8 SQL-99: Schema Definition, Basic Constraints, and Queries

The SQL language may be considered one of the major reasons for the success of relational databases in the commercial world. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. The reason is that even if users became dissatisfied with the particular relational DBMS product they chose to use, converting to another relational DBMS product would not be expected to be too expensive and time-consuming, since both systems would follow the same language standards. In practice, of course, there are many differences between various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if both relational systems faithfully support the standard, then conversion between the two systems should be much simplified. Another advantage of having such a standard is that users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sublanguage (SQL) if both relational DBMSs support standard SQL.

This chapter presents the main features of the SQL standard for *commercial* relational DBMSs, whereas Chapter 5 presented the most important concepts underlying the *formal* relational data model. In Chapter 6 (Sections 6.1 through 6.5) we discussed the *relational algebra* operations, which are very important for understanding the types of requests that may be specified on a relational database. They are also important for query processing and optimization in a relational DBMS, as we shall see in Chapters 15 and 16. However, the

relational algebra operations are considered to be too technical for most commercial DBMS users because a query in relational algebra is written as a sequence of operations that, when executed, produces the required result. Hence, the user must specify how—that is, *in what order*—to execute the query operations. On the other hand, the SQL language provides a higher-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. Although SQL includes some features from relational algebra, it is based to a greater extent on the *tuple relational calculus*, which we described in Section 6.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is derived from Structured Query Language. Originally, SQL was called SEQUEL (for *S*tructured *E*nglish *QUE*ry *L*anguage) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by ANSI (the American National Standards Institute) and ISO (the International Standards Organization) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL2 (also referred to as SQL-92) was subsequently developed. The next version of the standard was originally called SQL3, but is now called SQL-99. We will try to cover the latest version of SQL as much as possible.

SQL is a comprehensive database language: It has statements for data definition, query, and update. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java or COBOL or C/C++.[1] We will discuss most of these topics in the following subsections.

Because the specification of the SQL standard is expanding, with more features in each version of the standard, the latest **SQL-99** standard is divided into a **core** specification plus optional specialized **packages.** The core is supposed to be implemented by all RDBMS vendors that are SQL-99 compliant. The packages can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, on-line analytical processing (OLAP), multimedia data, and so on. We give a summary of some of these packages—and where they are discussed in the book—at the end of this chapter.

Because SQL is very important (and quite large) we devote two chapters to its basic features. In this chapter, Section 8.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 8.2 presents how basic constraints such as key and referential integrity are specified. Section 8.3 discusses statements for modifying schemas, tables, and constraints. Section 8.4 describes the basic SQL constructs for specifying retrieval queries, and Section 8.5 goes over more complex features of SQL queries, such as aggregate functions and grouping. Section 8.6 describes the SQL commands for insertion, deletion, and updating of data.

1. Originally, SQL had statements for creating and dropping indexes on the files that represent relations, but these have been dropped from the SQL standard for some time.

Section 8.7 lists some SQL features that are presented in other chapters of the book; these include transaction control in Chapter 17, security/authorization in Chapter 23, active databases (triggers) in Chapter 24, object-oriented features in Chapter 22, and OLAP features in Chapter 27. Section 8.8 summarizes the chapter.

In the next chapter, we discuss the concept of views (virtual tables), and then describe how more general constraints may be specified as assertions or checks. This is followed by a description of the various database programming techniques for programming with SQL.

For the reader who desires a less comprehensive introduction to SQL, parts of Section 8.5 may be skipped.

8.1 SQL Data Definition and Data Types

SQL uses the terms **table, row,** and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers). Before we describe the relevant CREATE statements, we discuss schema and catalog concepts in Section 8.1.1 to place our discussion in perspective. Section 8.1.2 describes how tables are created, and Section 8.1.3 describes the most important data types available for attribute specification. Because the SQL specification is very large, we give a description of the most important features. Further details can be found in the various SQL standards documents (see bibliographic notes).

8.1.1 Schema and Catalog Concepts in SQL

Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema. The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application. An **SQL schema** is identified by a **schema name,** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier JSMITH:

```
CREATE SCHEMA COMPANY AUTHORIZATION JSMITH;
```

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

In addition to the concept of a schema, SQL2 uses the concept of a **catalog—**a named collection of schemas in an SQL environment. An SQL **environment** is basically an installation of an SQL-compliant RDBMS on a computer system.[2] A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

8.1.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command (see Section 8.3). Figure 8.1 shows sample data definition statements in SQL for the relational database schema shown in Figure 5.7.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

```
CREATE TABLE COMPANY.EMPLOYEE . . .
```

rather than

```
CREATE TABLE EMPLOYEE . . .
```

as in Figure 8.1, we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.

The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from **virtual relations,** created through the CREATE VIEW statement (see Section 9.2), which may or may not correspond to an actual physical file. In SQL the attributes in a base table are considered to be *ordered in the sequence in which they are specified* in the CREATE TABLE statement. However, rows (tuples) are not considered to be ordered within a relation.

2. SQL also includes the concept of a *cluster* of catalogs within an environment, but it is not very clear if so many levels of nesting are required in most applications.

(a)

```
CREATE TABLE EMPLOYEE
        ( FNAME             VARCHAR(15)        NOT NULL ,
          MINIT             CHAR ,
          LNAME             VARCHAR(15)        NOT NULL ,
          SSN               CHAR(9)            NOT NULL ,
          BDATE             DATE
          ADDRESS           VARCHAR(30) ,
          SEX               CHAR ,
          SALARY            DECIMAL(10,2) ,
          SUPERSSN          CHAR(9) ,
          DNO               INT                NOT NULL ,
    PRIMARY KEY (SSN) ,
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN) ,
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE DEPARTMENT
        ( DNAME             VARCHAR(15)        NOT NULL ,
          DNUMBER           INT                NOT NULL ,
          MGRSSN            CHAR(9)            NOT NULL ,
          MGRSTARTDATE      DATE ,
        PRIMARY KEY (DNUMBER) ,
        UNIQUE (DNAME) ,
        FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) ) ;
CREATE TABLE DEPT_LOCATIONS
        ( DNUMBER           INT                NOT NULL ,
          DLOCATION         VARCHAR(15)        NOT NULL ,
        PRIMARY KEY (DNUMBER, DLOCATION) ,
        FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE PROJECT
        ( PNAME             VARCHAR(15)        NOT NULL ,
          PNUMBER           INT                NOT NULL ,
          PLOCATION         VARCHAR(15) ,
          DNUM              INT                NOT NULL ,
        PRIMARY KEY (PNUMBER) ,
        UNIQUE (PNAME) ,
        FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE WORKS_ON
        ( ESSN              CHAR(9)            NOT NULL ,
          PNO               INT                NOT NULL ,
          HOURS             DECIMAL(3,1)       NOT NULL ,
        PRIMARY KEY (ESSN, PNO) ,
        FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ,
        FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) ) ;
CREATE TABLE DEPENDENT
        ( ESSN              CHAR(9)            NOT NULL ,
          DEPENDENT_NAME    VARCHAR(15)        NOT NULL ,
          SEX               CHAR ,
          BDATE             DATE ,
          RELATIONSHIP      VARCHAR(8) ,
                PRIMARY KEY (ESSN, DEPENDENT_NAME) ,
                FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ) ;
```

FIGURE 8.1 SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7

8.1.3 Attribute Data Types and Domains in SQL

The basic **data types** available for attributes include numeric, character string, bit string, boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(*i*,*j*)—or DEC(*i*,*j*) or NUMERIC(*i*,*j*)—where *i*, the *precision*, is the total number of decimal digits and *j*, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.
- **Character-string** data types are either fixed length—CHAR(*n*) or CHARACTER(*n*), where *n* is the number of characters—or varying length—VARCHAR(*n*) or CHAR VARYING(*n*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase).[3] For fixed-length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed. Padded blanks are generally ignored when strings are compared. For comparison purposes, strings are considered ordered in alphabetic (or lexicographic) order; if a string *str1* appears before another string *str2* in alphabetic order, then *str1* is considered to be less than *str2*.[4] There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'.
- **Bit-string** data types are either of fixed length *n*—BIT(*n*)—or varying length—BIT VARYING(*n*), where *n* is the maximum number of bits. The default for *n*, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'.[5]
- A **boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a boolean data type is UNKNOWN. We discuss the need for UNKNOWN and the three-valued logic in Section 8.5.1.
- New data types for **date** and **time** were added in SQL2. The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by

3. This is not the case with SQL keywords, such as CREATE or CHAR. With keywords, SQL is *case insensitive*, meaning that SQL treats uppercase and lowercase letters as equivalent in keywords.

4. For nonalphabetic characters, there is a defined order.

5. Bit strings whose length is a multiple of 4 can also be specified in *hexadecimal* notation, where the literal string is preceded by X and each hexadecimal character represents 4 bits.

the SQL implementation. The < (less than) comparison can be used with dates or times—an *earlier* date is considered to be smaller than a later date, and similarly with time. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2002-09-27' or TIME '09:12:47'. In addition, a data type TIME(*i*), where *i* is called *time fractional seconds precision*, specifies *i* + 1 additional positions for TIME—one position for an additional separator character, and *i* positions for specifying decimal fractions of a second. A TIME WITH TIME ZONE data type includes an additional six positions for specifying the *displacement* from the standard universal time zone, which is in the range +13:00 to –12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

- A **timestamp** data type (TIMESTAMP) includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single-quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2002-09-27 09:12:47 648302'.
- Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.
- The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string. Hence, they can generally be used in string comparisons by being **cast** (or **coerced** or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, as in Figure 8.1; alternatively, a domain can be declared, and the domain name used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

```
CREATE DOMAIN SSN_TYPE AS CHAR(9);
```

We can use SSN_TYPE in place of CHAR(9) in Figure 8.1 for the attributes SSN and SUPERSSN of EMPLOYEE, MGRSSN of DEPARTMENT, ESSN of WORKS_ON, and ESSN of DEPENDENT. A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes.

8.2 Specifying Basic Constraints in SQL

We now describe the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, as well as restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation. We discuss the specification of more general constraints, called assertions, in Secion 9.1.

8.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the *primary key* of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 8.1.

It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. Figure 8.2 illustrates an example of specifying a default manager for a new department and a default department for a new employee. If no default clause is specified, the default *default value* is NULL for attributes *that do not have* the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.[6] For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of DNUMBER in the DEPARTMENT table (see Figure 8.1) to the following:

```
DNUMBER INT NOT NULL CHECK (DNUMBER > 0 AND DNUMBER < 21);
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER CHECK
(D_NUM > 0 AND D_NUM < 21);
```

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers in Figure 8.1, such as DNUMBER of DEPARTMENT, DNUM of PROJECT, DNO of EMPLOYEE, and so on.

8.2.2 Specifying Key and Referential Integrity Constraints

Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them. Some examples to illustrate the specification of keys and referential integrity are shown in Figure 8.1.[7] The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a *single* attribute, the clause can follow the attribute directly. For example,

6. The CHECK clause can also be used for other purposes, as we shall see.

7. Key and referential integrity constraints were not included in early versions of SQL. In some earlier implementations, keys were specified implicitly at the internal level via the CREATE INDEX command.

```
CREATE TABLE EMPLOYEE
        ( . . . ,
          DNO              INT    NOT NULL    DEFAULT 1,
        CONSTRAINT EMPPK
          PRIMARY KEY (SSN) ,
        CONSTRAINT EMPSUPERFK
          FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
                        ON DELETE SET NULL    ON UPDATE CASCADE ,
        CONSTRAINT EMPDEPTFK
          FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
                        ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPARTMENT
        ( . . . ,
          MGRSSN    CHAR(9)  NOT NULL DEFAULT '888665555' ,
          . . . ,
          CONSTRAINT DEPTPK
            PRIMARY KEY (DNUMBER) ,
          CONSTRAINT DEPTSK
            UNIQUE (DNAME),
          CONSTRAINT DEPTMGRFK
            FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
                  ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPT_LOCATIONS
        ( . . . ,
          PRIMARY KEY (DNUMBER, DLOCATION),
          FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
            ON DELETE CASCADE    ON UPDATE CASCADE ) ;
```

FIGURE 8.2 Example illustrating how default attribute values and referential triggerred actions are specified in SQL

the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure 8.1):

```
DNUMBER INT PRIMARY KEY;
```

The UNIQUE clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 8.1.

Referential integrity is specified via the FOREIGN KEY clause, as shown in Figure 8.1. As we discussed in Section 5.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation. However, the schema designer can specify an alternative action to be taken if a referential integrity constraint is violated, by attaching a **referential triggered action** clause to any foreign key constraint. The options include

SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the examples shown in Figure 8.2. Here, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE. This means that if the tuple for a supervising employee is *deleted*, the value of SUPERSSN is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to SUPERSSN for all employee tuples referencing the updated employee tuple.

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE or ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL, and to the specified default value for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the foreign key to the updated (new) primary key value for all referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for "relationship" relations (see Section 7.1), such as WORKS_ON; for relations that represent multivalued attributes, such as DEPT_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

8.2.3 Giving Names to Constraints

Figure 8.2 also illustrates how a constraint may be given a **constraint name,** following the keyword **CONSTRAINT.** The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Section 8.3. Giving names to constraints is optional.

8.2.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 8.1 had an additional attribute DEPT_CREATE_DATE, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date:

```
CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);
```

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Section 9.1 because it requires the full power of queries, which are discussed in Sections 8.4 and 8.5.

8.3 Schema Change Statements in SQL

In this section, we give an overview of the **schema evolution commands** available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

8.3.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is not needed any more, the DROP SCHEMA command can be used. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed.

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 8.1, we can get rid of the DEPENDENT relation by issuing the following command:

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 9.2). With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

8.3.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command

```
ALTER TABLE COMPANY.EMPLOYEE ADD JOB VARCHAR(12);
```

We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command (see Section 8.6). If no default clause is specified, the new attribute will have

NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other elements) reference the column. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

```
ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS CASCADE;
```

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

```
ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN DROP
DEFAULT;
ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT
"333445555";
```

One can also change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 8.2 from the EMPLOYEE relation, we write:

```
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the **ADD** keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

The preceding subsections gave an overview of the schema evolution commands of SQL. There are many other details and options, and we refer the interested reader to the SQL documents listed in the bibliographical notes. The next two sections discuss the querying capabilities of SQL.

8.4 Basic Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. The SELECT statement *has no relationship* to the SELECT operation of relational algebra, which was discussed in Chapter 6. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use example queries specified on the schema of Figure 5.5 and will refer to the sample database state shown in Figure 5.6 to show the results of some of the example queries.

Before proceeding, we must point out an important distinction between SQL and the formal relational model discussed in Chapter 5: SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an **SQL** table is not a *set of tuples*, because a set does not allow two identical members; rather, it is a **multiset** (sometimes called a *bag*) of tuples. Some SQL relations are *constrained to be sets* because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement (described later in this section). We should be aware of this distinction as we discuss the examples.

8.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner. The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block,** is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT   <attribute list>
FROM     <table list>
WHERE    <condition>;
```

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively, and to the C/C++ programming language operators =, <, <=, >, >=, and !=. The main difference is the *not equal* operator. SQL has many additional comparison operators that we shall present gradually as needed.

We now illustrate the basic SELECT statement in SQL with some example queries. The queries are labeled here with the same query numbers that appear in Chapter 6 for easy cross reference.

QUERY 0

Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'.

```
Q0: SELECT BDATE, ADDRESS
    FROM   EMPLOYEE
    WHERE  FNAME='John' AND MINIT='B' AND LNAME='Smith';
```

This query involves only the EMPLOYEE relation listed in the FROM clause. The query *selects* the EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the BDATE and ADDRESS attributes listed in the SELECT clause. Q0 is similar to the following relational algebra expression, except that duplicates, if any, would *not* be eliminated:

$$\pi_{\text{BDATE,ADDRESS}}(\sigma_{\text{FNAME='JOHN' AND MINIT='B' AND LNAME='SMITH'}}(\text{EMPLOYEE}))$$

Hence, a simple SQL query with a single relation name in the FROM clause is similar to a SELECT-PROJECT pair of relational algebra operations. The SELECT clause of SQL specifies the *projection attributes*, and the WHERE clause specifies the *selection condition*. The only difference is that in the SQL query we may get duplicate tuples in the result, because the constraint that a relation is a set is not enforced. Figure 8.3a shows the result of query Q0 on the database of Figure 5.6.

The query Q0 is also similar to the following tuple relational calculus expression, except that duplicates, if any, would again *not* be eliminated in the SQL query:

Q0: {t.BDATE, t.ADDRESS | EMPLOYEE(t) **and** t.FNAME='John' **and** t.MINIT='B' **and** t.LNAME='Smith'}

Hence, we can think of an implicit tuple variable in the SQL query ranging over each tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is, those tuples for which the condition evaluates to TRUE after substituting their corresponding attribute values—are selected.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT  FNAME, LNAME, ADDRESS
    FROM    EMPLOYEE, DEPARTMENT
    WHERE   DNAME='Research' AND DNUMBER=DNO;
```

Query Q1 is similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations. Such queries are often called **select-project-join queries.** In the WHERE clause of Q1, the condition DNAME = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra. The condition DNUMBER = DNO is a **join condition,** which corresponds to a JOIN condition in the relational algebra. The result of query Q1 is shown in Figure 8.3b. In general, any number of select and join conditions may be specified in a single SQL query. The next example is a select-project-join query with *two* join conditions.

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
Q2: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM    PROJECT, DEPARTMENT, EMPLOYEE
```

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

(b)

FNAME	LNAME	ADDRESS
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

PNUMBER	DNUM	LNAME	ADDRESS	BDATE
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(d)

E.FNAME	E.LNAME	S.FNAME	S.LNAME
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

SSN
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

SSN	DNAME
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(g)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

FIGURE 8.3 Results of SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C

```
    WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND
            PLOCATION='Stafford';
```

The join condition DNUM = DNUMBER relates a project to its controlling department, whereas the join condition MGRSSN = SSN relates the controlling department to the employee who manages that department. The result of query Q2 is shown in Figure 8.3c.

8.4.2 Ambiguous Attribute Names, Aliasing, and Tuple Variables

In SQL the same name can be used for two (or more) attributes as long as the attributes are in *different relations*. If this is the case, and a query refers to two or more attributes with the same name, we must **qualify** the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 5.5 and 5.6 the DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME, and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes NAME and DNUMBER in Q1A to specify which ones we are referring to, because the attribute names are used in both relations:

```
Q1A: SELECT  FNAME, EMPLOYEE.NAME, ADDRESS
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DEPARTMENT.NAME='Research' AND
             DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.

QUERY 8

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:  SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
     FROM    EMPLOYEE AS E, EMPLOYEE AS S
     WHERE   E.SUPERSSN=S.SSN;
```

In this case, we are allowed to declare alternative relation names E and S, called **aliases** or **tuple variables,** for the EMPLOYEE relation. An alias can follow the keyword **AS,** as shown in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8. It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write

```
EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)
```

in the FROM clause, FN becomes an alias for FNAME, MI for MINIT, LN for LNAME, and so on.

In Q8, we can think of E and S as two *different copies* of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees; the second, S, represents employees in the role of supervisors. We can now join the two copies. Of course, in reality there is *only one* EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E.SUPERSSN = S.SSN. Notice that this is an example of a one-level recursive query, as we discussed in Section 6.4.2. In earlier versions of SQL, as in relational algebra, it was not possible to specify a general recursive query, with

an unknown number of levels, in a single SQL statement. A construct for specifying recursive queries has been incorporated into SQL-99, as described in Chapter 22.

The result of query Q8 is shown in Figure 8.3d. Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query. Notice that, if we want to, we can use this alias-naming mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once. In fact, this practice is recommended since it results in queries that are easier to comprehend. For example, we could specify query Q1A as in Q1B:

```
Q1B: SELECT  E.FNAME, E.NAME, E.ADDRESS
     FROM    EMPLOYEE E, DEPARTMENT D
     WHERE   D.NAME='Research' AND D.DNUMBER=E.DNUMBER;
```

If we specify tuple variables for every table in the WHERE clause, a select-project-join query in SQL closely resembles the corresponding tuple relational calculus expression (except for duplicate elimination). For example, compare Q1B with the following tuple relational calculus expression:

```
Q1:  {e.FNAME, e.LNAME, e.ADDRESS | EMPLOYEE(e) and (∃d)
     (DEPARTMENT(d) and d.DNAME='Research' and d.DNUMBER=e.DNO) }
```

Notice that the main difference—other than syntax—is that in the SQL query, the existential quantifier is not specified explicitly.

8.4.3 Unspecified WHERE Clause and Use of the Asterisk

We discuss two more features of SQL here. A *missing* WHERE clause indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected. For example, Query 9 selects all EMPLOYEE SSNs (Figure 8.3e), and Query 10 selects all combinations of an EMPLOYEE SSN and a DEPARTMENT DNAME (Figure 8.3f).

QUERIES 9 AND 10

Select all EMPLOYEE SSNs (Q9), and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME (Q10) in the database.

```
Q9:  SELECT  SSN
     FROM    EMPLOYEE;

Q10: SELECT  SSN, DNAME
     FROM    EMPLOYEE, DEPARTMENT;
```

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra. If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the CROSS PRODUCT (except for duplicate elimination, if any).

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes*. For example, query Q1C retrieves all the attribute values of EMPLOYEE who work in DEPARTMENT number 5 (Figure 8.3g), query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q1C:    SELECT  *
        FROM    EMPLOYEE
        WHERE   DNO=5;

Q1D:    SELECT  *
        FROM    EMPLOYEE, DEPARTMENT
        WHERE   DNAME='Research' AND DNO=DNUMBER;

Q10A:   SELECT  *
        FROM    EMPLOYEE, DEPARTMENT;
```

8.4.4 Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a **multiset;** *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function (see Section 8.5.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.[8] If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT—as in our previous examples—is equivalent to SELECT ALL. For

8. In general, an SQL table is not required to have a key, although in most cases there will be one.

example, Query 11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 8.4a. If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword **DISTINCT** as in Q11A, we accomplish this, as shown in Figure 8.4b.

QUERY 11

Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:   SELECT ALL SALARY
       FROM   EMPLOYEE;

Q11A:  SELECT DISTINCT SALARY
       FROM   EMPLOYEE;
```

SQL has directly incorporated some of the set operations of relational algebra. There are set union (**UNION**), set difference (**EXCEPT**), and set intersection (**INTERSECT**) operations. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result*. Because these set operations apply only to *union-compatible relations*, we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

QUERY 4

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4:    (SELECT DISTINCT PNUMBER
       FROM   PROJECT, DEPARTMENT, EMPLOYEE
```

(a)

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARY
30000
40000
25000
43000
38000
55000

(c)

FNAME	LNAME

(d)

FNAME	LNAME
James	Borg

FIGURE 8.4 Results of additional SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

```
WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND LNAME='Smith')
UNION
(SELECT DISTINCT PNUMBER
FROM    PROJECT, WORKS_ON, EMPLOYEE
WHERE   PNUMBER=PNO AND ESSN=SSN AND LNAME='Smith');
```

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result.

SQL also has corresponding multiset operations, which are followed by the keyword **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated). The behavior of these operations is illustrated by the examples in Figure 8.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

8.4.5 Substring Pattern Matching and Arithmetic Operators

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This

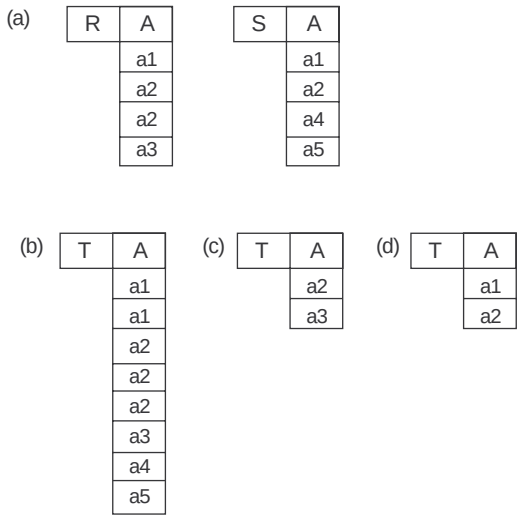

FIGURE 8.5 The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

can be used for string **pattern matching.** Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

QUERY 12

Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ADDRESS LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1950s, we can use Query 12A. Here, '5' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a placeholder for an arbitrary character.

QUERY 12A

Find all employees who were born during the 1950s.

```
Q12A: SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   BDATE LIKE '_ _ 5 _ _ _ _ _ _ _';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character,* which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF' ESCAPE '\' represents the literal string 'AB_CD%EF', because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (") if they are to be included in a string, because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as two consecutive apostrophes (") so that it will not be interpreted as ending the string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

QUERY 13

Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
Q13:  SELECT  FNAME, LNAME, 1.1*SALARY AS INCREASED_SAL
      FROM    EMPLOYEE, WORKS_ON, PROJECT
```

```
WHERE   SSN=ESSN AND PNO=PNUMBER AND
        PNAME='ProductX';
```

For string data types, the concatenate operator || can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (–) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator that can be used for convenience is **BETWEEN,** which is illustrated in Query 14.

QUERY 14

Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT  *
      FROM    EMPLOYEE
      WHERE   (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
```

The condition (SALARY **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((SALARY >= 30000) **AND** (SALARY <= 40000)).

8.4.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the **ORDER BY** clause. This is illustrated by Query 15.

QUERY 15

Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

```
Q15:  SELECT    DNAME, LNAME, FNAME, PNAME
      FROM      DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
      WHERE     DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
      ORDER BY  DNAME, LNAME, FNAME;
```

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly. For example, if we want descending order on DNAME and ascending order on LNAME, FNAME, the ORDER BY clause of Q15 can be written as

```
ORDER BY DNAME DESC, LNAME ASC, FNAME ASC
```

8.5 More Complex SQL Queries

In the previous section, we described some basic types of queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex queries. We discuss several of these features in this section.

8.5.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 5.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value unknown (exists but is not known), value not available (exists but is purposely withheld), or attribute not applicable (undefined for this tuple). Consider the following examples to illustrate each of the three meanings of NULL.

1. *Unknown value:* A particular person has a date of birth but it is not known, so it is represented by NULL in the database.
2. *Unavailable or withheld value:* A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. *Not applicable attribute:* An attribute LastCollegeDegree would be NULL for a person who has no college degrees, because it does not apply to that person.

It is often not possible to determine which of the three meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL.

In general, each NULL is considered to be different from every other NULL in the database. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued logic with values TRUE or FALSE. It is therefore necessary to define the results of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 8.1 shows the resulting values.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. However, there are exceptions to that rule for certain operations, such as outer joins, as we shall see.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses **IS** or **IS NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 8.5.6). Query 18 illustrates this; its result is shown in Figure 8.4d.

TABLE 8.1 LOGICAL CONNECTIVES IN THREE-VALUED LOGIC

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

QUERY 18

Retrieve the names of all employees who do not have supervisors.

```
Q18: SELECT FNAME, LNAME
     FROM   EMPLOYEE
     WHERE  SUPERSSN IS NULL;
```

8.5.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries,** which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query.** Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator **IN**, which compares a value v with a set (or multiset) of values V and evaluates to **TRUE** if v is one of the elements in V.

```
Q4A: SELECT DISTINCT PNUMBER
     FROM   PROJECT
     WHERE  PNUMBER IN (SELECT PNUMBER
                        FROM   PROJECT, DEPARTMENT,
                               EMPLOYEE
                        WHERE  DNUM=DNUMBER AND
```

```
                                    MGRSSN=SSN AND
                                    LNAME='Smith')
OR
PNUMBER IN  (SELECT   PNO
             FROM     WORKS_ON, EMPLOYEE
             WHERE    ESSN=SSN AND
                      LNAME='Smith');
```

The first nested query selects the project numbers of projects that have a 'Smith' involved as manager, while the second selects the project numbers of projects that have a 'Smith' involved as worker. In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT  DISTINCT ESSN
FROM    WORKS_ON
WHERE   (PNO, HOURS) IN (SELECT PNO, HOURS FROM WORKS_ON
                         WHERE SSN='123456789');
```

This query will select the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (PNO, HOURS) for each tuple in WORKS_ON with the set of union-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The keywords ANY and SOME have the same meaning. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > ALL (SELECT SALARY FROM EMPLOYEE
                      WHERE DNO=5);
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query*, and another in a relation in the FROM clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query.** For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we can specify and refer to an *alias* (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16, whose result is shown in Figure 8.4c.

QUERY 16

Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

```
Q16: SELECT  E.FNAME, E.LNAME
     FROM    EMPLOYEE AS E
     WHERE   E.SSN IN  (SELECT  ESSN
                        FROM    DEPENDENT
                        WHERE   E.FNAME=DEPENDENT_NAME
                         AND E.SEX=SEX);
```

In the nested query of Q16, we must qualify E.SEX because it refers to the SEX attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called SEX. All unqualified references to SEX in the nested query refer to SEX of DEPENDENT. However, we do not *have to* qualify FNAME and SSN because the DEPENDENT relation does not have attributes called FNAME and SSN, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities.

8.5.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated.** We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query*. For example, we can think of Q16 as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the ESSN values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E, DEPENDENT AS D
      WHERE   E.SSN=D.ESSN AND E.SEX=D.SEX AND
              E.FNAME=D.DEPENDENT_NAME;
```

The original SQL implementation on SYSTEM R also had a **CONTAINS** comparison operator, which was used to compare two sets or multisets. This operator was subsequently dropped from the language, possibly because of the difficulty of implementing it efficiently. Most commercial implementations of SQL do *not* have this operator. The CONTAINS operator compares two sets of values and returns TRUE if one set contains all values in the other set. Query 3 illustrates the use of the CONTAINS operator.

QUERY 3

Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

```
Q3:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   (       (SELECT  PNO
                      FROM    WORKS_ON
                      WHERE   SSN=ESSN)
                      CONTAINS
                      (SELECT  PNUMBER
                      FROM    PROJECT
                      WHERE   DNUM=5) );
```

In Q3, the second nested query (which is not correlated with the outer query) retrieves the project numbers of all projects controlled by department 5. For *each* employee tuple, the first nested query (which is correlated) retrieves the project numbers on which the employee works; if these contain all projects controlled by department 5, the employee tuple is selected and the name of that employee is retrieved. Notice that the CONTAINS comparison operator has a similar function to the DIVISION operation of the relational algebra (see Section 6.3.4) and to universal quantification in relational calculus (see Section 6.6.6). Because the CONTAINS operation is not part of SQL, we have to use other techniques, such as the EXISTS function, to specify these types of queries, as described in Section 8.5.4.

8.5.4 The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We illustrate the use of EXISTS—and NOT

EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS. This is shown as Q16B:

```
Q16B: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   EXISTS  (SELECT *
                       FROM    DEPENDENT
                       WHERE   E.SSN=ESSN AND E.SEX=SEX
                               AND E.FNAME=DEPENDENT_NAME);
```

EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query. In Q16B, the nested query references the SSN, FNAME, and SEX attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same social security number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. In general, EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and it returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and it returns **FALSE** otherwise. Next, we illustrate the use of NOT EXISTS.

QUERY 6

Retrieve the names of employees who have no dependents.

```
Q6:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   NOT EXISTS (SELECT  *
                         FROM    DEPENDENT
                         WHERE   SSN=ESSN);
```

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected. We can explain Q6 as follows: For *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose ESSN value matches the EMPLOYEE SSN; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its FNAME and LNAME.

QUERY 7

List the names of managers who have at least one dependent.

```
Q7:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   EXISTS      (SELECT  *
                          FROM    DEPENDENT
                          WHERE   SSN=ESSN)
```

```
AND
EXISTS      (SELECT   *
             FROM     DEPARTMENT
             WHERE    SSN=MGRSSN);
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

Query 3 ("Retrieve the name of each employee who works on *all* the projects controlled by department number 5," see Section 8.5.3) can be stated using EXISTS and NOT EXISTS in SQL systems. There are two options. The first is to use the well-known set theory transformation that (*S1* CONTAINS *S2*) is logically equivalent to (*S2* EXCEPT *S1*) is empty.[9] This option is shown as Q3A.

```
Q3A:  SELECT   FNAME, LNAME
      FROM     EMPLOYEE
      WHERE    NOT EXISTS
      (        (SELECT   PNUMBER
                FROM     PROJECT
                WHERE    DNUM=5)
      EXCEPT
                (SELECT   PNO
                FROM     WORKS_ON
                WHERE    SSN=ESSN) );
```

In Q3A, the first subquery (which is not correlated) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery MINUS (EXCEPT) the second subquery is empty, it means that the employee works on all the projects and is hence selected.

The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3, which used the CONTAINS comparison operator, and Q3A, which uses NOT EXISTS and EXCEPT. However, CONTAINS is not part of SQL, and not all relational systems have the EXCEPT operator even though it is part of SQL-99.

```
Q3B:  SELECT   LNAME, FNAME
      FROM     EMPLOYEE
```

9. Recall that EXCEPT is the set difference operator.

```
WHERE   NOT EXISTS
(SELECT  *
FROM    WORKS_ON B
WHERE   (B.PNO IN     (SELECT   PNUMBER
                       FROM     PROJECT
                       WHERE    DNUM=5) )
        AND
        NOT EXISTS  (SELECT   *
                     FROM     WORKS_ON C
                     WHERE    C.ESSN=SSN
                     AND      C.PNO=B.PNO) );
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose PNO is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same PNO and the same SSN as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we wrote this query in tuple relation calculus in Section 6.6.6.

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

8.5.5 Explicit Sets and Renaming of Attributes in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

QUERY 17

Retrieve the social security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:  SELECT  DISTINCT ESSN
      FROM    WORKS_ON
      WHERE   PNO IN (1, 2, 3);
```

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier **AS** followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example, Q8A shows how query Q8 can be slightly changed to retrieve the last name of each employee and his or her supervisor, while renaming the resulting

attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME. The new names will appear as column headers in the query result.

```
Q8A: SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
             SUPERVISOR_NAME
     FROM    EMPLOYEE AS E, EMPLOYEE AS S
     WHERE   E.SUPERSSN=S.SSN;
```

8.5.6 Joined Tables in SQL

The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the FROM clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier first to specify the join of the EMPLOYEE and DEPARTMENT relations, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

```
Q1A: SELECT  FNAME, LNAME, ADDRESS
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

The FROM clause in Q1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations *R* and *S*, no join condition is specified; an implicit equijoin condition for *each pair of attributes with the same name* from *R* and *S* is created. Each such pair of attributes is included only once in the resulting relation (see Section 6.4.3).

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as DNAME, DNO (to match the name of the desired join attribute DNO in EMPLOYEE), MSSN, and MSDATE. The implied join condition for this NATURAL JOIN is EMPLOYEE.DNO = DEPT.DNO, because this is the only pair of attributes with the same name after renaming.

```
Q1B: SELECT FNAME, LNAME, ADDRESS
     FROM   (EMPLOYEE NATURAL JOIN
             (DEPARTMENT AS DEPT (DNAME, DNO, MSSN, MSDATE)))
     WHERE  DNAME='Research;
```

The default type of join in a joined table is an **inner join,** where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query

Q8A, only employees that *have a supervisor* are included in the result; an EMPLOYEE tuple whose value for SUPERSSN is NULL is excluded. If the user requires that all employees be included, an OUTER JOIN must be used explicitly (see Section 6.4.3 for the definition of OUTER JOIN). In SQL, this is handled by explicitly specifying the OUTER JOIN in a joined table, as illustrated in Q8B:

```
Q8B: SELECT  E.LNAME AS EMPLOYEE_NAME,
             S.LNAME AS SUPERVISOR_NAME
     FROM    (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
             ON E.SUPERSSN=S.SSN);
```

The options available for specifying joined tables in SQL include INNER JOIN (same as JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one may also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the Cartesian product operation (see Section 6.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This is illustrated by Q2A, which is a different way of specifying query Q2, using the concept of a joined table:

```
Q2A: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
     FROM    ((PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)
             JOIN EMPLOYEE ON MGRSSN=SSN)
     WHERE   PLOCATION='Stafford';
```

8.5.7 Aggregate Functions in SQL

In Section 6.4.1, we introduced the concept of an aggregate function as a relational operation. Because grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts. A number of built-in functions exist: **COUNT, SUM, MAX, MIN,** and **AVG.**[10] The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[11] We illustrate the use of these functions with example queries.

10. Additional aggregate functions for more advanced statistical calculation have been added in SQL-99.

11. Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME, and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.

QUERY 19

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    EMPLOYEE;
```

If we want to get the preceding function values for employees of a specific department—say, the 'Research' department—we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

QUERY 20

Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

QUERIES 21 AND 22

Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21: SELECT  COUNT (*)
     FROM    EMPLOYEE;

Q22: SELECT  COUNT (*)
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DNO=DNUMBER AND DNAME='Research';
```

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

QUERY 23

Count the number of distinct salary values in the database.

```
Q23: SELECT  COUNT (DISTINCT SALARY)
     FROM    EMPLOYEE;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute).

The preceding examples summarize *a whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce single tuples or single values. They illustrate how functions are applied to retrieve a summary value or summary tuple from the database. These functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write the following:

```
Q5:  SELECT  LNAME, FNAME
     FROM    EMPLOYEE
     WHERE   (SELECT  COUNT (*)
              FROM    DEPENDENT
              WHERE   SSN=ESSN)  >=  2;
```

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

8.5.8 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation*, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees *in each department* or the number of employees who work *on each project*. In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s).** We can then apply the function to each such group independently. SQL has a **GROUP BY** clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should *also appear in the* SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

QUERY 24

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24: SELECT    DNO, COUNT (*), AVG (SALARY)
     FROM      EMPLOYEE
     GROUP BY  DNO;
```

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each

such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples. Figure 8.6a illustrates how grouping works on Q24; it also shows the result of Q24.

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute*. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute DNO, there would be a separate group for those tuples in the result of Q24.

QUERY 25

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME;
```

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations. Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions*. For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

QUERY 26

For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME
     HAVING    COUNT (*) > 2;
```

Notice that, while selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*. Figure 8.6b illustrates the use of HAVING and displays the result of Q26.

(a)

FNAME	MINIT	LNAME	SSN	• • •	SALARY	SUPERSSN	DNO
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	• • •	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	null	1

Grouping EMPLOYEE tuples by the value of DNO.

DNO	COUNT (*)	AVG (SALARY)
5	4	33250
4	3	31000
1	1	55000

Result of Q24.

(b)

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	• • •	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING.

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10	• • •	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition.

PNAME	COUNT (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(PNUMBER not shown).

FIGURE 8.6 Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

QUERY 27

For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON, EMPLOYEE
     WHERE     PNUMBER=PNO AND SSN=ESSN AND DNO=5
     GROUP BY  PNUMBER, PNAME;
```

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

```
SELECT     DNAME, COUNT (*)
FROM       DEPARTMENT, EMPLOYEE
WHERE      DNUMBER=DNO AND SALARY>40000
GROUP BY   DNAME
HAVING     COUNT (*) > 5;
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000*. The rule is that the WHERE clause is executed first, to select individual tuples; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000, *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

QUERY 28

For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28: SELECT    DNUMBER, COUNT (*)
     FROM      DEPARTMENT, EMPLOYEE
     WHERE     DNUMBER=DNO AND SALARY>40000 AND
               DNO IN    (SELECT    DNO
                          FROM      EMPLOYEE
                          GROUP BY  DNO
                          HAVING    COUNT (*) > 5)
     GROUP BY  DNUMBER;
```

8.5.9 Discussion and Summary of SQL Queries

A query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets [. . .] being optional:

SELECT <ATTRIBUTE AND FUNCTION LIST>

FROM <TABLE LIST>

[**WHERE** <CONDITION>]

[**GROUP BY** <GROUPING ATTRIBUTE(S)>]

[**HAVING** <GROUP CONDITION>]

[**ORDER BY** <ATTRIBUTE LIST>];

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selection of tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

A query is evaluated *conceptually*[12] by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, and ORDER BY) are specified, we can *think conceptually* of a query as being executed as follows: For *each combination of tuples*—one from each of the relations specified in the FROM clause—evaluate the WHERE clause; if it evaluates to TRUE, place the values of the attributes specified in the SELECT clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient. We discuss query processing and optimization in Chapters 15 and 16.

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas others may be more comfortable with another. From the programmer's and the

12. The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.

system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient. Ideally, the user should worry only about specifying the query correctly. It is the responsibility of the DBMS to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others.

8.6 Insert, Delete, and Update Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

8.6.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure 5.5 and specified in the CREATE TABLE EMPLOYEE . . . command in Figure 8.1, we can use U1:

```
U1:  INSERT INTO  EMPLOYEE
     VALUES       ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                  Oak Forest,Katy,TX', 'M', 37000, '987654321', 4);
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification *and* no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be *left out*. For example, to enter a tuple for a new EMPLOYEE for whom we know only the FNAME, LNAME, DNO, and SSN attributes, we can use U1A:

```
U1A: INSERT INTO  EMPLOYEE (FNAME, LNAME, DNO, SSN)
     VALUES       ('Richard', 'Marini', 4, '653298653');
```

Attributes not specified in U1A are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the* INSERT command itself. It is also possible to insert into a relation *multiple tuples* separated by commas in a single INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully implements SQL-99 should support and enforce all the integrity constraints that can be specified in the DDL. However, some DBMSs do not incorporate all the constraints, in order to maintain the efficiency of the DBMS and because of the complexity of enforcing all constraints. If a system does not support some constraint—say, referential integrity—the users or programmers must enforce the constraint. For example, if we issue the command in U2 on the database shown in Figure 5.6, a DBMS not supporting referential integrity will do the insertion even though no DEPARTMENT tuple exists in the database with DNUMBER = 2. It is the responsibility of the user to check that any such constraints *whose checks are not implemented by the DBMS* are not violated. However, the DBMS must implement checks to enforce all the SQL integrity constraints *it supports*. A DBMS enforcing NOT NULL will reject an INSERT command in which an attribute declared to be NOT NULL does not have a value; for example, U2A would be *rejected* because no SSN value is provided.

```
U2:  INSERT INTO   EMPLOYEE (FNAME, LNAME, SSN, DNO)
     VALUES        ('Robert', 'Hatcher', '980760540', 2);
     (* U2 is rejected if referential integrity checking is provided by dbms *)

U2A: INSERT INTO   EMPLOYEE (FNAME, LNAME, DNO)
     VALUES        ('Robert', 'Hatcher', 5);
     (* U2A is rejected if not null checking is provided by dbms *)
```

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the name, number of employees, and total salaries for each department, we can write the statements in U3A and U3B:

```
U3A: CREATE TABLE  DEPTS_INFO
     (DEPT_NAME    VARCHAR(15),
     NO_OF_EMPS    INTEGER,
     TOTAL_SAL     INTEGER);
U3B: INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                   TOTAL_SAL)
     SELECT        DNAME, COUNT (*), SUM (SALARY)
     FROM          (DEPARTMENT JOIN EMPLOYEE ON
                   DNUMBER=DNO)
     GROUP BY      DNAME;
```

A table DEPTS_INFO is created by U3A and is loaded with the summary information retrieved from the database by the query in U3B. We can now query DEPTS_INFO as we

would any other relation; when we do not need it any more, we can remove it by using the DROP TABLE command. Notice that the DEPTS_INFO table may not be up to date; that is, if we update either the DEPARTMENT or the EMPLOYEE relations after issuing U3B, the information in DEPTS_INFO *becomes outdated*. We have to create a view (see Section 9.2) to keep such a table up to date.

8.6.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL (see Section 8.2.2).[13] Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.[14] The DELETE commands in U4A to U4D, if applied independently to the database of Figure 5.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A: DELETE FROM  EMPLOYEE
     WHERE        LNAME='Brown';

U4B: DELETE FROM  EMPLOYEE
     WHERE        SSN='123456789';

U4C: DELETE FROM  EMPLOYEE
     WHERE        DNO IN (SELECT   DNUMBER
                          FROM     DEPARTMENT
                          WHERE    DNAME='Research');

U4D: DELETE FROM  EMPLOYEE;
```

8.6.3 The UPDATE Command

The **UPDATE** command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints of the DDL (see Section 8.2.2). An additional **SET** clause in the UPDATE command specifies the attributes to be modified and

13. Other actions can be automatically applied through triggers (see Section 24.1) and other mechanisms.
14. We must use the DROP TABLE command to remove the table definition (see Section 8.3.1).

their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

```
U5: UPDATE   PROJECT
    SET      PLOCATION = 'Bellaire', DNUM = 5
    WHERE    PNUMBER=10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown in U6. In this request, the modified SALARY value depends on the original SALARY value in each tuple, so two references to the SALARY attribute are needed. In the SET clause, the reference to the SALARY attribute on the right refers to the old SALARY value *before modification*, and the one on the left refers to the new SALARY value *after modification:*

```
U6: UPDATE   EMPLOYEE
    SET      SALARY = SALARY *1.1
    WHERE    DNO IN (SELECT    DNUMBER
                     FROM      DEPARTMENT
                     WHERE     DNAME='Research');
```

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

8.7 Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter but discuss elsewhere in the book. These are as follows:

- SQL has the capability to specify more general constraints, called assertions, using the CREATE ASSERTION statement. This is described in Section 9.1.
- SQL has language constructs for specifying views, also known as virtual tables, using the CREATE VIEW statement. Views are derived from the base tables declared through the CREATE TABLE statement, and are discussed in Section 9.2.
- SQL has several different techniques for writing programs in various programming languages that can include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Language Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Program Stored Modules). We discuss the differences among these techniques in Section 9.3, then discuss each technique in Sections 9.4 through 9.6. We also discuss how to access SQL databases through the Java programming language using JDBC and SQLJ.
- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition lan-*

guage (*SDL*) in Chapter 2. Earlier versions of SQL had commands for **creating indexes,** but these were removed from the language because they were not at the conceptual schema level (see Chapter 2).

- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes. We discuss these commands in Chapter 17 after we discuss the concept of transactions in more detail.
- SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE**—are discussed in Chapter 23 where we discuss database security and authorization.
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates. We discuss these features in Section 24.1, where we discuss active database concepts.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational.** Capabilities such as creating complex-structured attributes (also called **nested relations**), specifying abstract data types (called **UDT**s or user-defined types) for attributes and tables, creating **object identifiers** for referencing tuples, and specifying **operations** on types are discussed in Chapter 22.
- SQL and relational databases can interact with new technologies such as XML (eXtended Markup Language; see Chapter 26) and OLAP (On Line Analytical Processing for Data Warehouses; see Chapter 27).

8.8 Summary

In this chapter we presented the SQL database language. This language or variations of it have been implemented as interfaces to many commercial relational DBMSs, including Oracle, IBM's DB2 and SQL/DS, Microsoft's SQL Server and ACCESS, INGRES, INFORMIX, and SYBASE. The original version of SQL was implemented in the experimental DBMS called SYSTEM R, which was developed at IBM Research. SQL is designed to be a comprehensive language that includes statements for data definition, queries, updates, view definition, and constraint specification. We discussed many of these in separate sections of this chapter. In the final section we discussed additional features that are described elsewhere in the book. Our emphasis was on the SQL-99 standard.

Table 8.2 summarizes the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive nor to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available

TABLE 8.2 SUMMARY OF SQL SYNTAX

```
CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                          {, <column name> <column type> [<attribute constraint>] }
                          [<table constraint> {,<table constraint>}])

DROP TABLE <table name>

ALTER TABLE <table name> ADD <column name> <column type>

SELECT [DISTINCT] <attribute list>
FROM (<table name> [ <alias>] | <joined table>) {, (<table name> [ <alias>] | <joined table>) }
[WHERE <condition>]
[GROUP BY <grouping attributes> [HAVING <group selection condition> ] ]
[ORDER BY <column name> [<order>] {, <column name> [<order>] } ]

<attribute list>::= (* | ( <column name> | <function>(([DISTINCT]<column name> | *)))
                        {,( <column name> | <function>(([DISTINCT] <column name> | *)) } ) )
<grouping attributes>::= <column name> { , <column name>}
<order>::= (ASC | DESC)

INSERT INTO <table name> [( <column name>{, <column name>} ) ]
(VALUES ( <constant value> , { <constant value>} ){,(<constant value>{,<constant value>})}
 | <select statement>)

DELETE FROM <table name>
[WHERE <selection condition>]

UPDATE <table name>
SET <column name>=<value expression> { , <column name>=<value expression> }
[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

*The last two commands are not part of standard SQL2.

in SQL. We use BNF notation, where nonterminal symbols are shown in angled brackets < . . . >, optional parts are shown in square brackets [. . .], repetitions are shown in braces { . . . }, and alternatives are shown in parentheses (. . . | . . . | . . .).[15]

Review Questions

8.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 5? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

8.2. List the data types that are allowed for SQL attributes.

8.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 5? What about referential triggered actions?

8.4. Describe the six clauses in the syntax of an SQL query, and show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

8.5. Describe conceptually how an SQL query will be executed by specifying the conceptual order of executing each of the six clauses.

8.6. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?

Exercises

8.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. What are the referential integrity constraints that should hold on the schema? Write appropriate SQL DDL statements to define the database.

8.8. Repeat Exercise 8.7, but use the AIRLINE database schema of Figure 5.8.

8.9. Consider the LIBRARY relational database schema of Figure 6.12. Choose the appropriate action (reject, cascade, set to null, set to default) for each referential integrity constraint, both for *deletion* of a referenced tuple, and for *update* of a primary key attribute value in a referenced tuple. Justify your choices.

8.10. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 6.12. Specify appropriate keys and referential triggered actions.

8.11. Write SQL queries for the LIBRARY database queries given in Exercise 6.18.

8.12. How can the key and foreign key constraints be enforced by the DBMS? Is the enforcement technique you suggest difficult to implement? Can the constraint checks be executed efficiently when updates are applied to the database?

8.13. Specify the queries of Exercise 6.16 in SQL. Show the result of each query if it is applied to the COMPANY database of Figure 5.6.

8.14. Specify the following additional queries on the database of Figure 5.5 in SQL. Show the query results if each query is applied to the database of Figure 5.6.

15. The full syntax of SQL-99 is described in many voluminous documents of hundreds of pages.

a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
b. Suppose that we want the number of *male* employees in each department rather than all employees (as in Exercise 8.14a). Can we specify this query in SQL? Why or why not?

8.15. Specify the updates of Exercise 5.10, using the SQL update commands.

8.16. Specify the following queries in SQL on the database schema of Figure 1.2.
a. Retrieve the names of all senior students majoring in 'CS' (computer science).
b. Retrieve the names of all courses taught by Professor King in 1998 and 1999.
c. For each section taught by Professor King, retrieve the course number, semester, year, and number of students who took the section.
d. Retrieve the name and transcript of each senior student (Class = 5) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.
e. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).
f. Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.

8.17. Write SQL update statements to do the following on the database schema shown in Figure 1.2.
a. Insert a new student, <'Johnson', 25, 1, 'MATH'>, in the database.
b. Change the class of student 'Smith' to 2.
c. Insert a new course, <'Knowledge Engineering', 'CS4390', 3, 'CS'>.
d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

8.18. Specify the queries and updates of Exercises 6.17 and 5.11, which refer to the AIRLINE database (see Figure 5.8), in SQL.

8.19. a. Design a relational database schema for your database application.
b. Declare your relations, using the SQL DDL.
c. Specify a number of queries in SQL that are needed by your database application.
d. Based on your expected use of the database, choose some attributes that should have indexes specified on them.
e. Implement your database, if you have a DBMS that supports SQL.

8.20. Specify the answers to Exercises 6.19 through 6.21 and Exercise 6.23 in SQL.

Selected Bibliography

The SQL language, originally named SEQUEL, was based on the language SQUARE (Specifying Queries as Relational Expressions), described by Boyce et al. (1975). The syntax of SQUARE was modified into SEQUEL (Chamberlin and Boyce 1974) and then into SEQUEL 2 (Chamberlin et al. 1976), on which SQL is based. The original implementation of SEQUEL was done at IBM Research, San Jose, California.

Reisner (1977) describes a human factors evaluation of SEQUEL in which she found that users have some difficulty with specifying join conditions and grouping correctly.

Date (1984b) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard, and ANSI (1992) describes the SQL2 standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, Oracle, INGRES, INFORMIX, and other commercial DBMS products. Melton and Simon (1993) is a comprehensive treatment of SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of SQL queries.