

## 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, older 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 differences among various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if two relational DBMSs faithfully support the standard, then conversion between two systems should be 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), as long as both/all of the relational DBMSs support standard SQL.

This chapter presents the *practical* relational model, which is based on the SQL standard for *commercial* relational DBMSs, whereas Chapter 5 presented the most important concepts underlying the *formal* relational data model. In Chapter 8 (Sections 8.1 through 8.5), 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 Chapters 18 and 19. However, the relational algebra operations are too low-level 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 8.6. 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. The standardization of SQL is a joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO), and the first SQL standard is 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. Additional updates to the standard are SQL:2003 and SQL:2006, which added XML features (see Chapter 13) among other updates to the language. Another update in 2008 incorporated more object database features into SQL (see Chapter 12), and a further update is SQL:2011. We will try to cover the latest version of SQL as much as possible, but some of the newer features are discussed in later chapters. It is also not possible to cover the language in its entirety in this text. It is important to note that when new features are added to SQL, it usually takes a few years for some of these features to make it into the commercial SQL DBMSs.

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 or C/C++.<sup>1</sup>

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 the subject of SQL is both important and extensive, we devote two chapters to its basic features. In this chapter, Section 6.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 6.2 presents how basic constraints such as key and referential integrity are specified. Section 6.3 describes the basic SQL constructs for

---

<sup>1</sup>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.

specifying retrieval queries, and Section 6.4 describes the SQL commands for insertion, deletion, and update.

In Chapter 7, 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, and the concept of triggers, which is presented in more detail in Chapter 26. We discuss the SQL facility for defining views on the database in Chapter 7. 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 6.5 lists some SQL features that are presented in other chapters of the book; these include object-oriented features in Chapter 12, XML in Chapter 13, transaction control in Chapter 20, active databases (triggers) in Chapter 26, online analytical processing (OLAP) features in Chapter 29, and security/authorization in Chapter 30. Section 6.6 summarizes the chapter. Chapters 10 and 11 discuss the various database programming techniques for programming with SQL.

## 6.1 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), types, 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 6.1.1 to place our discussion in perspective. Section 6.1.2 describes how tables are created, and Section 6.1.3 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).

### 6.1.1 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 (in some systems, a *schema* is called a *database*). 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, types, 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.<sup>2</sup> Database installations typically have a default environment and schema, so when a user connects and logs in to that database installation, the user can refer directly to tables and other constructs within that schema without having to specify a particular schema name. 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 type and domain definitions.

### 6.1.2 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 possibly 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 7). Figure 6.1 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 6.1, 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 table and its rows are actually created

---

<sup>2</sup>SQL also includes the concept of a *cluster* of catalogs.

```

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),
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 6.1**  
SQL CREATE  
TABLE data  
definition statements  
for defining the  
COMPANY schema  
from Figure 5.7.

and stored as a file by the DBMS. Base relations are distinguished from **virtual relations**, created through the CREATE VIEW statement (see Chapter 7), 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 table (relation).

It is important to note that in Figure 6.1, 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 EMPLOYEE table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has 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 7). We displayed all the foreign keys in Figure 6.1 to show the complete COMPANY schema in one place.

### 6.1.3 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).<sup>3</sup> 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*.<sup>4</sup> There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings

<sup>3</sup>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.

<sup>4</sup>For nonalphabetic characters, there is a defined order.

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 them from character strings; for example, B'10101'.<sup>5</sup> 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 7.
- 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 days must be between 01 and 31; furthermore, a day should be a valid day 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 '2014-09-27' 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

---

<sup>5</sup>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.



strings preceded by the keyword `TIMESTAMP`, with a blank space between data and time; for example, `TIMESTAMP '2014-09-27 09:12:47.648302'`.

- Another data type related to `DATE`, `TIME`, and `TIMESTAMP` is the `INTERVAL` data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either `YEAR/MONTH` intervals or `DAY/TIME` intervals.

The format of `DATE`, `TIME`, and `TIMESTAMP` can be considered as a special type of string. Hence, they can generally be used in string comparisons by being **cast** (or **coerced** or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, as in Figure 6.1; alternatively, a domain can be declared, and the domain name can be 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 6.1 for the attributes `Ssn` and `Super_ssn` of `EMPLOYEE`, `Mgr_ssn` of `DEPARTMENT`, `Esn` of `WORKS_ON`, and `Esn` 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.

In SQL, there is also a **CREATE TYPE** command, which can be used to create user defined types or UDTs. These can then be used either as data types for attributes, or as the basis for creating tables. We shall discuss `CREATE TYPE` in detail in Chapter 12, because it is often used in conjunction with specifying object database features that have been incorporated into more recent versions of SQL.

## 6.2 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 `NULL`s, and constraints on individual tuples within a relation using the `CHECK` clause. We discuss the specification of more general constraints, called assertions, in Chapter 7.

### 6.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows `NULL`s 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 6.1.

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



```

CREATE TABLE EMPLOYEE
(
    ...,
    Dno          INT          NOT NULL          DEFAULT 1,
    CONSTRAINT EMPCHK
    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 6.2**

Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

new tuple if an explicit value is not provided for that attribute. Figure 6.2 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.<sup>6</sup> 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 6.1) to the following:

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

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

```

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

```

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

We can then use the created domain `D_NUM` as the attribute type for all attributes that refer to department numbers in Figure 6.1, such as `Dnumber` of `DEPARTMENT`, `Dnum` of `PROJECT`, `Dno` of `EMPLOYEE`, and so on.

## 6.2.2 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 6.1.<sup>7</sup> 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 6.1):

```
Dnumber INT PRIMARY KEY,
```

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the `DEPARTMENT` and `PROJECT` table declarations in Figure 6.1. The **UNIQUE** clause can also be specified directly for a unique key if it 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 6.1. As we discussed in Section 5.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is updated. 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 6.2. 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.<sup>8</sup>

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

<sup>7</sup>Key and referential integrity constraints were not included in early versions of SQL.

<sup>8</sup>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 6.1.2.

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.

### 6.2.3 Giving Names to Constraints

Figure 6.2 also illustrates how a constraint may be given a **constraint name**, following the keyword **CONSTRAINT**. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Chapter 7. Giving names to constraints is optional. It is also possible to temporarily *defer* a constraint until the end of a transaction, as we shall discuss in Chapter 20 when we present transaction concepts.

### 6.2.4 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 **row-based** constraints because they apply to each row *individually* and are checked whenever a row is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 6.1 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 7 because it requires the full power of queries, which are discussed in Sections 6.3 and 7.1.

## 6.3 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 shall discuss in Chapter 8. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use example queries specified on the schema of Figure 5.5 and will

refer to the sample database state shown in Figure 5.6 to show the results of some of these 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 7.1.

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

### 6.3.1 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:<sup>9</sup>

```
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 8 for easy cross-reference.

---

<sup>9</sup>The **SELECT** and **FROM** clauses are required in all SQL queries. The **WHERE** is optional (see Section 6.3.3).

**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** in relational algebra (see Chapter 8) and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition** in relational algebra. Figure 6.3(a) shows the result of query Q0 on the database of Figure 5.6.

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, 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 6.3(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'
```

**Figure 6.3**

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

(a)

<u>Bdate</u>	<u>Address</u>
1965-01-09	731Fondren, Houston, TX

(b)

<u>Fname</u>	<u>Lname</u>	<u>Address</u>
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)

<u>Pnumber</u>	<u>Dnum</u>	<u>Lname</u>	<u>Address</u>	<u>Bdate</u>
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

(d)

<u>E.Fname</u>	<u>E.Lname</u>	<u>S.Fname</u>	<u>S.Lname</u>
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)

<u>E.Fname</u>
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

<u>Ssn</u>	<u>Dname</u>
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)

<u>Fname</u>	<u>Minit</u>	<u>Lname</u>	<u>Ssn</u>	<u>Bdate</u>	<u>Address</u>	<u>Sex</u>	<u>Salary</u>	<u>Super_ssn</u>	<u>Dno</u>
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

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 (that controls the project), and one employee (that manages the department). 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 6.3(c).

### 6.3.2 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 tables*. 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 5.5 and 5.6 the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the 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 can be rewritten as Q1' below with fully qualified attribute names. We can also rename the table names to shorter names by creating 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 8.4.2. 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 7).

The result of query Q8 is shown in Figure 6.3(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 or **renaming** 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 AS E, DEPARTMENT AS D
      WHERE     D.DName = 'Research' AND D.Dnumber = E.Dno;
```

### 6.3.3 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 6.3(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 6.3(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 8). 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*. The \* can also be prefixed by the relation name or alias; for example, EMPLOYEE.\* refers to all attributes of the EMPLOYEE table.

Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 6.3(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;

```

### 6.3.4 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 7.1.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

**Figure 6.4**

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

(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

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.<sup>10</sup> 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 6.4(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 6.4(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;

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra (see Chapter 8). There are set union (**UNION**), set difference (**EXCEPT**),<sup>11</sup> 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 *type-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.

<sup>10</sup>In general, an SQL table is not required to have a key, although in most cases there will be one.

<sup>11</sup>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 6.5**

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).

**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 6.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

### 6.3.5 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 1970s, we can use Query Q12A. Here, '7' 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 '  7  _ _ _ _ _ _ _ _';
```

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 5.1).

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% 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% 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)).

### 6.3.6 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 AS D, EMPLOYEE AS E, WORKS_ON AS W,
              PROJECT AS 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
```

### 6.3.7 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 7.1.8.

In Chapter 7, 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.

## 6.4 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.

### 6.4.1 The INSERT Command

In its simplest form, **INSERT** is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the **CREATE TABLE** command. For example, to add a new tuple to the **EMPLOYEE** relation shown in Figure 5.5 and specified in the **CREATE TABLE EMPLOYEE ...** command in Figure 6.1, 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 5.6, 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.

```

U2:      INSERT INTO      EMPLOYEE (Fname, Lname, Ssn, Dno)
VALUES      ('Robert', 'Hatcher', '980760540', 2);
(U2 is rejected if referential integrity checking is provided by DBMS.)

U2A:     INSERT INTO      EMPLOYEE (Fname, Lname, Dno)
VALUES      ('Robert', 'Hatcher', 5);
(U2A is rejected if NOT NULL checking is provided by DBMS.)

```

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the 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 anymore, we can remove it by using the DROP TABLE command (see Chapter 7). 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 7) to keep such a table up to date.

Most DBMSs have *bulk loading* tools that allow a user to load formatted data from a file into a table without having to write a large number of INSERT commands. The user can also write a program to read each record in the file, format it as a row in the table, and insert it using the looping constructs of a programming language (see Chapters 10 and 11, where we discuss database programming techniques).

Another variation for loading data is to create a new table TNEW that has the same attributes as an existing table T, and load some of the data currently in T into TNEW. The syntax for doing this uses the LIKE clause. For example, if we

want to create a table D5EMPS with a similar structure to the EMPLOYEE table and load it with the rows of employees who work in department 5, we can write the following SQL:

```
CREATE TABLE      D5EMPS LIKE EMPLOYEE
(SELECT           E.*
FROM             EMPLOYEE AS E
WHERE            E.Dno = 5) WITH DATA;
```

The clause WITH DATA specifies that the table will be created and loaded with the data specified in the query, although in some implementations it may be left out.

### 6.4.2 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 6.2.2).<sup>12</sup> 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 7). The DELETE commands in U4A to U4D, if applied independently to the database state shown in Figure 5.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A:  DELETE FROM      EMPLOYEE
      WHERE            Lname = 'Brown';
U4B:  DELETE FROM      EMPLOYEE
      WHERE            Ssn = '123456789';
U4C:  DELETE FROM      EMPLOYEE
      WHERE            Dno = 5;
U4D:  DELETE FROM      EMPLOYEE;
```

### 6.4.3 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 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

<sup>12</sup>Other actions can be automatically applied through triggers (see Section 26.1) and other mechanisms.

constraints of the DDL (see Section 6.2.2). 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% 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.

## 6.5 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 7, 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, case statements, 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 10. We also describe 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; but they require a special privilege. We describe this in Chapter 17.

- 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 20 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 20, 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, specifying abstract data types (called **UDTs** or user-defined types) for attributes and tables, creating **object identifiers** for referencing tuples, and specifying **operations** on types are discussed in Chapter 12.
- SQL and relational databases can interact with new technologies such as XML (see Chapter 13) and OLAP/data warehouses (Chapter 29).

## 6.6 Summary

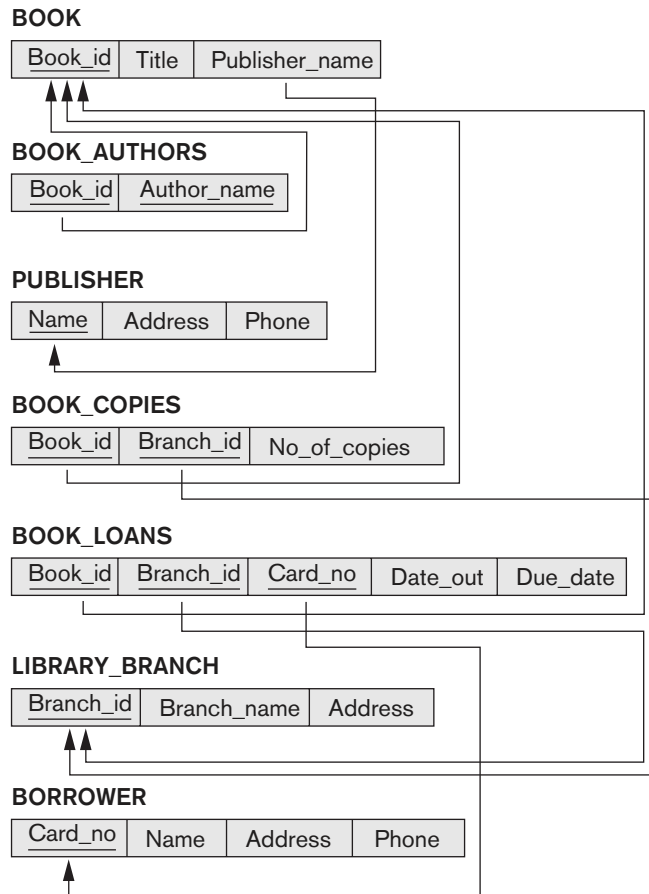
In this chapter, we introduced the SQL database language. This language and its variations have been implemented as interfaces to many commercial relational DBMSs, including Oracle's Oracle; ibm's DB2; Microsoft's SQL Server; and many other systems including Sybase and INGRES. Some open source systems 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, SQL basic data types, commands for constraint specification, simple retrieval queries, and database update commands. In the next chapter, we will present the following features of SQL: complex retrieval queries; views; triggers and assertions; and schema modification commands.

## Review Questions

- 6.1. 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?
- 6.2. List the data types that are allowed for SQL attributes.
- 6.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 3? What about referential triggered actions?
- 6.4. 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

- 6.5. 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.
- 6.6. Repeat Exercise 6.5, but use the AIRLINE database schema of Figure 5.8.
- 6.7. Consider the LIBRARY relational database schema shown in Figure 6.6. 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.
- 6.8. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 6.6. Specify the keys and referential triggered actions.
- 6.9. 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?
- 6.10. Specify the following queries in SQL on the COMPANY relational database schema shown in Figure 5.5. Show the result of each query if it is applied to the COMPANY database in Figure 5.6.
  - 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.
  - c. Find the names of all employees who are directly supervised by 'Franklin Wong'.



**Figure 6.6**  
A relational database  
schema for a  
LIBRARY database.

- 6.11. Specify the updates of Exercise 3.11 using the SQL update commands.
- 6.12. 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.

- 6.13.** Write SQL update statements to do the following on the database schema shown in Figure 1.2.
- Insert a new student, <'Johnson', 25, 1, 'Math'>, in the database.
  - Change the class of student 'Smith' to 2.
  - Insert a new course, <'Knowledge Engineering', 'cs4390', 3, 'cs'>.
  - Delete the record for the student whose name is 'Smith' and whose student number is 17.
- 6.14.** Design a relational database schema for a database application of your choice.
- Declare your relations using the SQL DDL.
  - Specify a number of queries in SQL that are needed by your database application.
  - Based on your expected use of the database, choose some attributes that should have indexes specified on them.
  - Implement your database, if you have a DBMS that supports SQL.
- 6.15.** Consider that the EMPLOYEE table's constraint EMPSUPERFK as specified in Figure 6.2 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:

- What happens when the following command is run on the database state shown in Figure 5.6?  
  
**DELETE EMPLOYEE WHERE** Lname = 'Borg'
  - Is it better to CASCADE or SET NULL in case of EMPSUPERFK constraint ON DELETE?
- 6.16.** Write SQL statements to create a table EMPLOYEE\_BACKUP to back up the EMPLOYEE table shown in Figure 5.6.

## 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 & 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 7.



This page intentionally left blank

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

This chapter describes more advanced features of the SQL language for relational databases. We start in Section 7.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping, and case statements. In Section 7.2, 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 7.3, 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 7.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 7.5 is the chapter summary.

This chapter is a continuation of Chapter 6. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

### 7.1 More Complex SQL Retrieval Queries

In Section 6.3, 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.

7.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 5.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (value exists but is not known, or it is not known whether or not the value exists), value *not available* (value exists but is purposely withheld), or value *not applicable* (the attribute does not apply to this tuple or 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. An example of the other case of unknown would be NULL for a person’s home phone because it is not known whether or not the person has a home phone.
- 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 among 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 record with NULL in one of its attributes 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 7.1 shows the resulting values.

Table 7.1 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 7.1(a) and 7.1(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 7.1(a). Table 7.1(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 7.1(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 7.1.6.

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 7.1.6). Query 18 illustrates NULL comparison by retrieving any employees who do not have a supervisor.

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

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

### 7.1.2 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 another SQL query. That other query is called the **outer query**. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed. 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  $V$ .

In Q4A, the first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, whereas 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 7.1.7).

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  D.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.

### 7.1.3 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;

```

### 7.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition. The EXISTS function in SQL is used to check whether the result of a 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. EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and 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 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 8.6.7. One way to write this query is to use the construct (S2 EXCEPT S1) as explained next, and checking whether the result is empty.<sup>1</sup> 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.

<sup>1</sup>Recall that EXCEPT is the set difference operator. The keyword MINUS is also sometimes used, for example, in Oracle.

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.

```

Q3B:  SELECT  Lname, Fname
        FROM    EMPLOYEE
        WHERE    NOT EXISTS ( SELECT  *
                                FROM    WORKS_ON B
                                WHERE    ( B.Pno IN ( SELECT  Pnumber
                                                         FROM    PROJECT
                                                         WHERE    Dnum = 5 )
                                AND
                                NOT EXISTS ( SELECT  *
                                                FROM    WORKS_ON C
                                                WHERE    C.Essn = Ssn
                                                AND      C.Pno = B.Pno )));

```

In Q3B, the outer nested query selects any WORKS\_ON (B) tuples whose Pno is of a project controlled by department 5, *if* there is not a WORKS\_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we will write this query in tuple relation calculus (see Section 8.6.7).

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

### 7.1.5 Explicit Sets and Renaming in SQL

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

**Query 17.** Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```

Q17:  SELECT  DISTINCT Essn
        FROM    WORKS_ON
        WHERE    Pno IN (1, 2, 3);

```

In SQL, it is possible to **rename** any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names in general, and it can be used in appropriate parts of a query. For example, Q8A shows how query Q8 from Section 4.3.2 can be slightly changed to retrieve the last name of each employee and his or her supervisor while renaming the resulting attribute names

as `Employee_name` and `Supervisor_name`. The new names will appear as column headers for the query result.

```
Q8A:  SELECT  E.Lname AS Employee_name, S.Lname AS Supervisor_name
      FROM    EMPLOYEE AS E, EMPLOYEE AS S
      WHERE   E.Super_ssn = S.Ssn;
```

### 7.1.6 Joined Tables in SQL and Outer Joins

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

```
Q1A:  SELECT  Fname, Lname, Address
      FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
      WHERE   Dname = ‘Research’;
```

The FROM clause in Q1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a **NATURAL JOIN** on two relations *R* and *S*, no join condition is specified; an implicit *EQUIJOIN condition* for *each pair of attributes with the same name* from *R* and *S* is created. Each such pair of attributes is included *only once* in the resulting relation (see Sections 8.3.2 and 8.4.4 for more details on the various types of join operations in relational algebra).

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

```
Q1B:  SELECT  Fname, Lname, Address
      FROM    (EMPLOYEE NATURAL JOIN
              (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
      WHERE   Dname = ‘Research’;
```

The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, only employees who *have a supervisor* are included in the result;

an **EMPLOYEE** tuple whose value for `Super_ssn` is **NULL** is excluded. If the user requires that all employees be included, a different type of join called **OUTER JOIN** must be used explicitly (see Section 8.4.4 for the definition of **OUTER JOIN** in relational algebra). There are several variations of **OUTER JOIN**, as we shall see. In the SQL standard, this is handled by explicitly specifying the keyword **OUTER JOIN** in a joined table, as illustrated in Q8B:

```
Q8B:  SELECT    E.Lname AS Employee_name,
                S.Lname AS Supervisor_name
FROM      (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
            ON E.Super_ssn = S.Ssn);
```

In SQL, the options available for specifying joined tables include **INNER JOIN** (only pairs of tuples that match the join condition are retrieved, same as **JOIN**), **LEFT OUTER JOIN** (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with **NULL** values for the attributes of the right table), **RIGHT OUTER JOIN** (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with **NULL** values for the attributes of the left table), and **FULL OUTER JOIN**. In the latter three options, the keyword **OUTER** may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword **NATURAL** before the operation (for example, **NATURAL LEFT OUTER JOIN**). The keyword **CROSS JOIN** is used to specify the **CARTESIAN PRODUCT** operation (see Section 8.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**. For example, Q2A is a different way of specifying query Q2 from Section 6.3.1 using the concept of a joined table:

```
Q2A:  SELECT    Pnumber, Dnum, Lname, Address, Bdate
FROM      ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
            JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE      Plocation = 'Stafford';
```

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators `+`, `= +`, and `++` for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

```
Q8C:  SELECT    E.Lname, S.Lname
FROM      EMPLOYEE E, EMPLOYEE S
WHERE      E.Super_ssn ++ S.Ssn;
```

### 7.1.7 Aggregate Functions in SQL

**Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database

applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.<sup>2</sup> The **COUNT** function returns the *number of tuples or values* as specified in a query. The functions **SUM**, **MAX**, **MIN**, and **AVG** can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the **SELECT** clause or in a **HAVING** clause (which we introduce later). The functions **MAX** and **MIN** can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.<sup>3</sup> We illustrate the use of these functions with several queries.

**Query 19.** Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

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

This query returns a *single-row* summary of all the rows in the **EMPLOYEE** table. We could use **AS** to rename the column names in the resulting single-row table; for example, as in Q19A.

```
Q19A:     SELECT      SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal,  
                   MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal  
          FROM        EMPLOYEE;
```

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

**Query 20.** Find the sum of the salaries of all employees of the ‘Research’ department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:      SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)  
          FROM        (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)  
          WHERE       Dname = ‘Research’;
```

**Queries 21 and 22.** Retrieve the total number of employees in the company (Q21) and the number of employees in the ‘Research’ department (Q22).

```
Q21:      SELECT      COUNT (*)  
          FROM        EMPLOYEE;  
  
Q22:      SELECT      COUNT (*)  
          FROM        EMPLOYEE, DEPARTMENT  
          WHERE       DNO = DNUMBER AND DNAME = ‘Research’;
```

<sup>2</sup>Additional aggregate functions for more advanced statistical calculation were added in SQL-99.

<sup>3</sup>Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, **DATE**, **TIME**, and **TIMESTAMP** domains have total orderings on their values, as do alphabetic strings.

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

**Query 23.** Count the number of distinct salary values in the database.

**Q23:**     **SELECT**     **COUNT (DISTINCT Salary)**  
          **FROM**       EMPLOYEE;

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(\*) because tuples instead of values are counted. In the previous examples, any Salary values that are NULL are not included in the aggregate function calculation. The general rule is as follows: when an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation; if the collection becomes empty because all values are NULL, the aggregate function will return NULL (except in the case of COUNT, where it will return 0 for an empty collection of values).

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

**Q5:**     **SELECT**     Lname, Fname  
          **FROM**       EMPLOYEE  
          **WHERE**     ( **SELECT**     **COUNT (\*)**  
                      **FROM**       DEPENDENT  
                      **WHERE**     Ssn = Essn ) >= 2;

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

SQL also has aggregate functions SOME and ALL that can be applied to a collection of Boolean values; SOME returns TRUE if at least one element in the collection is TRUE, whereas ALL returns TRUE if all elements in the collection are TRUE.

### 7.1.8 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation*, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees *in each department* or the number

of employees who work *on each project*. In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**. We can then apply the function to each such group independently to produce summary information about each group. SQL has a **GROUP BY** clause for this purpose. The **GROUP BY** clause specifies the grouping attributes, which should *also appear in the SELECT clause*, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

**Query 24.** For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24:   SELECT    Dno, COUNT (*), AVG (Salary)
        FROM      EMPLOYEE
        GROUP BY  Dno;
```

In Q24, the **EMPLOYEE** tuples are partitioned into groups—each group having the same value for the **GROUP BY** attribute **Dno**. Hence, each group contains the employees who work in the same department. The **COUNT** and **AVG** functions are applied to each such group of tuples. Notice that the **SELECT** clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples. Figure 7.1(a) illustrates how grouping works and shows the result of Q24.

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

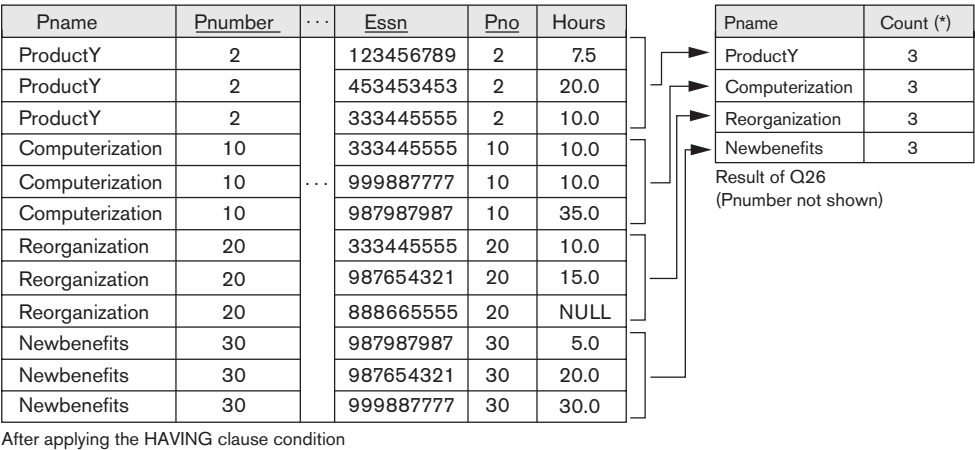
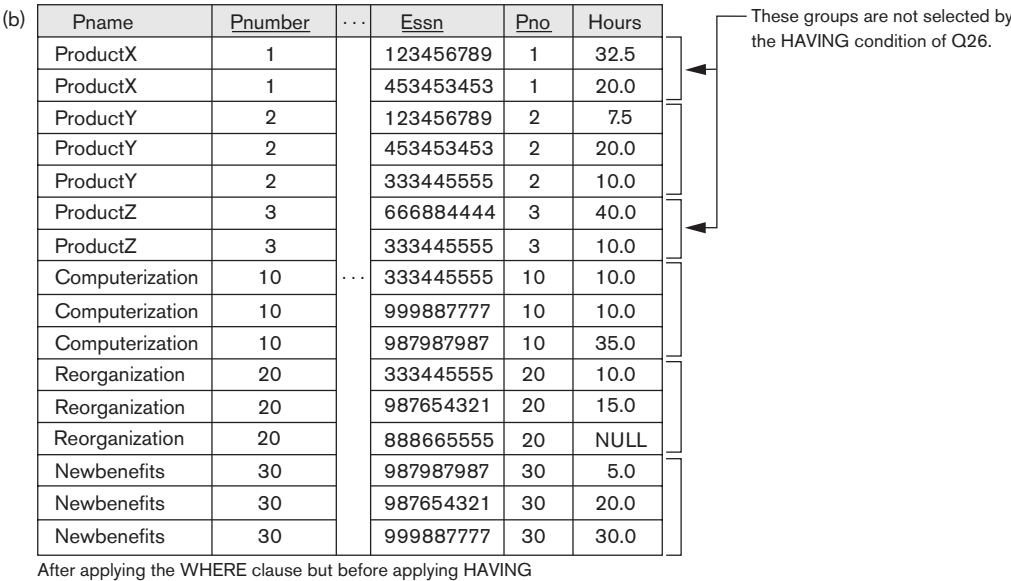
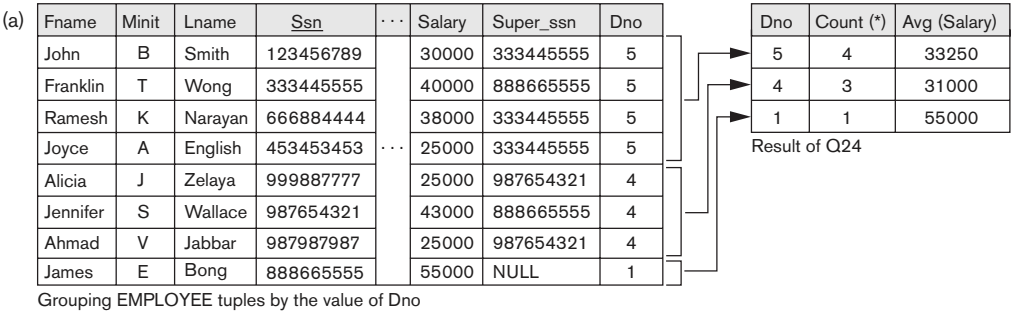
**Query 25.** For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25:   SELECT    Pnumber, Pname, COUNT (*)
        FROM      PROJECT, WORKS_ON
        WHERE     Pnumber = Pno
        GROUP BY  Pnumber, Pname;
```

Q25 shows how we can use a join condition in conjunction with **GROUP BY**. In this case, the grouping and functions are applied *after* the joining of the two relations in the **WHERE** clause.

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

**Figure 7.1**  
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.





**Query 26.** For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:  SELECT    Pnumber, Pname, COUNT (*)
FROM      PROJECT, WORKS_ON
WHERE      Pnumber = Pno
GROUP BY   Pnumber, Pname
HAVING     COUNT (*) > 2;
```

Notice that although selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*. Figure 7.1(b) illustrates the use of HAVING and displays the result of Q26.

**Query 27.** For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27:  SELECT    Pnumber, Pname, COUNT (*)
FROM      PROJECT, WORKS_ON, EMPLOYEE
WHERE      Pnumber = Pno AND Ssn = Essn AND Dno = 5
GROUP BY   Pnumber, Pname;
```

In Q27, we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

```
SELECT    Dno, COUNT (*)
FROM      EMPLOYEE
WHERE      Salary > 40000
GROUP BY   Dno
HAVING     COUNT (*) > 5;
```

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

**Query 28.** For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

```

Q28:      SELECT      Dno, COUNT (*)
           FROM        EMPLOYEE
           WHERE       Salary>40000 AND Dno IN
                    ( SELECT      Dno
                      FROM        EMPLOYEE
                      GROUP BY   Dno
                      HAVING      COUNT (*) > 5)
           GROUP BY   Dno;

```

### 7.1.9 Other SQL Constructs: WITH and CASE

In this section, we illustrate two additional SQL constructs. The WITH clause allows a user to define a table that will only be used in a particular query; it is somewhat similar to creating a view (see Section 7.3) that will be used only in one query and then dropped. This construct was introduced as a convenience in SQL:99 and may not be available in all SQL based DBMSs. Queries using WITH can generally be written using other SQL constructs. For example, we can rewrite Q28 as Q28':

```

Q28':    WITH        BIGDEPTS (Dno) AS
           ( SELECT      Dno
             FROM        EMPLOYEE
             GROUP BY   Dno
             HAVING      COUNT (*) > 5)
           SELECT      Dno, COUNT (*)
           FROM        EMPLOYEE
           WHERE       Salary>40000 AND Dno IN BIGDEPTS
           GROUP BY   Dno;

```

In Q28', we defined in the WITH clause a temporary table BIG\_DEPTS whose result holds the Dno's of departments with more than five employees, then used this table in the subsequent query. Once this query is executed, the temporary table BIGDEPTS is discarded.

SQL also has a CASE construct, which can be used when a value can be different based on certain conditions. This can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples. We illustrate this with an example. Suppose we want to give employees different raise amounts depending on which department they work for; for example, employees in department 5 get a \$2,000 raise, those in department 4 get \$1,500 and those in department 1 get \$3,000 (see Figure 5.6 for the employee tuples). Then we could re-write the update operation U6 from Section 6.4.3 as U6':

```

U6':      UPDATE     EMPLOYEE
           SET        Salary =
           CASE       WHEN      Dno = 5      THEN Salary + 2000
                     WHEN      Dno = 4      THEN Salary + 1500
                     WHEN      Dno = 1      THEN Salary + 3000
                     ELSE        Salary + 0 ;

```

In U6', the salary raise value is determined through the CASE construct based on the department number for which each employee works. The CASE construct can also be used when inserting tuples that can have different attributes being NULL depending on the type of record being inserted into a table, as when a specialization (see Chapter 4) is mapped into a single table (see Chapter 9) or when a union type is mapped into relations.

### 7.1.10 Recursive Queries in SQL

In this section, we illustrate how to write a recursive query in SQL. This syntax was added in SQL:99 to allow users the capability to specify a recursive query in a declarative manner. An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor. This relationship is described by the foreign key `Super_ssn` of the `EMPLOYEE` relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of a supervisory employee  $e$  at all levels—that is, all employees  $e'$  directly supervised by  $e$ , all employees  $e'$  directly supervised by each employee  $e'$ , all employees  $e''$  directly supervised by each employee  $e''$ , and so on. In SQL:99, this query can be written as follows:

```

Q29:      WITH RECURSIVE  SUP_EMP (SupSsn, EmpSsn) AS
           ( SELECT        SupervisorSsn, Ssn
             FROM          EMPLOYEE
             UNION
             SELECT        E.Ssn, S.SupSsn
             FROM          EMPLOYEE AS E, SUP_EMP AS S
             WHERE         E.SupervisorSsn = S.EmpSsn)
           SELECT*
           FROM            SUP_EMP;
```

In Q29, we are defining a view `SUP_EMP` that will hold the result of the recursive query. The view is initially empty. It is first loaded with the first level (supervisor, supervisee) `Ssn` combinations via the first part (**SELECT** `SupervisorSsn, Ssn` **FROM** `EMPLOYEE`), which is called the **base query**. This will be combined via **UNION** with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations, which are **UNIONed** with the first level. This is repeated with successive levels until a **fixed point** is reached, where no more tuples are added to the view. At this point, the result of the recursive query is in the view `SUP_EMP`.

### 7.1.11 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—**SELECT** and **FROM**—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are

specified in the following order, with the clauses between square brackets [ ... ] being optional:

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

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

In order to formulate queries correctly, it is useful to consider the steps that define the *meaning* or *semantics* of each query. A query is evaluated *conceptually*<sup>4</sup> by first applying the **FROM** clause (to identify all tables involved in the query or to materialize any joined tables), followed by the **WHERE** clause to select and join tuples, and then by **GROUP BY** and **HAVING**. Conceptually, **ORDER BY** is applied at the end to sort the query result. If none of the last three clauses (**GROUP BY**, **HAVING**, and **ORDER BY**) are specified, we can *think conceptually* of a query as being executed as follows: For *each combination of tuples*—one from each of the relations specified in the **FROM** clause—evaluate the **WHERE** clause; if it evaluates to **TRUE**, place the values of the attributes specified in the **SELECT** clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute. We discuss query processing and optimization in Chapters 18 and 19.

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

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify

---

<sup>4</sup>The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.



The constraint name `SALARY_CONSTRAINT` is followed by the keyword `CHECK`, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any `WHERE` clause condition can be used, but many constraints can be specified using the `EXISTS` and `NOT EXISTS` style of SQL conditions. Whenever some tuples in the database cause the condition of an `ASSERTION` statement to evaluate to `FALSE`, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition*. By including this query inside a `NOT EXISTS` clause, the assertion will specify that the result of this query must be empty so that the condition will always be `TRUE`. Thus, the assertion is violated if the result of the query is not empty. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Note that the `CHECK` clause and constraint condition can also be used to specify constraints on *individual* attributes and domains (see Section 6.2.1) and on *individual* tuples (see Section 6.2.4). A major difference between `CREATE ASSERTION` and the individual domain constraints and tuple constraints is that the `CHECK` clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use `CHECK` on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*. On the other hand, the schema designer should use `CREATE ASSERTION` only in cases where it is not possible to use `CHECK` on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.

## 7.2.2 Introduction to Triggers in SQL

Another important statement in SQL is `CREATE TRIGGER`. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates. The `CREATE TRIGGER` statement is used to implement such actions in SQL. We discuss triggers in detail in Section 26.1 when we describe *active databases*. Here we just give a simple example of how triggers may be used.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database (see Figures 5.5 and 5.6). Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY\_VIOLATION,<sup>5</sup> which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

```
R5: CREATE TRIGGER SALARY_VIOLATION
    BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
    ON EMPLOYEE
    FOR EACH ROW
    WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                        WHERE SSN = NEW.SUPERVISOR_SSN ) )
    INFORM_SUPERVISOR(NEW.Superervisor_ssn,
                     NEW.Ssn );
```

The trigger is given the name SALARY\_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM\_SUPERVISOR.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically. A complete discussion is given in Section 26.1.

---

<sup>5</sup>Assuming that an appropriate external procedure has been declared. We discuss stored procedures in Chapter 10.



## 7.3 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

### 7.3.1 Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables.<sup>6</sup> These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically. For example, referring to the COMPANY database in Figure 5.5, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS\_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS\_ON, and PROJECT tables the **defining tables** of the view.

### 7.3.2 Specification of Views in SQL

In SQL, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 7.2 when applied to the database schema of Figure 5.5.

<b>V1:</b>	<b>CREATE VIEW</b>	WORKS_ON1
	<b>AS SELECT</b>	Fname, Lname, Pname, Hours
	<b>FROM</b>	EMPLOYEE, PROJECT, WORKS_ON
	<b>WHERE</b>	Ssn = Essn <b>AND</b> Pno = Pnumber;
<b>V2:</b>	<b>CREATE VIEW</b>	DEPT_INFO(Dept_name, No_of_ems, Total_sal)
	<b>AS SELECT</b>	Dname, <b>COUNT</b> (*), <b>SUM</b> (Salary)
	<b>FROM</b>	DEPARTMENT, EMPLOYEE
	<b>WHERE</b>	Dnumber = Dno
	<b>GROUP BY</b>	Dname;

<sup>6</sup>As used in SQL, the term *view* is more limited than the term *user view* discussed in Chapters 1 and 2, since a user view would possibly include many relations.



**WORKS\_ON1**

Fname	Lname	Pname	Hours
-------	-------	-------	-------

**DEPT\_INFO**

Dept_name	No_of_emps	Total_sal
-----------	------------	-----------

**Figure 7.2**

Two views specified on the database schema of Figure 5.5.

In V1, we did not specify any new attribute names for the view WORKS\_ON1 (although we could have); in this case, WORKS\_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS\_ON. View V2 explicitly specifies new attribute names for the view DEPT\_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on the ‘ProductX’ project, we can utilize the WORKS\_ON1 view and specify the query as in QV1:

```
QV1:  SELECT    Fname, Lname
      FROM      WORKS_ON1
      WHERE     Pname = ‘ProductX’;
```

The same query would require the specification of two joins if specified on the base relations directly; one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism (see Section 7.3.4 and Chapter 30).

A view is supposed to be *always up-to-date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view does not have to be realized or materialized at the time of *view definition* but rather at the time when we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date. We will discuss various ways the DBMS can utilize to keep a view up-to-date in the next subsection.

If we do not need a view anymore, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

```
V1A:  DROP VIEW    WORKS_ON1;
```

### 7.3.3 View Implementation, View Update, and Inline Views

The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the

user) into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

```

SELECT      Fname, Lname
FROM        EMPLOYEE, PROJECT, WORKS_ON
WHERE       Ssn = Essn AND Pno = Pnumber
              AND Pname = 'ProductX';

```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied to *one of the defining base tables*. The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

A user can always issue a retrieval query against any view. However, issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in *multiple ways*. Hence, it is often not possible for the DBMS to determine which of the updates is intended. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS\_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```

UV1:      UPDATE WORKS_ON1
SET       Pname = 'ProductY'
WHERE     Lname = 'Smith' AND Fname = 'John'
              AND Pname = 'ProductX';

```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create

additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a):  UPDATE WORKS_ON
      SET      Pno = ( SELECT Pnumber
                        FROM    PROJECT
                        WHERE    Pname = 'ProductY' )
      WHERE    Essn IN ( SELECT Ssn
                        FROM    EMPLOYEE
                        WHERE    Lname = 'Smith' AND Fname = 'John' )
      AND
      Pno = ( SELECT Pnumber
              FROM    PROJECT
              WHERE    Pname = 'ProductX' );

(b):  UPDATE PROJECT SET      Pname = 'ProductY'
      WHERE    Pname = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total\_sal attribute of the DEPT\_INFO view does not make sense because Total\_sal is defined to be the sum of the individual employee salaries. This incorrect request is shown as UV2:

```
UV2:  UPDATE  DEPT_INFO
      SET      Total_sal = 100000
      WHERE    Dname = 'Research';
```

Generally, a view update is feasible when only *one possible update* on the base relations can accomplish the desired update operation on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, it is usually not permitted. Some researchers have suggested that the DBMS have a certain procedure for choosing one of the possible updates as the most likely one. Some researchers have developed methods for choosing the most likely update, whereas other researchers prefer to have the user choose the desired update mapping during view definition. But these options are generally not available in most commercial DBMSs.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have* default values specified.

- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause **WITH CHECK OPTION** should be added at the end of the view definition if a view *is to be updated* by INSERT, DELETE, or UPDATE statements. This allows the system to reject operations that violate the SQL rules for view updates. The full set of SQL rules for when a view may be modified by the user are more complex than the rules stated earlier.

It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

### 7.3.4 Views as Authorization Mechanisms

We describe SQL query authorization statements (GRANT and REVOKE) in detail in Chapter 30, when we present database security and authorization mechanisms. Here, we will just give a couple of simple examples to illustrate how views can be used to hide certain attributes or tuples from unauthorized users. Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW    DEPT5EMP    AS
SELECT        *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible as follows:

```
CREATE VIEW    BASIC_EMP_DATA    AS
SELECT        Fname, Lname, Address
FROM          EMPLOYEE;
```

Thus by creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view. Chapter 30 discusses security and authorization in detail, including the GRANT and REVOKE statements of SQL.

## 7.4 Schema Change Statements in SQL

In this section, we give an overview of the **schema evolution commands** available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema. Certain

checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it inconsistent.

### 7.4.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 6.1, we can get rid of the DEPENDENT relation by issuing the following command:

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 7.3) or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command (see Section 6.4.2) should be used instead of DROP TABLE.

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

### 7.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema (see Figure 6.1), we can use the command

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

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple (see Section 6.4.3). If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

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

```
ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;
```

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

```
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn  
DROP DEFAULT;  
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn  
SET DEFAULT '333445555';
```

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 6.2 from the EMPLOYEE relation, we write:

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

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

The preceding subsections gave an overview of the schema evolution commands of SQL. It is also possible to create new tables and views within a database schema using the appropriate commands. There are many other details and options; we refer the interested reader to the SQL documents listed in the Selected Bibliography at the end of this chapter.

## 7.5 Summary

In this chapter we presented additional features of the SQL database language. We started in Section 7.1 by presenting more complex features of SQL retrieval queries, including nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 7.2, we described the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and introduced the



are shown in angled brackets `< ... >`, optional parts are shown in square brackets `[ ... ]`, repetitions are shown in braces `{ ... }`, and alternatives are shown in parentheses `( ... | ... | ... )`.<sup>7</sup>

## Review Questions

- 7.1. Describe the six clauses in the syntax of an SQL retrieval query. Show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?
- 7.2. Describe conceptually how an SQL retrieval query will be executed by specifying the conceptual order of executing each of the six clauses.
- 7.3. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?
- 7.4. Discuss how each of the following constructs is used in SQL, and discuss the various options for each construct. Specify what each construct is useful for.
  - a. Nested queries
  - b. Joined tables and outer joins
  - c. Aggregate functions and grouping
  - d. Triggers
  - e. Assertions and how they differ from triggers
  - f. The SQL WITH clause
  - g. SQL CASE construct
  - h. Views and their updatability
  - i. Schema change commands

## Exercises

- 7.5. Specify the following queries on the database in Figure 5.5 in SQL. Show the query results if each query is applied to the database state in Figure 5.6.
  - a. For each department whose average employee salary is more than \$30,000, retrieve the department name and the number of employees working for that department.
  - b. Suppose that we want the number of *male* employees in each department making more than \$30,000, rather than all employees (as in Exercise 7.5a). Can we specify this query in SQL? Why or why not?

---

<sup>7</sup>The full syntax of SQL is described in many voluminous documents of hundreds of pages.



- 7.6. Specify the following queries in SQL on the database schema in Figure 1.2.
- Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).
  - Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.
- 7.7. In SQL, specify the following queries on the database in Figure 5.5 using the concept of nested queries and other concepts described in this chapter.
- Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.
  - Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.
  - Retrieve the names of employees who make at least \$10,000 more than the employee who is paid the least in the company.
- 7.8. Specify the following views in SQL on the COMPANY database schema shown in Figure 5.5.
- A view that has the department name, manager name, and manager salary for every department
  - A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department
  - A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project
  - A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project *with more than one employee working on it*
- 7.9. Consider the following view, DEPT\_SUMMARY, defined on the COMPANY database in Figure 5.6:

```

CREATE VIEW      DEPT_SUMMARY (D, C, Total_s, Average_s)
AS SELECT       Dno, COUNT (*), SUM (Salary), AVG (Salary)
FROM            EMPLOYEE
GROUP BY        Dno;

```

State which of the following queries and updates would be allowed on the view. If a query or update would be allowed, show what the corresponding query or update on the base relations would look like, and give its result when applied to the database in Figure 5.6.

- ```

SELECT      *
FROM        DEPT_SUMMARY;

```
- ```

SELECT      D, C
FROM        DEPT_SUMMARY
WHERE       TOTAL_S > 100000;

```

- c. **SELECT** D, AVERAGE\_S  
**FROM** DEPT\_SUMMARY  
**WHERE** C > ( **SELECT** C **FROM** DEPT\_SUMMARY **WHERE** D = 4);
- d. **UPDATE** DEPT\_SUMMARY  
**SET** D = 3  
**WHERE** D = 4;
- e. **DELETE** **FROM** DEPT\_SUMMARY  
**WHERE** C > 4;

## Selected Bibliography

Reisner (1977) describes a human factors evaluation of SEQUEL, a precursor of SQL, in which she found that users have some difficulty with specifying join conditions and grouping correctly. Date (1984) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, Oracle, INGRES, Informix, and other commercial DBMS products. Melton and Simon (1993) give a comprehensive treatment of the ANSI 1992 standard called SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

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

There are many books that describe various aspects of SQL. For example, two references that describe SQL-99 are Melton and Simon (2002) and Melton (2003). Further SQL standards—SQL 2006 and SQL 2008—are described in a variety of technical reports; but no standard references exist.