CHAPTER 3

An Introduction to Relational Databases

Copia con fines educativos

3.1 INTRODUCTION

As explained in Chapter 1, the emphasis in this book is heavily on relational systems. In particular, Part II covers the theoretical foundations of such systems—that is, the relational model—in considerable depth. The purpose of the present chapter is to give a preliminary, intuitive, and very informal introduction to the material to be addressed in Part II (and to some extent in subsequent parts too), in order to pave the way for a better understanding of those later parts of the book. Most of the topics mentioned will be discussed much more formally, and in much more detail, in those later chapters.

3.2 AN INFORMAL LOOK AT THE RELATIONAL MODEL

We claimed in Chapter 1 that relational systems are based on a formal foundation, or theory, called *the relational model of data*. The relational model is often described as having the following three aspects:

- *Structural aspect:* The data in the database is perceived by the user as tables, and nothing but tables.
- *Integrity aspect:* Those tables satisfy certain integrity constraints, to be discussed toward the end of this section.
- *Manipulative aspect:* The operators available to the user for manipulating those tables—for example, for purposes of data retrieval—are operators that derive tables from tables. Of those operators, three particularly important ones are *restrict*, *project*, and *join*.

A simple relational database, the departments-and-employees database, is shown in Fig. 3.1. As you can see, that database is indeed "perceived as tables" (and the meanings of those tables are intended to be self-evident). Fig. 3.2 shows some sample restrict, project, and join operations against the database of Fig. 3.1. Here are (very loose!) definitions of those operations:

- The *restrict* operation extracts specified rows from a table. *Note:* Restrict is sometimes called *select*; we prefer *restrict* because the operator is not the same as the SELECT of SQL.
- The *project* operation extracts specified columns from a table.
- The *join* operation combines two tables into one on the basis of common values in a common column.

Of the examples in Fig. 3.2, the only one that seems to need any further explanation is the join example. Join requires the two tables to have a common column, which tables DEPT and EMP do (they both have a column called DEPT#), and so they can be joined on

DEPT

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.1 The departments-and-employees database (sample values)

Copia con fines educativos

Restrict:

DEPTs where BUDGET > 8M

Result:

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M

Project:

DEPTs over DEPT#, BUDGET

Result:

DEPT#	BUDGET
D1	10M
D2	12M
D3	5M

Join:

DEPTs and EMPs over DEPT#

Result:

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K
D1	Marketing	10M	E2	Cheng	42K
D2	Development	12M	E3	Finzi	30K
D2	Development	12M	E4	Saito	35K

Fig. 3.2 Restrict, project, and join (examples)

the basis of common values in that column. To be specific, a given row from table DEPT will join to a given row in table EMP (to yield a row of the result table) if and only if the two rows in question have a common DEPT# value. For example, the DEPT and EMP rows

DEPT#	DNAME	BUDGET
D1	Marketing	10M

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K

(column names shown for explicitness) join together to produce the result row

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

because they have the same value, D1, in the common column. Note that the common value appears once, not twice, in the result row. The overall result of the join contains all possible rows that can be obtained in this manner, and no other rows. Observe in particular that since no EMP row has a DEPT# value of D3 (i.e., no employee is currently assigned to that department), no row for D3 appears in the result, even though there *is* a row for D3 in table DEPT.

Now, one point that Fig. 3.2 clearly shows is that *the result of each of the three operations is another table* (in other words, the operators are indeed "operators that derive tables from tables," as required). This is the closure property of relational systems, and it is very important. Basically, because the output of any operation is the same kind of object as the input—they are all tables—*the output from one operation can become input*

Copia con fines educativos

to another. Thus it is possible, for example, to take a projection of a join, a join of two restrictions, a restriction of a projection, and so on. In other words, it is possible to write *nested relational expressions*—that is, relational expressions in which the operands themselves are represented by relational expressions, not necessarily just by simple table names. This fact in turn has numerous important consequences, as we will see later, both in this chapter and in many subsequent ones.

By the way, when we say that the output from each operation is another table, it is important to understand that we are talking *from a conceptual point of view.* We do not mean to imply that the system actually has to materialize the result of every individual operation in its entirety.[1] For example, suppose we are trying to compute a restriction of a join. Then, as soon as a given row of the join is formed, the system can immediately test that row against the specified restriction condition to see whether it belongs in the final result, and immediately discard it if not. In other words, the intermediate result that is the output from the join might never exist as a fully materialized table in its own right at all. As a general rule, in fact, the system tries very hard *not* to materialize intermediate results in their entirety, for obvious performance reasons. *Note:* If intermediate results are fully materialized, the overall expression evaluation strategy is called (unsurprisingly) materialized evaluation; if intermediate results are passed piecemeal to subsequent operations, it is called pipelined evaluation.

Another point that Fig. 3.2 also clearly illustrates is that the operations are all set-at-a-time, not row-at-a-time; that is, the operands and results are whole tables, not just single rows, and tables contain *sets* of rows. (A table containing a set of just one row is legal, of course, as is an *empty* table, i.e., one containing no rows at all.) For example, the join in Fig. 3.2 operates on two tables of three and four rows respectively, and returns a result table of four rows. By contrast, the operations in nonrelational systems are typically at the row- or record-at-a-time level; thus, this *set processing capability* is a major distinguishing characteristic of relational systems (see further discussion in Section 3.5).

Let us return to Fig. 3.1 for a moment. There are a couple of additional points to be made in connection with the sample database of that figure:

- First, note that relational systems require only that the database be *perceived by the user* as tables. Tables are the logical structure in a relational system, not the physical structure. At the physical level, in fact, the system is free to store the data any way it likes—using sequential files, indexing, hashing, pointer chains, compression, and so on—provided only that it can map that stored representation to tables at the logical level. Another way of saying the same thing is that tables represent an *abstraction* of the way the data is physically stored—an abstraction in which numerous storage-level details (such as stored record placement, stored record sequence, stored data value representations, stored record prefixes, stored access structures such as indexes, and so forth) are all *hidden from the user.*

  Incidentally, the term *logical structure* in the foregoing paragraph is intended to encompass both the conceptual and external levels, in ANSI/SPARC terms. The point is that—as explained in Chapter 2—the conceptual and external levels in a relational

[1] In other words, to repeat from Chapter 1, the relational model is indeed a *model*—it has nothing to say about implementation.

Copia con fines educativos

system will both be relational, but the internal level will not be. In fact, relational theory as such has nothing to say about the internal level at all; it is, to repeat, concerned with how the database looks to the *user*.[2] The only requirement is that, to repeat, whatever physical structure is chosen at the internal level must fully support the required logical structure.

- Second, relational databases abide by a very nice principle, called *The Information Principle: The entire information content of the database is represented in one and only one way—namely, as explicit values in column positions in rows in tables.* This method of representation is the *only* method available (at the logical level, that is) in a relational system. In particular, there are no *pointers* connecting one table to another. In Fig. 3.1, for example, there is a connection between the D1 row of table DEPT and the E1 row of table EMP, because employee E1 works in department D1; but that connection is represented, not by a pointer, but by the appearance of the *value* D1 in the DEPT# position of the EMP row for E1. In nonrelational systems such as IMS or IDMS, by contrast, such information is typically represented—as mentioned in Chapter 1—by some kind of *pointer* that is explicitly visible to the user.

  Note: We will explain in Chapter 26 just why allowing such user-visible pointers would constitute a violation of *The Information Principle*. Also, when we say there are no pointers in a relational database, we do not mean there cannot be pointers at the *physical level*—on the contrary, there certainly can, and indeed there almost certainly will. But, to repeat, all such physical storage details are concealed from the user in a relational system.

So much for the structural and manipulative aspects of the relational model; now we turn to the integrity aspect. Consider the departments-and-employees database of Fig. 3.1 once again. In practice, that database might be required to satisfy any number of integrity constraints—for example, employee salaries might have to be in the range 25K to 95K (say), department budgets might have to be in the range 1M to 15M (say), and so on. Certain of those constraints are of such major pragmatic importance, however, that they enjoy some special nomenclature. To be specific:

1. Each row in table DEPT must include a unique DEPT# value; likewise, each row in table EMP must include a unique EMP# value. We say, loosely, that columns DEPT# in table DEPT and EMP# in table EMP are the **primary keys** for their respective tables. (Recall from Chapter 1 that we indicate primary keys in our figures by double underlining.)
2. Each DEPT# value in table EMP must exist as a DEPT# value in table DEPT, to reflect the fact that every employee must be assigned to an existing department. We say, loosely, that column DEPT# in table EMP is a **foreign key**, referencing the primary key of table DEPT.

[2] It is an unfortunate fact that most of today's SQL products do not support this aspect of the theory properly. To be more specific, they typically support only rather restrictive conceptual/internal mappings; typically, in fact, they map one logical table directly to one stored file. This is one reason why (as noted in Chapter 1) those products do not provide as much data independence as relational technology is theoretically capable of. See Appendix A for further discussion.

Obra con fines educativos

A More Formal Definition

We close this section with a somewhat more formal definition of the relational model, for purposes of subsequent reference (despite the fact that the definition is quite abstract and will not make much sense at this stage). Briefly, the relational model consists of the following five components:

1. An open-ended collection of scalar types (including in particular the type *boolean* or *truth value*)
2. A relation type generator and an intended interpretation for relations of types generated thereby
3. Facilities for defining relation variables of such generated relation types
4. A relational assignment operation for assigning relation values to such relation variables
5. An open-ended collection of generic relational operators ("the relational algebra") for deriving relation values from other relation values

As you can see, the relational model is very much more than just "tables plus restrict, project, and join," though it is often characterized in such a manner informally.

By the way, you might be surprised to see no explicit mention of integrity constraints in the foregoing definition. The fact is, however, such constraints represent just one application of the relational operators (albeit a very important one); that is, such constraints are formulated in terms of those operators, as we will see in Chapter 9.

3.3 RELATIONS AND RELVARS

If it is true that a relational database is basically just a database in which the data is perceived as tables—and of course it *is* true—then a good question to ask is: Why exactly do we call such a database relational? The answer is simple (in fact, we mentioned it in Chapter 1): *Relation* is just a mathematical term for a table—to be precise, a table of a specific kind (details to be pinned down in Chapter 6). Thus, for example, we can say that the departments-and-employees database of Fig. 3.1 contains two *relations*.

Now, in informal contexts it is usual to treat the terms *relation* and *table* as if they were synonymous; in fact, the term *table* is used much more often than the term *relation* in practice. But it is worth taking a moment to understand why the term *relation* was introduced in the first place. Briefly:

- As we have seen, relational systems are based on the relational model. The relational model in turn is an abstract theory of data that is based on certain aspects of mathematics (mainly set theory and predicate logic).
- The principles of the relational model were originally laid down in 1969–70 by E. F. Codd, at that time a researcher at IBM. It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into a field (database management) that prior to

Copia con fines educativos

that time was all too deficient in any such qualities. Codd's ideas were first widely disseminated in a now classic paper, "A Relational Model of Data for Large Shared Data Banks" (reference [6.1] in Chapter 6).

- Since that time, those ideas—by now almost universally accepted—have had a wide-ranging influence on just about every aspect of database technology, and indeed on other fields as well, such as the fields of artificial intelligence, natural language processing, and hardware design.

Now, the relational model as originally formulated by Codd very deliberately made use of certain terms, such as the term *relation* itself, that were not familiar in IT circles at that time (even though the concepts in some cases were). The trouble was, many of the more familiar terms were very fuzzy—they lacked the precision necessary to a formal theory of the kind that Codd was proposing. For example, consider the term *record*. At different times and in different contexts, that single term can mean either a record *occurrence* or a record *type;* a *logical* record or a *physical* record; a *stored* record or a *virtual* record; and perhaps other things besides. The relational model therefore does not use the term *record* at all—instead, it uses the term *tuple* (rhymes with *couple*), to which it gives a very precise definition. We will discuss that definition in detail in Chapter 6; for present purposes, it is sufficient to say that the term *tuple* corresponds approximately to the notion of a row (just as the term *relation* corresponds approximately to the notion of a table).

In the same kind of way, the relational model does not use the term *field;* instead, it uses the term *attribute*, which for present purposes we can say corresponds approximately to the notion of a column in a table.

When we move on to study the more formal aspects of relational systems in Part II, we will make use of the formal terminology, but in this chapter we are not trying to be so formal (for the most part, at any rate), and we will mostly stick to terms such as *row* and *column* that are reasonably familiar. However, one formal term we will start using a lot from this point forward is the term *relation* itself.

We return to the departments-and-employees database of Fig. 3.1 to make another important point. The fact is, DEPT and EMP in that database are really relation variables: variables, that is, whose values are relation values (different relation values at different times). For example, suppose EMP currently has the value—the *relation* value, that is—shown in Fig. 3.1, and suppose we delete the row for Saito (employee number E4):

```
DELETE EMP WHERE EMP# = EMP# ('E4') ;
```

The result is shown in Fig. 3.3.

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K

Fig. 3.3 Relation variable EMP after deleting E4 row

Copia con fines educativos

Conceptually, what has happened here is that *the old relation value of EMP has been replaced* en bloc *by an entirely new relation value.* Of course, the old value (with four rows) and the new one (with three) are very similar, but conceptually they *are* different values. Indeed, the delete operation in question is basically just shorthand for a certain relational assignment operation that might look like this:

```
EMP  :=  EMP WHERE NOT ( EMP# = EMP# ('E4') ) ;
```

As in all assignments, what is happening here, conceptually, is that (a) the *expression* on the right side is evaluated, and then (b) the result of that evaluation is assigned to the *variable* on the left side (naturally that left side must identify a variable specifically). As already stated, the net effect is thus to replace the "old" EMP value by a "new" one. (As an aside, we remark that we have now seen our first examples of the use of the Tutorial D language—both the original DELETE and the equivalent assignment are expressed in that language.)

In analogous fashion, relational INSERT and UPDATE operations are also basically shorthand for certain relational assignments. See Chapter 6 for further details.

Now, it is an unfortunate fact that much of the literature uses the term *relation* when what it really means is a relation *variable* (as well as when it means a relation *per se*—that is, a relation *value*). Historically, however, this practice has certainly led to some confusion. Throughout this book, therefore, we will distinguish very carefully between relation variables and relations *per se*: following reference [3.3], in fact, we will use the term relvar as a convenient shorthand for *relation variable*, and we will take care to phrase our remarks in terms of relvars, not relations, when it really is relvars that we mean.[3] Please note, therefore, that from this point forward we take the unqualified term *relation* to mean a relation value specifically (just as we take, e.g., the unqualified term *integer* to mean an integer value specifically), though we will also use the qualified term *relation value* on occasion, for emphasis.

Before going any further, we should warn you that the term *relvar* is not in common usage—but it should be! We really do feel it is important to be clear about the distinction between relations *per se* and relation variables. (We freely admit that earlier editions of this book failed in this respect, but then so did the rest of the literature. What is more, most of it still does.) Note in particular that, by definition, update operations and integrity constraints—see Chapters 6 and 9, respectively—both apply specifically to relvars, not relations.

3.4 WHAT RELATIONS MEAN

In Chapter 1, we mentioned the fact that columns in relations have associated data types (*types* for short, also known as *domains*). And at the end of Section 3.2, we said that the relational model includes "an open-ended set of . . . types." Note carefully that the fact that the set is open-ended implies among other things that users will be able to define their

[3] The distinction between relation values and relation variables is actually a special case of the distinction between values and variables in general. We will examine this latter distinction in depth in Chapter 5.

Copia com fins educativos

own types (as well as being able to make use of system-defined or *built-in* types, of course). For example, we might have user-defined types as follows (Tutorial D syntax again; the ellipses ". . ." denote portions of the definitions that are not germane to the present discussion):

```
TYPE EMP# ... ;
TYPE NAME ... ;
TYPE DEPT# ... ;
TYPE MONEY ... ;
```

Type EMP#, for example, can be regarded (among other things) as *the set of all possible employee numbers;* type NAME as *the set of all possible names;* and so on.

Now consider Fig. 3.4, which is basically the EMP portion of Fig. 3.1 expanded to show the column data types. As the figure indicates, every relation—to be more precise, every relation *value*—has two parts, a set of column-name:type-name pairs (the heading), together with a set of rows that conform to that heading (the body). *Note:* In practice we often ignore the type-name components of the heading, as indeed we have done in all of our examples prior to this point, but you should understand that, conceptually, they are always there.

Now, there is a very important (though perhaps unusual) way of thinking about relations, and that is as follows:

1. Given a relation *r*, the heading of *r* denotes a certain predicate (where a predicate is just a *truth-valued function* that, like all functions, takes a set of *parameters*).
2. As mentioned briefly in Chapter 1, each row in the body of *r* denotes a certain true proposition, obtained from the predicate by substituting certain *argument* values of the appropriate type for the parameters of the predicate ("instantiating the predicate").

   In the case of Fig. 3.4, for example, the predicate looks something like this:

   Employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY

(the parameters are EMP#, ENAME, DEPT#, and SALARY, corresponding of course to the four EMP columns). And the corresponding true propositions are:

Employee E1 is named Lopez, works in department D1, and earns salary 40K

(obtained by substituting the EMP# value E1, the NAME value Lopez, the DEPT# value D1, and the MONEY value 40K for the appropriate parameters);

EMP# : EMP#	ENAME : NAME	DEPT# : DEPT#	SALARY : MONEY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.4 Sample EMP relation value, showing column types

Copia con fines educativos

Employee E2 is named Cheng, works in department D1, and earns salary 42K

(obtained by substituting the EMP# value E2, the NAME value Cheng, the DEPT# value D1, and the MONEY value 42K for the appropriate parameters); and so on. In a nutshell, therefore:

- *Types* are (sets of) things we can talk about.
- *Relations* are (sets of) things we say about the things we can talk about.

(There is a nice analogy here that might help you appreciate and remember these important points: *Types are to relations as nouns are to sentences.*) Thus, in the example, the things we can talk about are employee numbers, names, department numbers, and money values, and the things we say are true utterances of the form "The employee with the specified employee number has the specified name, works in the specified department, and earns the specified salary."

It follows from all of the foregoing that:

1. Types and relations are both *necessary* (without types, we have nothing to talk about; without relations, we cannot say anything).
2. Types and relations are *sufficient*, as well as necessary—that is, we do not need anything else, logically speaking.
3. *Types and relations are not the same thing.* It is an unfortunate fact that certain commercial products—not relational ones, by definition!—are confused over this very point. We will discuss this confusion in Chapter 26 (Section 26.2).

By the way, it is important to understand that *every* relation has an associated predicate, including relations that are derived from others by means of operators such as join. For example, the DEPT relation of Fig. 3.1 and the three result relations of Fig. 3.2 have predicates as follows:

- DEPT: *Department DEPT# is named DNAME and has budget BUDGET*
- Restriction of DEPT where BUDGET > 8M: *Department DEPT# is named DNAME and has budget BUDGET, which is greater than eight million dollars*
- Projection of DEPT over DEPT# and BUDGET: *Department DEPT# has some name and has budget BUDGET*
- Join of DEPT and EMP over DEPT#: *Department DEPT# is named DNAME and has budget BUDGET and employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY* (note that this predicate has six parameters, not seven—the two references to DEPT# denote the same parameter)

Finally, we observe that relvars have predicates too: namely, the predicate that is common to all of the relations that are possible values of the relvar in question. For example, the predicate for relvar EMP is:

Employee EMP# is named ENAME, works in department DEPT#, and earns salary SALARY

Copia con fines educativos

3.5 OPTIMIZATION

As explained in Section 3.2, the relational operators (restrict, project, join, and so on) are all *set-level*. As a consequence, relational languages are often said to be nonprocedural, on the grounds that users specify *what*, not *how*—that is, they say what they want, without specifying a procedure for getting it. The process of "navigating" around the stored data in order to satisfy user requests is performed automatically by the system, not manually by the user. For this reason, relational systems are sometimes said to perform automatic navigation. In nonrelational systems, by contrast, navigation is generally the responsibility of the user. A striking illustration of the benefits of automatic navigation is shown in Fig. 3.5, which contrasts a certain SQL INSERT statement with the "manual navigation" code the user might have to write to achieve an equivalent effect in a nonrelational system (actually a CODASYL network system: the example is taken from the chapter on network databases in reference [1.5]). *Note:* The database is the well-known suppliers-and-parts database. See Section 3.9 for further explanation.

```
INSERT INTO SP ( S#, P#, QTY )
       VALUES ( 'S4', 'P3', 1000 ) ;
```

```
MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
   ACCEPT S-SP-ADDR FROM S-SP CURRENCY
   FIND OWNER WITHIN P-SP
   GET P
   IF P# IN P < 'P3'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN S-SP .
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
   ACCEPT P-SP-ADDR FROM P-SP CURRENCY
   FIND OWNER WITHIN S-SP
   GET S
   IF S# IN S < 'S4'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP
```

Fig. 3.5 Automatic *vs.* manual navigation

Cópia com fins educativos

Despite the remarks of the previous paragraph, it has to be said that *nonprocedural* is not a very satisfactory term, common though it is, because procedurality and nonprocedurality are not absolutes. The best that can be said is that some language *A* is either more or less procedural than some other language *B*. Perhaps a better way of putting matters would be to say that relational languages are at *a higher level of abstraction* than nonrelational languages (as Fig. 3.5 suggests). Fundamentally, it is this raising of the level of abstraction that is responsible for the increased productivity that relational systems can provide.

Deciding just how to perform the automatic navigation referred to above is the responsibility of a very important DBMS component called the optimizer (we mentioned this component briefly in Chapter 2). In other words, for each access request from the user, it is the job of the optimizer to choose an efficient way to implement that request. By way of an example, let us suppose the user issues the following query (Tutorial D once again):

```
( EMP WHERE EMP# = EMP# ('E4') ) { SALARY }
```

Explanation: The expression inside the outer parentheses ("EMP WHERE . . .") denotes a restriction of the current value of relvar EMP to just the row for employee E4. The column name in braces ("SALARY") then causes the result of that restriction to be projected over the SALARY column. The result of that projection is a single-column, single-row relation that contains employee E4's salary. (Incidentally, note that we are implicitly making use of the relational *closure* property in this example—we have written a nested relational expression, in which the input to the projection is the output from the restriction.)

Now, even in this very simple example, there are probably at least two ways of performing the necessary data access:

1. By doing a physical sequential scan of (the stored version of) relvar EMP until the required data is found
2. If there is an index on (the stored version of) the EMP# column—which in practice there probably will be, because EMP# values are supposed to be unique, and many systems in fact *require* an index in order to enforce uniqueness—then by using that index to go directly to the required data

The optimizer will choose which of these two strategies to adopt. More generally, given any particular request, the optimizer will make its choice of strategy for implementing that request on the basis of considerations such as the following:

- Which relvars are referenced in the request
- How big those relvars currently are
- What indexes exist
- How selective those indexes are
- How the data is physically clustered on the disk
- What relational operations are involved

Copia con fines educativos

and so on. To repeat, therefore: Users specify only what data they want, not how to get to that data; the access strategy for getting to that data is chosen by the optimizer ("automatic navigation"). Users and user programs are thus independent of such access strategies, which is of course essential if data independence is to be achieved.

We will have a lot more to say about the optimizer in Chapter 18.

3.6 THE CATALOG

As explained in Chapter 2, the DBMS must provide a catalog or dictionary function. The catalog is the place where—among other things—all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal, external/external) are kept. In other words, the catalog contains detailed information, sometimes called *descriptor information* or *metadata*, regarding the various objects that are of interest to the system itself. Examples of such objects are relvars, indexes, users, integrity constraints, security constraints, and so on. Descriptor information is essential if the system is to do its job properly. For example, the optimizer uses catalog information about indexes and other auxiliary structures, as well as much other information, to help it decide how to implement user requests (see Chapter 18). Likewise, the authorization subsystem uses catalog information about users and security constraints to grant or deny such requests in the first place (see Chapter 17).

Now, one of the nice features of relational systems is that, in such a system, *the catalog itself consists of relvars* (more precisely, *system* relvars, so called to distinguish them from ordinary user ones). As a result, users can interrogate the catalog in exactly the same way they interrogate their own data. For example, the catalog in an SQL system might include two system relvars called TABLE and COLUMN, the purpose of which is to describe the tables (or relvars) in the database and the columns in those tables. For the departments-and-employees database of Fig. 3.1, the TABLE and COLUMN relvars might look in outline as shown in Fig. 3.6.[4]

Note: As mentioned in Chapter 2, the catalog should normally be self-describing—that is, it should include entries describing the catalog relvars themselves (see Exercise 3.3).

Now suppose some user of the departments-and-employees database wants to know exactly what columns relvar DEPT contains (obviously we are assuming that for some reason the user does not already have this information). Then the expression

```
( COLUMN WHERE TABNAME = 'DEPT' ) { COLNAME }
```

does the job.

Here is another example: "Which relvars include a column called EMP#?"

```
( COLUMN WHERE COLNAME = 'EMP#' ) { TABNAME }
```

[4] Note that the presence of column ROWCOUNT in Fig. 3.6 suggests that INSERT and DELETE operations on the database will cause an update to the catalog es a side effect. In practice, ROWCOUNT might be updated only on request (e.g., when some utility is run), meaning that values of that column might not always be current.

Copia con fines informativos

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
DEPT	3	3	.....
EMP	4	4	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
DEPT	DEPT#	.....
DEPT	DNAME	.....
DEPT	BUDGET	.....
EMP	EMP#	.....
EMP	ENAME	.....
EMP	DEPT#	.....
EMP	SALARY	.....
.....	.....	.....

Fig. 3.6 . Catalog for the departments-and-employees database (in outline)

Exercise: What does the following do?

```
( ( TABLE JOIN COLUMN )
        WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

3.7 BASE RELVARS AND VIEWS

We have seen that, starting with a set of relvars such as DEPT and EMP, together with a set of relation values for those relvars, relational expressions allow us to obtain further relation values from those given ones. It is time to introduce a little more terminology. The original (given) relvars are called **base relvars**, and their values are called **base relations**; a relation that is not a base relation but can be obtained from the base relations by means of some relational expression is called a **derived**, or **derivable**, relation. *Note:* Base relvars are called real relvars in reference [3.3].

Now, relational systems obviously have to provide a means for creating the base relvars in the first place. In SQL, for example, this task is performed by the CREATE TABLE statement (TABLE here meaning, very specifically, a base relvar, or what SQL calls a base table). And base relvars obviously have to be *named*—for example:

```
CREATE TABLE EMP ... ;
```

However, relational systems usually support another kind of named relvar also, called a view, whose value at any given time is a *derived* relation (and so a view can be thought of, loosely, as a **derived relvar**). The value of a given view at a given time is whatever results from evaluating a certain relational expression at that time; the relational expression in question is specified when the view in question is created. For example, the statement

```
CREATE VIEW TOPEMP AS
     ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ;
```

Copia con fines educativos

TOPEMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Fig. 3.7 TOPEMP as a view of EMP (unshaded portions)

might be used to define a view called TOPEMP. (For reasons that are unimportant at this juncture, this example is expressed in a mixture of SQL and Tutorial D.)

When this statement is executed, the relational expression following the AS—the view-defining expression—is not evaluated but is merely remembered by the system in some way (actually by saving it in the catalog, under the specified name TOPEMP). To the user, however, it is now as if there really were a relvar in the database called TOPEMP, with current value as indicated in the unshaded portions (only) of Fig. 3.7. And the user should be able to operate on that view exactly as if it were a base relvar. *Note:* If (as suggested previously) DEPT and EMP are thought of as real relvars, then TOPEMP might be thought of as a *virtual* relvar—that is, a relvar that appears to exist in its own right, but in fact does not (its value at any given time depends on the value(s) of certain other relvar(s)). In fact, views are called virtual relvars in reference [3.3].

Note carefully, however, that although we say that the value of TOPEMP is the relation that would result if the view-defining expression were evaluated, we do *not* mean we now have *a separate copy* of the data; that is, we do not mean the view-defining expression actually *is* evaluated and the result materialized. On the contrary, the view is effectively just a kind of "window" into the underlying base relvar EMP. As a consequence, any changes to that underlying relvar will be automatically and instantaneously visible through that window (assuming they lie within the unshaded portion). Likewise, changes to TOPEMP will automatically and instantaneously be applied to relvar EMP, and hence be visible through the window (see later for an example).

Here is a sample retrieval operation against view TOPEMP:

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

Given the sample data of Fig. 3.7, the result will look like this:

EMP#	SALARY
E1	40K
E4	35K

Conceptually, operations against a view like the retrieval operation just shown are handled by replacing references to the view name by the view-defining expression (i.e., the expression that was saved in the catalog). In the example, therefore, the original expression

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

Copia con fines educativos

is modified by the system to become

```
( ( ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ) .
          WHERE SALARY < 42K ) .{ EMP#, SALARY }
```

(we have italicized the view name in the original expression and the replacement text in the modified version). The modified expression can then be simplified to just

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) { EMP#, SALARY }
```

(see Chapter 18), and this latter expression when evaluated yields the result shown earlier. In other words, the original operation against the view is effectively converted into an equivalent operation against the underlying base relvar, and that equivalent operation is then executed in the normal way (more accurately, *optimized* and executed in the normal way).

By way of another example, consider the following DELETE operation:

```
DELETE TOPEMP WHERE SALARY < 42K ;
```

The DELETE that is actually executed looks something like this:

```
DELETE EMP WHERE SALARY > 33K AND SALARY < 42K ;
```

Now, the view TOPEMP is very simple, consisting as it does just of a row-and-column subset of a single underlying base relvar (loosely speaking). In principle, however, a view definition, since it is essentially just a named relational expression, can be *of arbitrary complexity* (it can even refer to other views). For example, here is a view whose definition involves a join of two underlying base relvars:

```
CREATE VIEW JOINEX AS
   ( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) { EMP#, DEPT# } ;
```

We will return to the whole question of view definition and view processing in Chapter 10.

Incidentally, we can now explain the remark in Chapter 2, near the end of Section 2.2, to the effect that the term *view* has a rather specific meaning in relational contexts that is not identical to the meaning assigned to it in the ANSI/SPARC architecture. At the external level of that architecture, the database is perceived as an "external view," defined by an external schema (and different users can have different external views). In relational systems, by contrast, a view is, specifically, a *named, derived, virtual relvar,* as previously explained. Thus, the relational analog of an ANSI/SPARC "external view" is (typically) a collection of several relvars, each of which is a view in the relational sense, and the "external schema" consists of definitions of those views. (It follows that views in the relational sense are the relational model's way of providing **logical data independence**, though once again it has to be said that today's SQL products are sadly deficient in this regard. See Chapter 10.)

Now, the ANSI/SPARC architecture is quite general and allows for arbitrary variability between the external and conceptual levels. In principle, even the *types* of data structures supported at the two levels could be different; for example, the conceptual level

Copia con fines educativos

could be relational, while a given user could have an external view that was hierarchic.[5] In practice, however, most systems use the same type of structure as the basis for both levels, and relational products are no exception to this general rule—views are still relvars, just like the base relvars are. And since the same type of object is supported at both levels, the same data sublanguage (usually SQL) applies at both levels. Indeed, the fact that a view is a relvar is precisely one of the strengths of relational systems; it is important in just the same way as the fact that a subset is a set is important in mathematics. *Note:* SQL products and the SQL standard (see Chapter 4) often seem to miss this point, however, inasmuch as they refer repeatedly to "tables and views," with the tacit implication that a view is not a table. You are strongly advised *not* to fall into this common trap of taking "tables" (or "relvars") to mean, specifically, *base* tables (or relvars) only.

There is one final point that needs to be made on the subject of base relvars *vs.* views, as follows. The base relvar *vs.* view distinction is frequently characterized thus:

- Base relvars "really exist," in the sense that they represent data that is physically stored in the database.
- Views, by contrast, do not "really exist" but merely provide different ways of looking at "the real data."

However, this characterization, though perhaps useful in informal contexts, does not accurately reflect the true state of affairs. It is true that users can *think* of base relvars as if they were physically stored; in a way, in fact, the whole point of relational systems is to allow users to think of base relvars as physically existing, while not having to concern themselves with how those relvars are actually represented in storage. But—and it is a big but—this way of thinking should *not* be construed as meaning that a base relvar is physically stored in any kind of direct way (e.g., as a single stored file). As explained in Section 3.2, base relvars are best thought of as an *abstraction* of some collection of stored data—an abstraction in which all storage-level details are concealed. In principle, there can be an arbitrary degree of differentiation between a base relvar and its stored counterpart.[6]

A simple example might help to clarify this point. Consider the departments-and-employees database once again. Most of today's relational systems would probably implement that database with two stored files, one for each of the two base relvars. But there is absolutely no logical reason why there should not be just one stored file of *hierarchic* stored records, each consisting of (a) the department number, name, and budget for some given department, together with (b) the employee number, name, and salary for each employee who happens to be in that department. In other words, the data can be physically stored in whatever way seems appropriate (see Appendix A for a discussion of further possibilities), but it always looks the same at the logical level.

[5] We will see an example of this possibility in Chapter 27.

[6] The following quote from a recent book displays several of the confusions discussed in this paragraph, as well as others discussed in Section 3.3 earlier: "[It] is important to make a distinction between stored relations, which are *tables*, and virtual relations, which are *views* . . . [We] shall use *relation* only where a table or a view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term *base relation* or *base table*." The quote is, regrettably, not at all atypical.

Copia con fines educativos

3.8 TRANSACTIONS

Note: The topic of this section is not peculiar to relational systems. We cover it here nevertheless, because an understanding of the basic idea is needed in order to appreciate certain aspects of the material to come in Part II. However, our coverage at this point is deliberately not very deep.

In Chapter 1 we said that a transaction is a "logical unit of work," typically involving several database operations. Clearly, the user needs to be able to inform the system when distinct operations are part of the same transaction, and the BEGIN TRANSACTION, COMMIT, and ROLLBACK operations are provided for this purpose. Basically, a transaction begins when a BEGIN TRANSACTION operation is executed, and terminates when a corresponding COMMIT or ROLLBACK operation is executed. For example (pseudocode):

```
BEGIN TRANSACTION ; /* move $$$ from account A to account B */
UPDATE account A ;                              /* withdrawal   */
UPDATE account B ;                              /* deposit      */
IF everything worked fine
   THEN COMMIT ;                                /* normal end */
   ELSE ROLLBACK ;                              /* abnormal end */
END IF ;
```

Points arising:

1. Transactions are guaranteed to be atomic—that is, they are guaranteed (from a logical point of view) either to execute in their entirety or not to execute at all.[7] even if (say) the system fails halfway through the process.
2. Transactions are also guaranteed to be durable, in the sense that once a transaction successfully executes COMMIT, its updates are guaranteed to appear in the database, even if the system subsequently fails at any point. (It is this durability property of transactions that makes the data in the database *persistent*, in the sense of Chapter 1.)
3. Transactions are also guaranteed to be isolated from one another, in the sense that database updates made by a given transaction *T1* are not made visible to any distinct transaction *T2* until and unless *T1* successfully executes COMMIT. COMMIT causes database updates made by the transaction to become visible to other transactions; such updates are said to be *committed*, and are guaranteed never to be canceled. If the transaction executes ROLLBACK instead, all database updates made by the transaction are canceled (*rolled back*). In this latter case, the effect is as if the transaction never ran in the first place.
4. The interleaved execution of a set of concurrent transactions is usually guaranteed to be serializable, in the sense that it produces the same result as executing those same transactions one at a time in some unspecified serial order.

Chapters 15 and 16 contain an extended discussion of all of the foregoing points, and much else besides.

[7] Since a transaction is the execution of some piece of code, a phrase such as "the execution of a transaction" is really a solecism (if it means anything at all, it has to mean the execution of an execution). However, such phraseology is common and useful, and for want of anything better we will use it ourselves in this book.

Copia con fines educativos

3.9 THE SUPPLIERS-AND-PARTS DATABASE

Most of our examples in this book are based on the well-known suppliers-and-parts database. The purpose of this section is to explain that database, in order to serve as a point of reference for later chapters. Fig. 3.8 shows a set of sample data values; subsequent examples will actually assume these specific values, where it makes any difference.[8] Fig. 3.9 shows the database definition, expressed in Tutorial D once again (the Tutorial D keyword VAR means "variable"). Note the primary and foreign key specifications in particular. Note too that (a) several columns have data types of the same name as the column in question; (b) the STATUS column and the two CITY columns are defined in terms of system-defined types—INTEGER (integers) and CHAR (character strings of arbitrary length)—instead of user-defined ones. Note finally that there is an important point that needs to be made regarding the column values as shown in Fig. 3.8, but we are not yet in a position to make it; we will come back to it in Chapter 5, Section 5.3, near the end of the subsection "Possible Representations."

The database is meant to be understood as follows:

- Relvar S represents *suppliers* (more accurately, suppliers *under contract*). Each supplier has a supplier number (S#), unique to that supplier; a supplier name (SNAME), not necessarily unique (though the SNAME values do happen to be unique in Fig. 3.8); a rating or status value (STATUS); and a location (CITY). We assume that each supplier is located in exactly one city.
- Relvar P represents *parts* (more accurately, *kinds* of parts). Each kind of part has a part number (P#), which is unique; a part name (PNAME); a color (COLOR); a

S

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

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

SP

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

Fig. 3.8 The suppliers-and-parts database (sample values)

[8] For ease of reference, Fig. 3.8 is repeated on the inside back cover. For the benefit of readers who might be familiar with the sample data values from earlier editions, we note that part P3 has moved from Rome to Oslo. The same change has also been made in Fig. 4.5 in the next chapter.

Copia con fines educativos

```
TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ... ;
TYPE QTY ... ;

VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR
    PRIMARY KEY { S# } ;

VAR P BASE RELATION
    { P#     P#,
      PNAME  NAME,
      COLOR  COLOR,
      WEIGHT WEIGHT,
      CITY   CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#    S#,
       P#    P#,
       QTY   QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

Fig. 3.9 The suppliers-and-parts database (data definition)

Copia con fines educativos

weight (WEIGHT); and a location where parts of that kind are stored (CITY). We assume where it makes any difference that part weights are given in pounds (but see the discussion of *units of measure* in Chapter 5, Section 5.4). We also assume that each kind of part comes in exactly one color and is stored in a warehouse in exactly one city.

- Relvar SP represents *shipments*. It serves in a sense to link the other two relvars together, logically speaking. For example, the first row in SP as shown in Fig. 3.8 links a specific supplier from relvar S (namely, supplier S1) to a specific part from relvar P (namely, part P1)—in other words, it represents a shipment of parts of kind P1 by the supplier called S1 (and the shipment quantity is 300). Thus, each shipment has a supplier number (S#), a part number (P#), and a quantity (QTY). We assume there is at most one shipment at any given time for a given supplier and a given part; for a given shipment, therefore, the combination of S# value and P# value is unique with respect to the set of shipments currentiy appearing in SP. Note that the database of Fig. 3.8 includes one supplier, supplier S5, with no shipments at all.

We remark that (as already pointed out in Chapter 1, Section 1.3) suppliers and parts can be regarded as entities, and a shipment can be regarded as a relationship between a particular supplier and a particular part. As also pointed out in that chapter, however, a relationship is best regarded as just a special case of an entity. One advantage of relational databases over all other known kinds is precisely that all entities, regardless of whether

they are in fact relationships, are represented in the same uniform way: namely, by means of rows in relations, as the example indicates.

A couple of final remarks:

- First, the suppliers-and-parts database is clearly very simple, much simpler than any real database is likely to be; most real databases will involve many more entities and relationships (and, more important, many more *kinds* of entities and relationships) than this one does. Nevertheless, it is at least adequate to illustrate most of the points that we need to make in the rest of the book, and (as already stated) we will use it as the basis for most—not all—of our examples as we proceed.
- Second, there is nothing wrong with using more descriptive names such as SUPPLIERS, PARTS, and SHIPMENTS in place of the rather terse names S, P, and SP in Figs. 3.8 and 3.9; indeed, descriptive names are generally to be recommended in practice. But in the case of the suppliers-and-parts database specifically, the relvars are referenced so frequently in what follows that very short names seemed desirable. Long names tend to become irksome with much repetition.

3.10 SUMMARY

This brings us to the end of our brief overview of relational technology. Obviously we have barely scratched the surface of what by now has become a very extensive subject, but the whole point of the chapter has been to serve as a gentle introduction to the much more comprehensive discussions to come. Even so, we have managed to cover quite a lot of ground. Here is a summary of the major topics we have discussed.

A **relational database** is a database that is perceived by its users as a collection of **relation variables**—that is, *relvars*—or, more informally, tables. A **relational system** is a system that supports relational databases and operations on such databases, including in particular the operations **restrict**, **project**, and **join**. These operations, and others like them, are collectively known as **the relational algebra**,[9] and they are all set-level. The closure property of relational systems means the output from every operation is the same kind of object as the input (they are all relations), which means we can write **nested relational expressions**. Relvars can be updated by means of the **relational assignment** operation; the familiar update operations **INSERT**, **DELETE**, and **UPDATE** can be regarded as shorthands for certain common relational assignments.

The formal theory underlying relational systems is called **the relational model of data**. The relational model is concerned with logical matters only, not physical matters. It addresses three principal aspects of data: **data structure**, data **integrity**, and data **manipulation**. The *structural* aspect has to do with relations *per se;* the *integrity* aspect has to do with (among other things) **primary and foreign keys**; and the *manipulative* aspect has to do with the operators (restrict, project, join, etc.). ***The Information Principle***—which we

[9] We mentioned this term in the formal definition of the relational model in Section 3.2. However, we will not start using it in earnest until we reach Chapter 6.

Copia con fines educativos

now observe might better be called *The Principle of Uniform Representation*—states that the entire information content of a relational database is represented in one and only one way, as explicit values in column positions in rows in relations. Equivalently: *The only variables allowed in a relational database are, specifically, relvars.*

Every relation has a **heading** and a **body**; the heading is a set of column-name:type-name pairs, the body is a set of rows that conform to the heading. The heading of a given relation can be regarded as a **predicate**, and each row in the body denotes a certain true **proposition**, obtained by substituting certain **arguments** of the appropriate type for the **parameters** of the predicate. Note that these remarks are true of derived relations as well as base ones; they are also true of relvars, *mutatis mutandis*. In other words, *types* are (sets of) things we can talk about, and *relations* are (sets of) things we say about the things we can talk about. Together, types and relations are necessary and sufficient to represent any data we like (at the logical level, that is).

The **optimizer** is the system component that determines how to implement user requests (which are concerned with *what*, not *how*). Since relational systems therefore assume responsibility for navigating around the stored database to locate the desired data, they are sometimes described as **automatic navigation** systems. Optimization and automatic navigation are prerequisites for **physical data independence**.

The catalog is a set of system relvars that contain descriptors for the various items that are of interest to the system (base relvars, views, indexes, users, etc.). Users can interrogate the catalog in exactly the same way they interrogate their own data.

The original (given) relvars in a given database are called **base relvars**, and their values are called **base relations**; a relation that is not a base relation but is obtained from the base relations by means of some relational expression is called a **derived relation** (collectively, base and derived relations are sometimes referred to as **expressible relations**). A view is a relvar whose value at any given time is such a derived relation (loosely, it can be thought of as a **derived relvar**); the value of such a relvar at any given time is whatever results from evaluating the associated **view-defining expression** at that time. Note, therefore, that base relvars have *independent existence*, but views do not—they depend on the applicable base relvars. (Another way of saying the same thing is that base relvars are *autonomous*, but views are not.) Users can operate on views in exactly the same way as they operate on base relvars, at least in theory. The system implements operations on views by replacing references to the name of the view by the view-defining expression, thereby converting the operation into an equivalent operation on the underlying base relvar(s).

A **transaction** is a *logical unit of work*, typically involving several database operations. A transaction begins when **BEGIN TRANSACTION** is executed and terminates when **COMMIT** (normal termination) or **ROLLBACK** (abnormal termination) is executed. Transactions are **atomic**, **durable**, and **isolated** from one another. The interleaved execution of a set of concurrent transactions is usually guaranteed to be **serializable**.

Finally, the base example for most of the book is the **suppliers-and-parts database**. It is worth taking the time to familiarize yourself with that example now, if you have not done so already; that is, you should at least know which relvars have which columns and what the primary and foreign keys are (it is not as important to know exactly what the sample data values are!).

Copia con fines educativos

EXERCISES

3.1 Explain the following in your own words:

automatic navigation	primary key
base relvar	projection
catalog	proposition
closure	relational database
commit	relational DBMS
derived relvar	relational model
foreign key	restriction
join	rollback
optimization	set-level operation
predicate	view

3.2 Sketch the contents of the catalog relvars TABLE and COLUMN for the suppliers-and-parts database.

3.3 As explained in Section 3.6, the catalog is self-describing—that is, it includes entries for the catalog relvars themselves. Extend Fig. 3.6 to include the necessary entries for the TABLE and COLUMN relvars themselves.

3.4 Here is a query on the suppliers-and-parts database. What does it do? What is the predicate for the result?

```
( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY }
```

3.5 Suppose the expression in Exercise 3.4 is used in a view definition:

```
CREATE VIEW V AS
   ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY } ;
```

Now coasider this query:

```
( V WHERE CITY = 'London' ) { S# }
```

What does this query do? What is the predicate for the result? Show what is involved on the part of the DBMS in processing this query.

3.6 What do you understand by the terms *atomicity*, *durability*, *isolation*, and *serializability* as applied to transactions?

3.7 State *The Information Principle*.

3.8 If you are familiar with the hierarchic data model, identify as many differences as you can between it and the relational model as briefly described in this chapter.

REFERENCES AND BIBLIOGRAPHY

3.1 E. F. Codd: "Relational Database: A Practical Foundation For Productivity," *CACM 25*, No. 2 (February 1982). Republished in Robert L. Ashenhurst (ed.), *ACM Turing Award Lectures: The First Twenty Years 1966–1985*. Reading, Mass.: Addison-Wesley (*ACM Press Anthology Series*, 1987).

This is the paper Codd presented on the occasion of his receiving the 1981 ACM Turing Award for his work on the relational model. It discusses the well-known *application backlog* problem. To quote: "The demand for computer applications is growing fast—so fast that information

Copia con fines educativos

systems departments (whose responsibility it is to provide those applications) are lagging further and further behind in their ability to meet that demand." There are two complementary ways of attacking this problem:

1. Provide IT professionals with new tools to increase their productivity.
2. Allow end users to interact directly with the database, thus bypassing the IT professional entirely.

Both approaches are needed, and in this paper Codd gives evidence to suggest that the necessary foundation for both is provided by relational technology.

3.2 C. J. Date: "Why Relational?" in *Relational Database Writings 1985–1989*. Reading, Mass.: Addison-Wesley (1990).

An attempt to provide a succinct yet reasonably comprehensive summary of the major advantages of relational systems. The following observation from the paper is worth repeating here: Among all the numerous advantages of "going relational," there is one in particular that cannot be overemphasized, and that is *the existence of a sound theoretical base*. To quote: "Relational really is different. It is different because it is not *ad hoc*. Older systems, by contrast, were *ad hoc;* they may have provided solutions to certain important problems of their day, but they did not rest on any solid theoretical base. Relational systems, by contrast, do rest on such a base . . . which means [they] are *rock solid* . . . Thanks to this solid foundation, relational systems behave in well-defined ways; and (possibly without realizing the fact) users have a simple model of that behavior in their mind, one that enables them to predict with confidence what the system will do in any given situation. There are (or should be) no surprises. This predictability means that user interfaces are easy to understand, document, teach, learn, use, and remember."

3.3 C. J. Date and Hugh Darwen: *Foundation for Future Database Systems: The Third Manifesto* (2d edition). Reading, Mass.: Addison-Wesley (2000). See also *http://www.thethirdmanifesto.com*, which contains certain formal extracts from the book, an errata list, and much other relevant material. Reference [20.1] is also relevant.

The Third Manifesto is a detailed, formal, and rigorous proposal for the future direction of databases and DBMSs. It can be seen as an abstract blueprint for the design of a DBMS and the language interface to such a DBMS. It is based on the classical core concepts type, value, variable, and operator. For example, we might have a *type* INTEGER; the integer "3" might be a *value* of that type; N might be a *variable* of that type, whose value at any given time is some integer value (i.e., some value of that type); and "+" might be an *operator* that applies to integer values (i.e., to values of that type). *Note:* The emphasis on types in particular is brought out by the book's subtitle: *A Detailed Study of the Impact of Type Theory on the Relational Model of Data, Including a Comprehensive Model of Type Inheritance.* Part of the point here is that type theory and the relational model are more or less independent of each other. To be more specific, the relational model does not prescribe support for any particular types (other than type *boolean*); it merely says that attributes of relations must be of some type, thus implying that *some* (unspecified) types must be supported.

The term *relvar* is taken from this book. In this connection, the book also says this: "The first version of this *Manifesto* drew a distinction between database values and database variables, analogons to the distinction between relation values and relation variables. It also introduced the term *dbvar* as shorthand for *database variable*. While we still believe this distinction to be a valid one, we found it had little direct relevance to other aspects of these proposals. We therefore decided, in the interest of familiarity, to revert to more traditional terminology." This

Cópia com fins educativos

decision subsequently turned out to be a bad one . . . To quote reference [23.4]: "With hindsight, it would have been much better to bite the bullet and adopt the more logically correct terms *database value* and *database variable* (or dbvar), despite their lack of familiarity." In the present book we do stay with the familiar term *database*, but we decided to do so only against our own better judgment (somewhat).

One more point. As the book itself says: "We [confess] that we do feel a little uncomfortable with the idea of calling what is, after all, primarily a technical document a *manifesto*. According to *Chambers Twentieth Century Dictionary*, a manifesto is *a written declaration of the intentions, opinions, or motives* of some person or group (e.g., a political party). By contrast, *The Third Manifesto* is . . . a matter of science and logic, not mere intentions, opinions, or motives." However, *The Third Manifesto* was specifically written to be compared and contrasted with two previous ones, *The Object-Oriented Database System Manifesto* [20.2, 25.1] and *The Third-Generation Database System Manifesto* [26.44], and our title was thus effectively chosen for us.

3.4 C. J. Date: "Great News, The Relational Model Is Very Much Alive!," *http://www.dbdebunk.com* (Augnst 2000).

Ever since it first appeared in 1969, the relational model has been subjected to an extraordinary variety of attacks by a number of different writers. One recent example was entitled, not at all atypically, "Great News, The Relational Model Is Dead!" This article was written as a rebuttal to this position.

3.5 C. J. Date: "There's Only One Relational Model!" *http://www.dbdebunk.com* (February 2001).

Ever since it first appeared in 1969, the relational model has been subjected to an extraordinary variety of misrepresentation and obfuscation by a number of different writers. One recent example was a book chapter titled "Different Relational Models," the first sentence of which read: "There is no such thing as *the* relational model for databases anymore [*sic*] than there is just one geometry." This article was written as a rebuttal to this position.

Copia con fines educativos

Copia con fines educativos