```
7.11.36 SX WHERE EXISTS STATUSX EXISTS STATUSY
              ( S ( S#:SX, STATUS:STATUSX ) AND
                S ( S#:'S1', STATUS:STATUSY ) AND
                STATUSX < STATUSY )

7.11.37 JX WHERE EXISTS CITYX
              ( J ( J#:JX, CITY:CITYX ) AND
                FORALL CITYY ( IF J ( CITY:CITYY )
                               THEN CITYY ≥ CITYX ) )
```

7.11.38–39 Solutions omitted.

```
7.11.40 JX WHERE J ( J#:JX ) AND
                NOT EXISTS SX EXISTS PX
                  ( SPJ ( S#:SX, P#:PX, J#:JX ) AND
                    S ( S#:SX, CITY:'London' ) AND
                    P ( P#:PX, COLOR:'Red' ) )

7.11.41 JX WHERE J ( J#:JX )
           AND   FORALL SX ( IF SPJ ( S#:SX, J#:JX )
                             THEN SX = 'S1' )

7.11.42 PX WHERE P ( P#:PX )
           AND   FORALL JX ( IF J ( J#:JX, CITY:'London' )
                             THEN SPJ ( P#:PX, J#:JX ) )

7.11.43 SX WHERE S ( S#:SX )
           AND   EXISTS PX FORALL JX
              ( SPJ ( S#:SX, P#:PX, J#:JX ) )

7.11.44 JX WHERE J ( J#:JX )
           AND   FORALL PX ( IF SPJ ( S#:'S1', P#:PX )
                             THEN SPJ ( P#:PX, J#:JX ) )

7.11.45 CITYX WHERE S ( CITY:CITYX )
              OR    P ( CITY:CITYX )
              OR    J ( CITY:CITYX )

7.11.46 PX WHERE EXISTS SX ( SPJ ( S#:SX, P#:PX ) AND
                             S ( S#:SX, CITY:'London' ) )
           OR    EXISTS JX ( SPJ ( J#:JX, P#:PX ) AND
                             J ( J#:JX, CITY:'London' ) )

7.11.47 ( SX, PX ) WHERE S ( S#:SX ) AND P ( P#:PX )
                   AND   NOT SPJ ( S#:SX, P#:PX )

7.11.48 ( SX AS XS#, SY AS YS# ) WHERE S ( S#:SX ) AND S ( S#:SY )
          AND FORALL PZ
          ( ( IF SPJ ( S#:SX, P#:PZ ) THEN SPJ ( S#:SY, P#:PZ ) )
            AND
            ( IF SPJ ( S#:SY, P#:PZ ) THEN SPJ ( S#:SX, P#:PZ ) ) )
```

8 The SQL Language

8.1 Introduction

As mentioned in the introduction to this part of the book, SQL is very far from being a faithful implementation of the relational model. Nevertheless, it *is* the standard relational language, it is supported by just about every product on the market today, and so every database professional needs to know something about it. Hence this chapter.

The first thing that must be said is that SQL is an enormous language. The standard document itself [8.1] is well over 600 pages long. As a consequence, it is not possible in a book of this nature to treat the subject exhaustively; all we can hope to do is describe major aspects in a reasonably comprehensive manner, but the reader is warned that our discussions are necessarily sketchy and superficial in many places. In particular, we have not hesitated to omit material that is irrelevant to the purpose at hand, nor to make significant simplifications in the interests of brevity. More complete (but still tutorial) descriptions can be found in references [8.5–8.7].

The plan of the chapter is as follows. Following this introductory section, Section 8.2 treats SQL's data definition facilities and Sections 8.3–8.4 treat SQL's data manipulation facilities (Section 8.3 covers retrieval operations and Section 8.4 update operations). Sections 8.5–8.7 then focus on three key SQL constructs, namely table expressions (including in particular *select* expressions), conditional expressions, and scalar expressions, respectively. Next, Section 8.8 describes the special considerations that apply to "embedded" SQL (i.e., the facilities for embedding SQL statements in a host language program). Section 8.9 presents a brief summary. *Note:* Additional aspects of SQL, having to do with matters such as recovery, concurrency, etc., will be described briefly in subsequent chapters devoted to those topics.

A few further preliminary remarks are in order. First, our discussions are all at the level of the current standard [8.1] known informally as "SQL/92," also as "SQL-92" or just "SQL2"; the official name is **International Standard Database Language SQL (1992)**. The presentation is loosely based on material from reference [8.5], though it has been considerably revised to suit the needs of the present book. Please note also that:

- The "#" character, much used in our example domain and column names, is in fact not legal in SQL/92.

- We use the semicolon ";" as a statement terminator, but SQL/92 actually prescribes such a terminator only in the case of embedded SQL, and then only for certain host languages.
- We often use names for syntactic categories that are different from those of the standard, because the standard terminology is often not very apt.

Although our discussions are all at the SQL/92 level, we should make it clear that no product actually supports the whole of SQL/92 at the time of writing. Instead, products typically support what might be called "a superset of a subset" of SQL/92. In other words, any given product, while it fails to support certain aspects of the standard, will at the same time probably go beyond the standard in certain other respects. IBM's DB2 product, for example, certainly does not support all of the SQL/92 integrity features, but it does go beyond the standard in its rules regarding view updatability.

One final introductory remark: SQL uses the terms *table, row,* and *column* in place of the relational terms *relation, tuple,* and *attribute*. For consistency with the SQL standard and SQL products, therefore, we will do likewise in this chapter (and elsewhere in this book whenever we are concerned with SQL specifically).

8.2 Data Definition

In this section we examine the **basic data objects** and corresponding **data definition language (DDL) statements** of SQL. The principal DDL statements are as follows:

```
CREATE DOMAIN   CREATE TABLE
ALTER DOMAIN    ALTER TABLE
DROP DOMAIN     DROP TABLE
```

There are also statements for creating and destroying ("dropping") views, but we defer discussion of views to Chapter 17.

Domains

SQL's "domains" are unfortunately a long way from being true relational domains as described in Chapters 4 and 19; in fact, the two concepts are so far apart that it would have been preferable to use some other name for the SQL construct. Almost the sole purpose of domains in SQL is to allow a simple data type specification (such as "S# CHAR(5)") to be defined once and then shared by several columns in several base tables. For purposes of reference, we list below some of the principal differences between true domains and the SQL construct (many of these points will not make much sense until the reader has studied Chapter 19):

- As already suggested, SQL domains are really just a syntactic shorthand. They are certainly not true user-defined data types.
- There is no requirement that SQL domains even be used—columns in base tables can be defined directly in terms of the builtin, system-defined data types such as FLOAT or INTEGER.
- There is no SQL support for "domains on domains": An SQL domain must be defined in terms of one of the builtin, system-defined data types, not another user-defined domain.
- SQL does not provide anything like *strong typing*. There is no true type checking. In particular, domains do not "constrain comparisons"—the only requirement on comparisons is that the comparands must be of the same *basic* type, i.e., both numeric or both character strings or (etc.). An analogous remark applies to numeric expressions, character string expressions, bit string expressions, . . . (etc., etc.): In all cases, domains as such are essentially irrelevant.
- SQL does not support the ability for users to define the operations that apply to a given domain.
- SQL does not make a clear distinction between a domain as such (i.e., a user-defined data type) and the *representation* of that domain in terms of one of the system-defined data types.
- SQL does not have any concept of subtypes and supertypes, nor of inheritance.
- Finally, SQL does not even support what is arguably the most fundamental domain of all, *viz.* the domain of truth values!

Here is the syntax for creating an SQL domain. *Note:* As in Chapter 7 we use square brackets "[" and "]" to indicate optional material, and we will continue to do so throughout this chapter.

```
CREATE DOMAIN domain data-type
            [ default-definition ]
            [ domain-constraint-definition-list ] ;
```

Explanation:

1. SQL supports the following **scalar data types,** most of them self-explanatory. (A number of defaults, abbreviations, and alternative spellings—e.g., CHAR for CHARACTER—are also supported. We omit the details here.)

```
CHARACTER [ VARYING ] (n)    INTEGER      DATE
BIT [ VARYING ] (n)          SMALLINT     TIME
NUMERIC (p,q)                FLOAT (p)    TIMESTAMP
DECIMAL (p,q)                             INTERVAL
```

2. The optional *default-definition* specifies a default value that applies to every column that is defined on the domain and does not have an explicit default value of its own. It takes the form "DEFAULT *default*"—where *default* in turn is a literal, a reference to a niladic builtin function (e.g., CURRENT_DATE), or NULL.* *Note:* A niladic function is a function that takes no arguments.

* We defer detailed discussion of SQL's support for nulls to Chapter 20. Passing references to nulls in the present chapter are unavoidable, however.

3. The optional list of *domain constraint definitions* specifies a set of integrity constraints that apply to every column defined on the domain. Now, we will explain in Chapter 16 that a domain integrity constraint is—or, rather, should be—conceptually nothing more than an enumeration of the values that go to make up that domain. SQL, however, allows a domain constraint to involve a truth-valued expression *of arbitrary complexity*. We leave it as an exercise for the reader to meditate on some of the unfortunate implications of this unwarranted permissiveness.

Here is an example:

```
CREATE DOMAIN COLOR CHAR(6) DEFAULT '???'
       CONSTRAINT VALID_COLORS
       CHECK ( VALUE IN
             ( 'Red', 'Yellow', 'Blue', 'Green', '???' ) ) ;
```

Now the CREATE TABLE for base table P, the parts table, might look like this:

```
CREATE TABLE P ( ... , COLOR COLOR, ... ) ;
```

If the user inserts a row into table P and does not provide a value for the COLOR column within that row, then the value "???" will be placed in that position by default. Alternatively, if the user *does* provide a COLOR value but it is not one of the legal set, the operation will fail, of course, and the system will produce a diagnostic that mentions the VALID_COLORS constraint.

Next, an existing domain can be **altered** at any time in a variety of ways by means of the statement ALTER DOMAIN. Specifically, ALTER DOMAIN allows a new default definition to be specified for an existing domain (replacing the previous one, if any) or an existing one to be deleted. It also allows a new integrity constraint to be specified for an existing domain or an existing one to be deleted. The details of these various options are surprisingly complex, however, and beyond the scope of this book; the interested reader is referred to reference [8.1] or reference [8.5] for further discussion.

Finally, an existing domain can be **destroyed** by means of the statement DROP DOMAIN—syntax:

```
DROP DOMAIN domain option ;
```

where *option* is either RESTRICT or CASCADE. The general idea here is as follows: (a) If RESTRICT is specified, the DROP will fail if the domain is referenced anywhere; (b) if CASCADE is specified, the DROP will succeed and will "cascade" in various ways (for example, columns that were previously defined on the domain will now be considered to be directly defined on the domain's underlying data type instead). Once again the details are quite complex, and we omit them here. See reference [8.1] or reference [8.5] for further information.

Base Tables

Before we get into the details of base tables specifically, there are a couple of points to be made on the topic of SQL tables in general. First, SQL tables—unlike true relations—are allowed to include **duplicate rows;** they therefore do not necessarily have any candidate keys. Second, SQL tables—unlike true relations—are considered to have a **left-to-right column ordering;** in the suppliers table S, for example, column S# might be the first column, column SNAME might be the second column, and so on.

Turning to base tables specifically: Base tables are defined by means of the CREATE TABLE statement (note, therefore, that the keyword TABLE here refers to a base table specifically; the same is true of ALTER TABLE and DROP TABLE, *q.v.*). The syntax is as follows:

```
CREATE TABLE base-table ( base-table-element-commalist ) ;
```

where each *base-table-element* is either a *column-definition* or a *base-table-constraint-definition*. Each *column-definition* in turn (there must be at least one such) looks like this:

```
column representation [ default-definition ]
```

Here *representation* specifies the relevant data type or domain, and the optional *default-definition* specifies a default for the column, overriding any default specified at the domain level, if applicable. If a given column does not have an explicit default of its own and does not inherit one from an underlying domain, it is implicitly assumed to have a default of NULL—i.e., NULL is the "default default."

Each *base-table-constraint-definition* is one of the following:

- a candidate key definition
- a foreign key definition
- a "check constraint" definition

We proceed to discuss each of these in more detail. *Note:* Each can optionally be preceded by the phrase "CONSTRAINT *constraint*," thereby providing a name for the new constraint (the same is true for domain constraints, as we saw in the VALID_COLORS example earlier). For brevity, we ignore this option in our further discussions below.

Candidate keys: A candidate key definition takes the form

```
UNIQUE ( column-commalist )
```

or the form

```
PRIMARY KEY ( column-commalist )
```

The *column-commalist* must not be empty in either case. A given base table can have at most one PRIMARY KEY specification but any number of UNIQUE specifications. In the case of PRIMARY KEY, each specified column is additionally assumed to be NOT NULL, even if NOT NULL is not specified explicitly (see "Check constraints" below).

Foreign keys: A foreign key definition takes the form

```
FOREIGN KEY ( column-commalist )
        REFERENCES base-table [ ( column-commalist ) ]
      [ ON DELETE option ]
      [ ON UPDATE option ]
```

where *option* is NO ACTION or CASCADE or SET DEFAULT or SET NULL. CASCADE and SET NULL correspond directly to our CASCADES and NULLIFIES (refer back to Chapter 5 if you need to refresh your memory); NO ACTION, which is the default, is similar but not identical to our RESTRICTED (see reference [8.5] for an explanation of the differences), and SET DEFAULT is self-explanatory. *Note:* The second *column-commalist* is required if the foreign key references a candidate key that is not a primary key.

Check constraints: A "check constraint definition" takes the form

```
CHECK ( conditional-expression )
```

An attempt to create a row within base table *B* is considered to violate a check constraint for *B* if it causes the conditional expression specified within that constraint to evaluate to *false*. Note that the conditional expression can be arbitrarily complex; it is specifically *not* limited to a restriction condition referring just to table *B*, but can instead refer to anything in the database. Reference [8.19] offers some critical comments regarding this unnecessary generality.

Here is an example of CREATE TABLE:

```
CREATE TABLE SP
     ( S# S# NOT NULL, P# P# NOT NULL, QTY QTY NOT NULL,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       FOREIGN KEY ( P# ) REFERENCES P
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       CHECK ( QTY > 0 AND QTY < 5001 ) ) ;
```

We are assuming here that (a) domains S#, P#, and QTY have already been defined, and (b) S# and P# have been explicitly defined to be the primary keys for tables S and P, respectively. Also, we have deliberately made use of the convenient shorthand by which a check constraint of the form

```
CHECK ( column IS NOT NULL )
```

can be replaced by a simple NOT NULL specification in the definition of the column in question. In the example, we have thus replaced three slightly cumbersome check constraints by three simple NOT NULLs.

Next, an existing base table can be **altered** at any time by means of the ALTER TABLE statement. The following alterations are supported:

- A new column can be added
- A new default can be defined for an existing column (replacing the previous one, if any)
- An existing column default can be deleted
- An existing column can be deleted

- A new base table integrity constraint can be specified
- An existing base table integrity constraint can be deleted

We give an example of the first case only:

```
ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1 ;
```

This statement adds a DISCOUNT column (of type INTEGER) to the suppliers base table. All existing rows in that table are extended from four columns to five; the value of the new fifth column is minus 1 in every case.

Finally, an existing base table can be **destroyed** by means of DROP TABLE—syntax:

```
DROP TABLE base-table option ;
```

where (as with DROP DOMAIN) *option* is either RESTRICT or CASCADE. If RESTRICT is specified and the base table is referenced in any view definition or integrity constraint, the DROP will fail; if CASCADE is specified, the DROP will succeed (destroying the table along with all of its rows), and any referencing view definitions and integrity constraints will be dropped also.

The Information Schema

The SQL analog of what is more conventionally known as the *catalog* is called the **Information Schema**. In fact, the familiar terms "catalog" and "schema" are both used in SQL, but with highly SQL-specific meanings. Loosely speaking, a **catalog** in SQL consists of the descriptors for an individual database,* and a **schema** consists of the descriptors for that portion of that database that belongs to some individual user. In other words, there can be any number of catalogs, each divided up into any number of schemas. However, each catalog is required to include exactly one schema called INFORMATION_SCHEMA, and from the user's perspective it is that schema (as already indicated) that performs the normal "catalog" function.

The Information Schema thus consists of a set of SQL tables whose contents effectively echo, in a precisely defined way, all of the definitions from all of the other schemas in the catalog in question. More precisely, the Information Schema is defined to contain a set of **views** of a hypothetical "Definition Schema." The implementation is not required to support the Definition Schema as such, but it is required (a) to support *some* kind of "Definition Schema," and (b) to support views of that "Definition Schema" that do look like those of the Information Schema. Points arising:

1. The rationale for stating the requirement in terms of two separate pieces (a) and (b) as just described is as follows. First, existing products certainly do support something akin to the "Definition Schema." However, those "Definition Schemas" vary widely from one product to another (even when the products in question come

* In the interests of accuracy, we should point out that there is actually no such thing as a "database" in the SQL standard! Exactly what the collection of data is that is described by a given catalog is implementation-defined. However, it is not unreasonable to think of it as a database.

from the same vendor). Hence the idea of requiring only that the implementation support certain predefined views of its "Definition Schema" makes sense.

2. We should really say "an" (not "the") Information Schema, since as we have seen there is one such schema in every catalog. In general, therefore, the totality of data available to a given user will *not* be described by a single Information Schema. To simplify our discussion, however, we will continue to talk as if there really were just one such schema.

It is not worth going into great detail on the content of the Information Schema here. Instead, we simply list some of the more important Information Schema views, in the hope that their names alone will be sufficient to give some idea of what that schema covers (we remark, however, that the TABLES view includes information regarding *all* named tables, views as well as base tables; the VIEWS view contains information for views only, of course). We deliberately ignore some of the more esoteric features.

```
SCHEMATA                     REFERENTIAL_CONSTRAINTS
DOMAINS                      CHECK_CONSTRAINTS
TABLES                       KEY_COLUMN_USAGE
VIEWS                        ASSERTIONS
COLUMNS                      VIEW_TABLE_USAGE
TABLE_PRIVILEGES             VIEW_COLUMN_USAGE
COLUMN_PRIVILEGES            CONSTRAINT_TABLE_USAGE
USAGE_PRIVILEGES             CONSTRAINT_COLUMN_USAGE
DOMAIN_CONSTRAINTS           CONSTRAINT_DOMAIN_USAGE
TABLE_CONSTRAINTS
```

8.3 Data Manipulation: Retrieval Operations

Now we turn to the **data manipulation language (DML)** statements of SQL. The principal DML statements are SELECT, INSERT, UPDATE, and DELETE. The present section considers retrieval operations (SELECT) and the next section considers update operations (INSERT, UPDATE, DELETE). For simplicity, we assume throughout that all statements are entered interactively; the special considerations that apply to SQL statements embedded in application programs are discussed in Section 8.8.

A retrieval operation in SQL is essentially just a **table expression,** of potentially arbitrary complexity. We do not get into all of that complexity here; rather, we simply present a set of examples, in the hope that those examples will serve to highlight some of the most important points. (To facilitate comparisons with the relational algebra and relational calculus, we also give references to the corresponding examples in Chapters 6 and 7, where applicable.) A more complete, and more formal, treatment of table expressions in general is given in Section 8.5.

8.3.1 Get color and city for "nonParis" parts with weight greater than ten.

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10 ;
```

First of all, note the use of the symbol <> (not equals) in this example. The usual scalar comparison operators are written as follows in SQL: =, <>, <, >, <=, and >=.

Next (and much more important), note that—given our usual sample data—this query will return *four* rows, not two, even though three of those four rows are identical, all being of the form (Red,London). SQL does not eliminate redundant duplicate rows from the result of a SELECT unless the user explicitly requests it to do so via the keyword **DISTINCT**, as in:

```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10 ;
```

This query will return two rows only.

Incidentally, we could perfectly well have omitted the "P." qualifiers throughout this example. The general rule regarding **name qualification** in SQL is that unqualified names are acceptable if they cause no ambiguity. In our examples, however, we will generally include all qualifiers, even when they are technically redundant. (Unfortunately, however, there are certain contexts in which column names are explicitly required to be *un*qualified! An example is the ORDER BY clause—see below.)

Finally, note that the sequence of rows in a given result table is unpredictable, in general, *unless* the user explicitly requests some particular sequence, as here:

```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10
ORDER  BY CITY DESC ;
```

In general, the **ORDER BY** clause takes the form

```
ORDER BY order-item-commalist
```

where (a) the commalist must not be empty, and (b) each *order-item* consists of an *un*qualified column name, optionally followed by ASC or DESC (where ASC and DESC mean ascending and descending, respectively, and ASC is the default).

8.3.2 For all parts, get the part number and the weight of that part in grams.

```
SELECT P.P#, P.WEIGHT * 454 AS GMWT
FROM   P ;
```

The specification AS GMWT introduces an appropriate result column name for the "computed column." The two columns of the result table are thus called P# and GMWT, respectively. If the AS GMWT specification had been omitted, the corresponding result column would effectively have been unnamed. Observe, therefore, that SQL does not actually require the user to provide a result column name in such circumstances, but we will always do so in our examples.

8.3.3 Get full details of all suppliers.

```
SELECT *     -- or "SELECT S.*" (i.e., the "*" can be qualified)
FROM   S ;
```

The result is a copy of the entire S table; the star or asterisk is shorthand for a list of all column names in the table(s) referenced in the FROM clause, in the left-to-right order in which those column(s) are defined within those table(s). Notice the **comment** in this example, incidentally (SQL comments are introduced with a double hyphen and terminate with a newline character).

We remark that the star notation is convenient for interactive queries, since it saves keystrokes. However, it is potentially dangerous in embedded SQL—i.e., SQL within an application program—because the meaning of the "*" might change (e.g., if a column is added to or dropped from some table, via ALTER TABLE).

Note: In SQL/92 the expression SELECT * FROM *T* (where *T* is a table name) can be further abbreviated to just TABLE *T*.

8.3.4 Get all combinations of supplier and part information such that the supplier and part in question are colocated. SQL provides several different ways of formulating this query. We give three of the simplest here.

1.
```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY,
       P.P#, P.PNAME, P.COLOR, P.WEIGHT
FROM   S, P
WHERE  S.CITY = P.CITY ;
```
2.
```
S JOIN P USING CITY ;
```
3.
```
S NATURAL JOIN P ;
```

The result in each case is the **natural join** of tables S and P (on cities).

The first of the foregoing formulations—which is the only one of the three that would have been valid in SQL as originally defined (the explicit JOIN support was added in SQL/92)—merits further discussion. Conceptually, we can think of that version of the query as being implemented as follows:

- First, the FROM clause is executed, to yield the **Cartesian product** S TIMES SP.
- Next, the WHERE clause is executed, to yield a **restriction** of that product in which the two CITY values in each row are equal (in other words, we have now constructed the *equijoin* of suppliers and parts over cities).
- Finally, the SELECT clause is executed, to yield a **projection** of that restriction over the columns mentioned in the SELECT clause. The final result is the natural join.

Loosely speaking, therefore, FROM in SQL corresponds to Cartesian product, WHERE to restrict, and SELECT to project, and the SQL SELECT–FROM–WHERE represents a projection of a restriction of a product.

8.3.5 Get all pairs of city names such that a supplier located in the first city supplies a part stored in the second city.

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
FROM   S JOIN SP USING S# JOIN P USING P# ;
```

Notice that the following is *not* correct, because it includes CITY as a joining column in the second join:

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
FROM   S NATURAL JOIN SP NATURAL JOIN P ;
```

8.3.6 Get all pairs of supplier numbers such that the two suppliers concerned are colocated. (Examples 6.6.5, 7.3.2)

```
SELECT FIRST.S# AS SA, SECOND.S# AS SB
FROM   S AS FIRST, S AS SECOND
WHERE  FIRST.CITY = SECOND.CITY
AND    FIRST.S# < SECOND.S# ;
```

Note the explicit **range variables** FIRST and SECOND in this example. The range variables have all been implicit in our previous examples (see the annotation to reference [7.3] if you need to refresh your memory regarding implicit range variables). Note too that the introduced column names SA and SB refer to columns of the *result table*, and so cannot be used in the WHERE clause.

8.3.7 Get the total number of suppliers.

```
SELECT COUNT(*) AS N
FROM   S ;
```

The result here is a table with one column, called N, and one row, containing the value 5. SQL supports the usual set of **aggregate functions** (COUNT, SUM, AVG, MAX, and MIN), but there are a few SQL-specific points the user needs to be aware of, *viz.*:

- In general, the argument of the function can optionally be preceded by the keyword DISTINCT, to indicate that duplicates are to be eliminated before the function is applied. For MAX and MIN, however, DISTINCT is irrelevant and has no effect.
- The special function COUNT(*)—DISTINCT not allowed— is provided to count all rows in a table without any duplicate elimination.
- Any nulls in the argument column are always eliminated before the function is applied, regardless of whether DISTINCT is specified, except for the case of COUNT(*), where nulls are handled just like nonnull values.
- If the argument happens to be an empty set, COUNT returns a value of zero; the other functions all return null. (We have argued elsewhere that this behavior is logically incorrect—see reference [8.19]—but it is the way SQL is defined.)

8.3.8 Get the maximum and minimum quantity for part P2.

```
SELECT MAX ( SP.QTY ) AS MAXQ, MIN ( SP.QTY ) AS MINQ
FROM   SP
WHERE  SP.P# = 'P2' ;
```

Observe that the FROM and WHERE clauses here both effectively provide part of the argument to the two aggregate functions. They should therefore logically appear within the argument-enclosing parentheses. Nevertheless, the query is indeed written as shown. This unorthodox approach to syntax has significant negative repercussions on the structure, usability, and orthogonality* of the SQL language. For instance, one immediate consequence is that aggregate functions cannot be nested, with the result that a query such as "Get the average total-part-quantity" cannot be formulated without cumbersome circumlocutions (because the expression AVG(SUM(QTY)) is not legal). Further details of such matters are beyond the scope of this book.

8.3.9 For each part supplied, get the part number and the total shipment quantity. (Example 7.5.4)

```
SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY
FROM   SP
GROUP  BY SP.P# ;
```

The foregoing is the SQL analog of the relational algebra expression

```
SUMMARIZE SP BY ( P# ) ADD SUM ( QTY ) AS TOTQTY
```

Observe in particular that if the GROUP BY clause is specified, expressions in the SELECT clause must be **single-valued per group**.

Here is an alternative formulation of the same query:

```
SELECT P.P#, ( SELECT SUM ( SP.QTY )
               FROM   SP
               WHERE  SP.P# = P.P# ) AS TOTQTY
FROM   P ;
```

The ability to use nested select expressions to represent scalar items (e.g., within the SELECT clause, as here) was added in SQL/92 and represents a major improvement over SQL as originally defined. In the example, it allows us to generate a result that includes rows for parts that are not supplied at all, which the previous formulation (using GROUP BY) does not. (The TOTQTY value for such parts will unfortunately be given as null, however, not zero.)

8.3.10 Get part numbers for all parts supplied by more than one supplier.

```
SELECT SP.P#
FROM   SP
GROUP  BY SP.P#
HAVING COUNT ( SP.S# ) > 1 ;
```

The HAVING clause is to groups what the WHERE clause is to rows; in other words, HAVING is used to eliminate groups, just as WHERE is used to eliminate rows. Expressions in a HAVING clause must be single-valued per group.

8.3.11 Get supplier names for suppliers who supply part P2. (Examples 6.6.1, 7.3.3)

* **Orthogonality** means *independence.* A language is orthogonal if independent concepts are kept independent, not mixed together in confusing ways. Orthogonality is desirable because the less orthogonal a language is, the more complicated it is and— paradoxically but simultaneously—the less powerful it is.

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = 'P2' ) ;
```

Explanation: This example makes use of what is called a **subquery**. Loosely speaking, a subquery is a SELECT–FROM–WHERE–GROUP BY–HAVING expression that is nested inside another such expression. Subqueries are typically used to represent the set of values to be searched via an **IN condition,** as the example illustrates. The system evaluates the overall query by evaluating the subquery first (at least conceptually). That subquery returns the set of supplier *numbers* for suppliers who supply part P2, namely the set {S1,S2,S3,S4}. The original expression is thus equivalent to the following simpler one:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN ( 'S1', 'S2', 'S3', 'S4' ) ;
```

It is worth pointing out that the original problem—"Get supplier names for suppliers who supply part P2"—can equally well be formulated by means of a *join,* e.g., as follows:

```
SELECT DISTINCT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2' ;
```

8.3.12 Get supplier names for suppliers who supply at least one red part. (Examples 6.6.2, 7.3.4)

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# IN
            ( SELECT P.P#
              FROM   P
              WHERE  P.COLOR = 'Red' ) ) ;
```

Subqueries can be nested to any depth. *Exercise:* Give some equivalent join formulations of this query.

8.3.13 Get supplier numbers for suppliers with status less than the current maximum status in the S table.

```
SELECT S.S#
FROM   S
WHERE  S.STATUS <
     ( SELECT MAX ( S.STATUS )
       FROM   S ) ;
```

This example involves *two distinct implicit range variables,* both denoted by the same symbol "S" and both ranging over the S table.

8.3.14 Get supplier names for suppliers who supply part P2. (Same as Example 8.3.11)

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = 'P2' ) ;
```

Explanation: The SQL expression "EXISTS (SELECT . . . FROM . . .)" evaluates to *true* if and only if the result of evaluating the "SELECT . . . FROM . . ." is not empty. In other words, the SQL **EXISTS** *function* corresponds to the *existential quantifier* of relational calculus (but see Chapter 20).

8.3.15 Get supplier names for suppliers who do not supply part P2. (Examples 6.6.6, 7.3.7)

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = 'P2' ) ;
```

Alternatively:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# NOT IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = 'P2' ) ;
```

8.3.16 Get supplier names for suppliers who supply all parts. (Examples 6.6.3, 7.3.6)

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP
              WHERE  SP.S# = S.S#
              AND    SP.P# = P.P# ) ) ;
```

SQL does not include any direct support for the universal quantifier FORALL; hence,

"FORALL-type" queries typically have to be expressed in terms of a negated existential quantifier, as in this example.

It is worth pointing out that expressions such as the one just shown, daunting though they might appear at first glance, are easily constructed by a user who is familiar with relational calculus, as explained in reference [7.6]. Alternatively—if they are still thought too daunting—then there are several "workaround" approaches that can be used that avoid the need for negated quantifiers. In the example, for instance, we might write:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT COUNT ( SP.P# )
         FROM   SP
         WHERE  SP.S# = S.S# ) = ( SELECT COUNT ( P.P# )
                                   FROM   P ) ;
```

("names of suppliers where the count of the parts they supply is equal to the count of all parts"). Note, however, that:

- First, this latter formulation relies—as the NOT EXISTS formulation did not—on the fact that no part number appears in relation SP that does not also appear in relation P. In other words, the two formulations are equivalent (and the second is correct) only because a certain integrity constraint is in effect.
- Second, the technique used in the second formulation to compare two counts was not supported in SQL as originally defined but was added in SQL/92. It is still not supported in all products.
- We remark too that what we would *really* like to do is to compare two *tables* (see the discussion of relational comparisons in Chapter 6), thereby expressing the query as follows:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT SP.P#
         FROM   SP
         WHERE  SP.S# = S.S# ) = ( SELECT P.P#
                                   FROM   P ) ;
```

SQL does not directly support comparisons between tables, however, and so we have to resort to the trick of comparing table cardinalities instead (relying on our own external knowledge to ensure that if the cardinalities are the same then the tables are the same too, at least in the situation under consideration). See Exercise 8.8 at the end of the chapter.

8.3.17 Get part numbers for parts that either weigh more than 16 pounds or are supplied by supplier S2, or both. (Example 7.3.9)

```
SELECT P.P#
FROM   P
WHERE  P.WEIGHT > 16
UNION
```

```
SELECT SP.P#
FROM   SP
WHERE  SP.S# = 'S2' ;
```

Redundant duplicate rows are always eliminated from the result of an unqualified **UNION, INTERSECT,** or **EXCEPT** (EXCEPT is the SQL analog of our MINUS). However, SQL also provides the qualified variants **UNION ALL, INTERSECT ALL,** and **EXCEPT ALL,** where duplicates (if any) are retained. We deliberately omit examples of these variants.

This brings us to the end of our list of retrieval examples. The list is rather long; nevertheless, there are numerous SQL features that we have not even mentioned. The fact is, SQL is an extremely *redundant* language, in the sense that it almost always provides numerous different ways of formulating the same query, and space simply does not permit us to describe all possible formulations and all possible options, even for the comparatively small number of examples we have discussed in this section.

8.4 Data Manipulation: Update Operations

As already mentioned, the SQL DML includes three update operations: INSERT, UPDATE (i.e., modify), and DELETE. We content ourselves here with a few simple examples, all of them (we trust) self-explanatory.

8.4.1 Single-row INSERT.

```
INSERT
INTO   P ( P#, PNAME, COLOR, WEIGHT, CITY )
VALUES ('P8', 'Sprocket', 'Pink', 14, 'Nice' ) ;
```

8.4.2 Multi-row INSERT.

```
INSERT
INTO   TEMP ( S#, CITY )
       SELECT S.S#, S.CITY
       FROM   S
       WHERE  S.STATUS > 15 ;
```

8.4.3 Multi-row UPDATE.

```
UPDATE P
SET    COLOR = 'Yellow',
       WEIGHT = P.WEIGHT + 5
WHERE  P.CITY = 'Paris' ;
```

8.4.4 Multi-row UPDATE.

```
UPDATE P
SET    CITY = ( SELECT S.CITY
                FROM   S
                WHERE  S.S# = 'S5' )
WHERE  P.COLOR = 'Red' ;
```

8.4.5 Multi-row DELETE.

```
DELETE
FROM   SP
WHERE  'London' =
     ( SELECT S.CITY
       FROM   S
       WHERE  S.S# = SP.S# ) ;
```

8.5 Table Expressions

An exhaustive treatment of table expressions would be out of place in this book. For purposes of reference, however, we do at least give in Fig. 8.1 (overleaf) a fairly complete BNF grammar for such expressions (the grammar is complete except for a few options having to do with nulls). And we elaborate on one special case—arguably the most important case in practice—namely, *select expressions.*

A select expression can be thought of, loosely, as a table expression that does not involve any UNIONs, EXCEPTs, or INTERSECTs ("loosely," because, of course, such operators might be involved in expressions that are *nested inside* the select expression). As Fig. 8.1 indicates, a select expression consists of several components: a SELECT clause, a FROM clause, a WHERE clause, a GROUP BY clause, and a HAVING clause (the last three of these clauses are optional). We now proceed to explain each of these components one by one.

The SELECT Clause

The SELECT clause takes the form

```
SELECT [ ALL | DISTINCT ] select-item-commalist
```

Explanation:

1. The *select-item-commalist* must not be empty. See below for a detailed discussion of select-items.
2. If neither ALL nor DISTINCT is specified, ALL is assumed.
3. We assume for the moment that the FROM, WHERE, GROUP BY, and HAVING clauses have already been evaluated. No matter which of those clauses are specified and which omitted, the conceptual result of evaluating them is always a table (possibly a "grouped" table—see later), which we will refer to as table *T1* (though the conceptual result is in fact unnamed).
4. Let *T2* be the table that is derived from *T1* by evaluating the specified select-items against *T1* (see below).
5. Let *T3* be the table that is derived from *T2* by eliminating redundant duplicate rows from *T2* if DISTINCT is specified, or a table that is identical to *T2* otherwise.
6. Table *T3* is the final result.

We turn now to an explanation of select-items. There are two cases to consider, of

```
table-expression
    ::=   join-table-expression
        | nonjoin-table-expression

join-table-expression
    ::=   table-reference [ NATURAL ] JOIN
                table-reference [ ON conditional-expression
                                | USING ( column-commalist ) ]
        | table-reference CROSS JOIN table-reference
        | ( join-table-expression )

table-reference
    ::=   table [ [ AS ] range-variable
                      [ ( column-commalist ) ] ]
        | ( table-expression ) [ AS ] range-variable
                                  [ ( column-commalist ) ]
        | join-table-expression

nonjoin-table-expression
    ::=   nonjoin-table-term
        | table-expression UNION [ ALL ]
                [ CORRESPONDING [ BY ( column-commalist ) ] ]
                     table-term
        | table-expression EXCEPT [ ALL ]
                [ CORRESPONDING [ BY ( column-commalist ) ] ]
                     table-term

nonjoin-table-term
    ::=   nonjoin-table-primary
        | table-term INTERSECT [ ALL ]
                [ CORRESPONDING [ BY ( column-commalist ) ] ]
                     table-primary

table-term
    ::=   nonjoin-table-term
        | join-table-expression

table-primary
    ::=   nonjoin-table-primary
        | join-table-expression

nonjoin-table-primary
    ::=   TABLE table
        | table-constructor
        | select-expression
        | ( nonjoin-table-expression )

table-constructor
    ::=   VALUES row-constructor-commalist

row-constructor
    ::=   scalar-expression
        | ( scalar-expression-commalist )
        | ( table-expression )

select-expression
    ::=   SELECT [ ALL | DISTINCT ] select-item-commalist
            FROM table-reference-commalist
               [ WHERE conditional-expression ]
                  [ GROUP BY column-commalist ]
                     [ HAVING conditional-expression ]

select-item
    ::=   scalar-expression [ [ AS ] column ]
        | [ range-variable . ] *
```

FIG. 8.1 A BNF grammar for SQL table expressions

which the second is just shorthand for a commalist of select-items of the first form: thus, the first case is really the more fundamental.

Case 1: The select-item takes the form

```
scalar-expression [ [ AS ] column ]
```

- The scalar expression will typically (but not necessarily) involve one or more columns of table *T1* (see paragraph 2 above). For each row of *T1,* the scalar expression is evaluated, to yield a scalar result. The commalist of such results (corresponding to evaluation of all select-items in the SELECT clause against a single row of *T1*) constitutes a single row of table *T2* (see paragraph 3 above). If the select-item includes an AS clause, the *un*qualified name *column* from that clause is assigned as the name of the corresponding column of table *T2** (the optional keyword AS is just noise and can be omitted without affecting the meaning). If the select-item does not include an AS clause, then (a) if it consists simply of a (possibly qualified) column name, that column name is assigned as the name of the corresponding column of table *T2;* (b) otherwise the corresponding column of table *T2* effectively has no name.
- If a select-item includes an aggregate function reference *and* the select expression does not include a GROUP BY clause (see below), then no select-item in the SELECT clause can include any reference to a column of table *T1* unless that column reference is the argument (or part of the argument) to an aggregate function reference.

Case 2: The select-item takes the form

```
[ range-variable . ] *
```

- If the qualifier is omitted (i.e., the select-item is just an unqualified asterisk), then this select-item must be the only select-item in the SELECT clause. This form is shorthand for a commalist of all of the columns of table *T1,* in left-to-right order.
- If the qualifier is included (i.e., the select-item consists of an asterisk qualified by a range variable name *R,* thus: "*R*.*"), then the select-item represents a commalist of all of the columns of the table associated with range variable *R,* in left-to-right order. (Recall that a table name can and often will be used as an implicit range variable. Thus, the select-item will frequently be of the form "*T*.*" rather than "*R*.*".)

The FROM Clause

The FROM clause takes the form

```
FROM table-reference-commalist
```

* Because it is, specifically, the name of a column of table *T2,* not table *T1,* any name introduced by such an AS clause cannot be used in the WHERE, GROUP BY, and HAVING clauses (if any) directly involved in the construction of that table *T1.* It can, however, be referenced in an associated ORDER BY clause, and also in an "outer" table expression that contains the select expression under discussion nested within it.

The *table-reference-commalist* must not be empty. Let the specified table references evaluate to tables *A, B, . . . C*, respectively. Then the result of evaluating the FROM clause is a table that is equal to the Cartesian product of *A, B, . . . C. Note:* The Cartesian product of a single table *T* is defined to be equal to *T;* in other words, it is (of course) legal for the FROM clause to contain just a single table reference.

The WHERE Clause

The WHERE clause takes the form

```
WHERE conditional-expression
```

Let *T* be the result of evaluating the immediately preceding FROM clause. Then the result of the WHERE clause is a table that is derived from *T* by eliminating all rows for which the conditional expression does not evaluate to *true*. If the WHERE clause is omitted, the result is simply *T*.

The GROUP BY Clause

The GROUP BY clause takes the form

```
GROUP BY column-commalist
```

The *column-commalist* must not be empty. Let *T* be the result of evaluating the immediately preceding FROM clause and WHERE clause (if any). Each *column* mentioned in the GROUP BY clause must be the optionally qualified name of a column of *T*. The result of the GROUP BY clause is a **grouped table**—i.e., a set of groups of rows, derived from *T* by conceptually rearranging it into the minimum number of groups such that within any one group all rows have the same value for the combination of columns identified by the GROUP BY clause. Note carefully, therefore, that the result is thus *not* a proper table. However, a GROUP BY clause never appears without a corresponding SELECT clause whose effect is to derive a proper table from that improper intermediate result, so little harm is done by this temporary deviation from the pure tabular framework.

If a select expression includes a GROUP BY clause, then there are restrictions on the form that the SELECT clause can take. To be specific, each select-item in the SELECT clause (including any that are implied by an asterisk shorthand) must be **single-valued per group**. Thus, such select-items must not include any reference to any column of table *T* that is not mentioned in the GROUP BY clause itself—*unless* that reference is the argument, or part of the argument, to one of the aggregate functions COUNT, SUM, AVG, MAX, or MIN, whose effect is to reduce some collection of scalar values from a group to a single such value.

The HAVING Clause

The HAVING clause takes the form

```
HAVING conditional-expression
```

Let *G* be the grouped table resulting from the evaluation of the immediately preceding FROM clause, WHERE clause (if any), and GROUP BY clause (if any). If there is no GROUP BY clause, then *G* is taken to be the result of evaluating the FROM and WHERE clauses alone, considered as a grouped table that contains exactly one group;* in other words, there is an implicit, conceptual GROUP BY clause in this case that specifies *no grouping columns at all*. The result of the HAVING clause is a grouped table that is derived from *G* by eliminating all groups for which the conditional expression does not evaluate to *true*.

Note 1: If the HAVING clause is omitted but the GROUP BY clause is included, the result is simply *G*. If the HAVING and GROUP BY clauses are both omitted, the result is simply the "proper"—i.e., nongrouped—table *T* resulting from the FROM and WHERE clauses.

Note 2: Scalar expressions in a HAVING clause must be single-valued per group (like scalar expressions in the SELECT clause if there is a GROUP BY clause, as discussed above).

Note 3: It is worth mentioning that the HAVING clause is totally redundant—i.e., for every select expression that involves such a clause, there is a semantically identical select expression that does not (exercise for the reader!).

A Comprehensive Example

We conclude our discussion of select expressions with a reasonably complex example that illustrates some (by no means all) of the points explained above. The query is as follows:

For all red and blue parts such that the total quantity supplied is greater than 350 (excluding from the total all shipments for which the quantity is less than or equal to 200), get the part number, the weight in grams, the color, and the maximum quantity supplied of that part.

```
SELECT P.P#,
       'Weight in grams =' AS TEXT1,
       P.WEIGHT * 454 AS GMWT,
       P.COLOR,
       'Max quantity =' AS TEXT2,
       MAX ( SP.QTY ) AS MQY
FROM   P, SP
WHERE  P.P# = SP.P#
AND    ( P.COLOR = 'Red' OR P.COLOR = 'Blue')
AND    SP.QTY > 200
GROUP  BY P.P#, P.WEIGHT, P.COLOR
HAVING SUM ( SP.QTY ) > 350 ;
```

Explanation: First, note that (as explained above) the clauses of a select expression are conceptually executed in the order in which they are written—with the sole exception

* This is what SQL says, though logically it should say *at most* one group (there should be no group at all if the FROM and WHERE clauses yield an empty table).

of the SELECT clause itself, which is executed last. In the example, therefore, we can imagine the result being constructed as follows:

1. **FROM:** The FROM clause is evaluated to yield a new table that is the Cartesian product of tables P and SP.
2. **WHERE:** The result of Step 1 is reduced by the elimination of all rows that do not satisfy the WHERE clause. In the example, therefore, rows not satisfying the conditional expression

   ```
     P.P# = SP.P# AND
   ( P.COLOR = 'Red' OR P.COLOR = 'Blue') AND
     SP.QTY > 200
   ```

   are eliminated.
3. **GROUP BY:** The result of Step 2 is grouped by values of the column(s) named in the GROUP BY clause. In the example, those columns are P.P#, P.WEIGHT, and P.COLOR. *Note:* In theory P.P# alone would be sufficient as the grouping column here, since P.WEIGHT and P.COLOR are themselves single-valued per part number. However, SQL is not aware of this latter fact, and will raise an error condition if P.WEIGHT and P.COLOR are omitted from the GROUP BY clause, because they *are* mentioned in the SELECT clause. See reference [9.6] in Chapter 9.
4. **HAVING:** Groups not satisfying the condition

   ```
   SUM ( SP.QTY ) > 350
   ```

   are eliminated from the result of Step 3.
5. **SELECT:** Each group in the result of Step 4 generates a single result row, as follows. First, the part number, weight, color, and maximum quantity are extracted from the group. Second, the weight is converted to grams. Third, the two literal strings "Weight in grams =" and "Max quantity =" are inserted at the appropriate points in the row. Note, incidentally, that—as the phrase "appropriate points in the row" suggests—we are relying here on the fact that columns of tables have a left-to-right ordering in SQL. The literal strings would not make much sense if they did not appear at those "appropriate points."

The final result looks like this:

P#	TEXT1	GMWT	COLOR	TEXT2	MQY
P1	Weight in grams =	5448	Red	Max quantity =	300
P5	Weight in grams =	5448	Blue	Max quantity =	400
P3	Weight in grams =	7718	Blue	Max quantity =	400

In conclusion, please understand that the algorithm just described is intended purely as a **conceptual** explanation of how the SELECT statement is evaluated. The algorithm is certainly correct, in the sense that it is guaranteed to produce the correct result. However, it would probably be rather inefficient if actually executed. For example, it would be very unfortunate if the system were actually to construct the Cartesian

product in Step 1. Considerations such as these are exactly the reason why relational systems require an optimizer (see Chapter 18). Indeed, the task of the optimizer in an SQL system can be characterized as that of finding an implementation procedure that will produce the same result as the conceptual algorithm sketched above but is more efficient than that algorithm.

8.6 Conditional Expressions

Like table expressions, conditional expressions appear in numerous contexts throughout the SQL language; in particular, of course, they are used in WHERE clauses to qualify or disqualify rows for subsequent processing. Here we discuss some of the most important features of such expressions. Please note, however, that our treatment is definitely *not* meant to be exhaustive; in particular, we ignore everything to do with nulls. (Conditional expressions, perhaps more than most other parts of the language, require significantly extended treatment when the implications and complications of nulls are taken into account, and certain conditional expression formats, not discussed in this chapter, are provided purely to deal with certain aspects of null support.)

As in the previous section, we begin with a BNF grammar (Fig. 8.2, overleaf). The reader will see that most conditional expression formats either have already been illustrated in earlier sections or else are self-explanatory; here we just offer a few words of explanation regarding a couple of specific cases, namely MATCH conditions and all-or-any conditions.

MATCH Conditions

A MATCH condition takes the form

```
row-constructor MATCH UNIQUE ( table-expression )
```

Let *r1* be the row that results from evaluating *row-constructor* and let *T* be the table that results from evaluating *table-expression.* Then the MATCH condition evaluates to *true* if and only if *T* contains exactly one row, *r2* say, such that the comparison

```
r1 = r2
```

evaluates to *true.* Here is an example:

```
SELECT SP.*
FROM   SP
WHERE  NOT ( SP.S# MATCH UNIQUE ( SELECT S.S# FROM S ) ) ;
```

("Get shipments that do not have exactly one matching supplier in the suppliers table"). Such a query might be useful in checking the integrity of the database, because, of course, there should not *be* any such shipments if the database is correct. Note, however, that an IN condition could be used to perform the same check.

Incidentally, the UNIQUE can be omitted from MATCH UNIQUE, but then MATCH becomes synonymous with IN (at least in the absence of nulls).

```
conditional-expression
    ::=   conditional-term
        | conditional-expression OR conditional-term

conditional-term
    ::=   conditional-factor
        | conditional-term AND conditional-factor

conditional-factor
    ::=   [ NOT ] conditional-primary

conditional-primary
    ::=   simple-condition | ( conditional-expression )

simple-condition
    ::=   comparison-condition
        | in-condition
        | match-condition
        | all-or-any-condition
        | exists-condition

comparison-condition
    ::=   row-constructor comparison-operator row-constructor

comparison-operator
    ::=   = | < | <= | > | >= | <>

in-condition
    ::=   row-constructor [ NOT ] IN ( table-expression )
        | scalar-expression [ NOT ] IN
                            ( scalar-expression-commalist )

match-condition
    ::=   row-constructor MATCH UNIQUE ( table-expression )

all-or-any-condition
    ::=   row-constructor
                comparison-operator ALL ( table-expression )
        | row-constructor
                comparison-operator ANY ( table-expression )

exists-condition
    ::=   EXISTS ( table-expression )
```

FIG. 8.2 A BNF grammar for SQL conditional expressions

All-or-Any Conditions

An all-or-any condition has the general form

```
row-constructor
    comparison-operator qualifier ( table-expression )
```

where *comparison-operator* is any of the usual set (=, <>, etc.), and *qualifier* is ALL or ANY. In general, an all-or-any condition evaluates to *true* if and only if the corresponding comparison without the ALL (respectively ANY) evaluates to *true* for all (respectively any) of the rows in the table represented by *table-expression.* (If that table is empty, the ALL conditions evaluate to *true,* the ANY conditions evaluate to *false.*) Here is an example ("Get part names for parts whose weight is greater than that of every blue part"):

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' ) ;
```

The result looks like this:

PNAME
Cog

Explanation: The nested table expression returns the set of weights for blue parts, namely the set {17,12}. The outer SELECT then returns the name of the only part whose weight is greater than every value in this set, namely part P6. In general, of course, the final result might contain any number of part names (including zero).

A word of caution is appropriate here, at least for native English speakers. The fact is, all-or-any conditions are seriously error-prone. A very natural English formulation of the foregoing query would use the word "any" in place of "every," which could easily lead to the (incorrect) use of >ANY instead of >ALL. Analogous criticisms apply to every one of the ANY and ALL operators.

8.7 Scalar Expressions

Scalar expressions in SQL are essentially straightforward. We content ourselves in this section with a list of some of the most important operators that can be used in the construction of such expressions, and offering a few additional comments on a couple of those operators—CASE and CAST—whose meaning is perhaps not immediately apparent. Note that the aggregate functions also can appear within such expressions, since they return a scalar result. Furthermore, a table expression enclosed in parentheses can also be treated as a scalar value, so long as it evaluates to a table of exactly one row and one column. As mentioned earlier (in the discussion of Example 8.3.9), this last possibility, which was introduced with SQL/92, represents a *major* improvement over SQL as originally defined.

Here then is the list of operators, in alphabetic order.

```
arithmetic operators (+, -, *, /)    OCTET_LENGTH
BIT_LENGTH                           POSITION
CASE                                 SESSION_USER
```

```
CAST                             SUBSTRING
CHARACTER_LENGTH                 SYSTEM_USER
concatenation (||)               TRIM
CURRENT_USER                     UPPER
LOWER                            USER
```

We now elaborate slightly on the operators CASE and CAST.

CASE Operations

A CASE operation returns one of a specified set of values, depending on a specified condition. For example:

```
CASE
   WHEN S.STATUS <  5 THEN 'Last resort'
   WHEN S.STATUS < 10 THEN 'Dubious'
   WHEN S.STATUS < 15 THEN 'Not too good'
   WHEN S.STATUS < 20 THEN 'Mediocre'
   WHEN S.STATUS < 25 THEN 'Acceptable'
   ELSE                    'Fine'
END
```

CAST Operations

CAST converts a specified scalar value to a specified scalar data type (possibly a user-defined domain). For example:

```
CAST ( 'S8' AS S# )
```

Not all pairs of data types are mutually convertible; for example, conversions between numbers and bit strings are not supported. The reader is referred to reference [8.1] for details of precisely which data types can be converted to which.

8.8 Embedded SQL

As explained in Chapter 3, SQL statements can be executed interactively, or they can be executed as part of an application program (in which case the SQL statements are physically embedded within the program source code, intermixed with the statements of the host language). Up to this point, however, we have ignored the latter case and have tacitly assumed—where it made any difference—that the language was being used interactively. Now we turn our attention to embedded SQL specifically.

The fundamental principle underlying embedded SQL, which we refer to as the **dual-mode principle,** is that *any SQL statement that can be used interactively can also be used in an application program.* Of course, there are various differences of detail between a given interactive SQL statement and its embedded counterpart, and retrieval operations in particular require significantly extended treatment in a host program environment (see later); but the principle is nevertheless broadly true. (Its converse is not, by the way; that is, there are a number of embedded SQL statements that cannot be used interactively, as we will see.)

Note clearly also that the dual-mode principle applies to the entire SQL language, not just to the data manipulation operations. It is true that the DML operations are far and away the ones most frequently used in a programming context, but there is nothing wrong in embedding (for example) a CREATE TABLE statement in a program, if it makes sense to do so for the application at hand.

Before we can discuss the actual statements of embedded SQL, it is necessary to cover a number of preliminary details. Most of those details are illustrated by the program fragment shown in Fig. 8.3. (To fix our ideas we assume that the host language is PL/I. Most of the ideas translate into other host languages with only minor changes.) Points arising:

1. Embedded SQL statements are prefixed by **EXEC SQL,** so that they can easily be distinguished from statements of the host language, and are terminated by a special **terminator** symbol (a semicolon for PL/I).
2. An *executable* SQL statement (from now on we will usually drop the "embedded") can appear wherever an executable host statement can appear. Note the qualifier "executable" here: Unlike interactive SQL, embedded SQL includes some statements that are purely declarative, not executable. For example, DECLARE CURSOR is not an executable statement (see later), nor are BEGIN and END DECLARE SECTION (see paragraph 5 below), and nor is WHENEVER (see paragraph 9 below).
3. SQL statements can include references to **host variables;** such references must include a colon prefix to distinguish them from SQL column names. Host variables

```
EXEC SQL BEGIN DECLARE SECTION ;

   DCL SQLSTATE CHAR(5) ;
   DCL P#       CHAR(6) ;
   DCL WEIGHT   FIXED DECIMAL(3) ;

EXEC SQL END DECLARE SECTION ;

P# = 'P2' ;                      /* for example              */
EXEC SQL SELECT P.WEIGHT
         INTO   :WEIGHT
         FROM   P
         WHERE  P.P# = :P# ;
IF SQLSTATE = '00000'
THEN ... ;                       /* WEIGHT = retrieved value */
ELSE ... ;                       /* some exception occurred  */
```

FIG. 8.3 Fragment of a PL/I program with embedded SQL

can appear in embedded SQL (DML statements only) wherever a literal can appear in interactive SQL. They can also appear in an INTO clause on SELECT (see paragraph 4 below) or FETCH (see later) to designate targets for retrieval, and in certain "dynamic SQL" statements (again, see later).

4. Notice the **INTO clause** on the SELECT statement in Fig. 8.3. The purpose of that clause is (as just indicated) to specify the target variables into which values are to be retrieved; the *i*th target variable mentioned in the INTO clause corresponds to the *i*th value to be retrieved as specified by the SELECT clause.
5. All host variables that will be referenced in SQL statements must be defined within an **embedded SQL declare section,** which is delimited by the **BEGIN** and **END DECLARE SECTION** statements.
6. Every embedded SQL program must include a host variable called **SQLSTATE.*** After any SQL statement has been executed, a status code is returned to the program in that variable; in particular, a status code of 00000 means that the statement executed successfully, and a value of 02000 means that the statement did execute but no data was found to satisfy the request. In principle, therefore, every SQL statement in the program should be followed by a test on SQLSTATE, and appropriate action taken if the value is not what was expected. In practice, however, such testing is usually implicit. See paragraph 9 below.
7. Host variables must have a **data type** appropriate to the uses to which they are put. In particular, a host variable that is to be used as a target (e.g., on FETCH) must have a data type that is compatible with that of the expression that provides the value to be assigned to that target; likewise, a host variable that is to be used as a source (e.g., on UPDATE) must have a data type that is compatible with that of the SQL column to which values of that source are to be assigned. Similar remarks apply to a host variable that is to be used in a comparison, or indeed in any kind of scalar expression. For details of what it means for data types to be compatible in the foregoing sense, the reader is referred to the official standard document [8.1].
8. Host variables and SQL columns can have the same name.
9. As already mentioned, every SQL statement should in principle be followed by a test of the returned SQLSTATE value. The **WHENEVER** statement is provided to simplify this process. The WHENEVER statement has the syntax:

   ```
   EXEC SQL WHENEVER condition action terminator
   ```

   where *terminator* is as explained in paragraph 1 above, *condition* is either SQLERROR or NOT FOUND, and "action" is either CONTINUE or a GO TO statement. WHENEVER is not an executable statement; rather, it is a directive to

* Earlier versions of SQL used a variable called SQLCODE in place of SQLSTATE; SQLSTATE was added in SQL/92, and SQLCODE is now officially "deprecated," because most of its values (unlike those of SQLSTATE) are implementation-defined instead of being prescribed by the standard.

the SQL language processor. "WHENEVER *condition* GO TO *label*" causes that processor to insert an "IF *condition* GO TO *label*" statement after each executable SQL statement it encounters; "WHENEVER *condition* CONTINUE" causes it not to insert any such statements, the implication being that the programmer will insert such statements by hand. The two *conditions* are defined as follows:

NOT FOUND	*means*	no data was found (SQLSTATE = 02000)
SQLERROR	*means*	an error occurred (see reference [8.1] for SQLSTATE)

Each WHENEVER statement the SQL processor encounters on its sequential scan through the program text (for a particular condition) overrides the previous one it found (for that condition).

So much for the preliminaries. In the rest of this section we concentrate on DML operations specifically. As already indicated, most of those operations can be handled in a fairly straightforward fashion (i.e., with only minor changes to their syntax). Retrieval operations require special treatment, however. The problem is that such operations retrieve many rows (in general), not just one, and host languages are typically not equipped to handle the retrieval of more than one row at a time. It is therefore necessary to provide some kind of bridge between the set-at-a-time retrieval level of SQL and the row-at-a-time retrieval level of the host; and **cursors** provide such a bridge. A cursor is a new kind of SQL object, one that applies to embedded SQL only (because of course interactive SQL has no need of it). It consists essentially of a kind of *pointer* that can be used to run through a collection of rows, pointing to each of the rows in turn and thus providing addressability to those rows one at a time. However, we defer detailed discussion of cursors to a later subsection, and consider first those statements that have no need of them.

Operations Not Involving Cursors

The data manipulation statements that do not need cursors are as follows:

- "Singleton SELECT"
- INSERT
- UPDATE (except the CURRENT form—see later)
- DELETE (again, except the CURRENT form—see later)

We give examples of each of these statements in turn.

8.8.1 (Singleton SELECT) Get status and city for the supplier whose supplier number is given by the host variable GIVENS#.

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK, :CITY
         FROM   S
         WHERE  S# = :GIVENS# ;
```

We use the term **singleton SELECT** to mean a select expression* that evaluates to a table containing at most one row. In the example, if there exists exactly one row in table S satisfying the WHERE condition, then the STATUS and CITY values from that row will be assigned to the host variables RANK and CITY as requested, and SQLSTATE will be set to 00000. If no S row satisfies the WHERE condition, SQLSTATE will be set to 02000; and if more than one does, the program is in error, and SQLSTATE will be set to an error code.

8.8.2 (INSERT) Insert a new part (part number, name, and weight given by host variables P#, PNAME, PWT, respectively; color and city unknown) into table P.

```
EXEC SQL INSERT
         INTO   P ( P#, PNAME, COLOR, WEIGHT, CITY )
         VALUES ( :P#, :PNAME, DEFAULT, :PWT, DEFAULT ) ;
```

8.8.3 (UPDATE) Increase the status of all London suppliers by the amount given by the host variable RAISE.

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CITY = 'London' ;
```

If no supplier rows satisfy the WHERE condition, SQLSTATE will be set to 02000.

8.8.4 (DELETE) Delete all shipments for suppliers whose city is given by the host variable CITY.

```
EXEC SQL DELETE
         FROM   SP
         WHERE  :CITY =
              ( SELECT CITY
                FROM   S
                WHERE  S.S# = SP.S# ) ;
```

Again SQLSTATE will be set to 02000 if no rows satisfy the WHERE condition.

Operations Involving Cursors

Now we turn to the question of set-level retrieval—i.e., retrieval of an entire set of many rows, instead of just one row. As explained earlier, what is needed here is a mechanism for accessing the rows in the set one by one, and **cursors** provide such a mechanism. The process is illustrated in outline by the example of Fig. 8.4, which is intended to retrieve supplier details (S#, SNAME, and STATUS) for all suppliers in the city given by the host variable Y.

Explanation: The DECLARE X CURSOR . . . statement defines a cursor called X, with an associated table expression as specified by the SELECT that forms part of that DECLARE. That table expression is not evaluated at this point; DECLARE CURSOR is a purely declarative statement. The expression *is* evaluated when the cursor is opened. The FETCH statement is then used to retrieve rows one at a time from the

* It is not quite a select expression as defined in Section 8.5, owing to the presence of the INTO clause.

```
EXEC SQL DECLARE X CURSOR FOR      - define the cursor    */
         SELECT S.S#, S.SNAME, S.STATUS
         FROM   S
         WHERE  S.CITY = :Y ;

EXEC SQL OPEN X ;                  /* execute the query   */
         DO for all S rows accessible via X ;
            EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                   /* fetch next supplier */
            .........
         END ;
EXEC SQL CLOSE X ;                 /* deactivate cursor X */
```

FIG. 8.4 Multi-row retrieval

resulting set, assigning retrieved values to host variables in accordance with the specifications of the INTO clause in that statement. (For simplicity we have given the host variables the same names as the corresponding database columns. Notice that the SELECT in the cursor declaration does not have an INTO clause of its own.) Since there will be many rows in the result set, the FETCH will normally appear within a loop (DO . . . END in PL/I); the loop will be repeated so long as there are more rows still to come in that result set. On exit from the loop, cursor X is closed.

Now let us consider cursors and cursor operations in more detail. First, a cursor is declared by means of a **DECLARE CURSOR** statement, which takes the general form

```
EXEC SQL DECLARE cursor CURSOR
         FOR table-expression
       [ ORDER BY order-item-commalist ] ;
```

where *table-expression* and *order-item-commalist* are as described earlier in this chapter. For an example, see Fig. 8.4. *Note:* We are ignoring a few optional specifications in the interests of brevity. See reference [8.1] or reference [8.5] for further details.

As previously stated, the DECLARE CURSOR statement is declarative, not executable; it declares a cursor with the specified name and having the specified table expression permanently associated with it. The table expression can include host variable references. A program can include any number of DECLARE CURSOR statements, each of which must (of course) be for a different cursor.

Three executable statements are provided to operate on cursors: **OPEN, FETCH,** and **CLOSE.**

1. The statement

   ```
   EXEC SQL OPEN cursor ;
   ```

   opens or *activates* the specified cursor (which must not currently be open). In effect, the table expression associated with the cursor is evaluated (using the current values for any host variables referenced within that expression); a set of rows is thus identified and becomes the current **active set** for the cursor. The cursor also identifies a *position* within that active set, namely the position just before the first row in the set. (Active sets are always considered to have an ordering, so that the concept of position has meaning. The ordering is either that defined by the ORDER BY clause, or a system-determined ordering in the absence of such a clause.)

2. The statement

   ```
   EXEC SQL FETCH cursor INTO host-variable-commalist ;
   ```

   advances the specified cursor (which must be open) to the next row in the active set and then assigns values from that row to host variables as specified in the INTO clause. If there is no next row when FETCH is executed, then SQLSTATE is set to 02000 and no data is retrieved.

3. The statement

   ```
   EXEC SQL CLOSE cursor ;
   ```

   closes or *deactivates* the specified cursor (which must currently be open). The cursor now has no current active set. However, it can subsequently be opened again, in which case it will acquire another active set—probably not exactly the same set as before, especially if the values of any host variables referenced in the cursor declaration have changed in the meantime. Note that changing the values of those host variables while the cursor is open has no effect on the current active set.

Two further statements can include references to cursors. These are the **CURRENT** forms of **UPDATE** and **DELETE**. If a cursor, X say, is currently positioned on a particular row, then it is possible to UPDATE or DELETE the "current of X," i.e., the row on which X is positioned. For example:

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CURRENT OF X ;
```

UPDATE . . . WHERE CURRENT and DELETE . . . WHERE CURRENT are not permitted if the table expression in the cursor declaration would define a nonupdatable view if it were part of a CREATE VIEW statement (see Chapter 17).

Dynamic SQL

Dynamic SQL consists of a set of embedded SQL facilities that are provided specifically to allow the construction of generalized, online, and possibly interactive applications. (Recall from Chapter 1 that an online application is an application that supports access to the database from an online terminal.) Consider what a typical online application has to do. In outline, the steps it must go through are as follows.

1. Accept a command from the terminal.
2. Analyze that command.
3. Issue appropriate SQL statements to the database.
4. Return a message and/or results to the terminal.

If the set of commands the program can accept is fairly small, as in the case of (perhaps) a program handling airline reservations, then the set of possible SQL statements to be issued will probably also be small and can be "hardwired" into the program. In this case, Steps 2 and 3 above will consist simply of logic to examine the input command and then branch to the part of the program that issues the predefined SQL statement(s). If, on the other hand, there can be great variability in the input, then it might not be practicable to predefine and "hardwire" SQL statements for every possible command. Instead, it is probably much more convenient to *construct* the necessary SQL statements dynamically, and then to compile and execute those constructed statements dynamically. The facilities of dynamic SQL are provided to assist in this process.

The two principal dynamic statements are PREPARE and EXECUTE. Their use is illustrated in the following (unrealistically simple but accurate) example.

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE SP.QTY < 300' ;
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

Explanation:

1. The name SQLSOURCE identifies a PL/I varying length character string variable in which the program will somehow construct the source form (i.e., character string representation) of some SQL statement—a DELETE statement, in our particular example.
2. The name SQLPREPPED, by contrast, identifies an *SQL* variable, not a PL/I variable, that will be used (conceptually) to hold the compiled form of the SQL statement whose source form is given in SQLSOURCE. The names SQLSOURCE and SQLPREPPED are arbitrary, of course.
3. The assignment statement "SQLSOURCE = . . . ;" assigns to SQLSOURCE the source form of an SQL DELETE statement. In practice, of course, the process of constructing such a source statement is likely to be much more complex—perhaps involving the input and analysis of some request from the end-user, expressed in natural language or some other form more "user-friendly" than plain SQL.
4. The PREPARE statement then takes that source statement and "prepares" (i.e., compiles) it to produce an executable version, which it stores in SQLPREPPED.
5. Finally, the EXECUTE statement executes that SQLPREPPED version and thus causes the actual DELETE to occur. SQLSTATE information from the DELETE is returned exactly as if the DELETE had been executed directly in the normal way.

Note that since it denotes an SQL variable, not a PL/I variable, the name SQLPREPPED does not have a colon prefix when it is referenced in the PREPARE and EXECUTE statements. Note too that such SQL variables are not explicitly declared.

Incidentally, the process just described is exactly what happens when SQL statements themselves are entered interactively. Most systems provide some kind of interactive SQL query processor. That processor is in fact just a particular kind of generalized online application; it is ready to accept an extremely wide variety of input, *viz.* any valid (or invalid!) SQL statement. It uses the facilities of dynamic SQL to construct suitable SQL statements corresponding to its input, to compile and execute those constructed statements, and to return messages and results back to the terminal.

For more information regarding dynamic SQL, see reference [8.1] or reference [8.5].

8.9 Summary

This concludes our survey of the major features of the SQL standard ("SQL/92"). We began by discussing the basic data objects. To review, the **principal DDL statements** are as follows:

```
CREATE DOMAIN   CREATE TABLE
ALTER DOMAIN    ALTER TABLE
DROP DOMAIN     DROP TABLE
```

Two further DDL statements, CREATE and DROP VIEW, are discussed in Chapter 17.

1. Regarding **domains,** we stressed the point that domains in SQL are very far from being true relational domains; in fact, SQL domains are basically little more than a shorthand. More precisely, they provide (a) domain-level **data type** specifications, (b) domain-level **default** definitions, and (c) domain-level **integrity constraints**. We summarized SQL's scalar data types, but omitted much of the complexity (unwarranted complexity, in this writer's opinion) that attaches to other aspects of SQL-style domains.
2. Regarding **base tables,** we first pointed out that SQL tables in general differ from true relations in at least two respects: They permit duplicate rows, and they have a left-to-right ordering to their columns. Base tables in particular have one or more columns, zero or more declared **candidate keys** (of which at most one can be declared to be the **primary** key), zero or more declared **foreign keys,** and zero or more declared **check constraints**. The following foreign key delete and update rules are supported: NO ACTION, CASCADE, SET DEFAULT, and SET NULL.

We also briefly described the **Information Schema,** which consists of a set of prescribed views of a hypothetical "Definition Schema."

Next we moved on to discuss **data manipulation** operations. To be specific:

1. We described **retrieval operations** (which basically means **table expressions**). Usually such an operation consists of a single **select expression,** but various kinds of explicit **JOIN** expressions are also supported, and join expressions and select expressions can be combined together in arbitrary ways using the **UNION, INTERSECT,** and **EXCEPT** operators. We also mentioned the use of **ORDER BY** to order the table resulting from a table expression (of any kind).
2. Regarding **select expressions** in particular, we described:

   - The basic **SELECT clause** itself, including the use of **DISTINCT,** scalar expressions, the introduction of result column names, and "SELECT *"
   - The **FROM clause,** including the use of range variables and the use of table references within the FROM clause that are more complex than just a simple table name
   - The **WHERE clause,** including the use of **subqueries** and the **EXISTS** function
   - The **GROUP BY** and **HAVING clauses,** including the use of the **aggregate functions** COUNT, SUM, AVG, MAX, and MIN

   We also gave a **conceptual evaluation algorithm** (i.e., an outline of a formal definition) for select expressions.
3. We briefly described the update operations **INSERT, UPDATE,** and **DELETE.**

Next, we gave more details of (a) **table expressions** (including a BNF grammar); (b) **conditional expressions** (again including a BNF grammar, and elaborating on **MATCH conditions** and **all-or-any conditions** in particular); and (c) **scalar expressions** (elaborating on the operators **CASE** and **CAST**). We also stressed the point that a select expression that evaluates to a single-column, single-row table can be used as a scalar value (e.g., within a SELECT or WHERE clause).

Finally, we described the principal features of **embedded SQL**. The basic idea behind embedded SQL is **the dual-mode principle,** i.e., the principle that (insofar as possible) *any SQL statement that can be used interactively can also be used in an application program.* The major exception to this principle arises in connection with **multi-row retrieval operations,** which require the use of a **cursor** to bridge the gap between the set-at-a-time retrieval level of SQL and the row-at-a-time retrieval level of host languages such as PL/I. (Perhaps this is the place to mention that the SQL standard [8.1] also supports Ada, C, COBOL, Fortran, MUMPS, and Pascal in addition to PL/I.)

Following a number of necessary (though mostly syntactic) preliminaries—including in particular a brief explanation of **SQLSTATE**—we considered those operations, namely **singleton SELECT, INSERT, UPDATE,** and **DELETE,** that have no need for cursors. Then we turned to the operations that *do* need cursors, and discussed **DECLARE CURSOR, OPEN, FETCH, CLOSE,** and the **CURRENT** forms of **UPDATE** and **DELETE**. (The standard refers to the CURRENT forms of these operators as *positioned* UPDATE and DELETE, and uses the term *searched* UPDATE and DELETE for the nonCURRENT or "out of the blue" forms.) Finally, we gave a very brief introduction to the concept of **dynamic SQL,** mentioning the **PREPARE** and **EXECUTE** statements in particular.

Exercises

8.1 Give an SQL data definition for the suppliers-parts-projects database.

8.2 Write a sequence of DROP statements that will have the effect of destroying all of the contents of the suppliers-parts-projects database.

8.3 In Section 8.2 we described the CREATE TABLE statement as defined by the SQL standard [8.1]. Many commercial SQL products support additional options on that statement, however, typically having to do with indexes, disk space allocation, and other implementation matters, and thereby undermining the objectives of physical data independence and intersystem compatibility. Investigate any SQL product that might be available to you. Do the foregoing criticisms apply to that product? Specifically, what additional CREATE TABLE options does that product support?

8.4 Once again, investigate any SQL product that might be available to you. Does that product support the Information Schema? If not, what *does* its catalog support look like?

8.5 Show that SQL is *relationally complete* (see Chapter 6), in the sense that, for any arbitrary expression of the relational algebra, there exists a semantically equivalent SQL expression.

8.6 Does SQL have equivalents of the relational EXTEND and SUMMARIZE operations?

8.7 Is there an SQL equivalent of the relational assignment operation?

8.8 Are there SQL equivalents of the relational comparison operations?

8.9 Give as many different SQL formulations as you can think of for the query "Get supplier names for suppliers who supply part P2" (see Examples 8.3.11 and 8.3.14).

8.10 There are two formally equivalent approaches to the manipulative part of the relational model, the calculus and the algebra. One implication is that there are therefore two styles on which the design of a query language can be based. For example, QUEL is (at least arguably) calculus-based, and so is QBE; by contrast, the language ISBL of the system PRTV [6.8] is algebra-based. Is SQL algebra-based or calculus-based?

8.11 Give SQL solutions to Exercises 6.13-6.48.

8.12 Give SQL formulations for the following update problems.

(a) Insert a new supplier S10 into table S. The name and city are Smith and New York respectively; the status is not yet known.

(b) Change the color of all red parts to orange.

(c) Delete all projects for which there are no shipments.

8.13 Using the suppliers-parts-projects database, write a program with embedded SQL statements to list all supplier rows, in supplier number order. Each supplier row should be immediately followed in the listing by all project rows for projects supplied by that supplier, in project number order.

8.14 Given the tables

```
CREATE TABLE PARTS
     ( P# ... , DESCRIPTION ... ,
       PRIMARY KEY ( P# ) ) ;

CREATE TABLE PART_STRUCTURE
     ( MAJOR_P# ... , MINOR_P# ... , QTY ... ,
       PRIMARY KEY ( MAJOR_P#, MINOR_P# ),
       FOREIGN KEY ( MAJOR_P# ) REFERENCES PARTS,
       FOREIGN KEY ( MINOR_P# ) REFERENCES PARTS ) ;
```

where PART_STRUCTURE shows which parts (MAJOR_P#) contain which other parts (MINOR_P#) as first-level components, write an SQL program to list all component parts of a given part, to all levels (the **parts explosion** problem). The following sample values (repeated from Fig. 4.4) might help you visualize this problem:

PART_STRUCTURE

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

References and Bibliography

8.1 International Organization for Standardization (ISO). *Database Language SQL.* Document ISO/IEC 9075:1992. Also available as American National Standards Institute (ANSI) Document ANSI X3.135-1992.

The current version of the official ISO/ANSI SQL standard, known informally as *SQL2, SQL-92,* or *SQL/92.* The point is worth mentioning that, although SQL is widely recognized as the international "relational" standard, the standard document does not describe itself as such; in fact, it never actually mentions the term "relation" at all!

8.2 X/Open. *Structured Query Language (SQL): CAE Specification C201* (September 1992).

Defines the X/Open SQL standard.

8.3 U.S. Department of Commerce, National Institute of Standards and Technology. *Database Language SQL.* FIPS PUB 127-2 (1992).

Defines the Federal Information Processing (FIPS) SQL standard.

8.4 IBM Corp.. *Systems Application Architecture Common Programming Interface: Database Reference.* IBM Document No. SC26-4348.

Defines the IBM SAA SQL standard.

8.5 C. J. Date and Hugh Darwen. *A Guide to the SQL Standard* (3rd edition). Reading, Mass.: Addison-Wesley (1993).

Portions of this chapter are based on material from this reference, which is intended as a comprehensive tutorial on SQL/92. References [8.6] and [8.7] below are also SQL/92 tutorials.

8.6 Stephen Cannan and Gerard Otten. *SQL—The Standard Handbook.* Maidenhead, UK: McGraw-Hill International (1993).

8.7 Jim Melton and Alan R. Simon. *Understanding The New SQL: A Complete Guide.* San Mateo, Calif.: Morgan Kaufmann (1993).

8.8 Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." Proc. ACM SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich. (May 1974).

The paper that first introduced the SQL language (or SEQUEL, as it was originally called; the name was subsequently changed for legal reasons).

8.9 M. M. Astrahan and R. A. Lorie. "SEQUEL-XRM: A Relational System." Proc. ACM Pacific Regional Conference, San Francisco, Calif. (April 1975).

Describes the first prototype implementation of SEQUEL, the original version of SQL [8.8]. See also references [8.12–8.13], which perform an analogous function for System R.

8.10 Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin. "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL." Proc. NCC 44, Anaheim, Calif. Montvale, N.J.: AFIPS Press (May 1975).

SQL's predecessor SEQUEL [8.8] was based on an earlier language called SQUARE. The two languages were fundamentally the same, in fact, but SQUARE used a rather mathematical syntax whereas SEQUEL was based on English keywords such as SELECT, FROM, WHERE, etc. The present paper reports on a set of experiments that were carried out on the usability of the two languages, using college students as subjects. A number of revisions were made to SEQUEL as a result of that work [8.11].

8.11 Donald D. Chamberlin *et al.* "SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control." *IBM J. R&D. 20,* No. 6 (November 1976). See also errata: *IBM J. R&D. 21,* No. 1 (January 1977).

Experience from the early prototype implementation of SEQUEL discussed in reference [8.9] and results from the usability tests reported in reference [8.10] led to the design of a revised version of the language called SEQUEL/2. The language supported by System R [8.12–8.13] was basically SEQUEL/2 (with the conspicuous absence of the so-called "assertion" and "trigger" facilities), plus certain extensions suggested by early user experience [8.14].

8.12 M. M. Astrahan *et al.* "System R: Relational Approach to Database Management." *ACM TODS 1,* No. 2 (June 1976).

System R was *the* major prototype implementation of (an early version of) the SQL language. This paper describes the architecture of System R as originally planned.

8.13 M. W. Blasgen *et al.* "System R: An Architectural Overview." *IBM Sys. J. 20,* No. 1 (February 1981).

Describes the architecture of System R as it became by the time the system was fully implemented.

8.14 Donald D. Chamberlin. "A Summary of User Experience with the SQL Data Sublanguage." Proc. International Conference on Databases, Aberdeen, Scotland (July 1980). Also available as IBM Research Report RJ2767 (April 1980).

Discusses early user experience with System R and proposes some extensions to the SQL language in the light of that experience. A few of those extensions—EXISTS, LIKE (not discussed in the present chapter), PREPARE, and EXECUTE—were in fact implemented in the final version of System R.

8.15 Donald D. Chamberlin, Arthur M. Gilbert, and Robert A. Yost. "A History of System R and SQL / Data System." Proc. 7th International Conference on Very Large Data Bases, Cannes, France (September 1981).

Discusses the lessons learned from the System R prototype and describes the evolution of that prototype into the first of IBM's relational product family, namely SQL/DS (recently renamed "DB2 for VM and VSE").

8.16 Donald D. Chamberlin *et al.* "A History and Evaluation of System R." *CACM 24,* No. 10 (October 1981).

Describes the three principal phases of the System R project (preliminary prototype, multi-user prototype, evaluation), with emphasis on the technologies of compilation and optimization pioneered in System R. There is some overlap between this paper and reference

[8.15]. *Note:* It is interesting to compare and contrast this paper with reference [7.14], which performs an analogous function for the University INGRES project.

8.17 C. J. Date. "A Critique of the SQL Database Language." *ACM SIGMOD Record 14,* No. 3 (November 1984). Republished in C. J. Date, *Relational Database: Selected Writings,* Reading, Mass.: Addison-Wesley (1986).

SQL is very far from perfect. This paper presents a critical analysis of a number of the language's principal shortcomings (mainly from the standpoint of formal computer languages in general, rather than database languages specifically). *Note:* Certain of this paper's criticisms do not apply to SQL/92.

8.18 C. J. Date. "What's Wrong with SQL?" In C. J. Date, *Relational Database Writings 1985–1989.* Reading, Mass.: Addison-Wesley (1990).

Discusses some additional shortcomings of SQL, over and above those identified in reference [8.17], under the headings "What's wrong with SQL *per se,*" "What's wrong with the SQL standard," and "Application portability." *Note:* Again, certain of this paper's criticisms do not apply to SQL/92.

8.19 C. J. Date. "How SQL Missed the Boat." *Database Programming & Design 6,* No. 9 (September 1993).

A succinct summary of SQL's shortcomings with respect to its support (or lack thereof) for the structural, manipulative, and integrity aspects of the relational model.

8.20 C. J. Date. "SQL Dos and Don'ts." In C. J. Date, *Relational Database Writings 1985–1989.* Reading, Mass.: Addison-Wesley (1990).

This paper offers some practical advice on how to use SQL in such a way as (a) to avoid some of the potential pitfalls arising from the problems discussed in references [8.17–8.19] and (b) to realize the maximum possible benefits in terms of productivity, portability, connectivity, and so forth.

8.21 M. Negri, S. Pelagatti, and L. Sbattella. "Formal Semantics of SQL Queries." *ACM TODS 16,* No. 3 (September 1991).

To quote from the abstract: "The semantics of SQL queries are formally defined by stating a set of rules that determine a syntax-driven translation of an SQL query to a formal model called Extended Three Valued Predicate Calculus (E3VPC), which is largely based on well-known mathematical concepts. Rules for transforming a general E3VPC expression to a canonical form are also given; . . . problems like equivalence analysis of SQL queries are completely solved." Note, however, that the SQL dialect considered is only the first version of the standard ("SQL/86"), not SQL/92.

Answers to Selected Exercises

8.1
```
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 J#     CHAR(4) ;
CREATE DOMAIN QTY    NUMERIC(9) ;
```