- **Fifth normal form: A relvar R is in 5NF—also called projection-join normal form (PJ/NF)—if and only if every nontrivial join dependency that is satisfied by R is implied by the candidate key(s) of R, where:**
  a. The join dependency $*\ \{ A, B, \ldots, Z \}$ on R is trivial if and only if at least one of A, B, ..., Z is the set of all attributes of R.

b. The join dependency * { *A*, *B*, ..., *Z* } on *R* **is implied by the candidate key(s)** of *R* if and only if each of *A*, *B*, ..., *Z* is a superkey for *R*.

Relvar SPJ is not in 5NF; it satisfies a certain join dependency, Constraint 3D, that is certainly not implied by its sole candidate key (that key being the combination of all of its attributes). To state this differently, relvar SPJ is not in 5NF, because (a) it *can* be 3-decomposed and (b) that 3-decomposability is not implied by the fact that the combination {S#,P#,J#} is a candidate key. By contrast, after 3-decomposition, the three projections SP, PJ, and JS are each in 5NF, since they do not involve any nontrivial JDs at all.

Note that any relvar in 5NF is automatically in 4NF also, because as we have seen an MVD is a special case of a JD. What is more, Fagin shows in reference [13.15] that any MVD that is implied by a candidate key must in fact be an FD in which that candidate key is the determinant. Fagin also shows in that same paper that any given relvar can be nonloss-decomposed into an equivalent collection of 5NF relvars; that is, 5NF is always achievable.

Let us take a closer look at the question of what it means for a JD to be implied by candidate keys. Consider our familiar suppliers relvar S once again. That relvar satisfies several join dependencies—this one, for example:

```
* { { S#, SNAME, STATUS }, { S#, CITY } }
```

That is, relvar S is equal to the join of its projections on {S#,SNAME,STATUS} and {S#,CITY}, and hence can be nonloss-decomposed into those projections. (This fact does not mean it *should* be so decomposed, only that it *could* be.) This JD is implied by the fact that {S#} is a candidate key; in fact, it is implied by Heath's theorem [12.4].

Suppose now (as we did in Chapter 12, Section 12.5) that relvar S has a second candidate key, {SNAME}. Then here is another JD that is satisfied by that relvar:

```
* { { S#, SNAME }, { S#, STATUS }, { SNAME, CITY } }
```

This JD is implied by the fact that {S#} and {SNAME} are *both* candidate keys.

Both of these examples illustrate the fact that the JD * {*A*, *B*, ..., *Z*} is implied by candidate keys if and only if each of *A*, *B*, ..., *Z* is a superkey for the relvar in question. Thus, given a relvar *R*, we can tell if *R* is in 5NF as long as we know all candidate keys *and all JDs* in *R*. However, discovering all of those JDs might itself be a nontrivial exercise. That is, whereas it is relatively easy to identify FDs and MVDs (because they have a fairly straightforward real-world interpretation), the same cannot be said for JDs—JDs, that is, that are not MVDs and therefore not FDs—because the intuitive meaning of JDs might not be obvious. Hence, the process of determining when a given relvar is in 4NF but not 5NF, and so could probably be decomposed to advantage, is still somewhat unclear. Experience suggests that such relvars are likely to be rare in practice.

In conclusion, we note that it follows from the definition that 5NF is the **ultimate normal form** with respect to projection and join (which accounts for its alternative name, *projection-join* normal form). That is, a relvar in 5NF is **guaranteed to be free of anomalies** that can be removed by taking projections. (Of course, this remark does not mean it is free of anomalies; it just means, to repeat, that it is free of anomalies that can be removed by taking projections.) For if a relvar is in 5NF, the only join dependencies are those that are implied by candidate keys, and so the only valid decompositions are ones that are based on those candidate keys (each projection in such a decomposition will consist of one or more of those candidate keys, plus zero or more additional attributes). For example, the suppliers relvar S is in 5NF. It *can* be further decomposed in several nonloss ways, as we saw earlier, but every projection in any such decomposition will still include a candidate key of the original relvar, and hence there does not seem to be any particular advantage in that further decomposition.

13.4 THE NORMALIZATION PROCEDURE SUMMARIZED

Up to this point in this chapter (and throughout the previous chapter), we have been concerned with the technique of *nonloss decomposition* as an aid to database design. The basic idea is as follows: Given some 1NF relvar *R* and some set of FDs, MVDs, and JDs that apply to *R*, we systematically reduce *R* to a collection of "smaller" (i.e., lower-degree) relvars that are equivalent to *R* in a certain well-defined sense but are also in some way more desirable.[2] Each step of the reduction process consists of taking projections of the relvars resulting from the preceding step. The given constraints are used at each step to guide the choice of which projections to take next. The overall process can be stated informally as a set of rules, thus:

1. Take projections of the original 1NF relvar to eliminate FDs that are not irreducible. This step will produce a collection of 2NF relvars.
2. Take projections of those 2NF relvars to eliminate transitive FDs. This step will produce a collection of 3NF relvars.
3. Take projections of those 3NF relvars to eliminate remaining FDs in which the determinant is not a candidate key. This step will produce a collection of BCNF relvars. *Note:* Rules 1–3 can be condensed into the single guideline "Take projections of the original relvar to eliminate FDs in which the determinant is not a candidate key."
4. Take projections of those BCNF relvars to eliminate MVDs that are not also FDs. This step will produce a collection of 4NF relvars. *Note:* In practice it is usual—by "separating independent RVAs," as explained in Section 13.2—to eliminate such MVDs before applying Rules 1–3 above.
5. Take projections of those 4NF relvars to eliminate JDs that are not implied by the candidate keys—though perhaps we should add "if you can find them." This step will produce a collection of relvars in 5NF.

Several points arise from the foregoing summary:

1. First of all, the process of taking projections at each step must be done in a nonloss way, and preferably in a dependency-preserving way as well.

[2] If *R* includes any RVAs (which it might), we assume it does so intentionally. RVAs that are not wanted can be eliminated as explained in Section 13.2.

2. Observe that (as was first noted by Fagin in reference [13.15]) there is a very attractive parallelism among the definitions of BCNF, 4NF, and 5NF:
   - A relvar *R* is in BCNF if and only if every FD satisfied by *R* is implied by the candidate keys of *R*.
   - A relvar *R* is in 4NF if and only if every MVD satisfied by *R* is implied by the candidate keys of *R*.
   - A relvar *R* is in 5NF if and only if every JD satisfied by *R* is implied by the candidate keys of *R*.

   The update anomalies discussed in Chapter 12 and in earlier sections of the present chapter are precisely anomalies that are caused by FDs or MVDs or JDs that are not implied by candidate keys. (The FDs, MVDs, and JDs we are referring to here are all assumed to be nontrivial ones.)
3. The overall objectives of the normalization process are as follows:
   - To eliminate certain kinds of redundancy
   - To avoid certain update anomalies
   - To produce a design that is a "good" representation of the real world—one that is intuitively easy to understand and a good base for future growth
   - To simplify the enforcement of certain integrity constraints

   We elaborate a little on the last item in this list. The general point is that (as we already know from discussions elsewhere in this book) *some integrity constraints imply others*. As a trivial example, the constraint that salaries must be greater than $10,000 certainly implies the constraint that they must be greater than zero. Now, if constraint *A* implies constraint *B*, then enforcing *A* will enforce *B* automatically (it will not even be necessary to state *B* explicitly, except perhaps in the form of a comment). And normalization to 5NF gives us a simple way of enforcing certain important and commonly occurring constraints; basically, all we need do is enforce uniqueness of candidate keys, and then all JDs (and all MVDs and all FDs) will be enforced automatically—because, of course, all of those JDs (and MVDs and FDs) will be implied by those candidate keys.
4. Once again, we stress the point that the normalization guidelines are only guidelines, and occasionally there might be good reasons for not normalizing "all the way." The classic example of a case where complete normalization *might* not be a good idea is provided by the name and address relvar NADDR (see Exercise 12.7 in Chapter 12)—though, to be frank, that example is not very convincing. As a general rule, not normalizing all the way is usually a bad idea.
5. We also repeat the point from Chapter 12 that the notions of dependency and further normalization are semantic in nature; in other words, they are concerned with what the data means. By contrast, the relational algebra and relational calculus, and languages such as SQL that are based on such formalisms, are concerned only with actual data values; they do not and cannot require any particular level of normalization other than first. The further normalization guidelines should be regarded primarily as a discipline to help the database designer (and hence the user)—a discipline by

which the designer can capture a part, albeit a small part, of the semantics of the real world in a simple and straightforward manner.

6. Following on from the previous point: The ideas of normalization are useful in database design, but they are not a panacea. Here are some of the reasons why not (this list is amplified in reference [13.9]):
   - It is true that normalization can help to enforce certain integrity constraints very simply, but (as we know from Chapter 9) JDs, MVDs, and FDs are not the only kinds of constraints that can arise in practice.
   - The decomposition might not be unique (usually, in fact, there will be many ways of reducing a given collection of relvars to 5NF), and there are few objective criteria by which to choose among alternative decompositions.
   - The BCNF and dependency preservation objectives can be in conflict, as explained in Section 12.5 ("the SJT problem").
   - The normalization procedure eliminates redundancies by taking projections, but not all redundancies can be eliminated in this manner ("the CTXD problem"—see the annotation to reference [13.14]).

We should also mention the fact that good top-down design methodologies tend to generate fully normalized designs anyway (see Chapter 14).

13.5 A NOTE ON DENORMALIZATION

Up to this point in this chapter (and throughout the previous chapter), we have generally assumed that full normalization all the way to 5NF is desirable. In practice, however, it is often claimed that "denormalization" is necessary to achieve good performance. The argument goes something like this:

1. Full normalization means lots of logically separate relvars (and we assume here that the relvars in question are base relvars specifically).
2. Lots of logically separate relvars means lots of physically separate stored files.
3. Lots of physically separate stored files means lots of I/O.

Strictly speaking, this argument is invalid, because (as explained elsewhere in this book) the relational model nowhere stipulates that base relvars must map one for one to stored files. *Denormalization, if necessary, should be done at the level of stored files, not at the level of base relvars.*[3] But the argument is valid, somewhat, for today's SQL products, precisely because of the inadequate degree of separation between those two levels found in those products. In this section, therefore, we take a closer look at the notion of "denormalization." *Note:* The discussion that follows is based on material from reference [13.6].

[3] This remark is not really accurate: denormalization is an operation on relvars, not stored files, and so it cannot be applied "at the level of stored files." But it is not unreasonable to assume that some analog of denormalization can be carried out at the level of stored files.

What *Is* Denormalization?

To review briefly, normalizing a relvar *R* means replacing *R* by a set of projections *R1*, *R2*, ..., *Rn*, such that *R* is equal to the join of *R1*, *R2*, ..., *Rn*; the objective is to *reduce redundancy*, by making sure that each of the projections *R1*, *R2*, ..., *Rn* is at the highest possible level of normalization.

In order to define denormalization, then, let *R1*, *R2*, ..., *Rn* be a set of relvars. Then denormalizing those relvars means replacing them by their join *R*, such that for all *i* (*i* = 1, 2, ..., *n*) projecting *R* over the attributes of *Ri* is guaranteed to yield *Ri* again. The objective is to *increase redundancy*, by ensuring that *R* is at a lower level of normalization than the relvars *R1*, *R2*, ..., *Rn*. More specifically, the objective is to reduce the number of joins that need to be done at run time by (in effect) doing some of those joins ahead of time, as part of the database design.

By way of an example, we might consider denormalizing parts and shipments to produce a relvar PSQ as indicated in Fig. 13.6.[4] Observe that relvar PSQ is in 1NF and not in 2NF.

Some Problems

The concept of denormalization suffers from a number of well-known problems. One obvious one is that once we start denormalizing, it is not clear where we should stop. With normalization, there are clear logical reasons for continuing until we reach the highest possible normal form; do we then conclude that with denormalization we should proceed until we reach the *lowest* possible normal form? Surely not; yet there are no established *logical* criteria for deciding exactly where to stop. In choosing to denormalize, in other words, we are backing off from a position that does at least have some solid science and theory behind it, and replacing it by one that is purely pragmatic in nature, and subjective.

The second obvious point is that there are redundancy and update problems, precisely because we are dealing once again with relvars that are less than fully normalized. We have already discussed these problems at length. What is less obvious, however, is that there can be retrieval problems too; that is, denormalization can actually make certain queries harder

PSQ

P#	PNAME	COLOR	WEIGHT	CITY	S#	QTY
P1	Nut	Red	12.0	London	S1	300
P1	Nut	Red	12.0	London	S2	300
P2	Bolt	Green	17.0	Paris	S1	200
..	.....	.....	......	......	..	...
P6	Cog	Red	19.0	London	S1	100

Fig. 13.6 Denormalizing parts and shipments

[4] There is a problem with denormalizing *suppliers* and shipments, given our usual sample data, because supplier S5 is lost in the join. For such reasons, some people might argue that we should use "outer" joins in the denormalization process. But outer joins have problems of their own, as we will see in Chapter 19.

to express. For example, consider the query "For each part color, get the average weight." Given our usual normalized design, a suitable formulation is:

```
SUMMARIZE P BY { COLOR } ADD AVG ( WEIGHT ) AS AVWT
```

Given the denormalized design of Fig. 13.6, however, the formulation is a little trickier (not to mention the fact that it relies on the strong and generally invalid assumption that every part does have at least one shipment):

```
SUMMARIZE PSQ { P#, COLOR, WEIGHT } BY { COLOR }
                                ADD AVG ( WEIGHT ) AS AVWT
```

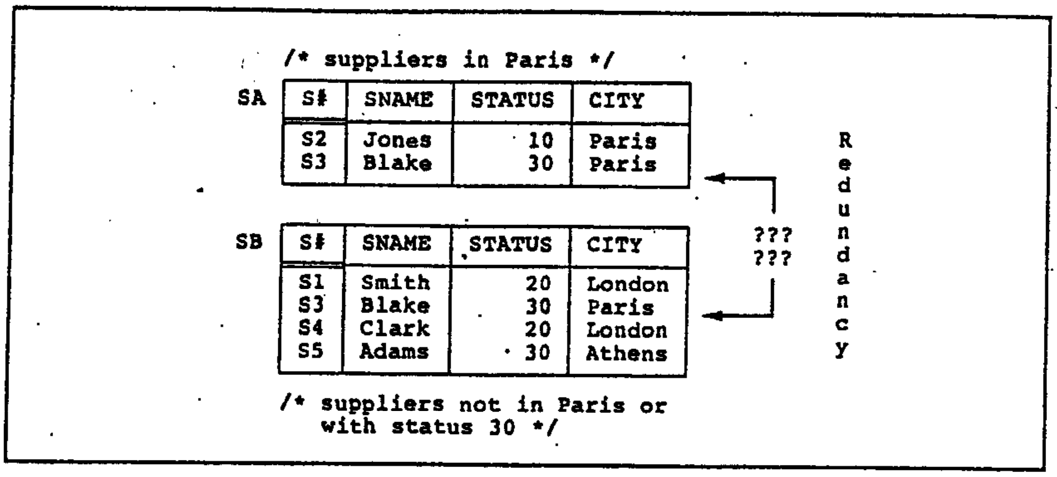
(Note that the latter formulation is likely to perform worse, too.) In other words, the common perception that denormalization is good for retrieval but bad for update is incorrect, in general, for both usability and performance reasons.

A third, and major, problem is as follows (and this point applies to "proper" denormalization—i.e., denormalization that is done at the level of stored files—as well as to the kind of denormalization that sometimes has to be done in today's SQL products): When we say that denormalization is good for performance, what we really mean is that it is good for the performance of *specific applications*. Any given physical design is, of necessity, good for some applications but bad for others (in terms of performance, that is). For example, assume that each base relvar does map to one physically stored file, and assume too that each stored file consists of a physically contiguous collection of stored records, one for each tuple in the corresponding relvar. Then:

- Suppose we represent the join of suppliers, shipments, and parts as one base relvar and hence one stored file. Then the query "Get supplier details for suppliers who supply red parts" will presumably perform well against this physical structure.
- However, the query "Get supplier details for London suppliers" will perform worse against this physical structure than it would if we had stayed with three base relvars and mapped them to three physically separate stored files. The reason is that, with the latter design, all supplier stored records will be physically contiguous, whereas in the former design they will be physically spread over a wider area, and will therefore require more I/O. Analogous remarks apply to any query that accesses suppliers only, or parts only, or shipments only, instead of performing some kind of join.

13.6 ORTHOGONAL DESIGN (A DIGRESSION)

In this section, following reference [13.12], we briefly examine another database design principle, one that is not part of further normalization *per se* but is closely related to it (and, like further normalization, is at least scientific). It is called *The Principle of Orthogonal Design*. Consider Fig. 13.7, which shows an obviously bad but possible design for suppliers; relvar SA in that design corresponds to suppliers who are located in Paris, while relvar SB corresponds to suppliers who are either not located in Paris or have status greater than 30 (i.e., these are the relvar predicates, loosely speaking). As the figure indicates, the design leads to certain redundancies; to be specific, the tuple for supplier S3



/* suppliers in Paris */

SA

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

SB

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

/* suppliers not in Paris or with status 30 */

Fig. 13.7 A bad but possible design for suppliers

appears twice, once in each relvar. And those redundancies in turn lead to update anomalies once again.

Note, incidentally, that the tuple for supplier S3 *must* appear in both places. For suppose, contrariwise, that it were to appear in (say) SB but not SA. Applying the Closed World Assumption to SA would then tell us that it is not the case that supplier S3 is located in Paris. However, SB tells us that it *is* the case that supplier S3 is located in Paris. In other words, we would have a contradiction on our hands, and the database would be inconsistent.

The problem with the design of Fig. 13.7 is obvious, of course: It is precisely the fact that it is possible for the very same tuple to appear in two distinct relvars. In other words, the two relvars have *overlapping meanings*, in the sense that it is possible for the very same tuple to satisfy the predicates for both. So an obvious rule is:

- *The Principle of Orthogonal Design (initial version):* Within a given database, no two distinct base relvars should have overlapping meanings.

Points arising:

1. Recall from Chapter 10 that, from the user's point of view, *all* relvars are base relvars (apart from views that are defined as mere shorthands). In other words, the principle applies to the design of all "expressible" databases, not just to the "real" database—*The Principle of Database Relativity* at work once again. (Of course, analogous remarks apply to the principles of normalization also.)
2. Note that two relvars cannot possibly have overlapping meanings unless they are of the same type (i.e., unless they have the same heading).
3. Adherence to the principle implies that (e.g.) when we insert a tuple, we can regard the operation as inserting a tuple *into the database*, rather than into some specific relvar—because there will be at most one relvar whose predicate the new tuple satisfies.

We elaborate briefly on this last point. Of course, it is true that when we insert a tuple, we do typically specify the name of the relvar *R* into which that tuple is to be inserted. But this observation does not invalidate the argument. In fact, that name *R* is really just *shorthand for the corresponding predicate, PR* say; we are really saying "INSERT tuple *t*—and by the way, *t* is required to satisfy predicate *PR*." Furthermore, *R* might be a view, perhaps defined by means of an expression of the form *A* UNION *B*—and, as we saw in Chapter 10, it is very desirable that the system know whether the new tuple is to go into *A* or *B* or both.

As a matter of fact, remarks analogous to the foregoing apply to *all* operations, not just to INSERTs; in all cases, relvar names are really just shorthand for relvar predicates. *The point cannot be emphasized too strongly that it is predicates, not names, that represent data semantics.*

Now, we have not yet finished with the orthogonal design principle—there is an important refinement that needs to be addressed. Consider Fig. 13.8, which shows another obviously bad but possible design for suppliers. Here the two relvars *per se* do not have overlapping meanings, but their projections on {S#,SNAME} clearly do. As a consequence, an attempt to insert, say, the tuple (S6,Lopez) into a view defined as the union of those two projections will cause the tuple (S6,Lopez,*t*) to be inserted into SX and the tuple (S6,Lopez,*c*) to be inserted into SY (where *t* and *c* are the applicable default values). Clearly, we need to extend the orthogonal design principle to take care of problems like that of Fig. 13.8:

- *The Principle of Orthogonal Design (final version):* Let *A* and *B* be distinct base relvars. Then there must not exist nonloss decompositions of *A* and *B* into *A1, A2, ..., Am* and *B1, B2, ..., Bn* (respectively) such that some projection *Ai* in the set *A1, A2, ..., Am* and some projection *Bj* in the set *B1, B2, ..., Bn* have overlapping meanings.

Points arising:

1. The term *nonloss decomposition* here means exactly what it always means—*viz.*, decomposition of a given relvar into a set of projections such that:
   - The given relvar can be reconstructed by joining the projections back together again.
   - None of those projections is redundant in that reconstruction process. (Strictly speaking, this second condition is not necessary in order for the decomposition to be nonloss, but it is usually desirable, as we saw in Chapter 12.)

SX

S#	SNAME	STATUS
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

SY

S#	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Paris
S4	Clark	London
S5	Adams	Athens

Fig. 13.8 Another bad but possible design for suppliers

2. This version of the principle subsumes the original version, because one nonloss decomposition that is always available for relvar *R* is the identity projection (i.e., the projection of *R* over all attributes).

Further Observations

We offer a few additional remarks concerning the orthogonal design principle.

1. First of all, the term *orthogonality* derives from the fact that what the design principle effectively says is that base relvars should have mutually independent meanings. The principle is common sense, of course, but *formalized* common sense (like the principles of normalization).
2. Suppose we start with the usual suppliers relvar S, but decide for design purposes to break that relvar down into a set of restrictions. Then the orthogonal design principle tells us that the restrictions in that breakdown should all be disjoint, in the sense that no supplier tuple can ever appear in more than one of them. (Also, of course, the union of those restrictions must give us back the original relvar.) We refer to such a breakdown as an orthogonal decomposition.
3. The overall objective of orthogonal design is to reduce redundancy and thereby to avoid update anomalies (again like normalization). In fact, it complements normalization, in the sense that—loosely speaking—normalization reduces redundancy *within* relvars, while orthogonality reduces redundancy *across* relvars.
4. Orthogonality might be common sense, but it is often flouted in practice (indeed, such flouting is sometimes even recommended). Designs like the following one, from a financial database, are all too common:

```
ACTIVITIES_2001 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2002 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2003 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2004 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2005 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
```

   In fact, encoding meaning into names—of relvars or anything else—violates *The Information Principle*, which says (just to remind you) that all information in the database must be cast explicitly in terms of values, and in no other way.
5. If *A* and *B* are base relvars of the same type, adherence to the orthogonal design principle implies that:

A UNION B	: Is always a disjoint union
A INTERSECT B	: Is always empty
A MINUS B	: Is always equal to *A*

13.7 OTHER NORMAL FORMS

Back to normalization *per se.* Recall from the introduction to Chapter 12 that there do exist other normal forms, over and above those discussed in that chapter and this one so far. The fact is, the theory of normalization and related topics—now usually known as dependency theory—has grown into a considerable field in its own right, with a very extensive literature. Research in the area continues; in fact, it flourished for several years (though more recently it might have tapered off somewhat). It is beyond the scope of this chapter to discuss that research in any depth; a good survey of the field, as of the mid 1980s, can be found in reference [13.18], and more recent surveys can be found in references [11.1] and [11.3]. Here we just mention a few specific issues:

1. **Domain-key normal form:** Domain-key normal form (DK/NF) was proposed by Fagin in reference [13.16]. DK/NF—unlike the normal forms we have been discussing—is not defined in terms of FDs, MVDs, or JDs at all. Instead, a relvar *R* is said to be in DK/NF if and only if every constraint on *R* is a logical consequence of the *domain constraints* and *key constraints* that apply to *R*, where:

   - A **domain constraint** is a constraint to the effect that values of a given attribute are taken from some prescribed domain. (Note therefore that, to use the terminology of Chapter 9, such a constraint is really an *attribute* constraint, not a type constraint, even though domains are types.)
   - A **key constraint** is a constraint to the effect that a certain set of attributes constitutes a candidate key.

   Enforcing constraints on a DK/NF relvar is thus conceptually simple, since it is sufficient to enforce just the "domain" (attribute) and key constraints and all other constraints will then be enforced automatically. Note too that "all other constraints" here means more than just FDs, MVDs, and JDs; in fact, it means the entire relvar predicate.

   Fagin shows in reference [13.16] that any DK/NF relvar is necessarily in 5NF (and hence in 4NF, etc.), and indeed in (3,3)NF also (see point 2). However, DK/NF is not always achievable, nor has the question "Exactly when *can* it be achieved?" been answered.

2. **"Restriction-union" normal form:** Consider the suppliers relvar S once again. Normalization theory as we have described it tells us that relvar S is in a "good" normal form; indeed, it is in 5NF, and is therefore guaranteed to be free of anomalies that can be eliminated by taking projections. But why keep all suppliers in a single relvar? What about a design in which London suppliers are kept in one relvar (LS, say), Paris suppliers in another (PS, say), and so on? In other words, what about the possibility of decomposing the original suppliers relvar via **restriction** instead of projection? Would the resulting structure be a good design or a bad one? (In fact it would almost certainly be bad—see Exercise 8.8 in Chapter 8—but the point is that classical normalization theory as such has absolutely nothing to say in answer to such questions.)

   Another direction for normalization research therefore consists of examining the implications of decomposing relvars by some operation other than projection. In the example, the decomposition operator is, as already mentioned, (disjoint) *restriction;* the corresponding recomposition operator is (disjoint) **union**. Thus, it might be possible to construct a "restriction-union" normalization theory, analogous but orthogonal once again to the projection-join normalization theory we have been discussing.[5] To

[5] Indeed, Fagin [13.15] originally called 5NF *projection-join* normal form precisely because it was *the* normal form with respect to the projection and join operators.

this writer's knowledge no such theory has ever been worked out in detail, but some initial ideas can be found in a paper by Smith [13.32], where a new normal form called "(3,3)NF" is defined. (3,3)NF implies BCNF; however, a (3,3)NF relvar need not be in 4NF, nor need a 4NF relvar be in (3,3)NF, so that (as already suggested) reduction to (3,3)NF is orthogonal to reduction to 4NF (and 5NF). Further ideas on this topic appear in references [13.15] and [13.23]. *The Principle of Orthogonal Design* is relevant, too [13.12] (and orthogonal design can thus be thought of as a kind of normalization after all).

3. **Sixth normal form:** Fifth normal form is the final normal form as far as classical projection and join are concerned. In Chapter 23, however, we will see that it is possible, and desirable, to define (a) generalized versions of those operators, and hence (b) a generalized form of join dependency, and hence (c) a new (sixth) normal form, 6NF. Note that it is reasonable to use the name "sixth normal form," because 6NF (unlike the normal forms discussed under points 1 and 2) really does represent another step along the road from 1NF to 2NF to . . . to 5NF. What is more, all 6NF relvars are necessarily in 5NF. See Chapter 23 for further explanation.

13.8 SUMMARY

In this chapter we have completed our discussion (begun in Chapter 12) of further normalization. We have discussed **multi-valued dependencies** (MVDs), which are a generalization of functional dependencies, and **join dependencies** (JDs), which are a generalization of multi-valued dependencies. To simplify considerably:

- A relvar $R\{A,B,C\}$ satisfies the MVDs $A \twoheadrightarrow B \mid C$ if and only if the set of B values matching a given AC value pair depends only on the A value, and similarly for the set of C values matching a given AB pair. Such a relvar can be nonloss-decomposed into its projections on $\{A,B\}$ and $\{A,C\}$; in fact, the existence of the MVDs is a necessary and sufficient condition for that decomposition to be nonloss (Fagin's theorem).
- A relvar satisfies the JD $* \{A, B, ..., Z\}$ if and only if it is equal to the join of its projections on $A, B, ..., Z$. Such a relvar can (obviously) be nonloss-decomposed into those projections.

A relvar is in 4NF if the only nontrivial MVDs it satisfies are in fact FDs out of superkeys. A relvar is in 5NF—also called **projection-join normal form**, PJ/NF—if and only if the only nontrivial JDs it satisfies are in fact FDs out of superkeys (i.e., if the JD is $* \{A, B, ..., Z\}$, then each of $A, B, ..., Z$ is a superkey). 5NF, which is always achievable, is the ***ultimate normal form*** with respect to projection and join.

We also summarized the **normalization procedure**, presenting it as an informal sequence of steps (but we remind you that database design is typically not done by following that procedure anyway). We then described ***The Principle of Orthogonal Design:*** Loosely, no two relvars should have projections with overlapping meanings. Finally, we briefly mentioned some ***additional normal forms***.

In conclusion, we should perhaps point out that research into issues such as those we have been discussing is very much a worthwhile activity. The reason is that the field of further normalization, or rather dependency theory as it is now more usually called, does represent a small piece of science in a field (database design) that is regrettably still far too much of an artistic endeavor—that is, it is still far too subjective and lacking in solid principles and guidelines. Thus, any further successes in dependency theory research are very much to be welcomed.

EXERCISES

13.1 Relvars CTX and SPJ as discussed in the body of the chapter—see Figs. 13.2 and 13.4 for some sample values—satisfied a certain MVD and a certain JD, respectively, that was not implied by the candidate keys of the relvar in question. Express that MVD and that JD as integrity constraints, using the Tutorial D syntax of Chapter 9. Give both calculus and algebraic versions.

13.2 Let *C* be a certain club, and let relvar *R*{*A*,*B*} be such that the tuple (*a*,*b*) appears in *R* if and only if *a* and *b* are both members of *C*. What FDs, MVDs, and JDs does *R* satisfy? What normal form is it in?

13.3 A database is to contain information concerning sales representatives, sales areas, and products. Each representative is responsible for sales in one or more areas; each area has one or more responsible representatives. Similarly, each representative is responsible for sales of one or more products, and each product has one or more responsible representatives. Every product is sold in every area; however, no two representatives sell the same product in the same area. Every representative sells the same set of products in every area for which that representative is responsible. Design a suitable set of relvars for this data.

13.4 In Chapter 12, Section 12.5, we gave an algorithm for nonloss decomposition of an arbitrary relvar *R* into a set of BCNF relvars. Revise that algorithm so that it yields 4NF relvars instead.

13.5 *(Modified version of Exercise 13.3)* A database is to contain information concerning sales representatives, sales areas, and products. Each representative is responsible for sales in one or more areas; each area has one or more responsible representatives. Similarly, each representative is responsible for sales of one or more products, and each product has one or more responsible representatives. Finally, each product is sold in one or more areas, and each area has one or more products sold in it. Moreover, if representative *R* is responsible for area *A*, and product *P* is sold in area *A*, and representative *R* is responsible for product *P*, then *R* sells *P* in *A*. Design a suitable set of relvars for this data.

13.6 Suppose we represent suppliers by the following two relvars SX and SY (as in Fig. 13.8 in Section 13.6):

```
SX { S#, SNAME, STATUS }
SY { S#, SNAME, CITY }
```

Does this design conform to the normalization guidelines described in this chapter and its predecessor? Justify your answer.

REFERENCES AND BIBLIOGRAPHY

13.1 A. V. Aho, C. Beeri, and J. D. Ullman: "The Theory of Joins in Relational Databases," *ACM TODS 4*, No. 3 (September 1979).

The paper that first pointed out that relvars could exist that were not equal to the join of any two of their projections, but were equal to the join of three or more. The major objective of the paper was to present an algorithm, now generally called the chase, for determining whether or not a given JD is a logical consequence of a given set of FDs (an example of the implication problem—see reference [13.18]). This problem is equivalent to the problem of determining whether a given decomposition is nonloss, given a certain set of FDs. The paper also discusses the question of extending the algorithm to deal with the case where the given dependencies are not FDs but MVDs.

13.2 Catriel Beeri, Ronald Fagin, and John H. Howard: "A Complete Axiomatization for Functional and Multi-Valued Dependencies," Proc. 1977 ACM SIGMOD Int. Conf. on Management of Data, Toronto, Canada (August 1977).

Extends the work of Armstrong [11.2] to include MVDs as well as FDs. In particular, it gives the following set of sound and complete inference rules for MVDs:

1. **Complementation:** If A, B, and C together include all attributes of the relvar and A is a superset of $B \cap C$, then $A \twoheadrightarrow B$ if and only if $A \twoheadrightarrow C$.
2. **Reflexivity:** If B is a subset of A, then $A \twoheadrightarrow B$.
3. **Augmentation:** If $A \twoheadrightarrow B$ and C is a subset of D, then $AD \twoheadrightarrow BC$.
4. **Transitivity:** If $A \twoheadrightarrow B$ and $B \twoheadrightarrow C$, then $A \twoheadrightarrow C - B$.

The following additional (and useful) inference rules can be derived from the first four:

5. **Pseudotransitivity:** If $A \twoheadrightarrow B$ and $BC \twoheadrightarrow D$, then $AC \twoheadrightarrow D - BC$.
6. **Union:** If $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$, then $A \twoheadrightarrow BC$.
7. **Decomposition:** If $A \twoheadrightarrow BC$, then $A \twoheadrightarrow B \cap C$, $A \twoheadrightarrow B - C$, and $A \twoheadrightarrow C - B$.

The paper then goes on to give two rules involving a mixture of MVDs and FDs:

8. **Replication:** If $A \rightarrow B$, then $A \twoheadrightarrow B$.
9. **Coalescence:** If $A \twoheadrightarrow B$ and $C \rightarrow D$ and D is a subset of B and $B \cap C$ is empty, then $A \rightarrow D$.

Armstrong's rules [11.2] plus rules 1–4 and 8–9 above form a sound and complete set of inference rules for FDs and MVDs taken together.

The paper also derives one more useful rule relating FDs and MVDs:

10. If $A \twoheadrightarrow B$ and $AB \rightarrow C$, then $A \rightarrow C - B$.

13.3 Volkert Brosda and Gottfried Vossen: "Update and Retrieval Through a Universal Schema Interface," *ACM TODS 13*, No. 4 (December 1988).

Earlier attempts at providing a "universal relation" interface (see reference [13.20]) dealt with retrieval operations only. This paper proposes an approach for dealing with update operations also.

13.4 C. Robert Carlson and Robert S. Kaplan: "A Generalized Access Path Model and Its Application to a Relational Data Base System," Proc. 1976 ACM SIGMOD Int. Conf. on Management of Data, Washington, D.C. (June 1976).