is based on how the database will be used. This exercise asks you to think about specific situations where dividing the SSN is useful.

3.19. Consider a STUDENT relation in a UNIVERSITY database with the following attributes (Name, Ssn, Local_phone, Address, Cell_phone, Age, Gpa). Note that the cell phone may be from a different city and state (or province) from the local phone. A possible tuple of the relation is shown below:

Name	Ssn	Local_phone	Address	Cell_phone	Age	Gpa
George Shaw William Edwards	123-45-6789	555-1234	123 Main St., Anytown, CA 94539	555-4321	19	3.75

a. Identify the critical missing information from the Local_phone and Cell_phone attributes. (*Hint*: How do you call someone who lives in a different state or province?)

b. Would you store this additional information in the Local_phone and Cell_phone attributes or add new attributes to the schema for STUDENT?

c. Consider the Name attribute. What are the advantages and disadvantages of splitting this field from one attribute into three attributes (first name, middle name, and last name)?

d. What general guideline would you recommend for deciding when to store information in a single attribute and when to split the information?

e. Suppose the student can have between 0 and 5 phones. Suggest two different designs that allow this type of information.

3.20. Recent changes in privacy laws have disallowed organizations from using Social Security numbers to identify individuals unless certain restrictions are satisfied. As a result, most U.S. universities cannot use SSNs as primary keys (except for financial data). In practice, Student_id, a unique identifier assigned to every student, is likely to be used as the primary key rather than SSN since Student_id can be used throughout the system.

a. Some database designers are reluctant to use generated keys (also known as *surrogate keys*) for primary keys (such as Student_id) because they are artificial. Can you propose any natural choices of keys that can be used to identify the student record in a UNIVERSITY database?

b. Suppose that you are able to guarantee uniqueness of a natural key that includes last name. Are you guaranteed that the last name will not change during the lifetime of the database? If last name can change, what solutions can you propose for creating a primary key that still includes last name but remains unique?

c. What are the advantages and disadvantages of using generated (surrogate) keys?

Selected Bibliography

The relational model was introduced by Codd (1970) in a classic paper. Codd also introduced relational algebra and laid the theoretical foundations for the relational model in a series of papers (Codd 1971, 1972, 1972a, 1974); he was later given the Turing Award, the highest honor of the ACM (Association for Computing Machinery) for his work on the relational model. In a later paper, Codd (1979) discussed extending the relational model to incorporate more meta-data and semantics about the relations; he also proposed a three-valued logic to deal with uncertainty in relations and incorporating NULLs in the relational algebra. The resulting model is known as RM/T. Childs (1968) had earlier used set theory to model databases. Later, Codd (1990) published a book examining over 300 features of the relational data model and database systems. Date (2001) provides a retrospective review and analysis of the relational data model.

Since Codd's pioneering work, much research has been conducted on various aspects of the relational model. Todd (1976) describes an experimental DBMS called PRTV that directly implements the relational algebra operations. Schmidt and Swenson (1975) introduce additional semantics into the relational model by classifying different types of relations. Chen's (1976) Entity-Relationship model, which is discussed in Chapter 7, is a means to communicate the real-world semantics of a relational database at the conceptual level. Wiederhold and Elmasri (1979) introduce various types of connections between relations to enhance its constraints. Extensions of the relational model are discussed in Chapters 11 and 26. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 4 to 6, 9, 11, 13, 15, 16, 24, and 25. Maier (1983) and Atzeni and De Antonellis (1993) provide an extensive theoretical treatment of the relational data model.

This page intentionally left blank

chapter 4

Basic SQL

The SQL language may be considered one of the major reasons for the commercial success of relational databases. 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. This is because even if the users became dissatisfied with the particular relational DBMS product they were using, converting to another relational DBMS product was not expected to be too expensive and time-consuming because both systems followed 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 3 presented the most important concepts underlying the *formal* relational data model. In Chapter 6 (Sections 6.1 through 6.5) we shall discuss 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 Chapter 19. 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 describe in Section 6.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) 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 the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL-92 (also referred to as SQL2) was subsequently developed. The next standard that is well-recognized is SQL:1999, which started out as SQL3. Two later updates to the standard are SQL:2003 and SQL:2006, which added XML features (see Chapter 12) among other updates to the language. Another update in 2008 incorporated more object database features in SQL (see Chapter 11). 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 definitions, queries, and updates. 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, COBOL, or C/C++.[1]

The later SQL standards (starting with **SQL:1999**) are divided into a **core** specification plus specialized **extensions**. The core is supposed to be implemented by all RDBMS vendors that are SQL compliant. The extensions can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, online analytical processing (OLAP), multimedia data, and so on.

Because SQL is very important (and quite large), we devote two chapters to its features. In this chapter, Section 4.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 4.2 presents how basic constraints such as key and referential integrity are specified. Section 4.3 describes the basic SQL constructs for specifying retrieval queries, and Section 4.4 describes the SQL commands for insertion, deletion, and data updates.

In Chapter 5, we will describe more complex SQL retrieval queries, as well as the ALTER commands for changing the schema. We will also describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers, which is presented in

[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.

more detail in Chapter 26 and we will describe the SQL facility for defining views on the database in Chapter 5. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables.

Section 4.5 lists some SQL features that are presented in other chapters of the book; these include transaction control in Chapter 21, security/authorization in Chapter 24, active databases (triggers) in Chapter 26, object-oriented features in Chapter 11, and online analytical processing (OLAP) features in Chapter 29. Section 4.6 summarizes the chapter. Chapters 13 and 14 discuss the various database programming techniques for programming with SQL.

4.1 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 4.1.1 to place our discussion in perspective. Section 4.1.2 describes how tables are created, and Section 4.1.3 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 end-of-chapter bibliographic notes).

4.1.1 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'. Note that each statement in SQL ends with a semicolon.

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, SQL 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.

4.1.2 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 Chapter 5). Figure 4.1 shows sample data definition statements in SQL for the COMPANY relational database schema shown in Figure 3.7.

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 4.1, 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 Chapter 5), 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.

It is important to note that in Figure 4.1, there are some *foreign keys that may cause errors* because they are specified either via circular references or because they refer to a table that has not yet been created. For example, the foreign key Super_ssn in the EMPLOYEE table is a circular reference because it refers to the table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has

[2]SQL also includes the concept of a *cluster* of catalogs within an environment.

```
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),
          Super_ssn       CHAR(9),
          Dno             INT                NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname           VARCHAR(15)        NOT NULL,
          Dnumber         INT                NOT NULL,
          Mgr_ssn         CHAR(9)            NOT NULL,
          Mgr_start_date  DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) 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 4.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

not been created yet. To deal with this type of problem, these constraints can be left out of the initial CREATE TABLE statement, and then added later using the ALTER TABLE statement (see Chapter 5). We displayed all the foreign keys in Figure 4.1 to show the complete COMPANY schema in one place.

4.1.3 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'. Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.
- **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

[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.

them from character strings; for example, B'10101'.[5] Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

- 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 Chapter 5.
- 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 the SQL implementation. This implies that months should be between 1 and 12 and dates must be between 1 and 31; furthermore, a date should be a valid date for the corresponding month. 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 '2008-09-27' 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 period (.) 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.

Some additional data types are discussed below. The list of types discussed here is not exhaustive; different implementations have added more data types to SQL.

- A **timestamp** data type (TIMESTAMP) includes 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 '2008-09-27 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.

---

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

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 4.1; 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 4.1 for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn 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. Notice that domains may not be available in some implementations of SQL.

4.2 Specifying Constraints in SQL

This section describes the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, 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 Chapter 5.

4.2.1 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 4.1.

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 4.2 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 4.1) to the following:

Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

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

```
CREATE TABLE EMPLOYEE
    (   . . . ,
        Dno         INT             NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK
        PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) 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
    (   . . . ,
        Mgr_ssn     CHAR(9)         NOT NULL        DEFAULT '888665555',
        . . . ,
    CONSTRAINT DEPTPK
        PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
        UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_ssn) 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 4.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

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 4.1, such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

4.2.2 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 4.1.[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, the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure 4.1):

```
Dnumber INT PRIMARY KEY;
```

---

[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.

The **UNIQUE** clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 4.1. The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

Dname VARCHAR(15) **UNIQUE**;

Referential integrity is specified via the **FOREIGN KEY** clause, as shown in Figure 4.1. As we discussed in Section 3.2.4, 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, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken 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 4.2. Here, the database designer chooses ON DELETE SET NULL and ON UPDATE CASCADE for the foreign key Super_ssn of EMPLOYEE. This means that if the tuple for a *supervising employee* is *deleted*, the value of Super_ssn 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 Super_ssn for all employee tuples referencing the updated employee tuple.[8]

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute 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 referencing foreign key attribute(s) to the updated (new) primary key value for all the 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 9.1), 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.

4.2.3 Giving Names to Constraints

Figure 4.2 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 con-

---

[8]Notice that the foreign key Super_ssn in the EMPLOYEE table is a circular reference and hence may have to be added later as a named constraint using the ALTER TABLE statement as we discussed at the end of Section 4.1.2.

straint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Chapter 5. Giving names to constraints is optional.

4.2.4 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 4.1 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 <= Mgr_start_date);
```

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Chapter 5 because it requires the full power of queries, which are discussed in Sections 4.3 and 5.1.

4.3 Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. The SELECT statement *is not the same as* the SELECT operation of relational algebra, which we discuss 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 sample queries specified on the schema of Figure 3.5 and will refer to the sample database state shown in Figure 3.6 to show the results of some of the sample queries. In this section, we present the features of SQL for *simple retrieval queries*. Features of SQL for specifying more complex retrieval queries are presented in Section 5.1.

Before proceeding, we must point out an *important distinction* between SQL and the formal relational model discussed in Chapter 3: 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.

4.3.1 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:[9]

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 syntactic difference is the *not equal* operator. SQL has additional comparison operators that we will present gradually.

We illustrate the basic SELECT statement in SQL with some sample queries. The queries are labeled here with the same query numbers used in Chapter 6 for easy cross-reference.

Query 0. Retrieve the birth date 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 individual 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.

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes**, and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition**. Figure 4.3(a) shows the result of query Q0 on the database of Figure 3.6.

We can think of an implicit **tuple variable** or *iterator* in the SQL query ranging or *looping* over each individual tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is,

[9]The SELECT and FROM clauses are required in all SQL queries. The WHERE is optional (see Section 4.3.3).

Figure 4.3
Results of SQL queries when applied to the COMPANY database state shown in Figure 3.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C.

(a)

Bdate	Address
1965-01-09	731Fondren, 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	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, 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)

E.Fname
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	Super_ssn	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

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;
```

In the WHERE clause of Q1, the condition Dname = 'Research' is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE. The result of query Q1 is shown in Figure 4.3(b). In general, any number of selection and join conditions may be specified in a single SQL query.

A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** 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 birth date.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
               Plocation='Stafford';
```

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a *combination* of one project, one department, and one employee that satisfies the join conditions. The projection attributes are used to choose the attributes to be displayed from each combined tuple. The result of query Q2 is shown in Figure 4.3(c).

4.3.2 Ambiguous Attribute Names, Aliasing, Renaming, 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 multitable 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 3.5 and 3.6 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 same 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;
```

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 is shown in this manner as is Q1′ below. We can also create an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

```
Q1′:  SELECT  EMPLOYEE.Fname, EMPLOYEE.LName,
              EMPLOYEE.Address
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DEPARTMENT.DName='Research' AND
              DEPARTMENT.Dnumber=EMPLOYEE.Dno;
```

The ambiguity of attribute names 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.Super_ssn=S.Ssn;
```

In this case, we are required 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 or subordinates; 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.Super_ssn = S.Ssn. Notice that this is an example of a one-level recursive query, as we will discuss in Section 6.4.2. In earlier versions of SQL, 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:1999 (see Chapter 5).

The result of query Q8 is shown in Figure 4.3(d). Whenever one or more aliases are given to a relation, we can use these names to represent different references to that same relation. This permits multiple references to the same relation within a query.

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 Q1 as in Q1B:

```
Q1B:  SELECT  E.Fname, E.LName, E.Address
      FROM    EMPLOYEE E, DEPARTMENT D
      WHERE   D.DName='Research' AND D.Dnumber=E.Dno;
```

4.3.3 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 4.3(e)), and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not (Figure 4.3(f)).

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 (see Chapter 6). If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the actual 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 any EMPLOYEE who works in DEPARTMENT number 5 (Figure 4.3(g)), 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;
```

4.3.4 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 5.1.7) 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.[10] 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 example, Q11 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 4.4(a). 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 4.4(b).

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;
```

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

(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 4.4
Results of additional SQL queries when applied to the COMPANY database state shown in Figure 3.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra (see Chapter 6). There are set union (UNION), set difference (**EXCEPT**),[11] 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.* These set operations apply only to *union-compatible relations*, so 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.

```
Q4A:  ( SELECT   DISTINCT Pnumber
        FROM     PROJECT, DEPARTMENT, EMPLOYEE
        WHERE    Dnum=Dnumber AND Mgr_ssn=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 4.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

[11] In some systems, the keyword MINUS is used for the set difference operation instead of EXCEPT.

(a) R

A
a1
a2
a2
a3

S

A
a1
a2
a4
a5

(b) T

A
a1
a1
a2
a2
a2
a3
a4
a5

(c) T

A
a2
a3

(d) T

A
a1
a2

Figure 4.5
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).

4.3.5 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 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 Q12A. 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.

```
Q12:  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. Notice that substring comparison implies that attribute values are not atomic (indivisible) values, as we had assumed in the formal relational model (see Section 3.1).

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   E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
      FROM     EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
      WHERE    E.Ssn=W.Essn AND W.Pno=P.Pnumber AND
               P.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, which 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)).

4.3.6 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 of the attributes that appear in the query result, by 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, then first name.

```
Q15:  SELECT    D.Dname, E.Lname, E.Fname, P.Pname
      FROM      DEPARTMENT D, EMPLOYEE E, WORKS_ON W,
                PROJECT P
      WHERE     D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
                W.Pno= P.Pnumber
      ORDER BY  D.Dname, E.Lname, E.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 alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of Q15 can be written as

ORDER BY D.Dname **DESC**, E.Lname **ASC**, E.Fname **ASC**

4.3.7 Discussion and Summary of Basic SQL Retrieval Queries

A *simple* retrieval query in SQL can consist of up to four 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 list>
FROM <table list>
[**WHERE** <condition>]
[**ORDER BY** <attribute list>];

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including join conditions if needed. ORDER BY specifies an order for displaying the results of a query. Two additional clauses GROUP BY and HAVING will be described in Section 5.1.8.

In Chapter 5, we will present more complex features of SQL retrieval queries. These include the following: nested queries that allow one query to be included as part of another query; aggregate functions that are used to provide summaries of the information in the tables; two additional clauses (GROUP BY and HAVING) that can be used to provide additional power to aggregate functions; and various types of joins that can combine records from various tables in different ways.

4.4 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.

4.4.1 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 3.5 and specified in the CREATE TABLE EMPLOYEE ... command in Figure 4.1, we can use U1:

```
U1:   INSERT INTO   EMPLOYEE
      VALUES        ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                      Oak Forest, Katy, TX', 'M', 37000, '653298653', 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 should support and enforce all the integrity constraints that can be specified in the DDL. For example, if we issue the command in U2 on the database shown in Figure 3.6, the DBMS should *reject* the operation because no DEPARTMENT tuple exists in the database with Dnumber = 2. Similarly, U2A would be *rejected* because no Ssn value is provided and it is the primary key, which cannot be NULL.

```
U3:   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 employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

```
U3A:  CREATE TABLE      WORKS_ON_INFO
      ( Emp_name        VARCHAR(15),
        Proj_name       VARCHAR(15),
        Hours_per_week  DECIMAL(3,1) );
```

```
U3B:  INSERT INTO   WORKS_ON_INFO ( Emp_name, Proj_name,
                    Hours_per_week )
      SELECT        E.Lname, P.Pname, W.Hours
      FROM          PROJECT P, WORKS_ON W, EMPLOYEE E
      WHERE         P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B. We can now query WORKS_ON_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 (see Chapter 5). Notice that the WORKS_ON_INFO table may not be up-to-date; that is, if we update any of the PROJECT, WORKS_ON, or EMPLOYEE relations after issuing U3B, the information in WORKS_ON_INFO *may become outdated.* We have to create a view (see Chapter 5) to keep such a table up-to-date.

4.4.2 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 4.2.2).[12] 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. We must use the DROP TABLE command to remove the table definition (see Chapter 5). The DELETE commands in U4A to U4D, if applied independently to the database in Figure 3.6, 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=5;

U4D:  DELETE FROM   EMPLOYEE;
```

4.4.3 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

[12]Other actions can be automatically applied through triggers (see Section 26.1) and other mechanisms.

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 4.2.2). An additional **SET** clause in the UPDATE command specifies the attributes to be modified and 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 = 5;
```

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.

4.5 Additional Features of SQL

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

- In Chapter 5, which is a continuation of this chapter, we will present the following SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules). We discuss these techniques in Chapter 13. 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 language* (*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. Many systems still have the CREATE INDEX commands.

- 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 21 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 24, 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 26.1, 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 11.
- SQL and relational databases can interact with new technologies such as XML (see Chapter 12) and OLAP (Chapter 29).

4.6 Summary

In this chapter we presented the SQL database language. This language and its variations have been implemented as interfaces to many commercial relational DBMSs, including Oracle's Oracle and Rdb[13]; IBM's DB2, Informix Dynamic Server, and SQL/DS; Microsoft's SQL Server and Access; and INGRES. Some open source systems also provide SQL, such as MySQL and PostgreSQL. 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, constraint specification, and view definition. We discussed the following features of SQL in this chapter: the data definition commands for creating tables, commands for constraint specification, simple retrieval queries, and database update commands. In the next chapter,

[13] Rdb was originally produced by Digital Equipment Corporation. It was acquired by Oracle from Digital in 1994 and is being supported and enhanced.

we will present the following features of SQL: complex retrieval queries; views; triggers and assertions; and schema modification commands.

Review Questions

4.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 3? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

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

4.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 3? What about referential triggered actions?

4.4. Describe the four clauses in the syntax of a simple SQL retrieval query. Show what type of constructs can be specified in each of the clauses. Which are required and which are optional?

Exercises

4.5. 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.

4.6. Repeat Exercise 4.5, but use the AIRLINE database schema of Figure 3.8.

4.7. Consider the LIBRARY relational database schema shown in Figure 4.6. Choose the appropriate action (reject, cascade, set to NULL, set to default) for each referential integrity constraint, both for the *deletion* of a referenced tuple and for the *update* of a primary key attribute value in a referenced tuple. Justify your choices.

4.8. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 4.6. Specify the keys and referential triggered actions.

4.9. 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?

4.10. Specify the following queries in SQL on the COMPANY relational database schema shown in Figure 3.5. Show the result of each query if it is applied to the COMPANY database in Figure 3.6.

a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.

b. List the names of all employees who have a dependent with the same first name as themselves.

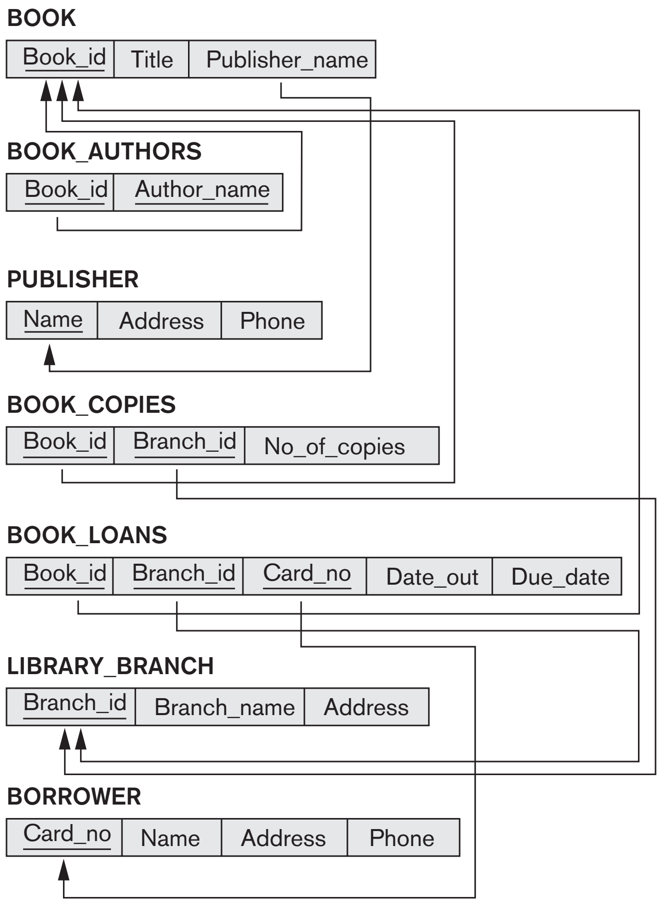

Figure 4.6
A relational database schema for a LIBRARY database.

c. Find the names of all employees who are directly supervised by 'Franklin Wong'.

4.11. Specify the updates of Exercise 3.11 using the SQL update commands.

4.12. 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 2007 and 2008.

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 = 4) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.

4.13. 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.

4.14. Design a relational database schema for a database application of your choice.

a. Declare your relations, using the SQL DDL.

b. Specify a number of queries in SQL that are needed by your database application.

c. Based on your expected use of the database, choose some attributes that should have indexes specified on them.

d. Implement your database, if you have a DBMS that supports SQL.

4.15. Consider the EMPLOYEE table's constraint EMPSUPERFK as specified in Figure 4.2 is changed to read as follows:

```
CONSTRAINT EMPSUPERFK
   FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                    ON DELETE CASCADE ON UPDATE CASCADE,
```

Answer the following questions:

a. What happens when the following command is run on the database state shown in Figure 3.6?

```
DELETE EMPLOYEE WHERE Lname = 'Borg'
```

b. Is it better to CASCADE or SET NULL in case of EMPSUPERFK constraint ON DELETE?

4.16. Write SQL statements to create a table EMPLOYEE_BACKUP to back up the EMPLOYEE table shown in Figure 3.6.

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. We will give additional references to various aspects of SQL at the end of Chapter 5.

chapter 5

More SQL: Complex Queries, Triggers, Views, and Schema Modification

This chapter describes more advanced features of the SQL language standard for relational databases. We start in Section 5.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 5.2, we describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers and the CREATE TRIGGER statement, which will be presented in more detail in Section 26.1 when we present the principles of active databases. Then, in Section 5.3, we describe the SQL facility for defining views on the database. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables. Section 5.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 5.5 is the chapter summary.

This chapter is a continuation of Chapter 4. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

5.1 More Complex SQL Retrieval Queries

In Section 4.3, we described some basic types of retrieval 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 retrievals from the database. We discuss several of these features in this section.

5.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 3.1.2 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 value *not applicable* (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

1. **Unknown value.** A person's date of birth 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 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 individual NULL value is considered to be different from every other NULL value in the various database records. 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 (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 5.1 shows the resulting values.

Table 5.1 Logical Connectives in Three-Valued Logic

(a)

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

(b)

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

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In Tables 5.1(a) and 5.1(b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table 5.1(a). Table 5.1(b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table 5.1(c) shows the result of the NOT logical operation. Notice that in standard Boolean logic, only TRUE or FALSE values are permitted; there is no UNKNOWN value.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause 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 in Section 5.1.6.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **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 5.1.6). Query 18 illustrates this.

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    Super_ssn IS NULL;
```

5.1.2 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.

The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name '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.

```
Q4A:  SELECT   DISTINCT Pnumber
      FROM     PROJECT
      WHERE    Pnumber IN
               ( SELECT    Pnumber
                 FROM      PROJECT, DEPARTMENT, EMPLOYEE
                 WHERE     Dnum=Dnumber AND
                           Mgr_ssn=Ssn AND Lname='Smith' )
               OR
               Pnumber IN
               ( SELECT    Pno
                 FROM      WORKS_ON, EMPLOYEE
                 WHERE     Essn=Ssn AND Lname='Smith' );
```

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    Essn='123456789' );
```

This query will select the Essns 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) within each tuple in WORKS_ON with the set of type-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 two keywords ANY and SOME have the same effect. 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 );
```

Notice that this query can also be specified using the MAX aggregate function (see Section 5.1.7).

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 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.

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT   E.Fname, E.Lname
      FROM     EMPLOYEE AS E
      WHERE    E.Ssn IN  ( SELECT   Essn
                           FROM     DEPENDENT AS D
                           WHERE    E.Fname=D.Dependent_name
                                    AND E.Sex=D.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. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query 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, as illustrated in Q16.

5.1.3 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;
```

5.1.4 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. The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples. We illustrate the use of EXISTS—and NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

```
Q16B:  SELECT   E.Fname, E.Lname
       FROM     EMPLOYEE AS E
       WHERE    EXISTS ( SELECT   *
                         FROM     DEPENDENT AS D
                         WHERE    E.Ssn=D.Essn AND E.Sex=D.Sex
                                  AND E.Fname=D.Dependent_name);
```

EXISTS and NOT EXISTS are typically 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 Essn, Sex, and Dependent_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 because the **WHERE**-clause condition will evaluate to **TRUE** in this case. 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=Mgr_ssn );
```

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?

The query Q3: *Retrieve the name of each employee who works on* all *the projects controlled by department number 5* can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B. This is an example of certain types of queries that require *universal quantification*, as we will discuss in Section 6.6.7. One way to write this query is to use the construct (*S2* EXCEPT *S1*) as explained next, and checking whether the result is empty.[1] 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 with the outer query) 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 result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore 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 Q3A, which uses NOT EXISTS and EXCEPT.

[1]Recall that EXCEPT is the set difference operator. The keyword MINUS is also sometimes used, for example, in Oracle.