CHAPTER 4

FDs and BCNF (Informal)

It is downright sinful to teach the abstract before the concrete.

—Z. A. Melzak: *Companion to Concrete Mathematics* (1973)

As noted in the previous chapter, Boyce/Codd normal form (BCNF for short) is defined in terms of functional dependencies (FDs); indeed, it's really *the* normal form with respect to functional dependencies, just as—to get ahead of ourselves for a moment—5NF is really *the* normal form with respect to join dependencies (JDs). The overall purpose of the present chapter is to explain BCNF and FDs in detail; as the chapter title indicates, however, the various explanations and associated definitions are all intentionally a little informal at this stage. (Informal, but not inaccurate; I won't tell any deliberate lies.) A more formal treatment of the material appears in the next chapter.

First Normal Form

Let relation *r* have attributes *A1*, ..., *An*, of types *T1*, ..., *Tn*, respectively. By definition, then, if tuple *t* appears in relation *r*, the value of attribute *Ai* in *t* is, and of course must be, of type *Ti* (*i* = 1, ..., *n*). For example, if *r* is the relation that's the current value of the shipments relvar SP (see Figure 1-1 in Chapter 1), then every tuple in *r* has an SNO value that's of type CHAR, a PNO value that's also of type CHAR, and a QTY value that's of type INTEGER.

Now, another way of saying what the first two sentences of the previous paragraph say is simply that relation *r* is in *first normal form* (1NF). Thus, *every* relation is in 1NF!—because a "relation" *r* for which those two sentences fail to hold simply isn't a relation

C. J. Date, *Database Design and Relational Theory*, https://doi.org/10.1007/978-1-4842-5540-7_4

in the first place. My apologies for the repetition, therefore, but here for the record is a precise definition:[1]

> **Definition (first normal form):** Let relation *r* have attributes *A1*, ..., *An*, of types *T1*, ..., *Tn*, respectively. Then *r* is in first normal form (1NF) if and only if, for all tuples *t* appearing in *r*, the value of attribute *Ai* in *t* is of type *Ti* (*i* = 1, ..., *n*).

To say it in different words, 1NF just means that each tuple in the relation in question contains exactly one value, of the appropriate type, for each attribute. *Observe in particular, therefore, that 1NF places absolutely no limitation on what those attribute types are allowed to be.*[2] They can even be relation types! That is, relations with relation valued attributes—RVAs for short—are legal (you might be surprised to hear this, but it's true). An example is given in Figure 4-1 below.

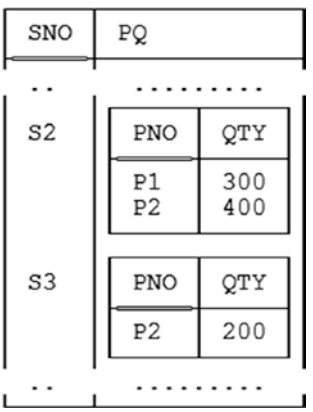

Figure 4-1. *A relation with a relation valued attribute*

[1]One reviewer accused me of rewriting history with this definition. Guilty as charged, perhaps, but I do have my reasons; to be specific, earlier "definitions" of the concept were all, in my opinion, either too vague to be useful or flat out wrong. See *SQL and Relational Theory* for further discussion, also Exercise 4.16 at the end of the chapter.

[2]This sentence is 100% correct as stated. However, I don't want to mislead you; the fact is, there *are* some exceptions—exceptions, that is, to the statement that relational attributes can be of any type whatsoever—but those exceptions have nothing to do with 1NF as such. (The exceptions in question were given in the answer to Exercise 2.2 in Chapter 2, but I repeat them here for convenience. First, if relation *r* is of type *T*, then no attribute of *r* can itself be of type *T* (think about it!). Second, no relation in the database can have an attribute of any pointer type.)

I'll have more to say about RVAs in just a moment, but first I need to get a couple of small points out of the way. To start with, I need to define what it means for a relation to be *normalized:*

> **Definition (normalized):** Relation *r* is normalized if and only if it's in 1NF.

In other words, *normalized* and *first normal form* mean exactly the same thing—all normalized relations are in 1NF, all 1NF relations are normalized. The reason for this slightly strange state of affairs is that *normalized* was the original (historical) term; the term *1NF* wasn't introduced until people started talking about 2NF and higher levels of normalization, when a term was needed to describe relations[3] that weren't in one of those higher normal forms. Of course, it's common nowadays for the term *normalized* to be used to mean some higher normal form (often 3NF specifically, or perhaps BCNF); indeed, I've been known to use it that way myself, though I generally try not to—because strictly speaking such usage is sloppy and incorrect, and it's probably better avoided unless there's no chance of confusion.

Turning to the second of my "small points": Observe now that all of the discussions in this section so far (the definitions in particular) have been framed in terms of relations, not relvars. But since every relation that can ever be assigned to a relvar is in 1NF by definition, no harm is done if we extend the 1NF concept in the obvious way to apply to relvars as well—and it's desirable to do so, because (as we'll see) all of the other normal forms are defined to apply to relvars, not relations. In fact, it could be argued that the reason 1NF *is* defined in terms of relations and not relvars has to do with the fact that it was, regrettably, many years before that distinction (I mean the distinction between relations and relvars) was properly drawn, anyway.

Back to RVAs. I've said, in effect, that relvars with RVAs are legal—but now I need to add that from a design point of view, at least, such relvars are usually (not always) contraindicated. Now, this fact doesn't mean you should avoid RVAs entirely (in particular, there's no problem if some attribute of some query result happens to be relation valued)—it just means we don't usually want RVAs "designed into the database,"

[3]Or relvars, rather (see my second "small point" in just a moment).

as it were. I don't want to get into a lot of detail on this issue in this book; let me just say that relvars with RVAs tend to look very much like the hierarchic structures found in older, nonrelational systems like IMS,[4] and all of the old problems that used to arise with hierarchies therefore raise their ugly head once again. Here for reference is a list of some of those problems:

- The fundamental point is that hierarchies are asymmetric. Thus, while they might make some tasks easier, they certainly make others more difficult.
- As a specific illustration of the previous point, queries in particular are asymmetric, as well as being more complicated than their symmetric counterparts. For example, consider what's involved in formulating the queries "Get part numbers for parts supplied by supplier S2" and "Get supplier numbers for suppliers who supply part P2" against the relation of Figure 4-1. The natural language versions of these queries are symmetric with respect to each other, but their formulations in SQL[5]—or **Tutorial D**, or any other formal language, come to that—most certainly aren't (see Exercise 4.14).
- Similar remarks apply to security and integrity constraints.
- Similar remarks apply to updates, perhaps with even more force.
- There's no guidance, in general, as to how to choose the "best" hierarchy. In the case of suppliers and parts, for example, should we make parts subordinate to suppliers—which is effectively what the design illustrated in Figure 4-1 does—or suppliers subordinate to parts?
- Even "natural" hierarchies like organization charts and bill of materials structures are still best represented, usually, by nonhierarchic designs.

[4]And, perhaps more to the point, newer ones like XML (see Exercise 4.12).

[5]I note in passing that SQL does support something a little bit like RVAs, in the form of columns whose type is *RT* MULTISET, where *RT* is some specified "row type."

Violating First Normal Form

By now you might be wondering, if all relvars are in 1NF by definition, what it could possibly mean for something *not* to be in 1NF. Perhaps surprisingly, this question does have a sensible answer. The point is, today's commercial DBMSs don't properly support relvars (or relations) at all—instead, they support a construct that for convenience I'll call a *table*, though by that term I don't necessarily mean to limit myself to the kinds of tables found in SQL systems specifically.[6] And tables, as opposed to relvars, might indeed not be in 1NF. To elaborate:

> **Definition (normalized table):** A table is in first normal form (1NF)—equivalently, such a table is normalized—if and only if it's a direct and faithful representation of some relvar.

So of course the question is: What does it mean for a table to be a direct and faithful representation of a relvar? The answer to this question involves five basic requirements, all of which are immediate consequences of the fact that the value of a relvar at any given time is (of course) always a relation specifically:

1. The table never contains any duplicate rows.
2. There's no left to right ordering to the columns.
3. There's no top to bottom ordering to the rows.
4. All columns are regular columns.
5. Every row and column intersection always contains exactly one value of the applicable type, and nothing else.

[6] As we've seen, the relational world in general very unfortunately uses the term *relation* to mean sometimes a relation value and sometimes a relation variable. In exactly the same kind of way, SQL in particular uses the term *table* to mean sometimes a table value and sometimes a table variable. Be aware, therefore, that in this section I use the term *table* to mean a table variable specifically, not a table value.

Requirements 1-3 are self-explanatory,[7] but the other two merit a little more explanation, perhaps. First, then, consider Requirement 4 ("All columns are regular columns"). In order to satisfy this requirement, the table in question must be such that both of the following are true:

a. Every column has a proper name (i.e., one that could be specified as a column name in a CREATE TABLE statement, in SQL terms), and that name is unique among the column names that apply to the table in question.

b. No row is allowed to contain anything extra, over and above values of those columns just mentioned. Thus, there are no "hidden" columns that can be accessed only by special operators instead of by regular column references (where a "regular column reference" is basically just a column name), and there are no columns that cause invocations of regular operators on rows to have irregular effects. In particular, therefore, there are no *identifiers* other than regular relational key values (no hidden "row IDs" or "object IDs," as are unfortunately found in some SQL products today), and there are no hidden *timestamps* as are found in certain "temporal database" proposals in the literature.

As for Requirement 5: Observe first of all that this requirement means that nulls are prohibited (since nulls, whatever else they might be, certainly aren't values). More generally, however, the requirement is intended to address the issue of *data value atomicity*. That's the one thing that "everybody knows" about relations in the relational model—namely, that attribute values within such relations are supposed to be atomic (right?). So what exactly does *atomic* mean in this context? Well, in his famous 1970 paper,[8] Codd merely said it meant "nondecomposable." And in later writings he went on to say that *nondecomposable* in turn meant "nondecomposable by the DBMS," which I take to mean there's no way the user can ask the DBMS to perform some such

[7]Though I note I passing that Requirement 2 in particular effectively means that SQL tables are *never* normalized—except, possibly, in the case of such a table with just one column (see footnote 14 in Chapter 1). However, the disciplines recommended in *SQL and Relational Theory* allow us among other things to treat such tables as if they were normalized after all (most of the time but, sadly, not *all* of the time).

[8]E. F. Codd: "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM 13*, No. 6 (June 1970).

decomposition, either explicitly or implicitly, on his or her behalf. All right, so let's consider a few examples:

- *Character strings:* Are character strings nondecomposable in the foregoing sense? Clearly not—think of SQL's SUBSTRING, LIKE, and concatenate operators, for example, all of which clearly rely on the fact that character strings in general have some internal structure and are thus decomposable into smaller pieces. Yet surely no one would argue that character strings shouldn't be allowed in relations.
- *Fixed point numbers:* Can be decomposed into integer and fractional parts.
- *Integers:* Can be decomposed into their prime factors. (Of course, I realize this isn't the kind of decomposability we usually consider in this context; I'm just trying to show that the notion of decomposability is itself open to a variety of interpretations.)
- *Dates and times:* Can be decomposed into year / month / day and hour / minute / second components, respectively.
- *Relational expressions:* Consider, e.g., view definitions in the catalog. Such expressions are certainly "decomposable"—decomposable by the DBMS, in fact—because if they weren't, there'd be no point in keeping them in the catalog in the first place.

The bottom line from all of these examples, and many others like them, is that if relation *r* has an attribute *A*, then the values of *A* within *r* can be *anything whatsoever*, just so long as they're values of the type *T* that's the type defined for that attribute *A*. And that type *T* in turn can be any type whatsoever![9] It can even be a relation type (whence the possibility of relation valued attributes, discussed a few pages back).

The 1NF "atomicity" requirement is sometimes stated in the form "no repeating groups." Indeed, I've stated it in that form myself in numerous earlier writings—in the previous edition of the present book in particular, where I attempted (but failed, I think) to give a precise definition of what a repeating group might be. On further reflection, I've come to the conclusion that it's better not to try to think about repeating groups at all in this context, but rather to focus on (and then, as I've just done, to debunk!) the

[9]Except as noted in footnote 2.

> notion of atomicity instead. In other words, I now think the injunction against repeating groups is and always was essentially meaningless—and I hereby apologize to anyone who might have been misled by my earlier efforts in this regard.

In conclusion, if any of the five requirements are violated, the table in question doesn't "directly and faithfully" represent a relvar, *and all bets are off*. In particular, relational operators such as join are no longer guaranteed to work as expected (as you'll already know if—as I assume—you're familiar with SQL). The relational model deals with *relation* values and variables, and relation values and variables *only*.

Functional Dependencies

So much for 1NF; now I can move on and begin to discuss some of the higher normal forms. Now, I've already said that Boyce/Codd normal form (BCNF) is defined in terms of functional dependencies (FDs), and of course the same is true of second normal form (2NF) and third normal form (3NF) as well. So here's a definition:

> **Definition (functional dependency):** Let X and Y be subsets of the heading of relvar R; then the functional dependency (FD)
>
> $X \rightarrow Y$
>
> holds in R if and only if, whenever two tuples of R agree on X, they also agree on Y.[10] X and Y here are called the determinant and the dependant, respectively, and the FD overall can be read as "X functionally determines Y," or as "Y is functionally dependent on X," or more simply just as "X arrow Y."

Here are a couple of examples:

- The FD {CITY} → {STATUS} holds in relvar S, as we know from Chapter 2. Note the braces, by the way; X and Y in the definition are subsets of the heading of R, and are therefore sets (of attributes), even when, as in the example, they happen to be singleton sets. By the same token, X and Y values are tuples, even when, as in the example, they happen to be tuples of degree one.

[10]"Agree on," in contexts like the one at hand, is standard shorthand for "have the same value for."

- The FD {SNO} → {SNAME,STATUS} also holds in relvar S, because {SNO} is a key—in fact, the only key—for that relvar, and there are *always* "arrows out of keys" (see the section "Keys," immediately following this one). *Note:* In case it isn't obvious, I use the phrase "arrow out of *X*" to mean there exists some *Y* such that the FD $X \rightarrow Y$ holds in the pertinent relvar (where *X* and *Y* are subsets of the heading of that relvar).

Now here's a useful thing to remember: If the FD $X \rightarrow Y$ holds in relvar *R*, then the FD $X'' \rightarrow Y'$ also holds in relvar *R* for all supersets X'' of *X* and all subsets Y' of *Y* (just so long as X'' is still a subset of the heading, of course). In other words, we can always add attributes to the determinant or subtract them from the dependant, and what we get will still be an FD that holds in the relvar in question. For example, here's another FD that holds in relvar S:

```
{ SNO , CITY } → { STATUS }
```

(I started with the FD {SNO} → {SNAME,STATUS} and added CITY to the determinant and subtracted SNAME from the dependant.)

I also need to explain what it means for an FD to be *trivial*:

> **Definition (trivial FD):** The FD $X \rightarrow Y$ is trivial if and only if there's no way it can be violated.

For example, the following FDs all hold trivially in any relvar with attributes called STATUS and CITY:[11]

```
{ CITY , STATUS } → { CITY }
{ CITY , STATUS } → { STATUS }
{ CITY }          → { CITY }
{ CITY }          → { }
```

To elaborate briefly (but considering just the first of these examples, for simplicity): If two tuples have the same value for CITY and STATUS, they certainly have the same value for CITY. In fact, it's easy to see that the FD $X \rightarrow Y$ is trivial if and only if *Y* is a subset of *X* (in symbols, $Y \subseteq X$). Now, when we're doing database design, we don't usually bother with trivial FDs because they're, well, trivial; but when we're trying to be formal and precise about these matters—in particular, when we're trying to develop a *theory* of design—then we need to take all FDs into account, trivial ones as well as nontrivial.

[11] In connection with the last of these examples in particular, see Exercise 4.10 at the end of the chapter.

Keys Revisited

I discussed the concept of keys in general terms in Chapter 1, but it's time to get a little more precise about the matter and to introduce some more terminology. First, here for the record is a precise definition of the term *candidate key*—which, as noted in Chapter 1, I abbreviate to just *key* throughout most of this book:

> **Definition (candidate key, key):** Let *K* be a subset of the heading of relvar *R*. Then *K* is a candidate key (or just a key for short) for *R* if and only if it possesses both of the following properties:
>
> 1. *Uniqueness:* No legitimate value for *R* contains two distinct tuples with the same value for *K*.
> 2. *Irreducibility:* No proper subset of *K* has the uniqueness property.

This is the first definition we've encountered that involves some kind of irreducibility, but we'll meet several more in the pages ahead—irreducibility of one kind or another is ubiquitous, and important, throughout the field of design theory in general, as we'll see. Regarding key irreducibility in particular, one reason (not the only one) why it's important is that if we were to specify a "key" that wasn't irreducible, the DBMS wouldn't be able to enforce the proper uniqueness constraint. For example, suppose we told the DBMS (lying!) that {SNO,CITY} was a key, and in fact the only key, for relvar S. Then the DBMS couldn't enforce the constraint that supplier numbers are "globally" unique; instead, it could enforce only the weaker constraint that supplier numbers are "locally" unique, in the sense that they're unique within the pertinent city.

I'm not going to discuss the foregoing definition any further here, since the concept is so familiar[12]—but observe how the next few definitions depend on it:

> **Definition (key attribute):** Attribute *A* of relvar *R* is a key attribute for *R* if and only if it's part of at least one key of *R*.

[12]Do note, however, that there's no suggestion that relvars have just one key. *Au contraire,* in fact: A relvar can have any number of distinct keys, subject only to a limit that's a logical consequence of the degree of the relvar in question (see Exercise 4.9 at the end of the chapter).

Definition (nonkey attribute): Attribute *A* of relvar *R* is a nonkey attribute for *R* if and only if it's not part of any key of *R*.[13]

For example, in relvar SP, SNO and PNO are key attributes and QTY is a nonkey attribute.

Definition ("all key" relvar): A relvar is "all key" if and only if the entire heading is a key (in which case it's the only key, necessarily)—equivalently, if and only if no proper subset of the entire heading is a key.

Note: If a relvar is "all key," then it certainly has no nonkey attributes, but the converse is false—a relvar can be such that all of its attributes are key attributes and yet not be "all key" (right?).

Definition (superkey): Let *SK* be a subset of the heading of relvar *R*. Then *SK* is a superkey for *R* if and only if it possesses the following property:

1. *Uniqueness:* No legitimate value for *R* contains two distinct tuples with the same value for *SK*.

More succinctly, a superkey for *R* is a subset of the heading of *R* that's unique but not necessarily irreducible. In other words, we might say, loosely, that a superkey is a superset of a key ("loosely," because of course the superset in question must still be a subset of the pertinent heading). Observe, therefore, that all keys are superkeys, but "most" superkeys aren't keys. *Note:* A superkey that isn't a key is sometimes said to be a *proper* superkey.

It's convenient to define the notion of a *subkey* also:

Definition (subkey): Let *SK* be a subset of the heading of relvar *R*. Then *SK* is a subkey for *R* if and only if it's a subset of at least one key of *R*.

Note: A subkey that isn't a key is sometimes said to be a *proper* subkey.

[13]As a historical note, I remark that key and nonkey attributes were called prime and nonprime attributes, respectively, in Codd's original normalization papers (see Appendix D).

By way of example, consider relvar SP, which has just one key, viz., {SNO,PNO}. That relvar has:

a. Two superkeys:

```
{ SNO , PNO }
{ SNO , PNO , QTY }
```

Note that the heading is always a superkey for any relvar *R*.

b. Four subkeys:

```
{ SNO , PNO }
{ SNO }
{ PNO }
{ }
```

Note that the empty set of attributes is always a subkey for any relvar *R*.

To close this section, note that if *H* and *SK* are the heading and a superkey, respectively, for relvar *R*, then the FD $SK \rightarrow H$ holds in *R*, necessarily. (Equivalently, the FD $SK \rightarrow Y$ holds in *R* for all subsets *Y* of *H*.) The reason is that if two tuples of *R* have the same value for *SK*, then they must in fact be the very same tuple, in which case they obviously must have the same value for *Y*. Of course, all of these remarks apply in the important special case in which *SK* is not just a superkey but a key; as I put it earlier (very loosely, of course), there are always arrows out of keys. In fact, we can now make a more general statement: There are always arrows out of superkeys.

Second Normal Form

There's one more concept I need to introduce, viz., *FD irreducibility* (another kind of irreducibility, observe), before I can get on to the definitions of 2NF, 3NF, and BCNF as such:

> **Definition (irreducible FD):** The FD $X \rightarrow Y$ is irreducible with respect to relvar *R* (or just irreducible, if *R* is understood) if and only if it holds in *R* and $X' \rightarrow Y$ doesn't hold in *R* for any proper subset X' of *X*.

For example, the FD {SNO,PNO} → {QTY} is irreducible with respect to relvar SP. *Note:* This kind of irreducibility is sometimes referred to more explicitly as *left* irreducibility (since it's really the left side of the FD that we're talking about), but I've chosen to elide that "left" here for simplicity.

Now—at last, you might be forgiven for thinking—I can define 2NF:

> **Definition (second normal form):** Relvar *R* is in second normal form (2NF) if and only if, for every key *K* of *R* and every nonkey attribute *A* of *R*, the FD $K \rightarrow \{A\}$ (which holds in *R*, necessarily) is irreducible.

Note: The following ("preferred") definition is logically equivalent to the one just given—see Exercise 4.4 at the end of the chapter—but can sometimes be more useful:

> **Definition (second normal form, preferred):** Relvar *R* is in second normal form (2NF) if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in *R*, at least one of the following is true:
>
> a. *X* is a superkey.
>
> b. *Y* is a subkey.
>
> c. *X* is not a subkey.

Points arising:

- First of all, please understand that it would be very unusual to regard 2NF as the ultimate goal of the design process. In fact, both 2NF and 3NF are mainly of historical interest; they're both regarded at best as stepping stones on the way to BCNF, which is of much more pragmatic (as well as theoretical) interest.
- Definitions of 2NF in the literature often take the form "*R* is in 2NF if and only if it's in 1NF and ... ." However, such definitions are usually based on a mistaken understanding of what 1NF is. As we've seen, *all* relvars are in 1NF, and the words "it's in 1NF and" therefore add nothing.

Let's look at an example. Actually, it's usually more instructive with the normal forms to look at a counterexample rather than an example per se. Consider, therefore, a revised version of relvar SP—let's call it SCP—that has an additional attribute CITY, representing the city of the applicable supplier. Here are some sample tuples:

SCP

SNO	CITY	PNO	QTY
S1	London	P1	300
S1	London	P2	200
S1	London	P3	400
..	.....	..	...
S2	Paris	P1	300
S2	Paris	P2	400
..	.....	..	...

This relvar clearly suffers from redundancy: Every tuple for supplier S1 tells us S1 is in London, every tuple for supplier S2 tells us S2 is in Paris, and so on. And (appealing to the first of the foregoing definitions of 2NF) the relvar isn't in second normal form—its sole key is {SNO,PNO}, and the FD {SNO,PNO} → {CITY} therefore certainly holds, but that FD isn't irreducible; to be specific, we can drop PNO from the determinant and what remains, {SNO} → {CITY}, is still an FD that holds in the relvar. Equivalently, we can say the FD {SNO} → {CITY} holds and is nontrivial; moreover, (a) {SNO} isn't a superkey, (b) {CITY} isn't a subkey, and (c) {SNO} *is* a subkey, and so again (appealing now to the second, "preferred" definition of 2NF) the relvar isn't in second normal form.

Third Normal Form

This time I'll just start with my preferred definition:

> **Definition (third normal form, preferred):** Relvar R is in third normal form (3NF) if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in R, at least one of the following is true:
>
> a. X is a superkey.
>
> b. Y is a subkey.

Points arising:

- To repeat something I said in the previous section (and contrary to popular opinion, perhaps), 3NF is mainly of historical interest—it should be regarded at best as no more than a stepping stone on the way to BCNF. *Note:* The reason I say *contrary to popular opinion* here is that many of the "definitions" of 3NF commonly found (at least in the popular literature) are actually definitions of BCNF—and BCNF, as I've already indicated, *is* important. *Caveat lector.*
- Definitions of 3NF in the literature often take the form "*R* is in 3NF if and only if it's in 2NF and ... ." I prefer a definition that makes no mention of 2NF. Note, however, that my definition of 3NF can in fact be derived from my preferred definition for 2NF by dropping condition (c) ("*X* is not a subkey"). It follows that 3NF implies 2NF—that is, if a relvar is in 3NF, then it's certainly in 2NF.

We've already seen an example of a relvar that's in 2NF but not 3NF: namely, the suppliers relvar S (see Figure 3-1 in Chapter 3). To elaborate: The nontrivial FD {CITY} → {STATUS} holds in that relvar, as we know; moreover, {CITY} isn't a superkey and {STATUS} isn't a subkey, and so the relvar isn't in 3NF. (It's certainly in 2NF, however. *Exercise:* Check this claim!)

Boyce/Codd Normal Form

As I said earlier, Boyce/Codd normal form (BCNF) is *the* normal form with respect to FDs—but now I can define it precisely:

> **Definition (Boyce/Codd normal form):** Relvar *R* is in Boyce/Codd normal form (BCNF) if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in *R*, the following is true:
>
> a. *X* is a superkey.

Points arising:

- It follows from the definition that the only FDs that hold in a BCNF relvar are either trivial ones (we can't get rid of those, obviously) or arrows out of superkeys (we can't get rid of those, either). Or as some people like to say: *Every fact is a fact about the key, the whole*

key, and nothing but the key—though I must immediately add that this informal characterization, intuitively attractive though it is, isn't really accurate, because it assumes among other things that there's just one key.

- The definition makes no reference to 2NF or 3NF. Note, however, that the definition can be derived from the 3NF definition by dropping condition (b) ("*Y* is a subkey"). It follows that BCNF implies 3NF—that is, if a relvar is in BCNF, then it's certainly in 3NF.

By way of an example of a relvar that's in 3NF but not BCNF, consider a revised version of the shipments relvar—let's call it SNP—that has an additional attribute SNAME, representing the name of the pertinent supplier. Suppose also that supplier names are necessarily unique (i.e., no two suppliers ever have the same name at the same time). Here are some sample tuples:

SNP

SNO	SNAME	PNO	QTY
S1	Smith	P1	300
S1	Smith	P2	200
S1	Smith	P3	400
..	.....	..	...
S2	Jones	P1	300
S2	Jones	P2	400
..	.....	..	...

Once again we observe some redundancy: Every tuple for supplier S1 tells us S1 is named Smith, every tuple for supplier S2 tells us S2 is named Jones, and so on; likewise, every tuple for Smith tells us Smith's supplier number is S1, every tuple for Jones tells us Jones's supplier number is S2, and so on. And the relvar isn't in BCNF. First of all, it has two keys, {SNO,PNO} and {SNAME,PNO}.[14] Second, every subset of the heading—the subset {QTY} in particular—is (of course) functionally dependent on both of those keys. Third, however, the FDs {SNO} → {SNAME} and {SNAME} → {SNO} also hold; these FDs are certainly not trivial, nor are they arrows out of superkeys, and so the relvar isn't in BCNF (though it is in 3NF).

[14]That's why I didn't show any double underlining when I showed the sample tuples—there are two candidate keys, and there doesn't seem to be any good reason to make either of them primary and thus somehow "more equal than the other."

Finally, as I'm sure you know, the normalization discipline says: If relvar *R* isn't in BCNF, then decompose it into projections that are. In the case of relvar SNP, either of the following decompositions will meet this objective:

- Projecting on {SNO,SNAME} and {SNO,PNO,QTY}
- Projecting on {SNO,SNAME} and {SNAME,PNO,QTY}

I can now explain why BCNF is the odd one out, as it were, in not simply being called "*n*th normal form" for some *n*. To quote from the paper in which Codd first described this new normal form:[15]

> More recently, Boyce and Codd developed the following definition: A [relvar] R is in third normal form if it is in first normal form and, for every attribute collection C of R, if any attribute not in C is functionally dependent on C, then all attributes in R are functionally dependent on C [*in other words, C is a superkey*].

So Codd was giving here what he regarded as a "new and improved" definition of *third* normal form. But the trouble was, the new definition was (and is) strictly stronger than the old one; that is, any relvar that's in 3NF by the new definition is certainly in 3NF by the old one, but the converse isn't true—a relvar can be in 3NF by the old definition and not in 3NF by the new one (relvar SNP, discussed above, is a case in point). So what that "new and improved" definition really defined was a new and stronger normal form, which therefore needed a distinct name of its own. However, by the time this point was adequately recognized, Fagin had already defined what he called fourth normal form, so *that* name wasn't available.[16] Hence the anomalous name *Boyce/Codd normal form*.

[15]E. F. Codd: "Recent Investigations into Relational Data Base Systems," Proc. IFIP Congress, Stockholm, Sweden (1974).

[16]Actually, when Raymond Boyce first came up with what became BCNF, he did call it fourth! The paper in which he first described the concept, *IBM Technical Disclosure Bulletin 16*, No. 1 (June 1973), had as its title "Fourth Normal Form and its Associated Decomposition Algorithm." Since that paper predated Fagin's paper on 4NF by several years (see Appendix D), Boyce's original name could perfectly well have been used at the time. It was Codd who insisted on calling the new normal form "third"—describing Boyce's definition as merely an improved version of one that already existed—and who thereby gave rise to a confusion (admittedly minor, but all logical differences are big differences) that continues to this day.

Exercises

4.1 How many FDs hold in relvar SP? Which ones are trivial? Which are irreducible?

4.2 Is it true that the FD concept relies on the notion of tuple equality?

4.3 Give examples from your own work environment of (a) a relvar not in 2NF; (b) a relvar in 3NF but not 2NF; (c) a relvar in BCNF but not 3NF.

4.4 Prove that the two definitions of 2NF given in the body of the chapter are logically equivalent.

4.5 Is it true that if a relvar isn't in 2NF, then it must have a composite key?

4.6 Is it true that every binary relvar is in BCNF?

4.7 (*Same as Exercise 1.4.*) Is it true that every "all key" relvar is in BCNF?

4.8 Write **Tutorial D** CONSTRAINT statements to express the fact that the pair of FDs {SNO} → {SNAME} and {SNAME} → {SNO} hold in relvar SNP (see the section "Boyce/Codd Normal Form").

Note: This is the first exercise in any chapter that asks you to give an answer in **Tutorial D**. Of course, I realize you might not be completely conversant with that language; in all such exercises, therefore—for example, in Exercises 4.14 and 4.15 below—please just do the best you can. I do think it's worth your while at least to attempt the exercises in question.

4.9 Let *R* be a relvar of degree *n*. What's the maximum number of FDs that can possibly hold in *R* (trivial ones as well as nontrivial)? What's the maximum number of keys it can have?

4.10 Given that *X* and *Y* in the FD $X \rightarrow Y$ are both sets of attributes, what happens if either of those sets is empty?

4.11 Can you think of a situation in which it really would be reasonable to have a base relvar with an RVA?

4.12 There's been a lot of discussion in the industry in recent years of the possibility of *XML databases*. But XML documents are inherently hierarchic in nature; so do you think the criticisms of hierarchies in the body of the chapter apply to XML databases? (Well, yes, they do, as I indicated in footnote 4 earlier in the chapter. So what do you conclude?)

4.13 In Chapter 1 I said I'd be indicating primary key attributes, in tabular pictures of relations, by double underlining. At that point, however, I hadn't properly discussed the difference between relations and relvars, and now we know that keys in general apply to relvars, not relations. Yet we've seen several tabular pictures since then that represent relations as such (I mean, relations that aren't just a sample value for some relvar)—see, e.g., Figure 4-1 for three examples[17]—and I've certainly been using the double underlining convention in those pictures. So what can we say about that convention now?

4.14 Give **Tutorial D** formulations of the following queries against the relation shown in Figure 4-1:

a. Get part numbers for parts supplied by supplier S2.

b. Get supplier numbers for suppliers who supply part P2.

4.15 Suppose we need to update the database to show that supplier S2 supplies part P5 in a quantity of 500. Give **Tutorial D** formulations of the required update against (a) the non RVA design of Figure 1-1, (b) the RVA design of Figure 4-1.

4.16 Given the RVA design illustrated in Figure 4-1, state as precisely as you can the corresponding relvar predicate.

[17]Yes, I do mean three.

4.17 Here are some definitions of 1NF from the technical literature. In view of the discussion of such matters in the body of the present chapter, do you have any comments on them?

- **First normal form** (1NF) … states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute … 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple* … 1NF disallows "relations within relations" or "relations as attribute values within tuples" … the only attribute values permitted by 1NF are single **atomic** (or **indivisible**) values (Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, 4th edition, Addison-Wesley, 2004)
- A relation is in **first normal form** if every field contains only atomic values, that is, no lists or sets (Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd edition, McGraw-Hill, 2003)
- *First normal form* is simply the condition that every component of every tuple is an atomic value (Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, *Database Systems: The Complete Book*, Prentice Hall, 2002)
- A domain is **atomic** if elements of the domain are considered to be indivisible units … we say that a relation schema *R* is in **first normal form** (1NF) if the domains of all attributes of *R* are atomic (Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, 4th edition, McGraw-Hill, 2002)
- A relation is said to be in **first normal form** (abbreviated 1NF) if and only if it satisfies the condition that it contains scalar values only (C. J. Date, *An Introduction to Database Systems*, 6th edition, Addison-Wesley, 1995)

Answers

4.1 The complete set of FDs—what's known, formally, as the *closure* (see Chapter 7), though it has nothing to do with the closure property of the relational algebra—for relvar SP contains a total of 31 distinct FDs, as follows:

```
{ SNO , PNO , QTY } → { SNO , PNO , QTY }
{ SNO , PNO , QTY } → { SNO , PNO }
{ SNO , PNO , QTY } → { SNO , QTY }
{ SNO , PNO , QTY } → { PNO , QTY }
{ SNO , PNO , QTY } → { SNO }
{ SNO , PNO , QTY } → { PNO }
{ SNO , PNO , QTY } → { QTY }
{ SNO , PNO , QTY } → { }

{ SNO , PNO }       → { SNO , PNO , QTY }
{ SNO , PNO }       → { SNO , PNO }
{ SNO , PNO }       → { SNO , QTY }
{ SNO , PNO }       → { PNO , QTY }
{ SNO , PNO }       → { SNO }
{ SNO , PNO }       → { PNO }
{ SNO , PNO }       → { QTY }
{ SNO , PNO }       → { }

{ SNO , QTY }       → { SNO , QTY }
{ SNO , QTY }       → { SNO }
{ SNO , QTY }       → { QTY }
{ SNO , QTY }       → { }

{ PNO , QTY }       → { PNO , QTY }
{ PNO , QTY }       → { PNO }
{ PNO , QTY }       → { QTY }
{ PNO , QTY }       → { }

{ SNO }             → { SNO }
{ SNO }             → { }
```

```
{ PNO }          → { PNO }
{ PNO }          → { }

{ QTY }          → { QTY }
{ QTY }          → { }

{ }              → { }
```

Of these, the only ones that aren't trivial are the following four:

```
{ SNO , PNO } → { SNO , PNO , QTY }
{ SNO , PNO } → { SNO , QTY }
{ SNO , PNO } → { PNO , QTY }
{ SNO , PNO } → { QTY }
```

And the only ones that are irreducible are the following eleven:

```
{ SNO , PNO } → { SNO , PNO , QTY }
{ SNO , PNO } → { SNO , PNO }
{ SNO , PNO } → { SNO , QTY }
{ SNO , PNO } → { PNO , QTY }
{ SNO , PNO } → { QTY }

{ SNO , QTY } → { SNO , QTY }

{ PNO , QTY } → { PNO , QTY }

{ SNO }       → { SNO }

{ PNO }       → { PNO }

{ QTY }       → { QTY }

{ }           → { }
```

4.2 Yes, it is ("whenever two tuples agree on *X*, they also agree on *Y*" implies an equality comparison between the projections of the tuples in question on the attributes of *X* and *Y*, respectively, and those two projections in turn are themselves tuples). *Note:* See the answer to Exercise 2.10 in Chapter 2 regarding the notion of tuple equality in general.

4.3 *No answer provided.*

4.4 First of all, here again are the two definitions, numbered for purposes of subsequent reference:

1. Relvar R is in 2NF if and only if, for every key K of R and every nonkey attribute A of R, the FD $K \rightarrow \{A\}$ is irreducible.

2. Relvar R is in 2NF if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in R, at least one of the following is true: (a) X is a superkey; (b) Y is a subkey; (c) X is not a subkey.

Let R not be in 2NF by Definition 1. Then there exists an FD—nontrivial by definition—$K \rightarrow \{A\}$, where K is a key of R and A is a nonkey attribute of R, that holds in R and is reducible. Since it's reducible, the (also nontrivial) FD $X \rightarrow \{A\}$ holds in R for some proper subkey X ($X \subset K$). Thus, denoting $\{A\}$ by Y, we have a nontrivial FD $X \rightarrow Y$ that holds in R such that X isn't a superkey, Y isn't a subkey, and X is a subkey. So R isn't in 2NF by Definition 2. So, loosely, Definition 2 implies Definition 1.[18]

Now let R not be in 2NF by Definition 2. Then there exists a nontrivial FD $X \rightarrow Y$ (F, say) that holds in R, such that X isn't a superkey, Y isn't a subkey, and X is a subkey. But if X is a subkey and not a superkey, it must be a *proper* subkey of some key K. Now there are two cases to consider:

a. Y contains a nonkey attribute A. In this case $K \rightarrow \{A\}$ holds in R but is reducible, and so R isn't in 2NF by Definition 1; so, again loosely, Definition 1 implies Definition 2.

b. No such F exists such that Y contains a nonkey attribute A. But then, for every F, every attribute A contained in Y is such that $\{A\}$ is a subkey. Hence R is in *3NF* (and therefore certainly in 2NF): Contradiction.

It follows that Definitions 1 and 2 are equivalent.

[18]You might think this is the wrong way round, but it isn't. What I've shown is that *not 2NF by Definition 1* implies *not 2NF by Definition 2*. Given that "*p* implies *q*" is equivalent to "(not *p*) or *q*," therefore, it follows that what I've shown is that *2NF by Definition 2* implies *2NF by Definition 1*, or (loosely) *Definition 2 implies Definition 1* as stated. Apologies if you find this confusing!

4.5 Consider the following (invalid!) argument.

Let relvar R not be in 2NF. Then there must be some key K of R and some nonkey attribute A of R such that the FD $K \rightarrow \{A\}$ (which holds in R, necessarily) is reducible—meaning some attribute can be dropped from K, yielding K' say, such that the FD $K' \rightarrow \{A\}$ still holds. Hence K must be composite.

This argument appears to show that the answer to the exercise must be *yes*—i.e., if a relvar isn't in 2NF, it must have a composite key. But the argument is incorrect! Here's a counterexample. Let USA be a binary relvar with attributes COUNTRY and STATE; the predicate is *STATE is part of COUNTRY*, but COUNTRY is the United States in every tuple. Now, {STATE} is the sole key for this relvar, and the FD {STATE} → {COUNTRY} thus certainly holds. However, the FD { } → {COUNTRY} clearly holds as well (see the answer to Exercise 4.10 below); the FD {STATE} → {COUNTRY} is thus reducible, and so the relvar isn't in 2NF, and yet the key {STATE} isn't composite.

4.6 No! By way of a counterexample, consider relvar USA from the answer to the previous exercise. That relvar is subject to the FD { } → {COUNTRY}, which is neither trivial nor an arrow out of a superkey, and so the relvar isn't in BCNF. (In fact, of course, it isn't even in 2NF, as we saw in the answer to the previous exercise.) It follows that the relvar can be nonloss decomposed into its two unary projections on {COUNTRY} and {STATE}, respectively. (Note that the corresponding join, needed to reconstruct the original relvar, in fact reduces to a cartesian product.)

4.7 Yes, it is. If no nontrivial FDs hold at all—which is certainly the case for an "all key" relvar—then there's certainly no nontrivial FD that holds for which the determinant isn't a superkey, and so the relvar is in BCNF.

4.8
```
CONSTRAINT ...
   COUNT ( SNP { SNO , SNAME } ) = COUNT ( SNP { SNO } ) ;

CONSTRAINT ...
   COUNT ( SNP { SNO , SNAME } ) = COUNT ( SNP { SNAME } ) ;
```

Note: This trick for specifying that an FD holds (i.e., by stating that two projections have the same cardinality) certainly does the job. As noted in Chapter 2, however, it's hardly very elegant, and for that reason I showed an alternative approach to formulating such constraints, using AND, JOIN, and RENAME, in that chapter. Here are revised formulations of the two constraints just shown that make use of that alternative approach:

```
CONSTRAINT ...
    WITH ( SS := S { SNO , SNAME } ) :
    AND ( ( SS JOIN ( SS RENAME { SNAME AS X } ) , SNAME = X ) ;

CONSTRAINT
    WITH ( SS := S { SNO , SNAME } ) :
    AND ( ( SS JOIN ( SS RENAME { SNO AS X } ) , SNO = X ) ;
```

Alternatively, Hugh Darwen and I have proposed[19] that **Tutorial D** should support another form of CONSTRAINT statement in which the usual boolean expression is replaced by a relational expression accompanied by one or more key specifications. Under this proposal, the foregoing constraints could be expressed as a single constraint, thus:

```
CONSTRAINT ... SNP { SNO , SNAME }
                   KEY { SNO }
                   KEY { SNAME } ;
```

Explanation: Think of the relational expression—SNP {SNO,SNAME}, in the example—as defining some temporary relvar (perhaps a view); then the key specifications—KEY {SNO} and KEY {SNAME}, in the example—indicates that the specified attributes would constitute keys for that relvar.[20]

[19]In our book *Database Explorations: Essays on The Third Manifesto and Related Topics* (Trafford, 2010) and elsewhere.

[20]Actually there's no need for the projection in the example—CONSTRAINT SNP KEY {SNO} KEY {SNAME} would suffice.

As an aside, I note that Darwen and I have also proposed allowing foreign key constraints to be specified for expressions in the same kind of way.

4.9 Let the FD $X \rightarrow Y$ hold in R. By definition, X and Y are subsets of the heading of R. Given that a set of n elements has 2^n possible subsets, it follows that each of X and Y has 2^n possible values, and hence an upper limit on the number of possible FDs that might hold in R is 2^{2n}. For example, if R is of degree five, the upper limit on the number of FDs that might hold is 1,024 (of which 243 are trivial). *Subsidiary exercises:*

a. Where did that figure of 243 come from?

Answer: I'll let you figure this one out for yourself!

b. Suppose those 1,024 FDs do all in fact hold. What can we conclude about R in that case?

Answer: It must have cardinality less than two. The reason is that one FD that holds in such a case is $\{\} \rightarrow H$, where H is the heading; it follows that $\{\}$ is a key, and so R is constrained to contain at most one tuple, as explained in the answer to the next exercise below.

As for how many keys R can have: Let m be the smallest integer greater than or equal to $n/2$. R will have the maximum possible number of keys if either (a) every distinct set of m attributes is a key or (b) m is odd and every distinct set of $(m - 1)$ attributes is a key. Either way, it follows that the maximum number of keys is $n! / (m! \times (n - m)!)$.[21] For example, a relvar of degree five can have at most ten, and one of degree three can have at most three, distinct keys. (An example of this latter case can be found in Appendix C.)

4.10 Let the specified FD $X \rightarrow Y$ hold in relvar R. Now, *every* tuple (regardless of whether it's a tuple of R) has the same value—namely, the 0-tuple—for the projection of that tuple over the

[21]The symbol $r!$ is pronounced "r factorial" (sometimes "r bang") and denotes the product $r \times (r\text{-}1) \times \ldots \times 2 \times 1$.

empty set of attributes (see the answer to Exercise 2.12 in Chapter 2). If Y is empty, therefore, the FD $X \rightarrow Y$ holds for all possible sets X of attributes of R; in fact, it's a trivial FD (and so it isn't very interesting), because the empty set is a subset of every set and so Y is definitely a subset of X in this case. On the other hand, if X is empty, the FD $X \rightarrow Y$ means that, at any given time, all tuples of R have the same value for Y (since they certainly all have the same value for X). What's more, if Y in turn is the entire heading of R–in other words, if X is a superkey—then R is constrained to contain at most one tuple (for otherwise R would suffer from a superkey uniqueness violation). *Note:* In this latter case, X isn't just a superkey but in fact a key, since it's certainly irreducible. What's more, it's the *only* key, because every other subset of the heading includes it as a proper subset.

4.11 Consider a relvar (FDR, say) in the database catalog whose purpose is to record the FDs that hold in various relvars in the database. Given that an FD is an expression of the form $X \rightarrow Y$ where X and Y are sets of attribute names, a reasonable design for that relvar FDR is one with attributes R (relvar name), X (determinant), and Y (dependant), and predicate (deliberately stated here somewhat loosely) *The FD $X \rightarrow Y$ holds in relvar R*. For any given relvar R in the database, therefore, the corresponding tuple in relvar FDR has X and Y values that are each relations of degree one, the tuples of which contain names of attributes of relvar R (and so X and Y are RVAs).

For another example, involving a "user relvar" instead of a relvar in the catalog, you might like to think about the following problem:

I decided to throw a party, so I drew up a list of people I wanted to invite and made some preliminary soundings. The response was good, but several people made their acceptance conditional on the acceptance of certain other invitees. For example, Bob and Cal both said they would come if and only if Amy came; Fay said she would come if and only if Don and Eve both came; Guy said he would come anyway; Hal said he would come if and only if Bob

and Amy both came; and so on. Design a database to show whose acceptance is based on whose. (*With acknowledgments to Hugh Darwen.*)

It seems to me that a reasonable design here would involve a relvar with two attributes X and Y, both relation valued, and predicate (again deliberately stated somewhat loosely) *The set of people X will attend if and only if the set of people Y will attend.*

Subsidiary exercise: Can you think of any refinements you might want to make to this design? *Hint:* Is it true that Bob will attend if and only if Bob will attend?

4.12 Well, I don't know what you conclude, but I know what I do. One thing I conclude is that we should always be on our guard against getting seduced by the latest fad. (I could say quite a lot more regarding this latter, but I don't think this book is the right place for it.)

4.13 There are two cases to consider:

a. The relation depicted is a sample value for some relvar *R*.

b. The relation depicted is a sample value for some relational expression *rx*, where *rx* is something other than a simple relvar reference (where a relvar reference is basically just the pertinent relvar name).

In Case a., double underlining simply indicates that a primary key *PK* has been declared for *R*[22] and the pertinent attribute is part of *PK*. In Case b., you can think of *rx* as the defining expression for some temporary relvar *R* (think of it as a view defining expression and *R* as the corresponding view, if you like); then double underlining indicates that a primary key *PK* could in principle be declared for *R* and the pertinent attribute is part of *PK*.

[22]Well, that's not quite the situation as far as this book is concerned, because in this book I almost never declare primary keys as such (in fact **Tutorial D** provides no way of doing so)—but I think you see what I mean.

4.14 I assume for the sake of this exercise and the next that the relation shown in Figure 4-1 is a sample value for a relvar SPQ. Here then are **Tutorial D** formulations (not the only ones possible) for the two queries:

a. `( ( SPQ WHERE SNO = 'S2' ) UNGROUP ( PQ ) ) { PNO }`

b. `( ( SPQ UNGROUP ( PQ ) ) WHERE PNO = 'P2' ) { SNO }`

Observe that the first of these expressions involves a restriction followed by an ungrouping, while the second involves an ungrouping followed by a restriction (there's the asymmetry).

Note: The UNGROUP operator wasn't discussed in the body of the chapter, but its semantics should be obvious from the examples. Basically, it's used to map a relation with an RVA to one without such an attribute. (There's a GROUP operator too, for "going the other way"—that is, mapping a relation without an RVA to one with one.) For further explanation, see *SQL and Relational Theory*.

4.15 Here I think it might be helpful first to give part of the **Tutorial D** grammar for <*relation assign*>, which is the fundamental relational update operator in **Tutorial D**. (Please note that the grammar is slightly simplified for present purposes, though. As for the names of the various syntactic categories, they're meant to be intuitively self-explanatory.)

```
<relation assign>
    ::=   <relvar name> := <relation exp>
        | <insert> | <delete> | <update>

<insert>
    ::=   INSERT <relvar name> <relation exp>

<delete>
    ::=   DELETE <relvar name> <relation exp>
        | DELETE <relvar name> [ WHERE <boolean exp> ]
```

```
<update>
    ::=   UPDATE <relvar name> [ WHERE <boolean exp> ] :
                                  { <attribute assign commalist> }
```

And an <*attribute assign*>, if the attribute in question happens to be relation valued, is basically just a <*relation assign*> (except that the pertinent <*attribute name*> appears in place of the target <*relvar name*> in that <*relation assign*>), and that's where we came in. Here then are **Tutorial D** statements for the required updates:

a. `INSERT SP RELATION { TUPLE { 'S2' , 'P5' , 500 } } ;`

b. `UPDATE SPQ WHERE SNO = 'S2' :`
   `{ INSERT PQ RELATION { TUPLE { PNO 'P5' , QTY 500 } } } ;`

4.16 Supplier SNO supplies part PNO in quantity QTY if and only if (PNO,QTY) is a tuple in PQ.

4.17 One obvious comment is that I was just as confused as everybody else, back in 1995! Though I did inadvertently (?) conceal my confusion by using the comparatively respectable term *scalar* from the programming languages world in place of that rather suspect term *atomic*. However, I've come to realize that *scalar* is really no more formal or precise than *atomic* is, in this context. The fact is, neither term has any absolute meaning—it simply depends on what we want to do with the data. For example, sometimes we want to deal with an entire set of part numbers as a single thing; at other times, we want to deal with individual part numbers within that set—but then we're descending to a lower level of detail (in other words, to a lower level of abstraction). The following analogy might help clarify this point. In physics—which after all is where the "atomicity" terminology comes from—the situation is precisely parallel: Sometimes we want to think about individual physical atoms as indivisible objects; at other times, we want to think about the protons, neutrons, and electrons that go to make up those atoms. And of course those protons and neutrons aren't really indivisible, either—they contain a variety of "subsubatomic" particles called *quarks*. And so on, possibly (?).

One last point: An attempt *might* be made to rescue the notion of absolute atomicity, as follows. Let's agree to say a value is *not* atomic if and only if operators exist to "take the value apart," as it were. (I think this idea, or something very like it, was probably what was in Codd's mind when he said that *nondecomposable* meant, specifically, "nondecomposable by the DBMS.") Thus, a character string isn't atomic because it can be taken apart by means of the SUBSTRING operator; a set isn't atomic because it can be taken apart by means of operators that extract either elements or subsets from the set; and so on. By contrast, a fixed point number is atomic, if there are no operators to extract its integer and fractional parts (say). But even if we were to accept this argument, it seems to me we would then also have to accept the argument that the very same value might be atomic today and composite tomorrow! Such would be the case for a fixed point number, for example, if there were originally no operators to extract its integer and fractional parts but such operators were subsequently introduced. And it further seems to me that if the notion of atomicity is time dependent in this way, then (as previously claimed) it really doesn't have any absolute meaning.