Database In Depth

Relational Theory for Practitioners

Answers to Exercises

C. J. Date

1

Introduction

Exercise 1-1. *No answer provided.*

Exercise 1-2. E. F. Codd (1923-2003) was the original inventor of the relational model, among many other things. In December 2003 I published a brief tribute to him and his achievements, which you can find on the ACM SIGMOD website www.acm.org/sigmod and elsewhere. See also my book *The Database Relational Model: A Retrospective Review and Analysis* (Addison-Wesley, 2001), a brief description of which is given in Appendix B.

Exercise 1-3. A domain is basically a conceptual pool of values from which actual attributes in actual relations take their actual values. In other words, a domain is a type, and the terms *domain* and *type* are effectively interchangeable—but personally I much prefer *type*, as having a longer pedigree (in the computing world, at least). *Domain* is the term used in most of the older database literature, however. *Note:* Don't confuse domains as understood in the relational world with the construct of the same name in SQL, which at best can be regarded (to be extremely charitable about the matter) as a very weak kind of type. See the answer to Exercise 2.1 in Chapter 2.

Exercise 1-4. A database satisfies the referential integrity rule if and only if for every tuple containing a *reference* (in other words, a foreign key value) there exists a *referent* (in other words, a tuple in the pertinent referenced relation with that same value for the pertinent candidate key). Loosely: If *B* references *A*, then *A* must exist. See Chapter 6 for further discussion.

Exercise 1-5. Let *R* be a relvar and let *r* be the relation that's its value at some given time. Then the heading, attributes, and degree of *R* are defined to be identical to the heading, attributes, and degree of *r*, respectively. Likewise, the body, tuples, and cardinality of *R* are defined to be identical to the body, tuples, and cardinality of *r*, respectively. Note, however, that the body, tuples, and cardinality of *R* vary over time, while the heading, attributes, and degree don't.

By the way, it follows from the foregoing that if we use SQL's ALTER TABLE to add a column to or drop a column from some table *T*, the effect is to replace that table by some logically distinct table *T'* (the term *table* being, in such contexts, SQL's counterpart to the relational term *relvar*). *T'* is *not* "the same table as before"—speaking purely from a logical point of view, that is. Of course, it's convenient to overlook this nicety in informal contexts.

Exercise 1-6. See the section "Model *vs*. Implementation" in the body of the chapter.

Exercise 1-7. The model is the *abstract machine* with which the user interacts; the *implementation* is the realization of that abstract machine on some physical computer system. Users have to understand the model, since it defines the interface they have to deal with; they don't have to understand the implementation, because that's under the covers (at least, it should be). The following analogy might help: In order to drive a car, you don't have to know what goes on under the hood—all you have to know is how to steer, how to shift gear, and so on. So the rules for steering, shifting, and the rest are the model, and what's under the hood is the implementation. (Of course, it's true that you might drive better if you have some understanding of what goes on under the hood, but you don't *have* to know. Analogously, you might use a data model better if you have some knowledge of how it's implemented—but, ideally at least, you shouldn't *have* to know.)

Exercise 1-8. Rows in tables are ordered top to bottom but tuples in relations aren't; columns in tables are ordered left to right but attributes in relations aren't; tables might have duplicate rows but relations never have duplicate tuples. See the answer to Exercise 3.8 in Chapter 3 for several further differences.

Exercise 1-9. Data independence is the independence of users and application programs from the way the data is physically stored and accessed. It's a logical consequence of keeping a rigid separation between the model and its implementation. To the extent that such separation is observed, and hence to the extent that data independence is achieved, we have the freedom to make changes to the way the data is physically stored and accessed—probably for performance reasons—without at the same time having to make corresponding changes in queries and application programs. Data independence is desirable because it translates into protecting investment in training and applications.

Exercise 1-10. *No answer provided.*

Exercise 1-11. Throughout this book I use the term *relational model* to mean the abstract machine originally defined by Codd (though that abstract machine has been refined, clarified, and extended slightly since Codd's original vision). I don't use the term to mean just a relational design for some particular database. There are lots of relational models in the latter sense but only one in the former sense. (As noted in the body of the chapter, I'll have quite a bit more to say on this issue in Chapter 8.)

Exercise 1-12. Here are some:

- The relational model has nothing to say about "stored relations" at all; in particular, it doesn't say which relations are stored and which not. In fact, it doesn't say that relations have to be stored *as such* (that is, as "stored relations") at all—there might be a better way to do it (and indeed there is, a point I'll revisit briefly in Chapter 7).
- Even if we agree that the term "stored relation" might make some kind of sense—meaning a user-visible relation that's represented in storage in some direct and efficient manner, without getting too specific on just what *direct* and *efficient* might mean—which relations are "stored" should be of no significance whatsoever at the relational (user) level of the system.
- The relational model categorically does *not* say that "tables" are stored and views aren't.
- The extract quoted doesn't mention the crucial logical difference between relations and relvars.
- The extract also seems to assume that *table* and *base table* are interchangeable terms and concepts (a very serious error, in my opinion).
- The extract also seems to distinguish between tables and relations (and/or relvars). If "table" means, specifically, an SQL table, then I certainly agree there are some important distinctions to observe, but they're not the ones the extract seems to be interested in.
- "[It] is important to make a distinction between stored relations ... and virtual relations": Actually, it's extremely important from the user's perspective (and from the perspective of the relational model, come to that) *not* to make any such distinction at all!

Exercise 1-13. Here are a few things that are wrong with it:

- Of course, the relational model as such doesn't "define tables" at all, in the sense meant by the extract quoted. It doesn't even "define" *relations,* or relvars. Rather, such definitions are supplied by some *user* of the model. And anyway: What's a "simple" table? Are there any complex ones?
- What does the phrase "each relation and many-to-many relationships" mean? What does it mean to "define tables" for such things?
- The following concepts aren't part of the model, so far as I know: entities, relationships between entities, linking tables, "cross-reference keys." (It's true that the original model had a rule called "entity integrity," but that name was only a name, and in any case I don't believe in the rule.) *Note:* Of course, it's possible to put some charitable interpretations on all of these terms, but the statements that result from such interpretations are usually wrong. For example, relations do *not* always represent "entities" (what "entity" is represented by the projection of suppliers on STATUS and CITY?).

- Primary and secondary indexes and rapid access to data are all implementation notions—they're nothing to do with the model. (In particular, primary or candidate keys should *not* be equated with "primary indexes.")

- "Based upon qualifications"? Would it be possible to be a little more precise? It's truly distressing, in the relational context above all others (where precision of thought and articulation always was a key objective), to find such dreadfully sloppy phrasing. Well, yeah, you know, a relation is kind of like a table, or a kind of a table, or something ... if you know what I mean.

- What about the operators? It's an all too common error to think the relational model has to do with structure only and to forget about the operators. But the operators are crucial! As Codd himself once remarked: "Structure without operators is ... like anatomy without physiology."

Exercise 1-14. Here are some possible CREATE TABLE statements. Regarding the column data types, see the introductory section in Chapter 2.

```
CREATE TABLE S
     ( SNO    SNO         NOT NULL,
       SNAME  NAME        NOT NULL,
       STATUS INTEGER     NOT NULL,
       CITY   VARCHAR(15) NOT NULL,
       UNIQUE ( SNO ) ) ;

CREATE TABLE P
     ( PNO    PNO         NOT NULL,
       PNAME  NAME        NOT NULL,
       COLOR  COLOR       NOT NULL,
       WEIGHT WEIGHT      NOT NULL,
       CITY   VARCHAR(15) NOT NULL,
       UNIQUE ( PNO ) ) ;

CREATE TABLE SP
     ( SNO SNO NOT NULL,
       PNO PNO NOT NULL,
       QTY QTY NOT NULL,
       UNIQUE ( SNO, PNO ),
       FOREIGN KEY ( SNO ) REFERENCES S ( SNO ),
       FOREIGN KEY ( PNO ) REFERENCES P ( PNO ) ) ;
```

Note that SQL encloses both the column definitions and the key and foreign key specifications all inside the same set of parentheses (contrast this with what **Tutorial D** does). Note too that by default SQL columns permit nulls; if we want to prohibit them, therefore (and I do), we have to specify an explicit constraint to that effect. There are various ways of defining such a constraint; specifying NOT NULL as part of the column definition is probably the easiest, and is likely to be the only way available if the column is of some user-defined type.

Exercise 1-15. Tutorial D:

```
SP := SP UNION RELATION { TUPLE
       { SNO SNO('S5'), PNO PNO('P6'), QTY QTY(250) } } ;
```

The text between the keyword UNION and the closing semicolon is a *relation selector invocation* (see Chapter 3), and it denotes the relation that contains just the tuple to be inserted.

Exercise 1-16. I'll give an answer here for completeness, but I'll defer detailed explanations to Chapter 5:

```
S := WITH ( S WHERE CITY = 'Paris' ) AS R1 ,
          ( EXTEND R1 ADD ( 25 AS NEW_STATUS ) ) AS R2 ,
          R2 { ALL BUT STATUS } AS R3 :
          R3 RENAME ( NEW_STATUS AS STATUS ) ;
```

Exercise 1-17. First consider the general assignment:

```
R := rx ;
```

Here *R* is a relvar and *rx* is a relational expression, denoting the relation to be assigned to *R*. An SQL analog might look like this:

```
DELETE FROM T ;
INSERT INTO T tx ;
```

Here *T* is an SQL table corresponding to relvar *R* and *tx* is an SQL expression (perhaps a subquery) corresponding to the relational expression *rx*. Note the need for the preliminary DELETE; note too that anything could happen, loosely speaking, between that DELETE and the subsequent INSERT, whereas there's no notion in the relational case of there *being* anything "between the DELETE and the INSERT" (the assignment is an atomic operation).

Matters are somewhat simpler if the assignment in question is logically equivalent to an INSERT or a DELETE—of course, not all assignments are[1]—because then we can dispense with the preliminary DELETE. Here for example is an SQL analog of the answer to Exercise 1.15:

```
INSERT INTO SP ( SNO, PNO, QTY )
   SELECT SP.SNO, SP.PNO, SP.QTY FROM SP
   UNION
   VALUES ( SNO('S5'), PNO('P6'), QTY(250) ) ;
```

[1] Subsidiary exercise: Give one that isn't.

There's another point I need to clear up here, though. In the body of the chapter, I said that SQL doesn't support relational assignment directly, and that's true. However, one reviewer of that chapter objected that, for example, the following SQL expression could be "thought of as relational assignment" (I've simplified the reviewer's example somewhat):

```
SELECT LONDON_S.*
FROM ( SELECT S.SNO, S.SNAME, S.STATUS
       FROM   S
       WHERE  S.CITY = 'London' ) AS LONDON_S
```

In effect, the reviewer was suggesting that this expression is assigning some value to a table called LONDON_S. But of course it isn't. In particular, it won't be possible to go on and do further queries or updates on LONDON_S; LONDON_S isn't an independent table in its own right, it's just a temporary table that's conceptually materialized as part of the process of evaluating the specified expression. That expression is *not* a relational assignment.

And one further point: The SQL standard supports a variant of CREATE TABLE, "CREATE TABLE AS," that allows the base table being created to be initialized to the result of some query, thereby not only creating the table in question but also assigning an initial value to it. Once initialized, however, the table in question behaves just like any other base table; thus, CREATE TABLE AS doesn't really constitute support for relational assignment either.

Exercise 1-18. The discussions that follow are based on more extensive ones to be found in my book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004).

Regarding duplicate tuples: Essentially, the concept makes no sense. Suppose for simplicity that the suppliers relation had just two attributes, SNO and CITY, and suppose it contained a tuple showing that it's a "true fact" that supplier S1 is located in London. Then if it also contained a duplicate of that tuple, if that were possible, it would simply be informing us of that same "true fact" a second time. But (as Chapter 3 observes) if something is true, saying it twice doesn't make it *more* true! For further discussion, see the section "Why Duplicate Tuples Are Prohibited" in Chapter 3, also the paper "Double Trouble, Double Trouble" mentioned in Appendix B.

Regarding tuple ordering: The lack of such ordering means there's no such thing as "the first tuple" or "the fifth tuple" or "the 97th tuple" of a relation, and there's no such thing as "the next tuple"; in other words, there's no concept of positional addressing, and no concept of "nextness." If we did have such concepts, we would need certain additional operators as well—for example, "retrieve the *n*th tuple," "insert this new tuple *here*," "move this existing tuple from *here* to *there*," and so on. As a matter of fact (to lift some text from Chapter 8), it's axiomatic that if we have n different ways to represent information, then we need n different sets of operators. And if $n > 1$, then we have more operators to implement, document, teach, learn, remember, and use. But those extra operators add complexity, not power! There's nothing useful that can be done if $n > 1$ that can't be done if $n = 1$.

Another good argument against ordering is that positional addressing is fragile—the addresses change as insertions and deletions are performed. An analogous observation applies to the following discussion of attribute ordering also.

Regarding attribute ordering: The lack of such ordering means there's no such thing as "the first attribute" or "the second attribute" (and so on), and there's no "next attribute" (there's no concept of "nextness")—attributes are always referenced by name, never by position. As a result, the scope for errors and obscure programming is reduced. For example, there's no way to subvert the system by somehow "flopping over" from one attribute into another. This situation contrasts with that found in many programming systems, where it often is possible to exploit the physical adjacency of logically discrete items, deliberately or otherwise, in a variety of subversive ways.

Note: In the interests of accuracy, I add that for reasons that need not concern us here, relations in mathematics, unlike their counterparts in the relational model, do have a left-to-right ordering to their attributes (and likewise for tuples, of course).

2

Relations Versus Types

Exercise 2-1. A type is a finite, named set of values—*all possible* values of some specific kind: for example, all possible integers, or all possible character strings, or all possible supplier numbers, or all possible XML documents, or all possible relations with a certain heading (and so on and so forth). There's no difference between a domain and a type. *Note:* SQL does make a distinction between domains and types, however. In particular, it supports both a CREATE TYPE statement and a CREATE DOMAIN statement. To a first approximation, CREATE TYPE is SQL's counterpart to the TYPE statement of **Tutorial D,** which I'll be discussing in Chapter 6 (though there are many, many differences, not all of them trivial in nature, between the two). CREATE DOMAIN might be regarded, *very* charitably, as SQL's attempt to provide a tiny part of the total functionality of CREATE TYPE (it was added to the language in 1992, while CREATE TYPE wasn't added until 1999); now that CREATE TYPE exists, there seems to be no reason to use, or even support, CREATE DOMAIN at all.

Exercise 2-2. Every type has at least one associated selector; a selector is an operator that allows us to select, or specify, an arbitrary value of the type in question. Let *T* be a type and let *S* be a selector for *T;* then every value of type *T* must be returned by some invocation of *S,* and every invocation of *S* must return some value of type *T.* See Chapter 6 for further discussion, also the answer to Exercise 3.2 in Chapter 3. *Note:* Selectors are provided "automatically" in **Tutorial D** (since they're effectively required by the relational model) but not necessarily automatically in SQL. Further details are beyond the scope of this book.

Exercise 2-3. A THE_ operator is an operator that provides access to some component of some *possible representation* or "possrep" of some specified value of some specified type. See Chapter 6 for further discussion. *Note:* THE_ operators are effectively provided "automatically" in both **Tutorial D** and SQL, to a first approximation. Again, further details are beyond the scope of this book.

Exercise 2-4. True in principle; might not be completely true in practice (but to the extent it isn't, we're talking about a confusion over model *vs*. implementation).

Exercise 2-5. A *parameter* is a formal operand in terms of which some operator is defined. An *argument* is an actual operand, provided to be substituted for some parameter in some invocation of the operator in question. (People often use these terms as if they were interchangeable; much confusion is caused that way, and you need to be on the lookout for it.)

A *database* is a repository for data. (*Note:* Much more precise definitions are possible; one such can be found in Chapter 4 of this book.) A *DBMS* is a software system for managing databases; it provides recovery, concurrency, integrity, query/update, and other services.

A *generated* type is a type obtained by invoking some type generator such as ARRAY or RELATION; specific array and relation types are thus generated types. A *nongenerated* type is a type that's not a generated type.

A *scalar* type is a type that has no user-visible components; a *nonscalar* type is a type that's not a scalar type. Be aware, however, that these terms are neither very formal nor very precise, in the final analysis. In particular, we'll meet a couple of important relations in Chapter 3 called TABLE_DUM and TABLE_DEE that are apparently "scalar" by the foregoing definition!

Type is a model concept; types have semantics that must be understood by the user. *Representation* is an implementation concept; representations are supposed to be hidden from the user. In particular, if *X* is a value or variable of type *T*, then the operators that apply to *X* are the operators defined for *T*, not the operators defined for the representation for *T*. For example, just because the representation for type ENO ("employee numbers") happens to be CHAR, say, it doesn't follow that we can concatenate two employee numbers; we can do that only if "||" is an operator that's defined for type ENO.

A *system-defined* (or *built-in*) type is a type that's available for use as soon as the system is installed (it "comes in the same box the system comes in"). A *user-defined* type is a type whose definition and implementation are provided by some suitably skilled user(s) after the system is installed. (To the user of such a type, however—as opposed to the designer and implementer of that type—that type should look and feel just like a system-defined type, of course.)

A *system-defined* (or *built-in*) operator is an operator that's available for use as soon as the system is installed (it comes in the same box the system comes in). A *user-defined* operator is an operator whose definition and implementation are provided by some suitably skilled user(s) after the system is installed. (To the user of such an operator, however—as opposed to the designer and implementer of that operator—that operator should look and feel just like a system-defined operator, of course.) User-defined operators can be defined in terms of parameters of either user- or system-defined types (or a mixture), but system-defined operators can be defined in terms of parameters of system-defined types only.

Exercise 2-6. Coercion is implicit type conversion. It's deprecated because it's error-prone (but note that this is purely a pragmatic issue; whether coercions are permitted or not has nothing to do with the relational model).

Exercise 2-7. Because it confuses type and representation.

Exercise 2-8. A type generator is an operator that returns a type instead of a value (and is invoked at compile time instead of run time). The relational model requires support for two such: namely, TUPLE and (of course) RELATION. *Note:* Types generated by these particular type generators are nonscalar, of course, but there's no reason in principle why generated types have to be nonscalar.

Exercise 2-9. A relation is in first normal form (1NF) if and only if every tuple contains a single value, of the appropriate type, in every attribute position; in other words, *all* relations are in first normal form. Given this state of affairs, therefore, you might be forgiven for wondering why we even bother to talk about the concept at all. The reason is that (as you probably know, and as I'll be explaining in detail in Chapter 7) we can extend it to apply to relvars as well as relations, and then we can define a series of "higher" normal forms for relvars that turn out to be important in database design. In other words, 1NF is the base on which those higher normal forms build. But it really isn't all that important as a concept in itself.

Note: I should add that 1NF is one of those concepts whose definition has evolved somewhat over time. It used to be defined to mean that every tuple had to contain a single "atomic" value in every attribute position. As we've come to realize, however (and as I tried to show in the body of the chapter), the concept of data value atomicity actually has no objective meaning.

Exercise 2-10. The type of *X* is the type specified as the type of the result of the operator to be executed last—"the outermost operator"—when *X* is evaluated. That type is significant because it means *X* can be used in exactly (that is, in all and only) those positions where a literal value of type *X* can appear.

Exercise 2-11.

```
OPERATOR CUBE ( I INTEGER ) RETURNS INTEGER ;
   RETURN ( I * I * I ) ;
END OPERATOR ;
```

Exercise 2-12.

```
OPERATOR FGP ( P POINT ) RETURNS POINT ;
   RETURN ( POINT ( F ( THE_X ( P ) ) ,
                    G ( THE_Y ( P ) ) ) ) ;
END OPERATOR ;
```

Exercise 2-13. The following relation type is the type of the suppliers relvar S:

```
RELATION { SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
```

The suppliers relvar S itself is, of course, a variable of this type. And every legal value of that variable—for example, the value shown in Fig. 1.3 in Chapter 1—is a value of this type.

Exercise 2-14. SQL definitions are given in the answer to Exercise 1.14 in Chapter 1. **Tutorial D** definitions:

```
VAR P BASE RELATION
  { PNO PNO, PNAME NAME, COLOR COLOR,
                         WEIGHT WEIGHT, CITY CHAR }
    KEY { PNO } ;

VAR SP BASE RELATION
  { SNO SNO, PNO PNO, QTY QTY }
    KEY { SNO }
    FOREIGN KEY { SNO } REFERENCES S
    FOREIGN KEY { PNO } REFERENCES P ;
```

Some differences between the SQL and **Tutorial D** definitions:

- As noted in the answer to Exercise 1.14 in Chapter 1, SQL specifies keys and foreign keys, along with table columns,[2] all inside the same set of parentheses—a fact that makes it hard to determine exactly what the pertinent *type* is. (As a matter of fact, SQL doesn't really support the concept of a relation type at all, as we'll see in Chapter 3.)
- SQL's FOREIGN KEY specifications sometimes have to specify the target columns. **Tutorial D**'s never do, because keys and matching foreign keys are required to consist of exactly the same attributes, meaning in particular that the attribute names are required to be the same. See Chapter 4 for further discussion.
- SQL tables don't have to have keys at all.
- The left-to-right order in which columns are listed matters in SQL.

The significance of the fact that relvar P, for example, is of a certain relation type is as follows:

- The only values that can ever be assigned to relvar P are relations of that type.
- A reference to relvar P can appear wherever a value of that type can appear (as in, for example, the expression P JOIN SP), in which case it denotes the relation that happens to be the current value of that relvar at the pertinent time. (In other words, a *relvar reference* is a valid relational expression. See Chapter 5 for further discussion.)

Exercise 2-15. a. Valid; BOOLEAN. b. Not valid; THE_N(SNAME)||THE_N(PNAME) (I'm assuming that type NAME has a single "possrep component"—see Chapter 6—called N, of type CHAR). c. Presumably valid; QTY (I'm assuming that multiplying a quantity by an integer returns another quantity). d. Not valid; QTY+QTY(100). e. Valid; INTEGER. f. Valid; BOOLEAN. g. Not valid; THE_C(COLOR) = CITY (I'm assuming that type COLOR has a single "possrep component" called C, of type CHAR). h. Valid; CHAR.

Exercise 2-16. Such an operation logically means replacing one type by another, not "updating a type" (types aren't variables). Consider the following. First of all, the

[2] And certain other items, too, beyond the scope of the present discussion.

operation of defining a type doesn't actually create the corresponding set of values; conceptually, those values already exist, and always will exist (think of type INTEGER, for example). All the "define type" operation (the TYPE statement in **Tutorial D**—see Chapter 6) really does is introduce a *name* by which that set of values can be referenced. Likewise, dropping a type doesn't actually drop the corresponding values, it just drops the name that was introduced by the corresponding "define type" operation. It follows that "updating a type" really means dropping the type name and then reintroducing that name to refer to a different set of values. Of course, there's nothing to preclude support for some kind of "alter type" shorthand to simplify matters—and SQL does support such an operator, in fact—but using such a shorthand isn't really "updating the type."

Exercise 2-17. The empty type is certainly a valid type; however, it obviously makes no sense to define a variable to be of such a type, because no value could ever be assigned to such a variable! Despite this fact, the empty type turns out to be crucially important in connection with type inheritance—but that's a topic that's (sadly) beyond the scope of this book. See the book *Databases, Types, and the Relational Model: The Third Manifesto* (Addison-Wesley, 2006), by Hugh Darwen and myself, if you want to know more.

Exercise 2-18. Let *T* be a type for which "=" is not defined and let *C* be a column of type *T*. Then *C* can't be part of a candidate key or a foreign key, nor can it be part of the argument to DISTINCT or GROUP BY or ORDER BY, nor can restrictions or joins or unions or intersections or differences be defined in terms of it. And what about implementation constructs such as indexes? There are probably other implications as well.

Second, let *T* be a type for which the semantics of "=" are user-defined and let *C* be a column of type *T*. Then the effects of making *C* part of a candidate key or foreign key or applying DISTINCT to it (etc., etc.) will be, at best, user-defined as well.

Exercise 2-19. Here's a trivial example of such violation. Let X be the character string 'AB ' (note the trailing space), let Y be the character string 'AB', and let PAD SPACE apply to the pertinent "collation." Then the comparison X = Y gives TRUE, and yet the operator invocations CHAR_LENGTH(X) and CHAR_LENGTH(Y) give 3 and 2, respectively. I leave the detailed implications for you to think about, but it should be clear that problems are likely to surface in connection with DISTINCT, GROUP BY, and ORDER BY operations among others (as well as in connection with certain implementation constructs, such as indexes).

Exercise 2-20. Because (a) they're logically unnecessary, (b) they're error-prone, (c) end-users can't use them, (d) they're clumsy—in particular, they have a direction to them, which foreign keys in particular don't—and (e) they undermine type inheritance. (Details of the last point are beyond the scope of this book.) There might be other reasons too.

Exercise 2-21. One obvious answer has to do with nulls; if we "set X to null" (which isn't really assigning a value to X, because nulls aren't values, but never mind), the comparison X = NULL certainly doesn't give TRUE. There are other examples too, not involving reliance on nulls. Again I leave the implications for you to think about.

Exercise 2-22. No! (Which database does type INTEGER belong to?) In an important sense, the whole subject of types and type management is orthogonal to the subject of databases and database management. We might even imagine the need for a *type administrator*, whose job it would be to look after types in a manner analogous to that in which the database administrator looks after databases.

Exercise 2-23. The primary problem here (not the only one) is that the "*" stands for *all of the columns in the pertinent table, in left-to-right order*. "All of the columns" can change at any time, and "left-to-right" order can change too. So can column names—in fact, the result of a "SELECT *" operation can even include columns with no name and/or several columns with the same name. Now, such considerations *might* not be very important to an interactive end-user (on the other hand they might be), but they're very important to an application programmer who's trying to develop an application that will be in production use for a long period of time. Code defensively!

Here are some further examples of SQL constructs that rely on left-to-right order of columns:

- VALUES (this is SQL's "table value constructor")—including in particular the case of VALUES in an INSERT statement that omits the target column names
- UNION, INTERSECT, EXCEPT (without CORRESPONDING in each case)
- CREATE VIEW (if column names are specified)
- Row comparisons and assignments
- IN (of rows and tables)

Exercise 2-24. An expression denotes a value; it can be thought of as a rule for computing or determining the value in question. A statement doesn't denote a value; instead, it causes some action to occur, such as assigning a value to some variable or changing the flow of control.

3

Tuples and Relations

Exercise 3-1. See the body of the chapter.

Exercise 3-2. A literal is a symbol that denotes a value that's fixed and determined by the particular symbol in question (and the type of that value is also fixed and determined by the symbol in question—necessarily so). Loosely, a literal is *self-defining.* Here are some **Tutorial D** examples:

```
4                           /* a literal of type INTEGER  */
'XYZ'                       /* a literal of type CHAR     */
FALSE                       /* a literal of type BOOLEAN  */
SNO('S1')                   /* a literal of type SNO      */
POINT(5.0,2.5)              /* a literal of type POINT    */
```

Actually, literals are selector invocations; to be precise, a literal is a selector invocation in which the arguments are themselves all literals in turn (implying in particular that a selector invocation with no arguments at all, like the INTEGER selector invocation 95, is a literal by definition). Every value of every type must be denotable by means of some literal.

Exercise 3-3. Two tuples are equal if and only if they're the very same tuple! For a detailed definition, see the body of the chapter.

Exercise 3-4. *No answer provided.*

Exercise 3-5. Tutorial D tuple selector invocations:

```
TUPLE { PNO PNO('P1'), PNAME NAME('Nut'), COLOR('Red'),
                       WEIGHT WEIGHT(12.0), CITY 'London' }

TUPLE { SNO SNO('S1'), PNO PNO('P1'), QTY QTY(300) }
```


SQL analogs:

```
ROW ( PNO('P1'), NAME('Nut'), COLOR('Red'),
                              WEIGHT(12.0), 'London' )

ROW ( SNO('S1'), PNO('P1'), QTY(300) )
```

Observe (a) the lack of column names,[3] (b) the reliance on left-to-right ordering, in these SQL expressions.

Exercise 3-6. The following selector invocation denotes a relation of two tuples:

```
RELATION
 { TUPLE { SNO SNO('S1'), PNO PNO('P1'), QTY QTY(300) } ,
   TUPLE { SNO SNO('S1'), PNO PNO('P2'), QTY QTY(200) } }
```

SQL analog:

```
VALUES ROW ( SNO('S1'), PNO('P1'), QTY(300) ) ,
       ROW ( SNO('S1'), PNO('P2'), QTY(200) )
```

By the way, the fact that there are no parentheses enclosing the two "row value constructors" here is *not* an error. In fact, the following SQL expression—

```
VALUES ( ROW ( SNO('S1'), PNO('P1'), QTY(300) ) ,
         ROW ( SNO('S1'), PNO('P2'), QTY(200) ) )
```

(which is certainly legal, syntactically speaking)—denotes something entirely different! See Exercise 3.21.

Exercise 3-7. Tuple literals and relation literals both make sense (of course); in fact, the tuple selector invocations in the answer to Exercise 3.5 are tuple literals, and the relation selector invocation in the answer to Exercise 3.6 is a relation literal. See the last sentence in the answer to Exercise 3.2.

Exercise 3-8. The list that follows is based on one in my book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004).

- Each attribute in the heading of a relation involves a type name, but those type names are usually omitted from tables (where by *tables* I mean tabular pictures of relations).
- Each component of each tuple in the body of a relation involves a type name and an attribute name, but those type and attribute names are usually omitted from tabular pictures.
- Each attribute value in each tuple in the body of a relation is a value of the applicable type, but those values are usually shown in some abbreviated form—for example, S1 instead of SNO('S1')—in tabular pictures.

[3] Oddly enough, the standard SQL term in this particular context is actually not *columns* but *fields*.

- The columns of a table have a left-to-right ordering, but the attributes of a relation don't. *Note:* One implication of this point is that columns can have duplicate names, or even no names at all. For example, consider the SQL expression

  ```
  SELECT DISTINCT S.CITY, S.STATUS * 2, P.CITY
  FROM   S, P
  ```

  What are the column names in the result of this expression?

- The rows of a table have a top-to-bottom ordering, but the tuples of a relation don't.
- A table might contain duplicate rows, but a relation never contains duplicate tuples.
- Tables are usually regarded as having at least one column, while relations are not required to have at least one attribute (see the section "TABLE_DUM and TABLE_DEE" in the body of the chapter).
- Tables (at least in SQL) are allowed to include nulls, but relations certainly aren't.
- Tables are "flat" or two-dimensional, but relations are *n*-dimensional.

Exercise 3-9. One exception is that no database relation can have an attribute of any pointer type—where a pointer type is any type that has associated *referencing* and *dereferencing* operators. (In case you're not familiar with these terms, here are some rough definitions: (a) Given a variable *V*, the referencing operator applied to *V* returns a pointer to *V;* (b) given a value *v* of type pointer, the dereferencing operator applied to *v* returns the variable that *v* points to.)

The other exception is a little harder to state, but what it boils down to is that if relation *r* has heading {*H*}, then no attribute of *r* can be defined in terms of a tuple or relation type with that same heading {*H*} (at any level of nesting).

A tuple with a tuple-valued attribute:

```
TUPLE { PRESIDENT 'Superman',
        ADDR TUPLE { STREET '1600 Pennsylvania Ave.',
                     CITY   'Washington',
                     STATE  'DC',
                     ZIP    '20500' } }
```

A tuple with a relation-valued attribute (you might find it helpful to draw a picture here):

```
TUPLE { SNO SNO('S2'),
        PNO_REL RELATION { TUPLE { PNO PNO('P1') } ,
                           TUPLE { PNO PNO('P2') } } }
```

Exercise 3-10. See Chapter 5.

Exercise 3-11. The operators "<" and ">" don't apply to tuples because tuples are sets, and the notion of some set *S1* being somehow "less than" or "greater than" some other set *S2* makes no sense. By contrast, SQL rows aren't sets but *sequences*. As a consequence, it's possible to define comparison operators—with some difficulty[4]—for SQL rows as follows. Let the rows to be compared be *a* and *b*. Then *a* and *b* must be of the same degree (*n*, say). Let the *i*th components of *a* and *b* be *ai* and *bi*, respectively (*i* = 1, 2, ..., *n*). Then:

- *a* = *b* is TRUE if and only if for all *i*, *ai* = *bi* is TRUE.
- *a* <> *b* is TRUE if and only if there exists some *i* such that *ai* <> *bi* is TRUE.
- *a* < *b* is TRUE if and only if there exists some *j* such that *aj* < *bj* is TRUE and for all *i* < *j*, *ai* = *bi* is TRUE.
- *a* > *b* is TRUE if and only if there exists some *j* such that *aj* > *bj* is TRUE and for all *i* < *j*, *ai* = *bi* is TRUE.
- *a* <= *b* is TRUE if and only if "*a* < *b*" is TRUE or *a* = *b* is TRUE.
- *a* >= *b* is TRUE if and only if *a* > *b* is TRUE or *a* = *b* is TRUE.
- *a* = *b* is FALSE if and only if *a* <> *b* is TRUE.
- *a* <> *b* is FALSE if and only if *a* = *b* is TRUE.
- *a* < *b* is FALSE if and only if *a* >= *b* is TRUE.
- *a* > *b* is FALSE if and only if *a* <= *b* is TRUE.
- *a* <= *b* is FALSE if and only if *a* > *b* is TRUE.
- *a* >= *b* is FALSE if and only if *a* < *b* is TRUE.

And if *op* stands for any of =, <>, <, <=, >, >=, then:

- *a op b* is UNKNOWN if and only if it's not TRUE and not FALSE.

[4] I speak advisedly here; the SQL standards committee took several iterations to get these definitions correct. You might want to try experimenting with some sample values of your own to see how the definitions work out in practice.

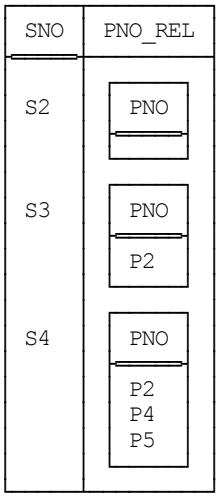
Exercise 3-12. For a relation with one RVA, see relation R4 in Fig. 2.2 in Chapter 2; for an equivalent relation with no RVA, see relation R1 in Fig. 2.1 in Chapter 2. Here's one with two RVAs:

CNO	TEACHER	TEXT
C1	TNO T2 T3 T5	XNO X1 X2
C2	TNO T4	TNO X2 X4 X5

The intended meaning is: Course CNO can be taught by every teacher TNO in TEACHER and uses every textbook XNO in TEXT. Here's a relation without RVAs that conveys the same information:

CNO	TNO	XNO
C1	T2	X1
C1	T2	X2
C1	T3	X1
C1	T3	X2
C1	T5	X1
C1	T5	X2
C2	T4	X2
C2	T4	X4
C2	T4	X5

As for a relation with an RVA such that there's no relation without an RVA that represents precisely the same information, one simple example can be obtained from Fig. 2.2 in Chapter 2 by just replacing the PNO_REL value for (say) supplier S2 by an empty relation:



However, it isn't necessary to invoke the notion of an empty relation in order to come up with an example of a relation with an RVA such that there's no relation without an RVA that represents precisely the same information. (*Subsidiary exercise:* Justify this remark! If you give up, take a look at the answer to Exercise 7.14.)

Note: Perhaps I should elaborate on what it means for two relations to represent the same information. Basically, relations *r1* and *r2* represent the same information if and only if it's possible to map *r1* into *r2* and *vice versa* by means of operations of the relational algebra, without introducing any additional information into the mapping in either direction. With reference to relations R4 in Fig. 2.2 in Chapter 2 and R1 in Fig. 2.1 in Chapter 2, for example, we have:

```
R4 = R1 GROUP ( { PNO } AS PNO_REL )

R1 = R4 UNGROUP ( PNO_REL )
```

These two relations thus do represent the same information. See Chapter 5 for further discussion of the GROUP and UNGROUP operators.

Exercise 3-13. In some ways a tuple does resemble a record and an attribute a field—but these resemblances are only approximate. A relvar shouldn't be regarded as "just a file," but rather as a *disciplined* file. The discipline in question is one that results in a considerable simplification in the structure of the data as seen by the user, and hence in a corresponding simplification in the operators needed to deal with that data, and indeed in the user interface in general. *Note:* In fact, I'm going to argue in Chapter 4 (in the section "Relvars and Predicates") that in some ways it's very misleading to think of a relvar as a file at all—and so the present answer should be taken with a considerable pinch of salt.

Exercise 3-14. TABLE_DEE and TABLE_DUM (DEE and DUM for short) are the only relations with no attributes; DEE contains exactly one tuple (the 0-tuple), DUM contains no tuples at all. SQL doesn't support them, because tables in SQL are always required to have at least one column. (As a consequence, SQL's version of the relational algebra is like an arithmetic that has no zero.)

Exercise 3-15. Here are the rules: Let *x* be an SQL row. Suppose for simplicity that *x* has just two components, *x1* and *x2* (in left-to-right order, of course!). Then *x* IS NULL is defined to be equivalent to *x1* IS NULL AND *x2* IS NULL, and *x* IS NOT NULL is defined to be equivalent to *x1* IS NOT NULL AND *x2* IS NOT NULL. It follows that the given row is neither null nor nonnull ... What do you conclude from this state of affairs?

By the way: At least one reviewer of the first draft of this book commented here that he'd never thought of a row being null. But rows are *values* (just as tuples and relations are values), and hence the idea of some row being unknown makes at least as much sense as, say, the idea of some salary being unknown. Thus, if the concept of representing an unknown value by a "null" makes any sense at all—which of course I don't think it does—then it surely applies to rows (and tables, and any other kind of value you can think of) just as much as it applies to scalars. And as this exercise demonstrates, SQL tries to support this position (in the case of rows, at least), but fails.

Exercise 3-16. To deal with this argument properly would take more space than we have here, but it all boils down to what's sometimes called *The Principle of Identity of Indiscernibles* (see Chapter 8). Let *a* and *b* be any two entities—for example, two pennies. Then, if there's *no way whatsoever* of distinguishing between *a* and *b*, there aren't two entities but only one. Of course, it might be true for certain purposes that the two entities can be *interchanged*, but that fact isn't sufficient to make them *indiscernible* (there's a logical difference between interchangeability and indiscernibility, and arguments to the effect that "duplicates occur naturally in the real world" tend to be based on a muddle over this difference). A detailed analysis of this whole issue can be found in the paper "Double Trouble, Double Trouble" (see Appendix B).

Exercise 3-17. The database of Fig. 3.1 is nonrelational because each of the tables contains duplicate rows and relations don't contain duplicate tuples. The database of Fig. 3.2 is nonrelational because one of the tables contains a row that "contains a null"—if you'll allow me to talk in this dreadfully sloppy way for just a moment[5]—and "tuples" that "contain nulls" aren't tuples.

Exercise 3-18. The question was: Do you think nulls occur naturally in the real world? Only you can answer this question—but if your answer is *yes*, I think you should examine your reasoning very carefully. For example, consider the statement "Joe's salary is $50,000." That statement *is* either true or false. Now, I might not know whether it's true or false; but my not knowing has nothing to do with whether it actually is true or false. In particular, my not knowing is certainly *not* the same as saying Joe's salary is null! "Joe's

[5] Given that nulls aren't values, talk of rows "containing" such things actually makes no sense. It's hard to talk coherently about things that make no sense. For example, if "the EMP row" for employee Joe contains nothing at all in the SALARY position, then that "EMP row" simply isn't an EMP row, by definition (nor is it any other kind of row).

salary is $50,000" is a statement about the real world. "Joe's salary is null" is a statement about my *knowledge* (or lack of knowledge, rather) about the real world. We shouldn't keep a mixture of these two very different kinds of statements in the same relation, or in the same relvar! *Note:* The discussion of predicates in the next chapter should give you further food for thought in this connection.

Suppose we had to represent the fact that we don't know Joe's salary, say on some paper form. Would we enter a null into that form? I don't think so. Rather, we would leave the box blank, or put a question mark, or write "unknown," or something along those lines. And of course that blank (or question mark, or "unknown," or whatever) is a *value*, not a null. Myself, therefore, I *don't* believe that nulls occur naturally in the real world.

Exercise 3-19. The relational model as such has little to say at the time of writing on how best to handle missing information (note that the issue is at least partly a database design issue anyway, not an issue of the relational model). A full discussion is thus beyond the scope of this book and, *a fortiori*, of these answers; see Hugh Darwen's presentation "How to Handle Missing Information Without Using Nulls" (*www.thethirdmanifesto.com*, May 9th, 2003) for some suggestions. See also Chapter 7 of the present book for a very brief comment on the subject.

Exercise 3-20. Can we dispense with type BOOLEAN? Probably not, though the question might deserve more study. Certainly it would be a little odd if all boolean expressions suddenly became relational expressions, and host languages thus suddenly all had to support relational data types. And there might be problems of circularity of definition if we tried, for reasons of user-friendliness, to make BOOLEAN a user-defined type (or a system-defined type, come to that) that's defined in terms of TABLE_DEE and TABLE_DUM.

Would it make sense to define a relvar of degree zero? It's hard but not impossible to imagine a situation in which such a relvar might be useful—but that's not the point. Rather, the point is that the system shouldn't include a prohibition against defining such a relvar. If it does, then that fact constitutes a violation of orthogonality, and such violations always come back to bite us eventually.

Exercise 3-21. It denotes a single row with four unnamed fields (those fields having values that are themselves rows in turn, each with two unnamed fields).

4

Relation Variables

Exercise 4-1. Loosely, the remark means "Update the STATUS attribute in tuples for suppliers in London." But tuples (and, *a fortiori*, attribute values within tuples) are values and simply can't be updated, by definition. Here's a more precise version of the remark:

- Let relation *s* be the current value of relvar S.
- Let *ls* be that restriction of *s* for which the CITY value is London.
- Let *ls′* be that relation that's identical to *ls* except that the STATUS value in each tuple is as specified in the given UPDATE operation.
- Let *s′* be the relation denoted by the expression (*s* MINUS *ls*) UNION *ls′*.
- Then *s′* is assigned to S.

For further discussion, see the section "Updating Is Set-at-a-Time" in the body of the chapter.

Exercise 4-2. Because relational operations are fundamentally set-at-a-time and SQL's "positioned update" operations are fundamentally tuple-at-a-time. Although set-at-a-time operations for which the set in question is of cardinality one are sometimes (perhaps even frequently) acceptable, they can't *always* work. In particular, tuple-level update operations might work for a while and then cease to work when integrity constraint support is improved.

Exercise 4-3.

```
CREATE TABLE TAX_BRACKET
  ( LOW        MONEY   NOT NULL,
    HIGH       MONEY   NOT NULL,
    PERCENTAGE INTEGER NOT NULL,
    UNIQUE ( LOW ),
    UNIQUE ( HIGH ),
```



```
    UNIQUE ( PERCENTAGE ) ) ;

CREATE TABLE ROSTER
  ( DAY   DAY_OF_WEEK NOT NULL,
    HOUR  TIME_OF_DAY NOT NULL,
    GATE  GATE        NOT NULL,
    PILOT NAME        NOT NULL,
    UNIQUE ( DAY, HOUR, GATE ),
    UNIQUE ( DAY, HOUR, PILOT ) ) ;

CREATE TABLE MARRIAGE
  ( SPOUSE_A         NAME NOT NULL,
    SPOUSE_B         NAME NOT NULL,
    DATE_OF_MARRIAGE DATE NOT NULL,
    UNIQUE ( SPOUSE_A, DATE_OF_MARRIAGE ),
    UNIQUE ( DATE_OF_MARRIAGE, SPOUSE_B ),
    UNIQUE ( SPOUSE_B, SPOUSE_A ) ) ;
```

Exercise 4-4. Because keys represent constraints and constraints apply to variables, not values. (That said, it's certainly possible, and sometimes useful, to think of subset *k* of the heading of relation *r* as a key for *r* if it's unique and irreducible with respect to the tuples of *r*. But thinking this way is strictly incorrect, and certainly much less useful than thinking about keys for relvars rather than relations.)

Exercise 4-5. Here's one: Suppose relvar *A* has a *reducible* "key" consisting of attributes *K* and *X*, say, where *K* is in fact a genuine irreducible key. Then relvar *A* satisfies the functional dependency $K \rightarrow X$. Suppose now that relvar *B* has a foreign key {*K*,*X*} referencing that "reducible key" in *A*. Then *B* too satisfies the functional dependency $K \rightarrow X$, and it's therefore likely that *B* is not in Boyce/Codd normal form (see Chapter 7).

Exercise 4-6. Keys are *sets* of attributes—in fact, every key is a subset of the pertinent heading—and key values are thus tuples by definition, even when the tuples in question have exactly one attribute. Thus, for example, the key for the parts relvar P is {PNO} and not just PNO, and the key value for the parts tuple for part P1 is TUPLE {PNO PNO('P1')} and not just PNO('P1')—and certainly not just 'P1' or P1.

Exercise 4-7. Let *m* be the largest 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) *n* is odd and every distinct set of *m*-1 attributes is a key. Either way, it follows that the maximum number of keys in *R* is:

```
n! / ( m! * (n - m)! )
```

Note: The expression *n*! is read "*n* factorial" and is defined as the product *n* * (*n*-1) * ... * 2 * 1. Relvars TAX_BRACKET and MARRIAGE (see Exercise 4.3) are both examples of relvars with the maximum possible number of keys; so is any relvar of degree zero. (If *n* = 0, the formula becomes 0!/(0!*0!), and 0! is 1. See Exercise 4.29.)

Exercise 4-8. Sample data:

EMP

ENO	MNO
E4	E2
E3	E2
E2	E1
E1	E1

I'm using the trick here of pretending that a certain employee (namely, employee E1) acts as his or her own manager, which is one way of avoiding the use of nulls in this kind of situation. Another and probably better approach to the problem is to separate the reporting-structure relationships out into a relvar of their own, excluding from that relvar any employee who has no manager:

EMP

ENO	...
E4	...
E3	...
E2	...
E1	...

EM

ENO	MNO
E4	E2
E3	E2
E2	E1

Subsidiary exercise: What are the predicates for relvars EM and the two versions of EMP here? (Thinking carefully about this exercise should serve to reinforce my suggestion that the second design is preferable. See also the brief discussion of missing information in Chapter 7 and the answer to Exercise 3.18 in Chapter 3.)

Exercise 4-9. The answer to this question is slightly nontrivial. First, the column names mentioned in an SQL FOREIGN KEY specification must be identical to those mentioned in the target UNIQUE specification (despite the fact that column-to-column matching is based on ordinal position and *not* on those column names as such). However, if and only if the target UNIQUE specification is formulated using the PRIMARY KEY variant, then the column names can be omitted from the referencing FOREIGN KEY specification and column names identical to those in that PRIMARY KEY specification are then assumed by default. As a consequence, the following is an SQL analog of the definition for relvar EMP from Exercise 4.8:

```
CREATE TABLE EMP
  ( ENO ENO NOT NULL, ..., MNO ENO NOT NULL, ...,
    PRIMARY KEY ( ENO ),
    FOREIGN KEY ( MNO ) REFERENCES EMP ) ;
```

By contrast, the following fails on a syntax error:

```
CREATE TABLE EMP
  ( ENO ENO NOT NULL, ..., MNO ENO NOT NULL, ...,
    UNIQUE ( ENO ),
    FOREIGN KEY ( MNO ) REFERENCES EMP ) ;
```


Exercise 4-10. Note that (by definition) such a situation must represent a one-to-one relationship. Thus, one obvious case arises if we decide to split some relvar "vertically," as in the following example (suppliers):

```
VAR SNT BASE RELATION
  { SNO SNO, SNAME NAME, STATUS INTEGER }
    KEY { SNO }
    FOREIGN KEY { SNO } REFERENCES SC ;

VAR SC BASE RELATION
  { SNO SNO, CITY CHAR }
    KEY { SNO }
    FOREIGN KEY { SNO } REFERENCES SNT ;
```

Subsidiary exercise: What are some of the implications of such a vertical split? *Hint:* See the discussion of *multiple assignment* in Chapter 6.

Exercise 4-11. It's obviously not possible to give a definitive answer to this exercise. I'll just mention the referential actions supported by the standard, which are NO ACTION (the default), CASCADE, RESTRICT, SET DEFAULT, and SET NULL. *Subsidiary exercise:* What's the difference between NO ACTION and RESTRICT?

Exercise 4-12. Triggered procedures do have certain pragmatic uses. However, they are of course procedural, not declarative, in nature, and I'm tempted to say that if you have to resort to using a triggered procedure for some purpose, it simply means the vendor hasn't done a good job of solving the corresponding problem declaratively. Integrity constraints in particular should *not* have to be implemented via triggered procedures. Note too that triggered procedures (especially the so-called INSTEAD OF triggers supported by some products, though not currently by the SQL standard) will often lead to a violation of *The Assignment Principle* and are thus somewhat suspect for that very reason.

Exercise 4-13. Substituting the view definition for the view reference in the outer FROM clause, we obtain:

```
SELECT DISTINCT LSSP.STATUS, LSSP.QTY
FROM ( SELECT S.SNO, S.SNAME, S.STATUS, SP.PNO, SP.QTY
       FROM   S, SP
       WHERE  S.SNO = SP.SNO
       AND    S.CITY = 'London' ) AS LSSP
WHERE  LSSP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'London' )
```

This simplifies to:

```
SELECT DISTINCT S.STATUS, SP.QTY
FROM   S, SP
WHERE  S.SNO = SP.SNO
AND    S.CITY = 'London'
AND    SP.PNO IN
     ( SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'London' )
```

Exercise 4-14. The sole key is {SNO,PNO}.

Exercise 4-15. *No answer provided*—except to note that if it's difficult to answer the question precisely for some product, then that very fact is part of the point of the exercise in the first place.

Exercise 4-16. As for the previous exercise (but more so, probably).

Exercise 4-17. Here are a couple of trivial examples (simplified notation):

Part a.

```
S { SNO, SNAME, STATUS, CITY }            /* join */
```

vs.

```
SS { SNO, SNAME }                         /* projections */
ST { SNO, STATUS }
SC { SNO, CITY }
```

Part b.

```
S { SNO, SNAME, STATUS, CITY }            /* union */
```

vs.

```
EUROPEANS { SNO, SNAME, STATUS, CITY }    /* restrictions */
AMERICANS { SNO, SNAME, STATUS, CITY }
```

In part b. here, EUROPEANS is a restriction of S where the city is a European city, AMERICANS is a restriction of S where the city is an American city. Their union is a disjoint union.

Exercise 4-18. *No answer provided.*

Exercise 4-19. For the distinction, see the body of the chapter. SQL doesn't support snapshots at the time of writing. (It does support CREATE TABLE AS—see the answer to Exercise 1.17 in Chapter 1—which allows a base table to be initialized when it's created, but CREATE TABLE AS has no REFRESH option.)

Exercise 4-20. "Materialized view" is a deprecated term for a snapshot. The term is deprecated because it muddies concepts that ought to be distinct—by definition, views simply *aren't* materialized, so far as the model is concerned—and it's leading us into a situation in which we no longer have a clear term for a concept we *did* have a clear term for, originally. The term should be firmly resisted.[6] In fact, I'm tempted to go further; it seems to me that people who advocate use of the term "materialized view" are betraying their lack of understanding of the relational model and of the importance of the distinction between model and implementation.

Exercise 4-21. An extensive discussion of these concepts, with examples, can be found in Appendix A.

---

[6] I realize I've probably already lost this battle, but I'm an eternal optimist.

Exercise 4-22. Relvar P: *Part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY.* Relvar SP: *Supplier SNO supplies part PNO in quantity QTY.*

Exercise 4-23. The intension of relvar *R* is the intended interpretation of *R*. The extension of relvar *R* at a given time is the set of tuples appearing in *R* at that time.

Exercise 4-24. *No answer provided.*

Exercise 4-25. LS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in London.* NLS: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY (which isn't London).* Whether LS and NLS are views or base relvars makes no difference.

Exercise 4-26. *Supplier SNO is under contract, is named SNAME, has status STATUS, is located in city CITY, and supplies part PNO in quantity QTY.*

Exercise 4-27. See the body of the chapter.

Exercise 4-28. Yes! To say that relvar *R* has an empty key is to say that *R* must never contain more than one tuple (because every tuple has the same value for the empty set of attributes—namely, the empty tuple; thus, if *R* contained two or more tuples, we would have a key uniqueness violation on our hands). And constraining *R* never to contain more than one tuple could certainly be useful. I'll leave finding an example of such a situation as a subsidiary exercise.

Exercise 4-29. The question certainly makes sense insofar as *every* relvar does have an associated predicate. However, just what the predicate is for some given relvar is in the mind of the definer of that relvar (and in the user's mind too, we hope). For example, if I define a relvar C as follows—

```
VAR C BASE RELATION { CITY CHAR } KEY { CITY } ;
```

—the corresponding predicate might be almost anything! It might, for example, be *CITY is a city in California;* or *CITY is a city in which at least one supplier is located;* or *CITY is a city that's the capital of some country;*[7] and so on. In the same way, the predicate for a relvar of degree zero—

```
VAR Z BASE RELATION { } KEY { } ;
```

—might also be "almost anything," except that (since the relvar has no attributes and the corresponding predicate therefore has no parameters) the predicate in question must in fact degenerate to a proposition and must therefore evaluate unequivocally either to TRUE or FALSE. (In fact, it will evaluate to TRUE if the value of Z is TABLE_DEE and FALSE if the value is TABLE_DUM.)

[7] Or even *CITY is the name of somebody's favorite teddy bear.* There's nothing in the relvar definition to say that CITY has to denote a city.

By the way, observe that relvar Z has an empty key. It's obvious that every degree-zero relvar must necessarily have an empty key; however, you shouldn't conclude that degree-zero relvars are the only ones with empty keys (see the previous exercise).

Exercise 4-30. Of course not. In fact, "most" relations aren't values of some relvar. As a trivial example, the relation denoted by S{CITY}, the projection of S on CITY, isn't a value of any relvar in the suppliers-and-parts database. Note, therefore, that throughout this book, when I talk about some relation, I *don't* necessarily mean a relation that's the value of some relvar.

5

Relational Algebra

Exercise 5-1. a. The result has duplicate column names (as well as left-to-right column ordering). b. The result has left-to-right column ordering. c. The result has an unnamed column (as well as left-to-right column ordering). d. Nothing, though it wouldn't hurt to specify DISTINCT. e. The result has duplicate rows, in general. f. Nothing, but it's not a relational expression.

Exercise 5-2. No! In particular, certain relational divides that you might expect to fail don't. Here are some examples:

a. Let relation PZ be of type RELATION {PNO PNO} and let its body be empty. Then the expression

```
SP { SNO, PNO } DIVIDEBY PZ { PNO }
```

reduces to the projection SP{SNO} of SP on SNO.

b. Let *z* be either TABLE_DEE or TABLE_DUM. Then the expression

```
r DIVIDEBY z
```

reduces to

```
r JOIN z.
```

c. Let relations *r* and *s* be of the same type. Then the expression

```
r DIVIDEBY s
```

gives TABLE_DEE if *r* is nonempty and every tuple of *s* appears in *r*, TABLE_DUM otherwise.

d. Finally, *r* DIVIDEBY *r* gives TABLE_DUM if *r* is empty, TABLE_DEE otherwise.

Exercise 5-3. The trap is that the join involves the CITY attributes as well as the SNO and PNO attributes. The result looks like this:

SNO	SNAME	STATUS	CITY	PNO	QTY	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12.0
S1	Smith	20	London	P4	200	Screw	Red	14.0
S1	Smith	20	London	P6	100	Cog	Red	19.0
S2	Jones	10	Paris	P2	400	Bolt	Green	17.0
S3	Blake	30	Paris	P2	200	Bolt	Green	17.0
S4	Clark	20	London	P4	200	Screw	Red	14.0

The predicate is: *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY; part PNO is used in the enterprise, is named PNAME, has color COLOR and weight WEIGHT, and is stored in city CITY; and supplier SNO supplies part PNO in quantity QTY.* Note that both appearances of SNO refer to the same parameter, as do both appearances of PNO and both appearances of CITY.

This example raises a significant issue, however, that deserves extended discussion here. Consider again this text from the body of the chapter:

> [In] operations like UNION or JOIN that need some correspondence to be established between operand attributes, **Tutorial D** does so by requiring the attributes in question *to have the same names* (as well as, necessarily, the same types). For example, here's a **Tutorial D** expression for the join of parts and suppliers on cities:
>
> ```
> P JOIN S
> ```
>
> By definition, this join is performed on the basis of part and supplier cities, P and S having just the CITY attribute in common.

Several reviewers objected that this reliance on matching attribute names could be dangerous. The following comment is typical (though I've edited the original somewhat):

> Suppose I add a CHANGE_DATE attribute to each of P and S; won't P JOIN S now give suppliers and parts with both the same city *and* the same date of change? That's unlikely to be a result that any developer would want. For that matter, what if the two name attributes were both called just NAME? That's a very likely scenario in real life. How do you deal with these issues in the relational algebra?

Well, the question of the two name attributes having the same name is handled by means of RENAME, of course. But the other question is deeper. The root of the problem is that —at least in today's mainstream SQL products—application programs are allowed to see the database "as it really is." And *that* fact can be traced back to a fundamental misunderstanding in the database community in the early 1970s (possibly even earlier). The perception at that time was that, in order to achieve data independence, it was necessary to move the database definition out of the application so that (in principle) that definition could be changed later without having to change the application. But of course that perception was incorrect! What was and still is really needed is *two separate definitions*, one inside the application and one outside; the one inside would represent the application's view of the database (and would allow for the necessary compile-time syntax checking and so forth), the one outside would represent the database "as it really was." Then, if it became necessary to change the definition of the database "as it really was,"

data independence would be preserved by changing the *mapping* between the two definitions.

Coming back to the example of P JOIN S: The foregoing strategy would allow us to add a CHANGE_DATE attribute to relvars P and S in the database *without* having to add such an attribute to the application's perception of those two relvars—and, of course, expressions such as P JOIN S are interpreted in terms of that application perception, not in terms of the database "as it really is." *Ergo,* no problem.

By the way, I used the term *view* in the foregoing explanation ("the application's view") advisedly. The fact is, views as conventionally understood should have been sufficient to solve the problem under discussion. But the trouble with views is that a view definition specifies both the application's perception of some portion of the database *and* the mapping between that perception and the database "as it really is." In order to achieve the kind of data independence I'm talking about, those two specifications need to be kept separate.

Exercise 5-4. In 2-dimensional cartesian geometry, (*x*,0) and (0,*y*) are the projections of the point (*x*,*y*) on to the X axis and the Y axis, respectively; equivalently, (*x*) and (*y*) are the projections of the point (*x*,*y*) in 2-dimensional space into certain 1-dimensional spaces. These notions are readily generalizable to *n* dimensions. (Recall from Chapter 3 that relations are indeed *n*-dimensional.)

Exercise 5-5.

a. Suppliers who supply part P2:

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

b. Parts not supplied by supplier S2:

PNO	PNAME	COLOR	WEIGHT	CITY
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

c. Cities in which at least one supplier is located but no parts are:

CITY
Athens

d. Supplier-number / part-number pairs for suppliers and parts in the same city:

SNO	PNO
S1	P1
S1	P4
S1	P6
S2	P2
S2	P5
S3	P2
S3	P5
S4	P1
S4	P4
S4	P6

e. All supplier-city / part-city pairs:

SC	PC
London	London
London	Paris
London	Oslo
Paris	London
Paris	Paris
Paris	Oslo
Athens	London
Athens	Paris
Athens	Oslo

Exercise 5-6. Intersection and cartesian product are both special cases of join, so we can ignore them here. The fact that union and join are commutative is obvious from the fact that the definitions are symmetric in the two relations concerned. I now show that union is associative. Let *t* be a tuple. Using "≡" to stand for "if and only if" and "∈" (as usual) to stand for "appears in," we have:

```
t ∈ r UNION (s UNION u)  ≡  t ∈ r OR t ∈ (s UNION u)
                         ≡  t ∈ r OR (t ∈ s OR t ∈ u)
                         ≡  (t ∈ r OR t ∈ s) OR t ∈ u
                         ≡  t ∈ (r UNION s) OR t ∈ u
                         ≡  t ∈ (r UNION s) UNION u
```

Note the appeal in the third line to the associativity of OR. The proof that join and semijoin are associative is analogous. Finally, it's obvious that semijoin isn't commutative, because the final step in evaluating *r* SEMIJOIN *s* involves projection on the attributes of *r;* hence, *r* SEMIJOIN *s* and *s* SEMIJOIN *r* are different, in general (except as indicated in the answer to the exercise immediately following).

Exercise 5-7. The expressions *r* SEMIJOIN *s* and *s* SEMIJOIN *r* are equivalent if and only if *r* and *s* are of the same type, in which case both expressions reduce to just JOIN{*r*,*s*}, which reduces in turn to INTERSECT{*r*,*s*}.

Exercise 5-8. The issue is slightly debatable, but I think it can be argued that *all* of the relational operators rely in some fashion on tuple equality.

Exercise 5-9. The cartesian product of a single table *t* is defined to be just *t*. But the question of what the product of *t1* and *t2* is if *t1* and *t2* both contain duplicate rows is a tricky one. The SQL standard says it's "the multiset of all rows *r* such that *r* is the concatenation of a row from each of the identified tables." But that multiset isn't well-defined!—even though the standard says "the cardinality of [the multiset] is the product of the cardinalities of the identified tables." Consider the tables T1 and T2 shown below:

T1

C1
0
0

T2

C2
1
2

Either of the following fits the foregoing definition for "the" multiset (that is, either one could be "the" multiset referred to):

CP1

C1	C2
0	1
0	1
0	2
0	2

CP2

C1	C2
0	1
0	2
0	2
0	2

In my opinion, moreover, the only way to resolve the ambiguity is by defining a mapping from each of the (multiset) argument tables to a proper *set*, and likewise defining a mapping of the (multiset) result table—in other words, the desired product—to a proper *set*. In other words, this discussion serves to emphasize the fact that one of the most fundamental concepts in the entire SQL language (namely, the concept that tables should permit duplicate rows) is *fundamentally flawed*, and cannot be repaired without, in effect, dispensing with the concept altogether.

Exercise 5-10. Rename isn't primitive because (for example) the expressions

```
S RENAME ( CITY AS SCITY )
```

and

```
( EXTEND S ADD ( CITY AS SCITY ) ) { ALL BUT CITY }
```

are obviously equivalent.

Exercise 5-11. An aggregate operator is a scalar operator (it returns a scalar value).[8] A summary is just an operand to SUMMARIZE—it doesn't "return" anything at all. It's true it might be thought of as "returning" one scalar value for each tuple in the PER relation, but that scalar value is then appended to that tuple to produce a tuple in the overall SUMMARIZE result (which is, of course, a relation).

Turning now to SQL: SQL does have something analogous to the BY form of SUMMARIZE, but not the more general PER form. For example, this expression—

```
SELECT SP.SNO, SUM ( SP.QTY ) AS TQ
```

[8] As noted in the body of the chapter, nonscalar aggregate operators can be defined as well, but they're beyond the scope of this book.


```
FROM   SP
GROUP  BY SP.SNO
```

—is SQL's analog of the following:

```
SUMMARIZE SP BY { SNO } ADD ( SUM ( QTY ) AS TQ )
```

However, SQL's analog of the following—

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TQ )
```

involves a certain amount of circumlocution:

```
SELECT S.SNO, TEMP.TQ
FROM   S, LATERAL ( SELECT SUM ( SP.QTY ) AS TQ
                    FROM   SP
                    WHERE  SP.SNO = S.SNO ) AS TEMP
```

Observe that this latter expression doesn't involve GROUP BY at all. Indeed, the previous example (SQL's analog of SUMMARIZE BY) could also be formulated without using GROUP BY:

```
SELECT DISTINCT SP.SNO, TEMP.TQ
FROM   SP, LATERAL ( SELECT SUM ( SPX.QTY ) AS TQ
                     FROM   SP AS SPX
                     WHERE  SPX.SNO = SP.SNO ) AS TEMP
```

So why do we need GROUP BY? *Answer:* We don't, logically speaking; it's just shorthand (though it wasn't originally designed as such). It's not even a very good shorthand; certainly it isn't very systematic.[9]

As for aggregate operators: The problem here is that COUNT, SUM, and similar operators can be invoked in SQL only in the context of a query (a table expression, in other words), and SQL queries always return a table, while aggregate operators return a scalar value. Here's a simple **Tutorial D** example (repeated from the body of the chapter):

```
VAR N INTEGER ;

N := COUNT ( S WHERE CITY = 'London' ) ;
```

The best that SQL can do by way of a counterpart here is something like this (note the "singleton select"):

```
DECLARE N INTEGER ;

SELECT COUNT ( * ) AS CT
INTO  :N
FROM   S
WHERE  S.CITY = 'London' ;
```

Observe the double coercion involved in the singleton SELECT: First of all, a table of one row and one column is derived; that table is then coerced to the single row it contains; and that row is then coerced to the single scalar value *it* contains. (And that value is then assigned to the variable N.)

[9] SUMMARIZE is shorthand too, but it *is* systematic.

As an aside, I remark that **Tutorial D** does not support coercions of any kind. Thus, if the expression *rx* denotes a relation with just one attribute *A* and just one tuple, an expression like the following must be used to extract the single *A* value from that relation:

```
A FROM ( TUPLE FROM ( rx ) )
```

In other words, TUPLE FROM (*rx*) extracts the sole tuple from the relation denoted by the relational expression *rx* (which must be of cardinality one) and *A* FROM *tx* extracts the value of attribute *A* from the tuple denoted by the tuple expression *tx* (which must have an attribute *A*).

Exercise 5-12.

```
WITH ( SP RENAME ( SNO AS X ) ) AS R :
EXTEND ( S { SNO } ) ADD ( COUNT ( R WHERE X = SNO ) AS NP )
```

Exercise 5-13.

a. Suppliers, so tagged:

SNO	SNAME	STATUS	CITY	TAG
S1	Smith	20	London	Supplier
S2	Jones	10	Paris	Supplier
S3	Blake	30	Paris	Supplier
S4	Clark	20	London	Supplier
S5	Adams	30	Athens	Supplier

b. The join of parts and shipments, with total shipment weights:

SNO	PNO	QTY	PNAME	COLOR	WEIGHT	CITY	SHIPWT
S1	P1	300	Nut	Red	12.0	London	3600.0
S1	P2	200	Bolt	Green	17.0	Paris	3400.0
S1	P3	400	Screw	Blue	17.0	Oslo	6800.0
S1	P4	200	Screw	Red	14.0	London	2800.0
S1	P5	100	Cam	Blue	12.0	Paris	1200.0
S1	P6	100	Cog	Red	19.0	London	1900.0
S2	P1	300	Nut	Red	12.0	London	3600.0
S2	P2	400	Bolt	Green	17.0	Paris	6800.0
S3	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P2	200	Bolt	Green	17.0	Paris	3400.0
S4	P4	300	Screw	Red	14.0	London	4200.0
S4	P5	400	Cam	Blue	12.0	Paris	4800.0

c. Parts with their weights in grams and ounces:

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT	OZWT
P1	Nut	Red	12.0	London	5448.0	192.0
P2	Bolt	Green	17.0	Paris	7718.0	204.0
P3	Screw	Blue	17.0	Oslo	7718.0	204.0
P4	Screw	Red	14.0	London	6356.0	168.0
P5	Cam	Blue	12.0	Paris	5448.0	192.0
P6	Cog	Red	19.0	London	8626.0	228.0

d. Suppliers with numbers of shipments:

SNO	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

e. Supplier cities with average status by city:

CITY	AVG_STATUS
London	20
Paris	20
Athens	30

Exercise 5-14. You can determine which of the expressions are equivalent from the following results of evaluating them:

a. *r* empty:

CT

r has *n* tuples (*n* > 0):

CT
n

b. *r* empty:

CT
0

r has *n* tuples (*n* > 0):

CT
n

c. *r* empty:

CT

r has *n* tuples (*n* > 0):

CT
n

d. *r* empty:

CT
0

r has *n* tuples (*n* > 0):

CT
n

In other words, the result is a relation of degree one in every case. If *r* is nonempty, all four expressions are equivalent; otherwise a. and c. are equivalent, and b. and d. are

equivalent. *Subsidiary exercise:* Why isn't there any double underlining in any of the foregoing pictures?

Exercise 5-15. The basic point is that the collection of values over which the aggregation is to be done is, in general, a *bag*, not a set. For example, if employees Joe and Alice happen to make the same salary, we want to include that salary twice, not once, in computing the average employee salary. Thus, the argument to AVG here is certainly not the projection of the employees relation on the salary attribute, since that projection would, by definition, eliminate one of those two salaries. It follows that, syntactically, the aggregate argument must *not* be specified as a relational expression. On the other hand, we really don't want to clutter up the language with "bag expressions" as well as relational expressions, especially since "bag expressions" would probably look rather similar to relational expressions and thereby cause some confusion. So what's the best thing to do?

Incidentally, it's tempting to suggest that this issue is the root (or one of the roots) of SQL's problem over duplicate rows. Recognizing that in one context at least—namely, that of aggregate operator invocations—duplicate elimination was contraindicated, the SQL language designers decided to make duplicate elimination the exception instead of the norm, and so built the language around bag expressions instead of relational expressions. The basic data object in SQL thus became, not the relation, but the row-bag; in other words, duplicate rows were allowed. The rest, as they say, is history.

The way out of this dilemma is to realize that, in general, aggregate operators take *two* arguments, as in **Tutorial D**. (In particular, this realization is what lets **Tutorial D** avoid SQL's *ad hoc* trick of sometimes requiring an aggregate operator argument to be qualified by DISTINCT.)

Exercise 5-16. SQL returns null in all cases except COUNT, where it does correctly return zero. As to why, your guess is as good as mine.

Exercise 5-17. Here's one reasonably straightforward formulation: *Supplier SNO supplies part PNO if and only part PNO is mentioned in relation PNO_REL.*

Exercise 5-18. Relation *r* has the same cardinality as SP and the same heading, except that it has one additional attribute, X, which is relation-valued. The relations that are values of X have degree zero; furthermore, each is TABLE_DEE, not TABLE_DUM, because every tuple *sp* in SP effectively includes the 0-tuple as its value for that subtuple of *sp* that corresponds to the empty set of attributes. Thus, each tuple in *r* effectively consists of the corresponding tuple from SP extended with the X value TABLE_DEE, and the original GROUP expression is logically equivalent to the following:

```
EXTEND SP ADD ( TABLE_DEE AS X )
```

The expression *r* UNGROUP (X) yields the original SP relation again.

Exercise 5-19. One answer is P{} = TABLE_DUM; another is COUNT(P) = 0. SQL analogs of these expressions are NOT EXISTS (SELECT * FROM P) and (SELECT COUNT(*) FROM P) = 0, respectively.

Exercise 5-20. Tutorial D on the left, SQL on the right (except for part n., for which the SQL solution is given after the **Tutorial D** one):

```
a. SP
```

```
SELECT SP.* FROM SP
or
TABLE SP
```

```
b. ( SP WHERE
       PNO = PNO('P1') )
                    { SNO }
```

```
SELECT SP.SNO
FROM   SP
WHERE  SP.PNO = PNO('P1')
```

```
c. S WHERE STATUS ≥ 15
      AND STATUS ≤ 25
```

```
SELECT S.*
FROM   S
WHERE  S.STATUS BETWEEN
       15 AND 25
```

```
d. ( SP MATCHING
     ( S WHERE CITY =
          'London' ) { PNO }
```

```
SELECT DISTINCT SP.PNO
FROM   SP, S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

```
e. ( P NOT MATCHING
     ( SP MATCHING
        ( S WHERE CITY =
            'London' ) ) )
                    { PNO }
```

```
SELECT P.PNO
FROM   P
EXCEPT
SELECT SP.PNO
FROM   SP, S
WHERE  SP.SNO = S.SNO
AND    S.CITY = 'London'
```

```
f. ( ( SUMMARIZE S
         BY { CITY } ADD
         ( COUNT ( ) AS NS ) )
    WHERE NS > 1 ) { CITY }
```

```
SELECT TEMP.CITY
FROM ( SELECT S.CITY,
              COUNT(*) AS NS
       FROM   S
       GROUP  BY S.CITY )
       AS TEMP
WHERE  TEMP.NS > 1
```

```
g. WITH SP { SNO, PNO } AS Z :
   ( ( Z RENAME ( PNO AS X ) )
     JOIN
     ( Z RENAME ( PNO AS Y ) ) )
   { X, Y }
```

```
SELECT XX.PNO AS X,
       YY.PNO AS Y
FROM   SP AS XX, SP AS YY
WHERE  XX.SNO = YY.SNO
```

```
h. EXTEND TABLE_DEE ADD ( COUNT
     ( SP WHERE SNO = SNO('S1') )
       AS N )
```

```
SELECT COUNT(*) AS N
FROM   SP
WHERE  SNO = SNO('S1')
```

Or simply (**Tutorial D**):

```
COUNT ( SP WHERE SNO = SNO('S1') )
```

But this **Tutorial D** expression returns a scalar, and an exact SQL analog isn't available (see the answer to Exercise 5.11).

```
i. ( S WHERE STATUS <
     STATUS FROM ( TUPLE FROM ( S
       WHERE SNO = SNO('S1') ) ) )
        { SNO }
```

```
SELECT S.SNO
FROM   S
WHERE  S.STATUS <
( SELECT S.STATUS
  FROM   S
  WHERE  S.SNO =
```



```
                                          |          SNO('S1') )
```

The **Tutorial D** expression STATUS FROM (TUPLE FROM *r*) extracts the STATUS value from the single tuple in relation *r* (see the answer to Exercise 5.11). Observe that—also as in the answer to Exercise 5.11—the SQL analog here effectively performs a double coercion: First, it coerces a table of one row to that row; second, it coerces a row containing a single scalar value to that scalar value.

```
j. ( S WHERE CITY =                       |  SELECT S.SNO
     MIN ( S { CITY } ) ) { SNO }         |  FROM   S
                                          |  WHERE  S.CITY =
                                          |    ( SELECT MIN ( S.CITY )
                                          |      FROM   S )
```

```
k. WITH ( S WHERE CITY =                  |  SELECT DISTINCT SPX.PNO
             'London' ) AS RX,            |  FROM   SP AS SPX
        ( SP RENAME ( PNO AS Y ) )        |  WHERE  NOT EXISTS (
                        AS RY :           |  SELECT S.SNO FROM S
   ( P WHERE RX { SNO } =                 |  WHERE  S.CITY = 'London'
       ( RY WHERE Y = PNO )               |  AND    NOT EXISTS (
                  { SNO } ) { PNO }       |  SELECT SPY.*
                                          |  FROM   SP AS SPY
                                          |  WHERE  SPY.PNO = S.SNO
                                          |  AND    SPY.PNO =
                                          |         SPX.PNO ) )
```

```
l. ( S { SNO } JOIN P { PNO } )           |  SELECT S.SNO, P.PNO
     NOT MATCHING SP                      |  FROM   S, P
                                          |  WHERE  NOT EXISTS
                                          |    ( SELECT *
                                          |      FROM   SP
                                          |      WHERE  SP.SNO = S.SNO
                                          |      AND    SP.PNO = P.PNO )
```

```
m. WITH ( SP WHERE SNO =                  |  SELECT S.SNO FROM S
              SNO('S2') ) AS RA,          |  WHERE  NOT EXISTS (
        ( SP RENAME ( SNO AS X ) )        |  SELECT P.* FROM P
                        AS RB :           |  WHERE  NOT EXISTS (
   S WHERE ( RB WHERE X = SNO )           |  SELECT SP.* FROM SP
             { PNO } ⊇ RA { PNO } )        |  WHERE  SP.PNO = P.PNO
                                          |  AND    SP.SNO =
                                          |         SNO('S2') )
                                          |  OR     EXISTS (
                                          |  SELECT SP.*
                                          |  FROM   SP
                                          |  WHERE  SP.PNO = P.PNO
                                          |  AND    SP.SNO = S.SNO ) )
```

```
n. WITH ( P WHERE CITY = 'London' ) { PNO } AS RA,
        ( SP WHERE PNO ∈ RA ) { SNO } AS RB,
        ( SP RENAME ( SNO AS X ) ) AS RC,
        ( EXTEND RB
            ADD ( ( SP WHERE X = SNO ) { PNO } AS Y )
                  { Y } AS RD,
        ( EXTEND SP
            ADD ( ( RC WHERE X = SNO ) { PNO } AS Z )
                  { SNO, Z } AS RE,
        ( RD JOIN RE ) AS RF :
        ( RF WHERE Y ⊆ Z ) { SNO }
```



```
SELECT S.SNO
FROM   S
WHERE  NOT EXISTS
     ( SELECT P.*
       FROM   P
       WHERE  EXISTS
            ( SELECT PP.*
              FROM   P AS PP
              WHERE  PP.CITY = 'London'
              AND    EXISTS
                   ( SELECT SS.*
                     FROM   S AS SS
                     WHERE  EXISTS
                          ( SELECT SP.*
                     FROM   SP
                     WHERE  SP.SNO = SS.SNO
                     AND    SP.PNO = PP.PNO
                     AND    EXISTS
                          ( SELECT SPX.*
                            FROM   SP AS SPX
                            WHERE  SPX.PNO = P.PNO
                            AND    SPX.SNO = SS.SNO ) ) )
              AND    NOT EXISTS
                   ( SELECT SP.*
                     FROM   SP
                     WHERE  SP.SNO = S.SNO
                     AND    SP.PNO = P.PNO ) ) )
```

Exercise 5-21. *No answer provided* (the exercise is easy).

Exercise 5-22. Union isn't idempotent in SQL, because the expression SELECT *R*.* FROM *R* UNION SELECT *R*.* FROM *R* isn't identically equal to SELECT *R*.* FROM *R*. That's because if *R* contains any duplicates, SQL's union will eliminate them. (And what does it do if *R* contains any nulls? Good question!)

Join is idempotent, and therefore so are intersection and cartesian product—in all cases, in the relational model but not in SQL, thanks again to duplicates and nulls.

Exercise 5-23. The expression *r*{} denotes the projection of *r* on no attributes; it returns TABLE_DUM if *r* is empty and TABLE_DEE otherwise (see the answer to Exercise 5.19).

Exercise 5-24. So far as I know, DB2 and Ingres both perform this kind of optimization. Other products might do so too.

Exercise 5-25. Of course, the point about this exercise is that (given our usual sample data values) there *are* no purple parts. The first expression returns all five suppliers, the second returns them all except supplier S5 (who supplies no parts at all). The first is correct, the second isn't, because if there are no purple parts, then every supplier supplies all of them—even supplier S5! See Appendix A for further explanation.

Exercise 5-26. I won't give a complete answer to this exercise, but at least let me observe that the following equivalences certainly allow the boolean expression to contain arbitrary combinations of ANDs, ORs, and NOTs:


```
a. r WHERE bx1 AND bx2  ≡  ( r WHERE bx1 ) JOIN ( r WHERE bx2 )

b. r WHERE bx1 OR bx2   ≡  ( r WHERE bx1 ) UNION ( r WHERE bx2 )

c. r WHERE NOT ( bx )   ≡  r MINUS ( r WHERE bx )
```

Exercise 5-27. Supplier numbers for suppliers who supply part P2 and part numbers for parts supplied by supplier S2, respectively. Note that these natural-language queries are symmetric, while their formal counterparts certainly aren't; that's because R4 is itself asymmetric, in the sense that it treats supplier numbers and part numbers very differently.

Exercise 5-28. It returns a relation looking like this (in outline):

SNO	SNAME	STATUS	CITY	PNO_REL
S1	Smith	20	London	PNO P1 P2 ... P6
S2	Jones	10	Paris	PNO P1 P2
...	.....	..	.....	.....
S5	Adams	30	Athens	PNO

Attribute PNO_REL here is an RVA (a relation-valued attribute). Note in particular that the empty set of parts supplied by supplier S5 is represented by an empty set, not—as it would be if we were to form "the outer join" of S and SP—by some strange *null*. To represent an empty set by an empty set seems like an obviously good idea; in fact, *there would be no need for outer join at all* if relation-valued attributes were properly supported! *Note:* I'm assuming here that you know what an outer join is, but if you don't it doesn't matter much; suffice it to say that it involves nulls and is thus not part of the relational model. (It *is* supported by SQL, though.)

Exercise 5-29. The first one is straightforward: It inserts a new tuple, with SNO value S6 and PNO_REL value a relation containing just one tuple, containing in turn the PNO value P5. As for the second one, I think it would help to show the relevant portion—at least, a simplified form of it—of the **Tutorial D** grammar for relational assignment (the names of the syntactic categories are supposed to be self-explanatory):

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



```
<relation insert>
    ::=   INSERT <relvar> <relation>

<relation delete>
    ::=   DELETE <relvar> [ WHERE <boolean> ]

<relation update>
    ::=   UPDATE <relvar> [ WHERE <boolean> ]
               ( <attribute assign commalist> )
```

And an *<attribute assign>*, if the attribute in question is relation-valued, is basically just a *<relation assign>*, of course, and that's where we came in. Thus, in the exercise, what the second update does is replace the tuple for supplier S2 by another in which the PNO_REL value additionally includes a tuple for supplier S5.

Observe, therefore, that the two updates are a formal representation of the following natural-language updates:

1. Add the fact that supplier S6 supplies part P5 to the database.
2. Add the fact that supplier S2 supplies part P5 to the database.

With our usual suppliers-and-parts design (without RVAs), there's no qualitative difference between these two—both involve the insertion of a single tuple into relvar SP, like this:

```
a. INSERT SP RELATION { TUPLE { SNO SNO('S6'),
                                PNO PNO('P5') } } ;

b. INSERT SP RELATION { TUPLE { SNO SNO('S2'),
                                PNO PNO('P5') } } ;
```

But with SSP, these symmetric updates are treated asymmetrically. Fundamentally, it's this lack of symmetry that's the problem (usually but not always) with relvars that include RVAs.

Exercise 5-30. Query a. is easy:

```
WITH ( SSP RENAME ( SNO AS XNO ) ) { XNO, PNO_REL } AS X ,
     ( SSP RENAME ( SNO AS YNO ) ) { YNO, PNO_REL } AS Y :
     ( X JOIN Y ) { SNO, YNO }
```

Note that the join here is being done on RVAs (and is thus implicitly performing relational comparisons).

Query b., by contrast, is not so straightforward. Query a. was easy because SSP "nests parts within suppliers," as it were; for Query b. we would really like to have suppliers nested within parts instead. So let's do that:

```
WITH ( SSP UNGROUP ( PNO_REL ) ) GROUP ( { SNO } AS SNO_REL )
     AS PPS ,
     ( PPS RENAME ( PNO AS XNO ) ) { XNO, SNO_REL } AS X ,
     ( PPS RENAME ( PNO AS YNO ) ) { YNO, SNO_REL } AS Y :
     ( X JOIN Y ) { XNO, YNO }
```


Exercise 5-31.

```
WITH ( P RENAME ( WEIGHT AS WT ) ) AS R1 ,
     ( EXTEND P ADD ( COUNT ( R1 WHERE WT > WEIGHT )
                                        AS N_HEAVIER ) AS R2 :
( R2 WHERE N_HEAVIER < 3 ) { ALL BUT N_HEAVIER }

SELECT PX.*
FROM   P AS PX
WHERE ( SELECT COUNT(*)
        FROM   P AS PY
        WHERE  PX.WEIGHT > PY.WEIGHT ) < 3
```

Note: Quota queries are quite common in practice. In our book *Databases, Types, and the Relational Model: The Third Manifesto* (Addison-Wesley, 2006), therefore, Hugh Darwen and I propose a "user-friendly" shorthand for expressing them, according to which the foregoing query could be expressed thus:

```
( ( RANK P BY ( DESC WEIGHT AS W ) ) WHERE W ≤ 3 )
                                          { ALL BUT W }
```


6

Integrity Constraints

Exercise 6-1. A type constraint is a definition of the set of values that go to make up a given type. Type constraints are checked when some selector is invoked; if the check fails, the selector invocation fails on a run-time type constraint violation. *Note:* Of course, the type constraint for type *T* must also be checked when type *T* is defined; if the specified boolean expression evaluates to FALSE at that time, the type definition must be rejected.

A database constraint is a constraint on the values that can appear in a given database. Database constraints are checked "at semicolons"—more specifically, at the end of any statement that assigns a value to some relvar in the database. If the check fails, the assignment fails on a run-time database constraint violation. *Note:* Of course, database constraints must also be checked when they're defined. If that check fails, the constraint definition must be rejected.

Exercise 6-2. *The Golden Rule* states that no update operation must ever cause any database constraint to evaluate to FALSE (and hence, *a fortiori*, that no update operation must ever cause any relvar constraint to evaluate to FALSE either).

Exercise 6-3. An attribute constraint is a statement to the effect that a certain attribute is of a certain type. A relvar constraint, also known as a single-relvar constraint, is a database constraint that mentions just a single relvar. A tuple constraint is a relvar constraint with the property that it can be checked for a given tuple by examining just that tuple in isolation. A multi-relvar constraint is a database constraint that mentions two or more distinct relvars. "The" relvar constraint for relvar *R* is the AND of all of the database constraints that mention *R*. "The" database constraint for database *DB* is the AND of all of the relvar constraints for relvars in *DB*.

Be aware, incidentally, that SQL classifies constraints rather differently. In particular (this isn't the whole story), it uses the term *base table constraint* for a constraint that's specified as part of the definition of some base table and the term *assertion* for a

constraint that's specified via CREATE ASSERTION—even though it's a fact that any constraint[10] that can be specified as a base table constraint can alternatively be specified as an assertion and *vice versa*.

Exercise 6-4. See the body of the chapter.

Exercise 6-5. See the body of the chapter, also the answers to Exercises 2.2 and 2.3 in Chapter 2.

Exercise 6-6.

```
TYPE CITY POSSREP { C CHAR CONSTRAINT C = 'London'
                                    OR C = 'Paris'
                                    OR C = 'Rome'
                                    OR C = 'Athens'
                                    OR C = 'Oslo'
                                    OR C = 'Stockholm'
                                    OR C = 'Madrid'
                                    OR C = 'Amsterdam' } ;
```

Now we can define the CITY attribute in relvars S and P to be of type CITY instead of just type CHAR.

By definition, there's no way to impose exactly the foregoing constraint without defining an explicit type. But we could impose the constraint that supplier cities in particular are limited to the same eight values by means of a suitable database constraint, and similarly for part cities. To be specific, we could define a relvar as follows:

```
VAR C BASE RELATION { CITY CHAR } KEY { CITY } ;
```

We could then "populate" this relvar with the eight city values:

```
INSERT C RELATION { TUPLE { CITY 'London'    },
                    TUPLE { CITY 'Paris'     },
                    TUPLE { CITY 'Rome'      },
                    TUPLE { CITY 'Athens'    },
                    TUPLE { CITY 'Oslo'      },
                    TUPLE { CITY 'Stockholm' },
                    TUPLE { CITY 'Madrid'    },
                    TUPLE { CITY 'Amsterdam' } } ;
```

Now we could define some foreign keys:

```
VAR S BASE RELATION ... FOREIGN KEY { CITY } REFERENCES C ;

VAR P BASE RELATION ... FOREIGN KEY { CITY } REFERENCES C ;
```

This approach has the advantage that it makes it easier to change the set of valid cities, if the requirement should arise.

---

[10] Well, almost any constraint. There's an interesting (?) wrinkle, though. Suppose constraint *C* is defined as an assertion, and the database is currently such as to violate *C*. Then *C* will *not* be regarded as violated if it's specified as part of the definition of base table *T* instead and *T* happens to be empty ... even if *C* is of the form "*T* must not be empty" (!).

Exercise 6-7.

```
TYPE SNO POSSREP { C CHAR CONSTRAINT
     CHAR_LENGTH ( C ) ≥ 2 AND CHAR_LENGTH ( C ) ≤ 5 AND
     SUBSTR ( C, 1, 1 ) = 'S' AND
     CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ≥ 0 AND
     CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ≤ 9999 } ;
```

I'm assuming that operators CHAR_LENGTH, SUBSTR, and CAST_AS_INTEGER are available and have the obvious semantics.

Exercise 6-8.

```
TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;
```

I'm assuming the existence of a user-defined type called POINT as defined in the body of the chapter.

Exercise 6-9. Type POINT is an obvious example, but there are many others—for example, you might like to think about a type PARALLELOGRAM, which can "possibly be represented" in numerous different ways (how many can you think of?). As for type constraints for such a type: Conceptually, each possrep specification *must* include a type constraint; moreover, those constraints must all be logically equivalent. For example:

```
TYPE POINT
     POSSREP CARTESIAN { X NUMERIC, Y NUMERIC
             CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 }
     POSSREP POLAR { R NUMERIC, THETA NUMERIC
             CONSTRAINT R ≤ LENGTH ( 100.0 ) ... } ;
```

Whether some shorthand could be provided that would allow us (in effect) to specify the constraint just once is a separate issue, beyond the scope of this book. *Note:* The ellipsis "..." in the foregoing example represents a further specification that I've deliberately omitted, since again it has to do with something beyond the scope of this book.

Exercise 6-10. A line segment can possibly be represented by its begin and end points or by its midpoint, length, and angle of inclination.

Exercise 6-11. I'll give answers in terms of the INSERT, DELETE, and UPDATE shorthands, not relational assignment as such:

C1: INSERT into S, UPDATE of STATUS in S

C2: INSERT into S, UPDATE of CITY or STATUS in S

C3: INSERT into S, UPDATE of SNO in S

C4: UPDATE of STATUS in S, INSERT into SP, UPDATE of SNO or PNO in SP (I'm assuming here that constraint C5, the foreign key constraint from SP to S, is being enforced)

C5: DELETE from S, UPDATE of SNO in S, INSERT into SP, UPDATE of SNO in SP

C6: INSERT into LS or NLS, UPDATE of SNO in LS or NLS

C7: INSERT into S or P, UPDATE of SNO or CITY in S, UPDATE of PNO or CITY in P

C8: UPDATE of SNO or STATUS in S

Exercise 6-12. The boolean expression in constraint C1 is a simple restriction condition; the one in constraint C4 is more complex. One implication is that a tuple presented for insertion into S can be checked against constraint C1 without even looking at any of the values currently existing in the database, whereas the same is not true for constraint C4.

Exercise 6-13. Yes, of course it's possible; constraint C3 does the trick. But note that neither a constraint like C3 nor an explicit KEY specification can guarantee that the specified attribute combination satisfies the irreducibility requirement, in general; in fact, there's no way the system can enforce that requirement, in general. (Though it would at least be possible to impose a syntax rule to the effect that if two distinct keys are specified for the same relvar, then neither is allowed to be a subset of the other. Such a rule would help, but it still wouldn't do the whole job.)

Exercise 6-14.

```
CREATE ASSERTION C7 CHECK
   ( ( SELECT COUNT(*) FROM
         ( SELECT S.CITY
           FROM   S
           WHERE  S.SNO = SNO('S1')
           UNION
           SELECT P.CITY
           FROM   P
           WHERE  P.PNO = PNO('P1') ) AS POINTLESS ) < 2 ) ;
```

Once again, note the coercion involved in this answer: The expression on the left side of the "<" comparison denotes a table of one (unnamed) column and one row, whereas the expression on the right side denotes a scalar value (an integer). A similar remark applies to several subsequent SQL answers.

Exercise 6-15. This one can't be done declaratively (SQL has no direct support for transition constraints). A triggered procedure can be used, but the details are slightly messy and are beyond the scope of this book.

Exercise 6-16. Space reasons make it too difficult to show **Tutorial D** and SQL formulations side by side here, so in each case I'll show the former first and the latter second. I omit details of which operations might cause the constraints to be violated.

```
a. CONSTRAINT CA IS_EMPTY
   ( P WHERE COLOR = COLOR('Red') AND WEIGHT ≥ WEIGHT(50.0) ) ;

CREATE ASSERTION CA CHECK ( NOT EXISTS (
   SELECT P.*
   FROM   P
   WHERE  P.COLOR = COLOR('Red')
   AND    P.WEIGHT >= WEIGHT(50.0) ) ) ;
```



```
b. CONSTRAINT CB IS_EMPTY (
      WITH ( SP RENAME ( SNO AS X ) ) AS R :
      S WHERE CITY = 'London' AND
        TUPLE { PNO PNO('P2') } ∉ ( R WHERE X = SNO ) { PNO } ) ;

   CREATE ASSERTION CB CHECK
    ( NOT EXISTS ( SELECT * FROM S
                   WHERE  S.CITY = 'London'
                   AND    NOT EXISTS
                            ( SELECT * FROM SP
                              WHERE  SP.SNO = S.SNO
                              AND    SP.PNO = PNO('P2') ) ) ) ;

c. CONSTRAINT CC COUNT ( S ) = COUNT ( S { CITY } ) ;

   CREATE ASSERTION CC CHECK ( UNIQUE ( SELECT S.CITY FROM S ) ) ;

d. CONSTRAINT CD COUNT ( S WHERE CITY = 'Athens' ) < 2 ;

   CREATE ASSERTION CD CHECK
    ( ( SELECT COUNT(*) FROM S WHERE S.CITY = 'Athens' ) < 2 ) ;

e. CONSTRAINT CE COUNT ( S WHERE CITY = 'London') > 0 ;

   CREATE ASSERTION CE CHECK
    ( EXISTS ( SELECT S.* FROM S WHERE S.CITY = 'London' ) ) ;

f. CONSTRAINT CF COUNT ( P WHERE COLOR = COLOR('Red')
                          AND WEIGHT < WEIGHT(50.0) ) > 0 ;

   CREATE ASSERTION CF CHECK
    ( EXISTS ( SELECT P.* FROM P
               WHERE  P.COLOR = COLOR('Red')
               AND    P.WEIGHT < WEIGHT(50.0) ) ) ;

g. CONSTRAINT CG CASE
                    WHEN IS_EMPTY ( S ) THEN TRUE
                    ELSE AVG ( S, STATUS ) > 10
                  END CASE ;

CREATE ASSERTION CG CHECK
 ( CASE
     WHEN NOT EXISTS ( SELECT * FROM S ) THEN TRUE
     ELSE ( SELECT AVG ( S.STATUS ) FROM S ) > 10
   END ) ;

h. CONSTRAINT CH
     CASE
        WHEN IS_EMPTY ( SP ) THEN TRUE
        ELSE IS_EMPTY ( SP WHERE QTY > 2 * AVG ( SP, QTY ) )
     END CASE ;

   CREATE ASSERTION CH CHECK
    ( CASE
        WHEN NOT EXISTS ( SELECT * FROM SP ) THEN TRUE
        ELSE NOT EXISTS ( SELECT * FROM SP
                          WHERE  SP.QTY > 2 *
                               ( SELECT AVG ( SP.QTY )
                                 FROM   SP ) )
     END ) ;
```



```
i. CONSTRAINT CI CASE
      WHEN COUNT ( S ) < 2 THEN TRUE
      ELSE IS_EMPTY ( JOIN
         { ( S WHERE STATUS = MAX ( S { STATUS } ) ) { CITY },
           ( S WHERE STATUS = MIN ( S { STATUS } ) ) { CITY } } )
      END CASE ;

   CREATE ASSERTION CI CHECK ( CASE
      WHEN ( SELECT COUNT(*) FROM S ) < 2 THEN TRUE
      ELSE NOT EXISTS
           ( SELECT * FROM S AS X, S AS Y
             WHERE  X.STATUS =
                  ( SELECT MAX ( S.STATUS ) FROM S )
             AND    Y.STATUS =
                  ( SELECT MIN ( S.STATUS ) FROM S )
             AND    X.CITY = Y.CITY )
                           END ) ;

j. CONSTRAINT CJ P { CITY } ⊆ S { CITY } ;

   CREATE ASSERTION CJ CHECK ( NOT EXISTS
        ( SELECT * FROM P
          WHERE  NOT EXISTS
               ( SELECT * FROM S
                 WHERE  S.CITY = P.CITY ) ) ) ;

k. CONSTRAINT CK IS_EMPTY
    ( EXTEND ( P RENAME ( PNO AS X ) )
        ADD ( ( S MATCHING ( SP WHERE PNO = X ) ) { CITY }
        AS SC_REL ) WHERE TUPLE { CITY CITY } ∉ SC_REL ) ;

   CREATE ASSERTION CK CHECK ( NOT EXISTS
        ( SELECT * FROM P
          WHERE  NOT EXISTS
               ( SELECT * FROM S
                 WHERE  S.CITY = P.CITY
                 AND    EXISTS
                      ( SELECT * FROM SP
                        WHERE  S.SNO = SP.SNO
                        AND    P.PNO = SP.PNO ) ) ) ) ;

l. CONSTRAINT CL
      COUNT ( ( ( S WHERE CITY = 'London' ) JOIN SP ) { PNO } ) >
      COUNT ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ) { PNO } ) ;

   CREATE ASSERTION CL CHECK (
    ( SELECT COUNT ( DISTINCT SP.PNO ) FROM S, SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'London' ) >
    ( SELECT COUNT ( DISTINCT SP.PNO ) FROM S, SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'Paris' ) ) ;
```



```
m. CONSTRAINT CM
       SUM ( ( ( S WHERE CITY = 'London' ) JOIN SP ), PNO ) >
       SUM ( ( ( S WHERE CITY = 'Paris'  ) JOIN SP ), PNO ) ;

   CREATE ASSERTION CM CHECK (
    ( SELECT SUM ( SP.QTY ) FROM S, SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'London' ) >
    ( SELECT SUM ( SP.QTY ) FROM S, SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'Paris' ) ) ;
```

Exercise 6-17. Suppose we defined a relvar SC{SNO,CITY} with predicate *Supplier SNO has no office in city CITY*. Suppose further that supplier S1 has an office in just ten cities. Then the Closed World Assumption would imply that relvar SC must have *n*-10 tuples for supplier S1, where *n* is the total number of cities in the world! (Or something.)

Exercise 6-18. We need a multiple assignment (if we are to do the delete in a single statement as requested):

```
DELETE S WHERE SNO = x , DELETE SP WHERE SNO = x ;
```

Exercise 6-19. I've named the constraints CA, CB, and so forth in the obvious way.

```
a. CONSTRAINT CA
       IS_EMPTY ( ( ( S' WHERE CITY = 'Athens' ) { SNO } ) JOIN S )
                  WHERE CITY ≠ 'Athens'
                  AND   CITY ≠ 'London'
                  AND   CITY ≠ 'Paris' )
   AND IS_EMPTY ( ( ( S' WHERE CITY = 'London' ) { SNO } ) JOIN S )
                  WHERE CITY ≠ 'London'
                  AND   CITY ≠ 'Paris' ) ;

b. CONSTRAINT CB IS_EMPTY
     ( ( SUMMARIZE ( ( SP' RENAME ( QTY AS QTY' ) ) { PNO, QTY' }
                     JOIN SP { PNO, QTY } ) BY { PNO }
         ADD ( SUM ( QTY' ) AS OLD, SUM ( QTY ) AS NEW ) )
       WHERE OLD > NEW ) ;

c. CONSTRAINT CC IS_EMPTY
     ( ( SUMMARIZE ( ( SP' RENAME ( QTY AS QTY' ) ) { SNO, QTY' }
                     JOIN SP { SNO, QTY } ) BY { SNO }
         ADD ( SUM ( QTY' ) AS OLD, SUM ( QTY ) AS NEW ) )
       WHERE NEW < 0.5 * OLD ) ;
```

The qualification "in a single update" is important because (taking constraint CA by way of example) we aren't trying to outlaw the possibility of an Athens supplier moving to Paris in one update and thence to Rome (say) in another.

Exercise 6-20. See the body of the chapter.

Exercise 6-21. Well, it might be syntactically valid, but it will certainly fail at run time (see the answer to Exercise 6.1).

Exercise 6-22. Same answer as Exercise 6.21.

Exercise 6-23. *No answer provided.*

Exercise 6-24. The answer has to do with type inheritance. The details are beyond the scope of this book; if you're interested, you can find a detailed discussion in the book *Databases, Types, and the Relational Model: The Third Manifesto* (Addison-Wesley, 2006), by Hugh Darwen and myself. The consequences are that when you define a type, you can't even specify the values that make up that type!—except for the *a priori* constraint imposed by the representation.

Exercise 6-25. Yes in principle—though **Tutorial D** in particular deliberately provides no way of defining type constraints, other than *a priori* ones, for either nonscalar or system-defined types.

7

Database Design Theory

Exercise 7-1. See the body of the chapter.

Exercise 7-2. The complete set of FDs—what's known, formally, as the *closure***,** though it's nothing to do with the closure property of the relational algebra—for relvar SP is as follows:

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

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

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



```
{ SNO, QTY }        → { }

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

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

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

{ }                 → { }
```

Exercise 7-3. True ("whenever two tuples agree on *A*, they also agree on *B*" implies a comparison between the projections of the tuples in question on {*A*,*B*}). *Note:* As I pointed out in the body of the chapter, it does make sense to talk about projections of tuples as well as of relations, and a similar remark applies to several other relational operators as well (for example, rename, extend). Technically, we say the operators in question are *overloaded*.

Exercise 7-4. Heath's theorem says: If *R*{*A*,*B*,*C*} satisfies the FD $A \rightarrow B$, then *R* is equal to the join of its projections *R1* on {*A*,*B*} and *R2* on {*A*,*C*}. In the following simple proof of this theorem, I use the same informal shorthand for tuples that I used in the body of the chapter.

First I show that no tuple of *R* is lost by taking the projections and then joining those projections back together again. Let <*a*,*b*,*c*> ∈ *R*. Then <*a*,*b*> ∈ *R1* and <*a*,*c*> ∈ *R2*, and so <*a*,*b*,*c*> ∈ *R1* JOIN *R2*.

Next I show that every tuple of the join is indeed a tuple of *R* (in other words, the join doesn't generate any "spurious" tuples). Let <*a*,*b*,*c*> ∈ *R1* JOIN *R2*. In order to generate such a tuple in the join, we must have <*a*,*b*> ∈ *R1* and <*a*,*c*> ∈ *R2*. Hence there must exist a tuple <*a*,*b'*,*c*> ∈ *R* for some *b'*, in order to generate the tuple <*a*,*c*> ∈ *R2*. We therefore must have <*a*,*b'*> ∈ *R1*. Now we have <*a*,*b*> ∈ *R1* and <*a*,*b'*> ∈ *R1;* hence we must have *b* = *b'*, because $A \rightarrow B$. Hence <*a*,*b*,*c*> ∈ *R*.

The converse of Heath's theorem would say that if *R*{*A*,*B*,*C*} is equal to the join of its projections on {*A*,*B*} and on {*A*,*C*}, then *R* satisfies the FD $A \rightarrow B$. This statement is false. To show this is so, it's sufficient to exhibit a single counterexample; I'll leave the question of finding such a counterexample to you. (If you give up, you can find one in the answer to Exercise 7.20 and another in the answer to Exercise 3.12 in Chapter 3. Consider also relvar SPJ as illustrated in Fig. 7.5 in the body of the chapter.)

Exercise 7-5. *No answer provided* (the exercise is easy).

Exercise 7-6. See the body of the chapter.

Exercise 7-7. See the answer to Exercise 6.13 in Chapter 6.

Exercise 7-8. For the definitions, see the body of the chapter. The former *is* a special case of the latter. For example, relvar S satisfies the trivial FD {CITY,STATUS} → {STATUS}. Applying Heath's theorem, therefore, we see that S satisfies the trivial JD *{*AB,AC*}, where *A* is {CITY,STATUS}, *B* is {STATUS}, and *C* is {SNO,SNAME}.

Exercise 7-9. An FD is basically a statement of the form $A \rightarrow B$, where *A* and *B* are each subsets of the heading of *R*. Given that a set of *n* elements has 2^n possible subsets, it follows that each of *A* and *B* has 2^n possible values, and hence an upper limit on the number of possible FDs in *R* is 2^{2n}. Thus, for example, the upper limit for a relvar of degree five is 1,024.

Exercise 7-10. Let the specified FD be satisfied by relvar *R*. Now, every tuple *t* of *R* has the same value (namely, the 0-tuple) for that subtuple of *t* that corresponds to the empty set of attributes. If *B* is empty, therefore, the FD $A \rightarrow B$ is trivially true for all possible sets *A* of attributes of *R;* in fact, it's a *trivial* FD (and it isn't very interesting). On the other hand, if *A* is empty, the FD $A \rightarrow B$ means all tuples of *R* have the same value for *B* (since they certainly all have the same value for *A*). And if *B* in turn is "all of the attributes of *R*"—in other words, if *R* has an empty key—then *R* is constrained to contain at most one tuple (see the answer to Exercise 4.28 in Chapter 4).

Exercise 7-11. Suppose we start with a relvar R with attributes D, P, S, L, T, and C (attribute names corresponding to parameters of the predicate in the obvious way). Then R satisfies the following nontrivial FDs:

```
{ L }       → { D, P, C, T }
{ D, P, S } → { L }
{ D, P, C } → { L }
{ D, P, T } → { L }
```

Thus, an obvious set of BCNF relvars (in outline) is:

```
SCHEDULE { L, D, P, C, T }
         KEY { L }
         KEY { D, P, C }
         KEY { D, P, T }

STUDYING { S, L }
         KEY { S, L }
```

STUDYING is in 6NF; SCHEDULE is in 5NF but not 6NF. Note, however, that if we decomposed SCHEDULE into its 6NF projections on {L,D}, {L,P}, {L,C}, and {L,T}—which we could certainly do if we wanted—we would fail to preserve three of the four original FDs. Thus, it's probably not a good idea to perform that further decomposition.

Exercise 7-12. Yes, sometimes, though probably not very often; in fact, such a possibility is the whole *raison d'être* for 5NF as opposed to 4NF. The subsection "More on 5NF" in the body of the chapter gives an example of a relvar that can be nonloss-decomposed into three projections and not into two. Moreover, that decomposition is probably desirable, because (once again) it reduces redundancy and thereby avoids certain update anomalies that might otherwise occur.

Exercise 7-13. Surrogate keys are *not* the same thing as tuple IDs. For one thing (to state the obvious), surrogates identify entities and tuple IDs identify tuples, and there's certainly nothing like a one-to-one correspondence between entities and tuples. (Think of derived tuples in particular—for example, tuples in the result of some query. In fact, it's not at all clear that derived tuples will have tuple IDs anyway.) Furthermore, tuple IDs usually have performance connotations, but surrogates don't (access to a tuple via its tuple ID is usually assumed to be fast, but no such observation applies to surrogates). Also, tuple IDs are usually concealed from the user, but surrogates mustn't be (because of *The Information Principle*—see Chapter 8); in other words, it's probably (and desirably!) not possible to store a tuple ID in a database relvar, while it certainly is possible, and desirable, to store a surrogate in a database relvar. In a nutshell: Surrogate keys have to do with logical design, tuple IDs have more to do with physical design.

Are surrogate keys a good idea? Well, observe first that the relational model has nothing to say on this question; like the whole business of database design, in fact, whether or not to use surrogate keys has to do with *how to apply* the relational model, not with the relational model as such.

That said, I now have to say that the question of whether surrogate keys are good or bad is far from straightforward. There are strong arguments on both sides: so many such, in fact, that I can't possibly do justice to them all here. Instead, I'll simply refer you to a detailed article of my own on the subject: "Composite Keys," in my book *Relational Database Writings 1989-1991* (Addison-Wesley, 1992). *Note:* The article is titled "Composite Keys" because, of course, surrogate keys are most likely to be useful in practice in situations in which primary keys (and matching foreign keys) would otherwise be composite keys specifically.

One last point on surrogates: Given that a BCNF relvar with no composite keys is "automatically" in 5NF, many people seem to think that simply introducing a surrogate key into a BCNF relvar "automatically" means the relvar is now in 5NF—but of course it doesn't mean that at all. In particular, if the relvar had a composite key before the introduction of the surrogate, it still has one afterward too.

Exercise 7-14. For the moment, I'll use the notation $X \rightarrow Y$ to mean that if everyone in the set of people X attends, then everyone in the set of people Y will attend as well (my choice of notation here isn't arbitrary, of course!). Until further notice, I'll refer to expressions such as $X \rightarrow Y$ as *statements*.

The first and most obvious design thus consists of a single relvar, IXAYWA ("if *X* attends, *Y* will attend"), containing a tuple for each of the given statements (see the following figure, where INV stands for "invitee"). Observe that *X* and *Y* in that relvar are RVAs.

IXAYWA

X	Y
INV Amy	INV Bob Cal
INV Don Eve	INV Hal
INV Fay	INV Hal
INV	INV Guy
INV Bob Amy	INV Joe

But we can do better than this. Of course, I chose the FD notation deliberately. Indeed, the statement "if *X* attends, *Y* will attend" is clearly isomorphic to the statement "the FD *X* $\rightarrow$ *Y* is satisfied" (imagine a relvar with a yes-or-no attribute for each of Amy, Bob, Cal, and so on, in which each tuple represents a kind of "bitmapped" attendance list that's consistent with the specified conditions). Hence we can use the theory of FDs to reduce the amount of information we need to record explicitly in the relvar. In fact, we can use that theory to find what's called an *irreducible equivalent* to the given set of statements—that is, a (usually small) set of statements that includes no redundant information and has the property that all of the original statements are implied by statements in that set.

Detailed consideration of how to find an irreducible equivalent is beyond the scope of this book; all I want to do here is give a sketch of what's involved. First, I pointed out in the body of the chapter that if $X \rightarrow Y$ is satisfied, then $X' \rightarrow Y'$ is satisfied for all supersets X' of X and all subsets Y' of Y. We can therefore reduce the amount of information we need to record explicitly by ensuring that every statement $X \rightarrow Y$ we do record is such that the set X is as small as possible and the set Y is as large as possible—meaning, more precisely, that:

- Nobody can be removed from the set X without the resulting statement no longer being true.
- Nobody can be added to the set Y without the resulting statement no longer being true.

Also, we know that the FD $X \rightarrow Y$ is necessarily satisfied for all sets Y such that Y is a subset of X (in particular, $X \rightarrow X$ is always satisfied; for example, "if Amy attends, Amy will attend" is obviously satisfied). Such cases are trivial, and we don't need to record them explicitly at all.

Note too that if relvar IXAYWA contains only those FDs that constitute an irreducible equivalent, then $\{X\}$ will be a key for the relvar.

For further details, I refer you to my book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004).

Exercise 7-15. The obvious design (in outline) is:

```
EMP  { ENO, ENAME, SALARY }
     KEY { ENO }

PGMR { ENO, LANG }
     KEY { ENO }
     FOREIGN KEY { ENO } REFERENCES EMP
```

Every employee has a tuple in EMP (and EMP has no other tuples). Employees who happen to be programmers additionally have a tuple in PGMR (and PGMR has no other tuples). Note that the join of EMP and PGMR gives full information—employee number, name, salary, and language skill—for programmers (only).

The only significant difference if programmers could have an arbitrary number of language skills is that relvar PGMR would be "all key" (its sole key would be {ENO,LANG}).

Exercise 7-16. Let the projections of R on $\{A,B\}$ and $\{A,C\}$ be $R1$ and $R2$, respectively, and let $\langle a,b1,c1\rangle \in R$ and $\langle a,b2,c2\rangle \in R$. Then $\langle a,b1\rangle \in R1$ and $\langle a,b2\rangle \in R1$, and $\langle a,c1\rangle \in R2$ and $\langle a,c2\rangle \in R2$; thus, $\langle a,b1,c2\rangle \in J$ and $\langle a,b2,c1\rangle \in J$, where $J = R1$ JOIN $R2$. But R satisfies the JD $*\{AB,AC\}$ and so $J = R$; thus, $\langle a,b1,c2\rangle \in R$ and $\langle a,b2,c1\rangle \in R$.

By the way, note that the definition of MVD is symmetric in *B* and *C;* thus, *R* satisfies the MVD *A* →→ *B* if and only if it satisfies the MVD *A* →→ *C*. MVDs always come in pairs in this way. For this reason, it's usual to write such pairs as a single statement, thus:

```
A →→ B | C
```

Exercise 7-17. Let *C* be the attributes of *R* not included in *A* or *B*. By Heath's theorem, if *R* satisfies the FD *A* → *B*, then it satisfies the JD *{*AB,AC*}. By definition, however (see Exercise 7.16), if *R* satisfies the JD *{*AB,AC*}, then it satisfies the MVDs *A* →→ *B* and *A* →→ *C*. Thus, *A* → *B* implies *A* →→ *B*.

Exercise 7-18. This result is immediate from the definition of multi-valued dependency (see Exercise 7.16).

Exercise 7-19. Exercise 7.17 shows that if *K* → *A*, then certainly *K* →→ *A*. But if *K* is a key, then *K* → *A*, and the desired conclusion follows immediately.

Exercise 7-20. 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*. Then *R* is equal to the cartesian product of its projections *R*{*A*} and *R*{*B*}; thus, it satisfies the JD *{*A*,*B*} and, equivalently, the following MVDs:

```
{ } →→ A | B
```

These MVDs aren't trivial, since they're certainly not satisfied by all binary relvars, and they're not implied by a superkey either. It follows that *R* isn't in 4NF. However, it is in BCNF, since it's "all key" (see Exercise 7.24).

Exercise 7-21. First let's introduce three relvars, with the obvious interpretations:

```
REP     { REPNO,  ... } KEY { REPNO  }
AREA    { AREANO, ... } KEY { AREANO }
PRODUCT { PRODNO, ... } KEY { PRODNO }
```

Second, we can represent the relationships (a) between sales representatives and sales areas and (b) between sales representatives and products by relvars like this:

```
RA { REPNO, AREANO } KEY { REPNO, AREANO }
RP { REPNO, PRODNO } KEY { REPNO, PRODNO }
```

Every product is sold in every area. So if we introduce a relvar

```
AP { AREANO, PRODNO } KEY { AREANO, PRODNO }
```

to represent the relationship between areas and products, then we have the following constraint:

```
CONSTRAINT C AP = AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

Note that this constraint implies that AP isn't in 4NF. In fact, AP doesn't give us any information we can't obtain from the other relvars; to be precise, we have:

```
AP { AREANO } = AREA { AREANO }
AP { PRODNO } = PRODUCT { PRODNO }
```

But let's assume for the moment that relvar AP is included in our design anyway.

No two representatives sell the same product in the same area. In other words, given an {AREANO,PRODNO} combination, there's exactly one responsible sales representative, REPNO, so we can introduce a relvar

```
APR { AREANO, PRODNO, REPNO } KEY { AREANO, PRODNO }
```

in which (to make the FD explicit)

```
{ AREANO, PRODNO } → REPNO
```

(Of course, specification of {AREANO,PRODNO} as a key is sufficient to express this FD.) Now, however, relvars RA, RP, and AP are all redundant, since they're all projections of APR; they can therefore all be dropped. In place of constraint C we now need constraint C1:

```
CONSTRAINT C1 APR { AREANO, PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;
```

This constraint must be stated separately and explicitly (it isn't "implied by keys").

Also, since every representative sells all of that representative's products in all of that representative's areas, we have the additional constraint C2 on relvar APR:

```
REPNO →→ AREANO | PRODNO
```

(These MVDs are nontrivial, and relvar APR isn't in 4NF.) Again the constraint must be stated separately and explicitly.

Thus the final design consists of the relvars REP, AREA, PRODUCT, and APR, together with the constraints C1 and C2:

```
CONSTRAINT C1 APR { AREANO, PRODNO } =
              AREA { AREANO } JOIN PRODUCT { PRODNO } ;

CONSTRAINT C2 APR =
              APR { REPNO, AREANO } JOIN APR { REPNO, PRODNO } ;
```

This exercise illustrates very clearly the point that, in general, normalization might be adequate to represent some of the semantic aspects of a given problem (basically, FDs, MVDs, and JDs that are implied by keys), but explicit statement of additional constraints is needed for other aspects. It also illustrates the point that it might not always be desirable to normalize "all the way" (relvar APR is in BCNF but not in 4NF).

Note: As a subsidiary exercise, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration. Might such a design mean that some of the comments in the previous paragraph no longer apply?

Exercise 7-22.

```
CONSTRAINT SPJ_JD SPJ = SPJ { SNO, PNO } JOIN
                        SPJ { PNO, JNO } JOIN
                        SPJ { JNO, SNO } ;
```


Exercise 7-23. This is a "cyclic constraint" example. The following design is suitable:

```
REP     { REPNO,  ... } KEY { REPNO  }
AREA    { AREANO, ... } KEY { AREANO }
PRODUCT { PRODNO, ... } KEY { PRODNO }

RA { REPNO,  AREANO } KEY { REPNO,  AREANO }
AP { AREANO, PRODNO } KEY { AREANO, PRODNO }
PR { PRODNO, REPNO  } KEY { PRODNO, REPNO  }
```

Also, the user needs to be informed that the join of RA, AP, and PR does *not* involve any "connection trap":

```
CONSTRAINT NO_TRAP
  ( RA JOIN AP JOIN PR ) { REPNO,  AREANO } = RA AND
  ( RA JOIN AP JOIN PR ) { AREANO, PRODNO } = AP AND
  ( RA JOIN AP JOIN PR ) { PRODNO, REPNO  } = PR ;
```

Note: As with Exercise 7.21, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration.

Exercise 7-24. a. True. b. True. c. False, though "almost" true. Here's a counterexample: We're given a relvar USA {COUNTRY, STATE} ("STATE is part of COUNTRY"), where COUNTRY is the USA in every tuple. This relvar satisfies the FD {} → {COUNTRY}, which is neither trivial nor implied by a key, and so the relvar is not in BCNF (it can be nonloss-decomposed into its two unary projections).

Exercise 7-25. Yes, they do, obviously. What do you conclude?

Exercise 7-26. I'm not going to attempt to show any E/R diagrams here, but the point of the exercise is simply to lend weight to the following remarks from the body of the chapter:

> The problem with E/R diagrams and similar pictures is that they're completely incapable of representing all but a few rather specialized constraints. Thus, while it might be OK to use such diagrams to explicate the overall design at a high level of abstraction, it's misleading, and in some respects quite dangerous, to think of such a diagram as actually *being* the design in its entirety. *Au contraire:* The design is the relvars, which the diagrams do show, *together with the constraints*, which they don't.

Exercise 7-27. As it stands, the design certainly permits the very same tuple to appear in both relvars—a state of affairs that might or might not matter, depending on circumstances. A better design might be:

```
PARENTS_OF { X NAME, Y NAME, Z NAME } KEY { X }
```

(The predicate is *The father and mother of X are Y and Z, respectively*.)

Exercise 7-28. Personally, I'm in favor of this strategy, even in an SQL context, because I find SQL rather a difficult language for all but the simplest of operations. Thus, for example, if I have a complicated query that I need to formulate in SQL, I don't even try to write it in SQL right off the bat; instead, I formulate it in terms of relational algebra or relational calculus first, and then I go through a kind of semimechanical process of mapping that formulation into an SQL equivalent as a follow-on step. The resulting SQL expression is often quite difficult to understand, but at least I know it's *correct*, because of

the way I constructed it. *Note:* I constructed the SQL answer to part n. of Exercise 5.20 in Chapter 5 in this way, and I think that answer illustrates very well the point I'm trying to make here.

8

What Is The Relational Model?

Exercise 8-1. For a definition of exactly what the relational model is, see the body of the chapter. Some of the biggest differences between SQL and the relational model are: (a) SQL tables allow duplicate rows; (b) SQL supports nulls; and (c) SQL tables have a left-to-right ordering to their columns. But there are many further differences too. Here are some of them (note that the list that follows does *not* include any of the numerous criticisms that might fairly be made of SQL *as a language*—regarding, for example, its lack of orthogonality and its failures with respect to syntactic and semantic consistency):

- SQL doesn't explicitly distinguish between relations and relvars (or tables and "tablevars").
- SQL's support for view updating is extremely limited (and in certain respects clearly incorrect).
- SQL has no direct support for relational assignment.
- SQL has no direct support for relational comparisons.
- SQL has no support for type constraints, apart from *a priori* ones.
- SQL doesn't support relation-valued attributes or the GROUP and UNGROUP operators.
- SQL has no direct support for several other relational operators, including in particular SUMMARIZE, DIVIDEBY, projection, SEMIJOIN, SEMIMINUS, and even (to some extent) JOIN.
- SQL doesn't support "=" for every type, nor does it prescribe the semantics of "=" for all types for which that operator *is* supported.
- SQL violates *The Assignment Principle* in various ways.
- SQL has no proper support for relation types or the RELATION type generator.

- SQL permits unnamed columns and duplicate column names.
- SQL fails to support TABLE_DUM and TABLE_DEE.
- SQL handles empty tables in a variety of incorrect ways.
- SQL does explicitly support certain tuple-level update operators.
- SQL permits tables to have no key at all.
- SQL doesn't allow empty keys.
- SQL doesn't allow keys or foreign keys on views, nor does it allow a foreign key to refer to a view.
- SQL permits deferred constraint checking.
- SQL doesn't fully support multiple assignment.
- SQL doesn't support declarative transition constraints.
- SQL aggregate operators can't be nested.
- SQL's support for EXISTS is logically flawed, and it doesn't directly support FORALL at all.

As to why such departures from the relational model are a bad thing: Part of the point of the relational model is that it provides a clear, precise, coherent, logical, carefully thought out definition of how database systems are supposed to behave. To the extent that SQL departs from the model, therefore, it will mean that systems that support it will be unclear, and/or imprecise, and/or incoherent, and/or illogical, and/or not carefully thought out. I don't think any further justification (for adhering to the model) should be necessary.

Exercise 8-2. Regarding *The Information Principle*, see the body of this chapter. Regarding row IDs, see Chapter 4.

Exercise 8-3. See Chapter 4.

Exercise 8-4. See Chapter 3.

Exercise 8-5. Join dependencies are explained in Chapter 7. If relvar *R* satisfies the FD $A \rightarrow B$, it satisfies the JD *{*AB*,*AC*}, where *C* is all of the attributes of *R* apart from *A* and *B* (this is Heath's theorem).

Exercise 8-6. A domain is a type, a relation is a value, and types aren't values. See Chapter 4 for further discussion.

Exercise 8-7. See Chapter 6.

Exercise 8-8. See Chapter 4.

Exercise 8-9. See Chapter 2.

Exercise 8-10. Fifth normal form, 5NF, is "the final normal form," so long as we limit our attention to projection as the decomposition operator and join as the recomposition operator. See Chapter 7 for further discussion.

Exercise 8-11. See Chapter 1.

Exercise 8-12. See Chapter 7.

Exercise 8-13. Yes, it might make sense, but such an operator would just be shorthand. To be specific, *r*/*s* would be shorthand for IS_EMPTY (*r* JOIN *s*) or, equivalently, IS_EMPTY (*r* INTERSECT *s*). Of course, *r* and *s* would have to be of the same type.

Exercise 8-14. See Chapter 5.

Exercise 8-15. False! See Chapter 1.

Exercise 8-16. See Chapter 3.

Exercise 8-17. False. See Chapter 2.

Exercise 8-18. A primary key is a "distinguished" candidate key. The distinction is primarily psychological, not logical, in nature; in SQL, for example, specifying a PRIMARY KEY constraint implies NOT NULL for all component columns, but specifying a UNIQUE constraint doesn't. See Chapter 1 for further discussion.

Exercise 8-19. See Chapter 7.

Exercise 8-20. In principle, no; in practice, possibly yes. See Chapter 7.

Exercise 8-21. True.

Exercise 8-22. The type constraint for type *T* is a specification of the set of values in *T*, and it's checked during the execution of some selector for *T*. See Chapter 6.

Exercise 8-23. Yes in all three cases. See Chapter 2.

Exercise 8-24. Because they're tuple-level operators, by definition. See Chapter 4.

Exercise 8-25. Yes. There are exactly two such relations, TABLE_DUM and TABLE_DEE. See Chapter 3.

Exercise 8-26. See Chapters 1 and 5.

Exercise 8-27. Relation *r* is identically equal to (a) the restriction *r* WHERE TRUE (the *identity restriction* of *r*) and (b) the projection *r*{*A*,*B*,...,*C*} where *A*, *B*, ..., *C* are all of the attributes of *r* (the *identity projection* of *r*).

Exercise 8-28. See Chapter 4.

Exercise 8-29. See Chapter 1.

Exercise 8-30. Relational databases contain exactly one kind of variable, the relvar. Other kinds of databases contain other kinds of variables, either instead of or as well as relvars.

Exercise 8-31. None. See Chapter 2.

Exercise 8-32. Yes in both cases.

Exercise 8-33. Yes for union, no for minus.

Exercise 8-34. A detailed discussion of this issue can be found in Chapter 27 of my book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004). The best brief answer is that XML documents are *values;* they're therefore values of some type, say type XMLDOC; and that type is a type just like any other, meaning in particular that relations can have attributes of that type. In other words, relations can contain XML documents as attribute values. The operators that apply to those values are precisely the operators—perhaps the operators of what's called *XPath* (see Chapter 2)—that are defined for values of type XMLDOC.

Exercise 8-35. It depends on "the PER relation" and the nature of the summarizing. The cardinality of the result is always the same as that of the PER relation, *n* say. In particular, therefore, if—as is often the case—the PER relation is a projection of the relation to be summarized, then *n* will be zero and the overall result will be empty. The value of the new attribute in each of the *n* tuples in that overall result will be whatever the specified summarization gives on an empty argument (for example, SUM gives zero), though of course this latter point is somewhat academic if *n* is in fact zero.

Exercise 8-36. See Chapter 4.

Exercise 8-37. See Chapter 5.

Exercise 8-38. For the definitions, see Chapter 7. As for the misnamings: Of course, "fourth" normal form, 4NF, is really fifth if you count (and 5NF is really sixth and 6NF seventh). The misnamings occurred because when 4NF was first defined, BCNF was still being referred to—in some circles, at least—as a "new, improved" *third* normal form.

Exercise 8-39. Yes in all three cases (at least in the relational model; what about SQL?).

Exercise 8-40. False (but "almost" true—see the answer to Exercise 7.24 in Chapter 7).

Exercise 8-41. TABLE_DEE.

Exercise 8-42. See Chapter 7.

Exercise 8-43. Empty candidate keys are discussed in the answer to Exercise 4.28 in Chapter 4. Empty foreign keys make sense, too; the referential constraint will be violated

if and only if the target relvar is empty and the referencing relvar is not. I'll leave finding a concrete example where this arrangement could be useful as a subsidiary exercise.

Exercise 8-44. True.

Exercise 8-45. It has a left-to-right ordering to its columns, it might include duplicate rows, it might involve nulls, it might involve anonymous columns, it might involve two or more columns with the same name, and it has a top-to-bottom ordering to its rows (even if no ORDER BY is specified!—note that the exercise specified a SELECT *statement*, not a SELECT *expression*).

Exercise 8-46. True.

Exercise 8-47. 8; this count includes the identity projection (the one on all attributes) and also the projection on no attributes. In general, in fact, a relation of n attributes has exactly 2^n distinct projections, precisely because a set of n elements has 2^n distinct subsets.

Exercise 8-48. Keys: 3. FDs: 64.

Exercise 8-49. This is rather an unfair question! As noted in Chapter 5, with what I called "the original divide" in Chapter 5, if *r* and *s* are relations with no common attributes and we form the cartesian product *r* TIMES *s* and then divide the result by *s*, we get back to *r* (so long as *s* isn't empty). Thus, cartesian product and divide are inverses of each other, in a very loose sense. But there are other, more general forms of divide for which no analogous property holds—and even in the "original" case, there's that problem over empty divisors ... Overall, therefore, the answer to the exercise is basically *no*.

As for the subsidiary question: In general, the cartesian product of two sets *A* and *B* is a set of ordered pairs—all possible ordered pairs <*a*,*b*> such that $a \in A$ and $b \in B$. If *A* and *B* are relations, therefore, the conventional cartesian product would be a set of ordered pairs of tuples. In the *relational* cartesian product, by contrast, each such pair of tuples is replaced by a single tuple, that tuple consisting of the set-theoretic union of the two tuples concerned. What's more, of course, those two tuples are required to have no attribute names in common.

Exercise 8-50. The expression *a* JOIN *b* JOIN *c* is equivalent to the join of the join of any two of *a*, *b*, *c* and the other one (we get the same overall result no matter which specific individual joins we perform). Moreover, *r* JOIN *s* is the same as *s* JOIN *r* for all *r* and *s*. The original expression is thus unambiguous; indeed, **Tutorial D** would allow it to be expressed as JOIN{*a*,*b*,*c*}. JOIN{*a*} is just *a*. JOIN{} is TABLE_DEE.

Exercise 8-51. Here's one way (illustrated by example). The relational comparison *a* = *b* can be simulated thus:

```
NOT EXISTS ( SELECT a.* FROM a
             EXCEPT
             SELECT b.* FROM b )
AND
NOT EXISTS ( SELECT b.* FROM b
             EXCEPT
             SELECT a.* FROM a )
```

Note, however, that if this expression evaluates to TRUE, it still doesn't guarantee that *a* is equal to *b* if either *a* or *b* includes any duplicate rows! Defining equality for two bags (as opposed to two sets) is nontrivial.

Exercise 8-52. Yes, for a variety of reasons:

- If users interact with views instead of base relvars, then they should be able to regard those views as being as much like base relvars as possible. Ideally, in fact, they shouldn't even have to know they *are* views, but should be able to treat them as if they really were base relvars. And just as the user of a base relvar needs to know what keys that base relvar has (in general), so the user of a view needs to know what keys that view has (again, in general). Explicitly declaring those keys is the obvious way to make that information available.
- The DBMS might not be able to infer such keys for itself (this is almost certainly the case with most if not all DBMSs on the market today). Explicit declarations are thus likely to be the only means available of informing the DBMS—as well as the user—of the existence of such keys.
- Even if the DBMS were able to deduce such keys for itself, explicit declarations would at least enable the system to check that its deductions and those explicit specifications were consistent.
- The view definer might have some knowledge that the DBMS doesn't, and might thus be able to improve on the DBMS's deductions.
- Such a facility could provide a simple and convenient way of stating certain constraints that could otherwise be stated only in a very circumlocutory fashion.

Exercise 8-53. If *B* is the empty set, it's necessarily a subset of *A*, so the FD is trivial. If *A* is the empty set, then every tuple of *R* must have the same value for *B* (why, exactly?). If *K* is the empty set, then *R* is limited to being of cardinality at most one (again, why exactly?).

Exercise 8-54. They all are.

Exercise 8-55. See Chapter 6.

Exercise 8-56. See Chapter 7.

Exercise 8-57. Just as 0 is the identity with respect to "+" in ordinary arithmetic, so TABLE_DEE is the identity with respect to JOIN in the relational algebra.

Exercise 8-58. Monadic operators:

- Any restriction of DEE yields DEE if the restriction condition evaluates to TRUE, DUM otherwise; any restriction of DUM yields DUM.
- Projection of any relation on no attributes yields DUM if the original relation is empty, DEE otherwise.
- Extending DEE or DUM to add a new attribute yields a relation of degree one and the same cardinality as its input.

Dyadic operators:

a	*b*	*a* UNION *b*	*a* INTERSECT *b*	*a* MINUS *b*
DEE	DEE	DEE	DEE	DUM
DEE	DUM	DEE	DUM	DEE
DUM	DEE	DEE	DUM	DUM
DUM	DUM	DUM	DUM	DUM

Note how reminiscent these tables are of the truth tables for OR, AND, and AND NOT, respectively; of course, the resemblance isn't a coincidence.

Exercise 8-59. See the body of the chapter. *Note:* There really is just one relatonal model (despite the fact that it's evolved over the years and will presumably continue to do so), but there are doubtless many equivalent ways of defining it. "Many different definitions" does *not* equate to "many different models"—despite popular misconceptions to the contrary.

Exercise 8-60. This is a rhetorical question, of course; if you don't know by now, go back and start reading the book again from the beginning! Thank you for your attention.