chapter 7

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.[1] This option is shown as Q3A.

```
Q3A:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    NOT EXISTS ( ( SELECT   Pnumber
                              FROM     PROJECT
                              WHERE    Dnum = 5)
                              EXCEPT   ( SELECT   Pno
                                         FROM     WORKS_ON
                                         WHERE    Ssn = Essn) );
```

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

[1]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**.[2] 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.[3] 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';
```

[2]Additional aggregate functions for more advanced statistical calculation were added in SQL-99.

[3]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.

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

(b)

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

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

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

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** SupervisorSss, 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*[4] 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

[4]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.

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

7.2 Specifying Constraints as Assertions and Actions as Triggers

In this section, we introduce two additional features of SQL: the **CREATE ASSERTION** statement and the **CREATE TRIGGER** statement. Section 7.2.1 discusses CREATE ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity) that we presented in Section 5.2. These built-in constraints can be specified within the **CREATE TABLE** statement of SQL (see Sections 6.1 and 6.2).

In Section 7.2.2 we introduce **CREATE TRIGGER**, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as **active databases**. We only introduce the basics of **triggers** in this chapter, and present a more complete discussion of active databases in Section 26.1.

7.2.1 Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints—those that do not fall into any of the categories described in Sections 6.1 and 6.2— via **declarative assertions**, using the **CREATE ASSERTION** statement. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query. For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT     *
                     FROM       EMPLOYEE E, EMPLOYEE M,
                                DEPARTMENT D
                     WHERE      E.Salary>M.Salary
                                AND     E.Dno = D.Dnumber
                                AND     D.Mgr_ssn = M.Ssn ) );
```

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,[5] 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.Supervisor_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.

[5]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.[6] 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.

```
V1:   CREATE VIEW     WORKS_ON1
      AS SELECT       Fname, Lname, Pname, Hours
         FROM         EMPLOYEE, PROJECT, WORKS_ON
         WHERE        Ssn = Essn AND Pno = Pnumber;

V2:   CREATE VIEW     DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT       Dname, COUNT (*), SUM (Salary)
         FROM         DEPARTMENT, EMPLOYEE
         WHERE        Dnumber = Dno
         GROUP BY     Dname;
```

[6]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.

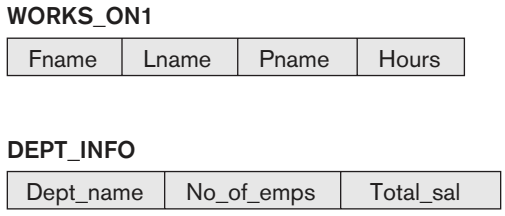

Figure 7.2
Two views specified on the database schema of Figure 5.5.

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

concept of triggers and the CREATE TRIGGER statement. Then, in Section 7.3, we described 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 introduced the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints.

Table 7.2 summarizes the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive or to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available in SQL. We use BNF notation, where nonterminal symbols

Table 7.2 Summary of SQL Syntax

CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>] { , <column name> <column type> [<attribute constraint>] } [<table constraint> { , <table constraint> }])
DROP TABLE <table name> ALTER TABLE <table name> ADD <column name> <column type>
SELECT [DISTINCT] <attribute list> FROM (<table name> { <alias> } | <joined table>) { , (<table name> { <alias> } | <joined table>) } [WHERE <condition>] [GROUP BY <grouping attributes> [HAVING <group selection condition>]] [ORDER BY <column name> [<order>] { , <column name> [<order>] }]
<attribute list> ::= (* | (<column name> | <function> (([DISTINCT] <column name> | *))) { , (<column name> | <function> (([DISTINCT] <column name> | *)) }))
<grouping attributes> ::= <column name> { , <column name> }
<order> ::= (ASC | DESC)
INSERT INTO <table name> [(<column name> { , <column name> })] (VALUES (<constant value> , { <constant value> }) { , (<constant value> { , <constant value> }) } | <select statement>)
DELETE FROM <table name> [WHERE <selection condition>]
UPDATE <table name> SET <column name> = <value expression> { , <column name> = <value expression> } [WHERE <selection condition>]
CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name> [<order>] { , <column name> [<order>] }) [CLUSTER]
DROP INDEX <index name>
CREATE VIEW <view name> [(<column name> { , <column name> })] AS <select statement>
DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

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

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?

[7]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.

a. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).

b. 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.

a. Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.

b. Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.

c. 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. A view that has the department name, manager name, and manager salary for every department

b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department

c. 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

d. 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.

a.
```
SELECT    *
FROM      DEPT_SUMMARY;
```

b.
```
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.