3 An Introduction to Relational Databases

3.1 Introduction

As explained in Chapter 1, the emphasis in this book is very much on the relational approach. In particular, the next part of the book, Part II, covers the theoretical foundations of that approach—namely, the relational model—in depth. The purpose of the present chapter is just to give a preliminary and very informal introduction to the material to be addressed in Part II (and to some extent in subsequent parts also), 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 again more formally, and in much more detail, in those later chapters.

3.2 Relational Systems

We begin by defining a **relational database management system** ("relational system" for short) as a system in which, at a minimum:

1. The data is perceived by the user as tables (and nothing but tables); and
2. The operators at the user's disposal—e.g., for data retrieval—are operators that generate new tables from old, and those operators include at least SELECT (also known as RESTRICT), PROJECT, and JOIN.

This definition, though still very brief, is slightly more specific than the one given in Chapter 1.

A sample 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 meaning of those tables is intended to be self-explanatory). Fig. 3.2 shows some sample SELECT, PROJECT, and JOIN operations against that database. Here are (very loose!) definitions of those operations:

- The **SELECT** operation (also known as **RESTRICT**) extracts specified rows from a table.

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)

- The **PROJECT** operation extracts specified columns from a table.
- The **JOIN** operation joins together two tables on the basis of common values in a common column.

Of the three examples, the only one that seems to need any further explanation is the JOIN example. First of all, observe that the two tables DEPT and EMP do indeed have a common column, namely DEPT#, so they can be joined together on the basis of

SELECT (RESTRICT): Result:

DEPTs where BUDGET > 8M

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

PROJECT: Result:

DEPTs over DEPT#, BUDGET

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 SELECT, PROJECT, and JOIN (examples)

common values in that column. That is, a given row from table DEPT will join to a given row in table EMP—to produce a new, wider row—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) can be joined 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. The set of all possible such joined rows constitutes the overall result. Observe that the common (DEPT#) value appears just once, not twice, in each result row. Observe too 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.

One point that Fig. 3.2 clearly illustrates is that *the result of each of the three operations is another table*. This is the relational property of **closure,** 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—so *the output from one operation can become input to another*. Thus it is possible (for example) to take a projection of a join, or a join of two restrictions, etc., etc. In other words, it is possible to write *nested expressions*—i.e., expressions in which the operands themselves are represented by expressions, instead of just 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).

Note: When we say that the output from each operation is another table, it is very important to understand that we are talking *from a conceptual point of view*. We do not necessarily mean to imply that the system actually has to materialize the result of every individual operation in its entirety. For example, suppose we are trying to compute a restriction of a join. Then, as soon as a given row of the join is constructed, the system can immediately apply the restriction to that row 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.

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 all entire tables, not just single rows, and tables contain *sets* of rows. 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. This **set processing capability** is a major distinguishing characteristic of relational systems (see further discussion in Section 3.6 below). By contrast, the operations in nonrelational systems are typically at the row- or record-at-a-time level.

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

- First, note that the "relational system" definition requires 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 use any or all of the usual storage structures—sequential files, indexing, hashing, pointer chains, compression, etc.—provided only that it can map those structures into 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 encodings, 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 system will be relational, but the internal or physical level will not. 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.*

- Second, relational databases like that of Fig. 3.1 satisfy a very nice property: *The entire information content of the database is represented in one and only one way, namely as explicit data values.* This method of representation (as explicit values in column positions in rows in tables) is the *only* method available in a relational database. In particular, there are no *pointers* connecting one table to another. 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, by contrast, such information is typically represented by some kind of pointer that is explicitly visible to the user.

  Note: When we say there are no pointers in a relational database, we do not mean that there cannot be pointers *at the physical level*—on the contrary, there certainly can be pointers at that level, and indeed there certainly will be. But as already explained, all such physical storage details are concealed from the user in a relational system.

- Finally, note that *all data values are atomic* (or **scalar**). That is, at every row-and-column position in every table there is always exactly one data value, never a group of several values. Thus, for example, in table EMP (considering the DEPT# and EMP# columns only, and for clarity showing them in that left-to-right order), we have

DEPT#	EMP#
D1	E1
D1	E2
..	..

instead of

DEPT#	EMP#
D1	E1,E2
..	..

Column EMP# in the second version of this table is an example of what is usually called a **repeating group**. A repeating group is a column, or combination of columns, that contains several data values in each row (different numbers of values in different rows, in general). *Relational databases do not allow repeating groups;* the second version of the table above would not be permitted in a relational system. (The reason for this apparent limitation is basically *simplicity.* See Chapters 4 and 19 for further discussion.)

We close this section by remarking that the definition given for "relational system" at the beginning of the section is only a *minimal* definition (it is taken from reference [3.1], and is essentially the definition that was current in the early 1980s). There is, of course, far more to a relational system than we can or need to describe in the present section. In particular, please note that the relational model consists of much more than just "tables plus SELECT, PROJECT, and JOIN." See Section 3.4.

3.3 A Note on Terminology

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 anyway? The answer is simple: "Relation" is just a mathematical term for a table (to be precise, a table of a certain specific kind—details to be discussed in Chapter 4). 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; indeed, the term "table" is used much more frequently than the term "relation" in such contexts. But it is worth taking a moment to understand why the latter term was introduced in the first place. Briefly, the explanation is as follows.

- As already indicated, relational systems are based on what is called *the relational model of data.* The relational model, in turn, is an abstract theory of data that is based on certain aspects of mathematics (principally set theory and predicate logic).
- The principles of the relational model were originally laid down in 1969–70 by Dr. E. F. Codd, at that time a researcher in 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 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" (see reference [4.1] in Chapter 4).

- 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 system 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.

- *Example:* Consider the term "record." At different times that single term can mean either a record *occurrence* or a record *type;* a *COBOL-style* record (which allows repeating groups) or a *flat* record (which does not); a *logical* record or a *physical* record; a *stored* record or a *virtual* record; and perhaps other things as well.

The formal relational model therefore does not use the term "record" at all; instead, it uses the term "tuple" (short for "*n*-tuple"), which was given a precise definition by Codd when he first introduced it. We do not give that definition here; for present purposes, it is sufficient to say that the term "tuple" corresponds approximately to the notion of a *flat record instance* (just as the term "relation" corresponds approximately to the notion of a table). When we move on (in Part II) to study the more formal aspects of relational systems, we will make use of the formal terminology, but in this chapter we are not trying to be very formal, and we will mostly stick to terms such as "table," "row," and "column" that are reasonably familiar.

3.4 The Relational Model

So what exactly is the relational model? A good way to characterize it is as follows: The relational model is *a way of looking at data*—that is, it is a prescription for a way of representing data (namely, by means of tables), and a prescription for a way of manipulating such a representation (namely, by means of operators such as JOIN). More precisely, the relational model is concerned with three aspects of data: data **structure,** data **integrity,** and data **manipulation**. The structural and manipulative aspects have already been illustrated; to illustrate the integrity aspect (*very* superficially, please note!), we consider the departments-and-employees database of Fig. 3.1 once again. In all likelihood, that database would be subject to numerous integrity rules; for example, employee salaries might have to be in the range 25K to 95K, department budgets might have to be in the range 1M to 15M, and so on. However, there are certain rules that the database *must* obey if it is to conform to the prescriptions of the relational model. 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.
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).

Columns DEPT# in table DEPT and EMP# in table EMP are the **primary keys** for their respective tables. Column DEPT# in table EMP is a **foreign** key, referencing the primary key of table DEPT. *Note:* The reader might already have noticed that we indicate primary keys by double underlining in our figures (see, e.g., Fig. 3.1). We will follow this convention throughout this book.

A word of warning is appropriate at this point. The relational model is, as already indicated, a *theory*. Note carefully, however, that (as suggested at the end of Section 3.2) it is not necessary for a system to support that theory in its entirety in order to qualify as relational according to the definition. Indeed, so far as this writer is aware, *there is no product on the market today that supports every last detail of the theory.* This is not to say that some parts of the theory are unimportant; on the contrary, *every detail* of the theory is important, and important, moreover, for genuinely practical reasons. Indeed, the point cannot be stressed too strongly that the purpose of the theory is not just "theory for its own sake"; rather, the purpose is to provide a base on which to build systems that are *100 percent practical.* But the sad fact is that the vendors have not yet really stepped up to the challenge of implementing the theory in its entirety. As a consequence, the relational products of today all fail, in one way or another, to deliver on the full promise of relational technology.

Note: When we say that every detail of the theory is important, we do not mean to imply that every portion of the theory is as important as every other. The fact is, some portions are not as widely accepted as others; indeed, there are some, such as the treatment of missing information, that are still subject to a considerable degree of controversy. Details of such matters are beyond the scope of the present chapter; see Parts II and V of this book (especially Chapters 5 and 20) for further discussion.

3.5 Optimization

As explained in Section 3.2, relational operations such as SELECT, PROJECT, and JOIN are all *set-level* operations. As a consequence, relational languages such as SQL are often said to be *nonprocedural,* on the grounds that users specify *what,* not *how*—i.e., they say what they want, without specifying a procedure for getting it. The process of "navigating" around the stored database in order to satisfy the user's request is performed automatically by the system, not manually by the user. For this reason, relational systems are sometimes referred to as **automatic navigation** systems. In nonrelational systems, by contrast, such navigation is generally the responsibility of the user. A striking illustration of the benefits of automatic navigation is shown in Fig. 3.3, 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.

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 such as SQL are at *a higher*


```
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. 3.3 Automatic *vs.* manual navigation

level of abstraction than programming languages such as C and COBOL (or data sublanguages such as are typically found in nonrelational DBMSs, come to that—see Fig. 3.3). 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**. In other words, for each relational 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 request:

```
RESULT := ( EMP WHERE EMP# = 'E4' ) [ SALARY ] ;
```

Explanation: The expression in parentheses ("EMP WHERE ...") requests a *restriction* of the EMP table to just the row where EMP# is E4. The column name in

square brackets ("SALARY") then requests a *projection* of the result of that restriction over the SALARY column. Finally, the *assignment* operation (":=") requests the result of that projection to be assigned to table RESULT. In other words, RESULT is a single-column, single-row table that— after the request has been executed—will contain employee E4's salary. (We are making use here of the syntax for relational operations to be described in detail in Chapter 6. Note too, incidentally, that we are implicitly making use of the relational *closure* property—we have written a nested expression, in which the input to the projection operation is the output from the restriction operation.)

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) table EMP until the required record is found;
2. If there is an index on (the stored version of) the EMP# column of that table—which in practice there probably will be, because it is the primary key, and most systems in fact *require* an index on the primary key—then by using that index and thus going directly to the E4 data.

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

- Which tables are referenced in the request (there may be more than one if, e.g., there are any joins involved)
- How big those tables are
- What indexes exist
- How selective those indexes are
- How the data is physically clustered on the disk
- What relational operations are involved

and so on. To repeat, therefore: User requests specify only what data the user wants, not how to get to that data; the access strategy for getting to the 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 (Section 2.8), every 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) are kept. In other words, the catalog contains

detailed information (sometimes called **descriptors**) regarding the various objects that are of interest to the system itself. Examples of such objects are tables, indexes, users, integrity rules, security rules, and so on. Descriptor information is essential if the system is to be able to do its job properly. For example, the optimizer uses catalog information about indexes (see Chapter 18), as well as much other information, to help it decide how to implement user requests. Likewise, the security subsystem uses catalog information about users and security rules (see Chapter 15) to grant or deny such requests in the first place.

Now, one of the nice features of relational systems is that, in such a system, *the catalog itself consists of tables* (more precisely, *system* tables, so called to distinguish them from ordinary user tables). As a result, users can interrogate the catalog in exactly the same way as they interrogate their own data. For example, the catalog will typically include two system tables called TABLES and COLUMNS, the purpose of which is to describe the tables known to the system and the columns of those tables. (We say "typically" because the catalog is not the same in every system; this is because the catalog for a particular system necessarily contains a good deal of information that is specific to that system.) For the departments-and-employees database, the TABLES and COLUMNS tables might look in outline as shown in Fig. 3.4.

Note: It would be more accurate to say that the TABLES and COLUMNS tables describe the *named* tables known to the system, as opposed to the *un*named tables that result from the evaluation of some relational expression. Note too that the category "named tables" includes the catalog tables themselves—i.e., the catalog is *self-describing*. The entries for the catalog tables themselves are not shown in Fig. 3.4, however.

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

TABLES

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

COLUMNS

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

FIG. 3.4 Catalog for the departments-and-employees database (in outline)

```
( COLUMNS WHERE TABNAME = 'DEPT' ) [ COLNAME ]
```

provides exactly what is required. *Note:* If we had wanted to *keep* the result of this query in some more permanent fashion, we could have assigned the value of the expression to some other table, say RESULT, as in the example in Section 3.5. However, we will omit this final assignment step from most of our examples (both here and in later chapters).

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

```
( COLUMNS WHERE COLNAME = 'EMP#' ) [ TABNAME ]
```

Exercise for the reader: What does the following do?

```
( ( TABLES JOIN COLUMNS )
          WHERE COLCOUNT < 5 ) [ TABNAME, COLNAME ]
```

3.7 Base Tables and Views

We have seen that, starting with a given set of tables such as DEPT and EMP, relational expressions allow us to obtain further tables from that given set—e.g., by joining two of the given tables together. It is time to introduce a little more terminology. The original (given) tables are called **base** tables; a table that is obtained from those base tables by means of some relational expression is called a **derived** table. Thus, base tables have *independent existence,* while derived tables do not—they depend on the base tables. Observe, therefore, that a derived table is, precisely, a table that is defined in terms of other tables—ultimately, in terms of base tables—and a base table is, precisely, a table that is not a derived table.

Now, relational systems obviously have to provide a means for creating the base tables in the first place. In SQL, for example, this function is performed by the CREATE TABLE statement (TABLE here meaning, very specifically, a *base* table). And base tables obviously have to be *named* (indeed, their name is specified in the statement that creates them). Most derived tables, by contrast, are not named. However, relational systems usually support one particular kind of derived table, called a *view,* that does have a name. A **view** is thus a named table that—unlike a base table—does not have an independent existence of its own, but is instead defined in terms of one or more underlying named tables (base tables or other views).

An example is in order. The statement

```
CREATE VIEW TOPEMPS AS
   ( EMP WHERE SALARY > 33K ) [ EMP#, ENAME, SALARY ] ;
```

might be used to define a view called TOPEMPS. When this statement is executed, the expression following the AS—which is in fact the view definition—is not evaluated but is merely "remembered" by the system in some way (actually by saving it in the catalog, under the specified name TOPEMPS). To the user, however, it is now as if there really were a table in the database called TOPEMPS, with rows and columns as shown in the unshaded portions (only) of Fig. 3.5 below. In other words, the name TOPEMPS

TOPEMPS

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

FIG. 3.5 TOPEMPS as a view of base table EMP (unshaded portions)

denotes a *virtual* table, *viz.* the table that would result if the view-defining expression were actually evaluated.

Note carefully, however, that although we say that the name TOPEMPS denotes "the table that would result if the view-defining expression were actually evaluated," we definitely do not mean to suggest that it refers to *a separate copy* of the data—i.e., we do not mean to suggest that the view-defining expression actually *is* evaluated. On the contrary, the view is effectively just a *window* into the underlying table EMP. Furthermore, of course, any changes to that underlying table will be automatically and instantaneously visible through that window (provided, of course, that those changes lie within the unshaded portion of EMP); likewise, changes to TOPEMPS will automatically and instantaneously be applied to the real table EMP, and hence of course be visible through the window.

Here then is an example of a query involving view TOPEMPS:

```
( TOPEMPS WHERE SALARY < 42K ) [ EMP#, SALARY ]
```

The result will look like this:

EMP#	SALARY
E1	40K
E4	35K

Operations against a view like that just shown are effectively handled by replacing *references* to the view by the expression that *defines* the view (i.e., the expression that was saved in the catalog). In the example, therefore, the expression

```
( TOPEMPS WHERE SALARY < 42K ) [ EMP#, SALARY ]
```

is modified by the system to become

```
( ( EMP WHERE SALARY > 33K ) [ EMP#, ENAME, SALARY ] )
                        WHERE SALARY < 42K ) [ EMP#, SALARY ]
```

which, after a certain amount of rearrangement (see Chapter 18), can be simplified to just

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) [ EMP#, SALARY ]
```

And this expression evaluates to the result shown earlier. In other words, the original operation against the view is effectively converted into an equivalent operation against

the underlying base table. That equivalent operation is then executed in the normal way (more accurately, *optimized and* executed in the normal way).

Now, the view TOPEMPS is very simple, consisting as it does just of a row-and-column-subset of a single underlying base table. In principle, however, a view definition—since it is essentially just a named relational expression—can be *of arbitrary complexity*. For example, here is a view whose definition includes a join of two underlying base tables:

```
CREATE VIEW JOINEX1 AS
   ( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) [ EMP#, DEPT# ] ;
```

We will return to the general question of view definition and view processing in Chapter 17.

Incidentally, we can now explain the remark in Chapter 2 (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 ascribed 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 (as explained above) is, specifically, a *named, derived, virtual table*. Thus, the relational analog of an ANSI/SPARC "external view" is (typically) a collection of several tables, each of which is a view in the relational sense. The "external schema" consists of definitions of those views.

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 structure supported at the two levels could be different—for example, the conceptual level could be based on relations, while a given user could have an external view of the database as a hierarchy. 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—a view is still a table, like a base table. 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 table is precisely one of the strengths of relational systems; it is important in just the same way that the fact that a subset is a set is important in mathematics. *Note:* SQL products, and the SQL standard (see Chapter 8) often seem to miss this point, however, inasmuch as they refer repeatedly to "tables and views" (with the implication that a view is not a table). The reader is advised *not* to fall into this common trap of taking "tables" to mean, specifically, *base* tables only.

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

- Base tables "really exist," in the sense that they represent data that is actually 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 arguably useful in an informal sense, does not accurately reflect the true state of affairs. It is true that users can *think* of base tables as if they physically existed; in a way, in fact, the whole point of the relational approach is to allow users to think of base tables as physically existing, while not having to concern themselves with how those tables are physically represented in storage. But—and it is a big but!—this way of thinking should *not* be construed as meaning that a base table is a physically stored table (i.e., a collection of physically adjacent, physically stored records, each one consisting of a direct copy of a row of the base table). As explained in Section 3.2, base tables 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 table and its stored counterpart.

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 tables. But there is absolutely no reason why there should not be just one stored file of *hierarchic* stored records, each one consisting of department number, name, and budget for some given department, followed by employee number, name, and salary for each employee who happens to be in that department.

3.8 The SQL Language

Most current relational products support some dialect of the standard relational language SQL. SQL was originally developed in IBM Research in the early 1970s; it was first implemented on a large scale in the IBM relational prototype System R, and subsequently reimplemented in numerous commercial products, from both IBM and other vendors. Dialects of SQL have since become an American national (ANSI) standard, an international (ISO) standard, a UNIX (X/Open) standard, an IBM standard (it forms the "common database interface" portion of IBM's System Applications Architecture, SAA), and a federal information processing standard (FIPS)—see the References and Bibliography section in Chapter 8. In this section we take a very brief look at the SQL language.

SQL is used to formulate relational operations (i.e., operations that define and manipulate data in relational form). We consider the definitional operations first. Fig. 3.6 shows how the departments-and-employees database of Fig. 3.1 might be defined, using SQL *data definition* operations.

As you can see, the definition includes one CREATE TABLE statement for each of the two tables. The CREATE TABLE statement is, as already indicated, an example of an SQL data definition operation. Each CREATE TABLE statement specifies the name of the (base) table to be created, the names and data types of the columns of that table, and the primary key and any foreign keys in that table (possibly some additional information also, not illustrated in Fig. 3.6). Refer back to Section 3.4 if you need to refresh your memory regarding primary and foreign keys.

```
CREATE TABLE DEPT
     ( DEPT#    CHAR(2),
       DNAME    CHAR(20),
       BUDGET   DECIMAL(7),
     PRIMARY KEY ( DEPT# ) ) ;

CREATE TABLE EMP
     ( EMP#     CHAR(2),
       ENAME    CHAR(20),
       DEPT#    CHAR(2),
       SALARY   DECIMAL(5),
     PRIMARY KEY ( EMP# ),
     FOREIGN KEY ( DEPT# ) REFERENCES DEPT ) ;
```

FIG. 3.6 The departments-and-employees database (SQL data definition)

Having created the tables, we can now start operating on them by means of the SQL *data manipulation* operations SELECT, INSERT, UPDATE, and DELETE. In particular, we can perform relational SELECT, PROJECT, and JOIN operations on the data, in each case by using the SQL data manipulation statement SELECT. Fig. 3.7 shows how the SELECT, PROJECT, and JOIN examples of Fig. 3.2 could be formu-

SELECT (RESTRICT): Result:

```
SELECT DEPT#, DNAME, BUDGET
FROM   DEPT
WHERE  BUDGET > 8M ;
```

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

PROJECT: Result:

```
SELECT DEPT#, BUDGET
FROM   DEPT ;
```

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

JOIN:

```
SELECT DEPT.DEPT#, DNAME, BUDGET, EMP#, ENAME, SALARY
FROM   DEPT, EMP
WHERE  DEPT.DEPT# = EMP.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. 3.7 SELECT, PROJECT, and JOIN examples in SQL

lated using SQL. *Note:* The join example in that figure illustrates the point that **qualified names** (e.g. DEPT.DEPT#, EMP.DEPT#) are sometimes necessary in SQL to "disambiguate" column references. If unqualified names were used—i.e., if the WHERE clause were of the form "WHERE DEPT# = DEPT#"—then the two "DEPT#" references would be ambiguous (it would not be clear in either case whether the reference stood for DEPT.DEPT# or EMP.DEPT#).

The reader will observe that the SELECT statement of SQL and the SELECT operation of the relational model are not the same thing! Indeed, SQL supports all three of the relational operations SELECT, PROJECT, and JOIN (and more besides), all within its own SELECT statement. For this reason among others, RESTRICT is to be preferred over SELECT as the name of the relational operation; referring to the two distinct operations by two distinct names should reduce the chance of confusion between them. (In fact, RESTRICT was the original name for the relational operation; furthermore, it is an intuitively good name, inasmuch as the operation has the effect of—for example—*restricting* the set of departments to just those with a budget in excess of 8M.)

We close this brief discussion of SQL with a few miscellaneous observations:

1. *Update operations:* Examples of the SQL update operations INSERT, UPDATE, and DELETE have already been given in Chapter 1. However, the examples in the body of that chapter happened all to be single-row operations. Like SELECT, however, INSERT, UPDATE, and DELETE are *set-level* operations, in general (and some of the exercises and answers in Chapter 1 did in fact illustrate this point). Here are some set-level update examples for the departments-and-employees database:

```
INSERT
INTO   TEMP ( EMP# )
       SELECT EMP#
       FROM   EMP
       WHERE  DEPT# = 'D1' ;
```

This example assumes that we have previously created another table TEMP with just one column, called EMP#. The INSERT statement inserts into that table employee numbers for all employees in department D1.

```
UPDATE EMP
SET    SALARY = SALARY * 1.1
WHERE  DEPT# = 'D1' ;
```

This UPDATE statement updates the database to reflect the fact that all employees in department D1 have been given a ten percent salary increase.

```
DELETE
FROM   EMP
WHERE  DEPT# = 'D2' ;
```

This DELETE statement deletes all EMP rows for employees in department D2.

2. *Catalog:* The SQL standard does include specifications for a standard catalog

called the *Information Schema* (see Chapter 8). At the time of writing, however, few products if any have actually implemented the standard Information Schema.

3. *Views:* Here are SQL analogs of the CREATE VIEW statement for TOPEMPS and the sample query against that view from Section 3.7:

```
CREATE VIEW TOPEMPS AS
       SELECT EMP#, ENAME, SALARY
       FROM   EMP
       WHERE  SALARY > 33K ;

SELECT EMP#, SALARY
FROM   TOPEMPS
WHERE  SALARY < 42K ;
```

4. *Means of invocation:* Most SQL products allow SQL statements to be executed both (a) "directly," i.e., interactively from an online terminal, and (b) as part of an application program (i.e., the SQL statements can be "embedded," meaning they can be intermixed with the programming language statements of such a program). In case (b), moreover, the program can typically be written in a variety of host languages (C, COBOL, Pascal, PL/I, etc.).
5. *SQL is not perfect:* We include numerous SQL examples in this book because SQL is the standard relational language and because its use and implementation are both very widespread. But it must be emphasized that SQL is very far from being the "perfect" relational language: It suffers from numerous sins of both omission and commission. See Chapter 8 for further discussion.

3.9 The Suppliers-and-Parts Database

Our running example throughout most of this book is the well-known **suppliers-and-parts** database. The purpose of this section is to introduce 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. Fig. 3.9 shows the database definition, expressed in a syntax to be explained in Chapter 4. Note the primary and foreign key specifications in particular.

The intended semantics of the database are as follows.

- Table S represents *suppliers.* Each supplier has a supplier number (S#), unique to that supplier; a supplier name (SNAME), not necessarily unique (though 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.
- Table P represents *parts* (more accurately, kinds of part). Each kind of part has a part number (P#), which is unique; a part name (PNAME); a color (COLOR); a weight (WEIGHT); and a location where parts of that type are stored (CITY). We assume—where it makes any difference—that part weights are given in pounds.

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	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	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)

```
CREATE DOMAIN S#     CHAR(5) ;
CREATE DOMAIN NAME   CHAR(20) ;
CREATE DOMAIN STATUS NUMERIC(5) ;
CREATE DOMAIN CITY   CHAR(15) ;
CREATE DOMAIN P#     CHAR(6) ;
CREATE DOMAIN COLOR  CHAR(6) ;
CREATE DOMAIN WEIGHT NUMERIC(5) ;
CREATE DOMAIN QTY    NUMERIC(9) ;

CREATE BASE RELATION S
     ( S#      DOMAIN ( S# ),
       SNAME   DOMAIN ( NAME ),
       STATUS  DOMAIN ( STATUS ),
       CITY    DOMAIN ( CITY ) )
     PRIMARY KEY ( S# ) ;

CREATE BASE RELATION P
     ( P#      DOMAIN ( P# ),
       PNAME   DOMAIN ( NAME ),
       COLOR   DOMAIN ( COLOR ),
       WEIGHT  DOMAIN ( WEIGHT ),
       CITY    DOMAIN ( CITY ) )
     PRIMARY KEY ( P# ) ;

CREATE BASE RELATION SP
     ( S#      DOMAIN ( S# ),
       P#      DOMAIN ( P# ),
       QTY     DOMAIN ( 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)

We also assume that each kind of part comes in exactly one color and is stored in a warehouse in exactly one city.

- Table SP represents *shipments*. It serves in a sense to connect the other two tables together. For example, the first row of table SP in Fig. 3.8 connects a specific supplier from table S (namely, supplier S1) with a specific part from table 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 that there can be 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 currently appearing in the SP table.

We remark that (as already pointed out in 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 Section 1.3, however, relationships are best regarded as just a special case of entities. One advantage of relational databases is precisely that all entities, regardless of whether they are in fact relationships, are represented in the same uniform way—namely, by means of tables, as the example shows.

One final remark: The suppliers-and-parts database is of course extremely simple, much simpler than any real database is likely to be in practice; most real databases will involve many more 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 next few parts of the book, and (as already stated) we will use it as the basis for most—not all—of our examples in the next few chapters. And another editorial comment: There is of course nothing wrong with using more descriptive names such as SUPPLIERS, PARTS, and SHIPMENTS in place of the rather terse names S, P, and SP used above; indeed, descriptive names are generally to be recommended in practice. But in the case of suppliers-and-parts specifically, the three tables are referenced so frequently in the chapters that follow 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 short 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 that follow in the remainder of the book. 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 **relations** or tables. All values in a relation are atomic or **scalar** (there are no repeating groups). A **relational system** is a system that supports relational databases and operations on such databases, including in particular the operations **RESTRICT** (often called SELECT), **PROJECT,** and **JOIN**. These operations, and others like them, are all **set-level**. The **closure** property of relational systems means that the output from every operation is the same kind of object as the input (they are all relations), which implies that we can write **nested relational expressions**.

The formal theory underlying relational systems is called **the relational model**. The relational model is concerned with logical matters only, not physical matters. It addresses three aspects of data—data **structure** (or **objects**), data **integrity,** and data **manipulation** (or **operators**). The *objects* are basically the tables; the *integrity* portion has to do with **primary and foreign keys;** and the *operators* are RESTRICT, PROJECT, JOIN, etc.

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, such systems are sometimes described as **automatic navigation** systems. Optimization and automatic navigation are prerequisites for **data independence** in a relational system.

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

A **derived** table is a table that is derived from other tables by means of some relational expression. A **base** table is a table that is not a derived table. A **view** is a **named** derived table, whose definition in terms of other tables is kept in the catalog. Users can operate on views in much the same way as they operate on base tables. The system implements operations on views by replacing references to the name of the view by the expression that defines the view, thereby converting the operation into an equivalent operation on the underlying base tables. We will refer to this method of implementation as the **substitution** method.

The standard language for interacting with relational databases is **SQL**. The SQL operation for creating a new base table is **CREATE TABLE**. The SQL retrieval operation is **SELECT** (often referred to as SELECT - FROM - WHERE); this operation provides the functionality of the relational RESTRICT, PROJECT, and JOIN operations, and more besides. The SQL update operations are **INSERT, UPDATE,** and **DELETE**. SQL is extremely important from a commercial point of view but is very far from being the "perfect" relational language.

Finally, the base example for much of the remainder of this book is **the suppliers-and-parts database**. It is worth taking the time to familiarize yourself with this example now, if you have not already done so. That is, you should at least know which columns exist in which tables and what the primary and foreign keys are (it is not so important to know exactly which scalar values occur where!).

By way of conclusion, let us try to relate the material discussed in this chapter to the components of the ANSI/SPARC architecture discussed in Chapter 2. The correspondence is not entirely clearcut, as will be seen, but it can nevertheless be useful as an aid to understanding.

1. Base tables correspond to the ANSI/SPARC conceptual level.
2. Views correspond to the ANSI/SPARC external level, as already explained in Section 3.7. *Note:* Actually, most relational products on the market today muddy the external/conceptual distinction somewhat, because they allow users to operate directly on base tables as well as on views.
3. The relational model has nothing to say regarding the ANSI/SPARC internal level. In principle—as explained in Section 3.2—the system is free to employ any storage structures it likes at the internal level, provided only that it can abstract from those storage structures and present the data at the conceptual level in pure tabular form. Unfortunately, this is another area where today's products have muddied the waters somewhat: Most of those products tend to map *one* base table to *one* stored file, and are far too inflexible with respect to the degree of difference they can tolerate between the two. In other words, those products do not provide as much data independence as we would really like, or as relational systems are theoretically capable of providing.

   Note: It is at least true, however, that user requests—i.e., SQL statements—in those products make no direct reference to access structures such as indexes. As a result, the DBA or DBMS can create and destroy such structures freely, for performance and tuning reasons, without invalidating existing applications. (At least, this is true in the SQL standard, though here again some products unfortunately violate the principle and do not conform to the standard in this regard.)
4. SQL is a typical (in fact, the standard) data sublanguage. As such, it includes both a **data definition** language (DDL) component and a **data manipulation** language (DML) component. As already indicated, the SQL DML can operate at both the external and the conceptual level. The SQL DDL, similarly, can be used to define objects at the external level (views), the conceptual level (base tables), and even—in most systems, though not in the standard—the internal level (e.g., indexes). Moreover, SQL also provides certain "data control" facilities—that is, facilities that cannot really be classified as belonging to either the DDL or the DML. An example of such a facility is the GRANT statement, which allows one user to grant certain *access privileges* to another (see Chapter 15).
5. Application programs in an SQL system can access the database from a host language such as COBOL by means of *embedded SQL statements* (see Chapter 8). Embedded SQL represents a "loose coupling" between SQL and the host language. Basically, any statement that can be used in interactive SQL can be used in embedded SQL also. In addition, certain special statements, also discussed in Chapter 8, are provided for use in the embedded environment only.

Exercises

3.1 Define the following terms:

automatic navigation	primary key
base table	projection
catalog	relational database
closure	relational DBMS
derived table	relational model
foreign key	restriction
join	set-level operation
optimization	view

3.2 Sketch the contents of the catalog tables TABLES and COLUMNS for the suppliers-and-parts database.

3.3 As explained in Section 3.6, the catalog is self-describing—i.e., it includes entries for the catalog tables themselves. Extend Fig. 3.4 to include the necessary entries for the TABLES and COLUMNS tables themselves.

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

```
RESULT  :=  ( ( S JOIN SP ) WHERE P# = 'P2' ) [ S#, CITY ] ;
```

3.5 Suppose the expression on the right-hand side of the assignment in Exercise 3.4 is used in a view definition:

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

Now consider the query

```
ANSWER  :=  ( V WHERE CITY = 'London' ) [ S# ] ;
```

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

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 that Codd presented on the occasion of his receiving the 1981 ACM Turing Award. It discusses the well-known *application backlog* problem. To paraphrase: "The demand for computer applications is growing fast—so fast that information 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 C. J. Date, *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 the relational approach. 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 that [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. "Relational Technology: A Brief Introduction." In C. J. Date and Hugh Darwen, *Relational Database Writings 1989–1991*. Reading, Mass.: Addison-Wesley (1992).

Portions of the present chapter were originally published in somewhat different form in this paper.

Answers to Selected Exercises

3.3 Fig. 3.10 shows the entries for the TABLES and COLUMNS tables (only; i.e., the entries for the user's own tables are omitted). It is obviously not possible to give precise COLCOUNT and ROWCOUNT values.

TABLES

TABNAME	COLCOUNT	ROWCOUNT	.....
TABLES	(>3)	(>2)	.....
COLUMNS	(>2)	(>5)	.....
.....	.....	.....	.....

COLUMNS

TABNAME	COLNAME	.....
TABLES	TABNAME	.....
TABLES	COLCOUNT	.....
TABLES	ROWCOUNT	.....
COLUMNS	TABNAME	.....
COLUMNS	COLNAME	.....
.....	.....	.....

FIG. 3.10 Catalog entries for TABLES and COLUMNS themselves (in outline)

3.4 The query retrieves supplier number and city for suppliers who supply part P2.

3.5 The meaning of the query is: "Retrieve supplier number for London suppliers who supply part P2." The first step in processing the query is to replace the name V by the expression that defines V, giving:

```
( ( ( ( S JOIN SP ) WHERE P# = 'P2' ) [ S#, CITY ] )
                                WHERE  CITY = 'London' ) [ S# ]
```

This simplifies to:

```
( ( S WHERE CITY = 'London' ) JOIN ( SP WHERE P# = 'P2' ) ) [ S# ]
```

For further discussion and explanation, see Chapters 17 and 18.