

6 SQL: Data Manipulation



Chapter Objectives

In this chapter you will learn:

- The purpose and importance of the Structured Query Language (SQL).
- The history and development of SQL. © CourseSmart
- How to write an SQL command.
- How to retrieve data from the database using the SELECT statement.
- How to build SQL statements that:
 - use the WHERE clause to retrieve rows that satisfy various conditions;
 - sort query results using ORDER BY;
 - use the aggregate functions of SQL;
 - group data using GROUP BY;
 - use subqueries;
 - join tables together;
 - perform set operations (UNION, INTERSECT, EXCEPT).
- How to perform database updates using INSERT, UPDATE, and DELETE.

© CourseSmart

In Chapters 4 and 5 we described the relational data model and relational languages in some detail. A particular language that has emerged from the development of the relational model is the Structured Query Language, or SQL as it is commonly called. Over the last few years, SQL has become the standard relational database language. In 1986, a standard for SQL was defined by the American National Standards Institute (ANSI) and was subsequently adopted in 1987 as an international standard by the International Organization for Standardization (ISO, 1987). More than one hundred DBMSs now support SQL, running on various hardware platforms from PCs to mainframes.

Owing to the current importance of SQL, we devote four chapters and an appendix of this book to examining the language in detail, providing a comprehensive treatment for both technical and nontechnical users including programmers, database professionals, and managers. In these chapters we largely

concentrate on the ISO definition of the SQL language. However, owing to the complexity of this standard, we do not attempt to cover all parts of the language. In this chapter, we focus on the data manipulation statements of the language.

Structure of this Chapter In Section 6.1 we introduce SQL and discuss why the language is so important to database applications. In Section 6.2 we introduce the notation used in this book to specify the structure of an SQL statement. In Section 6.3 we discuss how to retrieve data from relations using SQL, and how to insert, update, and delete data from relations.

Looking ahead, in Chapter 7 we examine other features of the language, including data definition, views, transactions, and access control. In Chapter 8 we examine more advanced features of the language, including triggers and stored procedures. In Section 29.4 we examine in some detail the features that have been added to the SQL specification to support object-oriented data management. In Appendix I we discuss how SQL can be embedded in high-level programming languages to access constructs that were not available in SQL until very recently. The two formal languages, relational algebra and relational calculus, that we covered in Chapter 5 provide a foundation for a large part of the SQL standard and it may be useful to refer to this chapter occasionally to see the similarities. However, our presentation of SQL is mainly independent of these languages for those readers who have omitted Chapter 5. The examples in this chapter use the *DreamHome* rental database instance shown in Figure 4.3.



6.1 Introduction to SQL

In this section we outline the objectives of SQL, provide a short history of the language, and discuss why the language is so important to database applications.

6.1.1 Objectives of SQL

Ideally, a database language should allow a user to:

- create the database and relation structures;
- perform basic data management tasks, such as the insertion, modification, and deletion of data from the relations;
- perform both simple and complex queries.

A database language must perform these tasks with minimal user effort, and its command structure and syntax must be relatively easy to learn. Finally, the language must be portable; that is, it must conform to some recognized standard so that we can use the same command structure and syntax when we move from one DBMS to another. SQL is intended to satisfy these requirements.

SQL is an example of a **transform-oriented language**, or a language designed to use relations to transform inputs into required outputs. As a language, the ISO SQL standard has two major components:

- a Data Definition Language (DDL) for defining the database structure and controlling access to the data;
- a Data Manipulation Language (DML) for retrieving and updating data.

Until the 1999 release of the standard, known as SQL:1999 or SQL3, SQL contained only these definitional and manipulative commands; it did not contain flow of control commands, such as IF . . . THEN . . . ELSE, GO TO, or DO . . . WHILE. These commands had to be implemented using a programming or job-control language, or interactively by the decisions of the user. Owing to this lack of *computational completeness*, SQL can be used in two ways. The first way is to use SQL *interactively* by entering the statements at a terminal. The second way is to embed SQL statements in a procedural language, as we discuss in Appendix I. We also discuss the latest release of the standard, SQL:2008 in Chapter 29.

SQL is a relatively easy language to learn:

- It is a nonprocedural language: you specify *what* information you require, rather than *how* to get it. In other words, SQL does not require you to specify the access methods to the data.
- Like most modern languages, SQL is essentially free-format, which means that parts of statements do not have to be typed at particular locations on the screen.
- The command structure consists of standard English words such as CREATE TABLE, INSERT, SELECT. For example:
 - **CREATE TABLE Staff (staffNo VARCHAR(5), lName VARCHAR(15), salary DECIMAL(7,2));**
 - **INSERT INTO Staff VALUES ('SG16', 'Brown', 8300);**
 - **SELECT staffNo, lName, salary
FROM Staff
WHERE salary > 10000;**
- SQL can be used by a range of users including database administrators (DBA), management personnel, application developers, and many other types of end-user.

An international standard now exists for the SQL language making it both the formal and de facto standard language for defining and manipulating relational databases (ISO, 1992, 2008a).

6.1.2 History of SQL

As stated in Chapter 4, the history of the relational model (and indirectly SQL) started with the publication of the seminal paper by E. F. Codd, written during his work at IBM's Research Laboratory in San José (Codd, 1970). In 1974, D. Chamberlin, also from the IBM San José Laboratory, defined a language called the Structured English Query Language, or SEQUEL. A revised version, SEQUEL/2, was defined in 1976, but the name was subsequently changed to SQL for legal reasons (Chamberlin and Boyce, 1974; Chamberlin *et al.*, 1976). Today, many people still pronounce SQL as "See-Quel," though the official pronunciation is "S-Q-L."

IBM produced a prototype DBMS based on SEQUEL/2 called System R (Astrahan *et al.*, 1976). The purpose of this prototype was to validate the feasibility of the relational model. Besides its other successes, one of the most important results that has been attributed to this project was the development of SQL. However, the roots of

SQL are in the language SQUARE (Specifying Queries As Relational Expressions), which predates the System R project. SQUARE was designed as a research language to implement relational algebra with English sentences (Boyce *et al.*, 1975).

In the late 1970s, the database system Oracle was produced by what is now called the Oracle Corporation, and was probably the first commercial implementation of a relational DBMS based on SQL. INGRES followed shortly afterwards, with a query language called QUEL, which although more “structured” than SQL, was less English-like. When SQL emerged as the standard database language for relational systems, INGRES was converted to an SQL-based DBMS. IBM produced its first commercial RDBMS, called SQL/DS, for the DOS/VSE and VM/CMS environments in 1981 and 1982, respectively, and subsequently as DB2 for the MVS environment in 1983.

© CourseSmart In 1982, ANSI began work on a Relational Database Language (RDL) based on a concept paper from IBM. ISO joined in this work in 1983, and together they defined a standard for SQL. (The name RDL was dropped in 1984, and the draft standard reverted to a form that was more like the existing implementations of SQL.)

The initial ISO standard published in 1987 attracted a considerable degree of criticism. Date, an influential researcher in this area, claimed that important features such as referential integrity constraints and certain relational operators had been omitted. He also pointed out that the language was extremely redundant; in other words, there was more than one way to write the same query (Date, 1986, 1987a, 1990). Much of the criticism was valid, and had been recognized by the standards bodies before the standard was published. It was decided, however, that it was more important to release a standard as early as possible to establish a common base from which the language and the implementations could develop than to wait until all the features that people felt should be present could be defined and agreed.

In 1989, ISO published an addendum that defined an “Integrity Enhancement Feature” (ISO, 1989). In 1992, the first major revision to the ISO standard occurred, sometimes referred to as SQL2 or SQL-92 (ISO, 1992). Although some features had been defined in the standard for the first time, many of these had already been implemented in part or in a similar form in one or more of the many SQL implementations. It was not until 1999 that the next release of the standard, commonly referred to as SQL:1999 (ISO, 1999a), was formalized. This release contains additional features to support object-oriented data management, which we examine in Section 29.4. There have been further releases of the standard in late 2003 (SQL:2003) and in summer 2008 (SQL:2008).

Features that are provided on top of the standard by the vendors are called **extensions**. For example, the standard specifies six different data types for data in an SQL database. Many implementations supplement this list with a variety of extensions. Each implementation of SQL is called a **dialect**. No two dialects are exactly alike, and currently no dialect exactly matches the ISO standard. Moreover, as database vendors introduce new functionality, they are expanding their SQL dialects and moving them even further apart. However, the central core of the SQL language is showing signs of becoming more standardized. In fact, SQL now has a set of features called **Core SQL** that a vendor must implement to claim **conformance** with the SQL standard. Many of the remaining features are divided into packages; for example, there are **packages** for object features and OLAP (OnLine Analytical Processing).

Although SQL was originally an IBM concept, its importance soon motivated other vendors to create their own implementations. Today there are literally hundreds of SQL-based products available, with new products being introduced regularly.

6.1.3 Importance of SQL

SQL is the first and, so far, only standard database language to gain wide acceptance. The only other standard database language, the Network Database Language (NDL), based on the CODASYL network model, has few followers. Nearly every major current vendor provides database products based on SQL or with an SQL interface, and most are represented on at least one of the standard-making bodies. There is a huge investment in the SQL language both by vendors and by users. It has become part of application architectures such as IBM's Systems Application Architecture (SAA) and is the strategic choice of many large and influential organizations, for example, the Open Group consortium for UNIX standards. SQL has also become a Federal Information Processing Standard (FIPS) to which conformance is required for all sales of DBMSs to the U.S. government. The SQL Access Group, a consortium of vendors, defined a set of enhancements to SQL that would support interoperability across disparate systems.

SQL is used in other standards and even influences the development of other standards as a definitional tool. Examples include ISO's Information Resource Dictionary System (IRDS) standard and Remote Data Access (RDA) standard. The development of the language is supported by considerable academic interest, providing both a theoretical basis for the language and the techniques needed to implement it successfully. This is especially true in query optimization, distribution of data, and security. There are now specialized implementations of SQL that are directed at new markets, such as OnLine Analytical Processing (OLAP).

6.1.4 Terminology

The ISO SQL standard does not use the formal terms of relations, attributes, and tuples, instead using the terms tables, columns, and rows. In our presentation of SQL we mostly use the ISO terminology. It should also be noted that SQL does not adhere strictly to the definition of the relational model described in Chapter 4. For example, SQL allows the table produced as the result of the SELECT statement to contain duplicate rows, it imposes an ordering on the columns, and it allows the user to order the rows of a result table.

6.2 Writing SQL Commands

In this section we briefly describe the structure of an SQL statement and the notation we use to define the format of the various SQL constructs. An SQL statement consists of **reserved words** and **user-defined words**. Reserved words are a fixed part of the SQL language and have a fixed meaning. They must be spelled *exactly* as required and cannot be split across lines. User-defined words are made up by the user (according to certain syntax rules) and represent the names of various database objects such as tables, columns, views, indexes, and so on. The words in a statement are also built according to a set of syntax rules. Although the standard

does not require it, many dialects of SQL require the use of a statement terminator to mark the end of each SQL statement (usually the semicolon “;” is used).

Most components of an SQL statement are **case-insensitive**, which means that letters can be typed in either upper- or lowercase. The one important exception to this rule is that literal character data must be typed *exactly* as it appears in the database. For example, if we store a person’s surname as “SMITH” and then search for it using the string “Smith,” the row will not be found.

Although SQL is free-format, an SQL statement or set of statements is more readable if indentation and lineation are used. For example:

- each clause in a statement should begin on a new line;
- the beginning of each clause should line up with the beginning of other clauses;
- if a clause has several parts, they should each appear on a separate line and be indented under the start of the clause to show the relationship.

Throughout this and the next two chapters, we use the following extended form of the Backus Naur Form (BNF) notation to define SQL statements:

- uppercase letters are used to represent reserved words and must be spelled exactly as shown;
- lowercase letters are used to represent user-defined words;
- a vertical bar (|) indicates a **choice** among alternatives; for example, a | b | c;
- curly braces indicate a **required element**; for example, {a};
- square brackets indicate an **optional element**; for example, [a];
- an ellipsis (. . .) is used to indicate **optional repetition** of an item zero or more times.

For example:

{a|b} (, c . . .)

means either a or b followed by zero or more repetitions of c separated by commas.

In practice, the DDL statements are used to create the database structure (that is, the tables) and the access mechanisms (that is, what each user can legally access), and then the DML statements are used to populate and query the tables. However, in this chapter we present the DML before the DDL statements to reflect the importance of DML statements to the general user. We discuss the main DDL statements in the next chapter.

6.3 Data Manipulation

This section looks at the SQL DML statements:

- SELECT – to query data in the database;
- INSERT – to insert data into a table;
- UPDATE – to update data in a table;
- DELETE – to delete data from a table.

Owing to the complexity of the SELECT statement and the relative simplicity of the other DML statements, we devote most of this section to the SELECT

statement and its various formats. We begin by considering simple queries, and successively add more complexity to show how more complicated queries that use sorting, grouping, aggregates, and also queries on multiple tables can be generated. We end the chapter by considering the INSERT, UPDATE, and DELETE statements.

We illustrate the SQL statements using the instance of the *DreamHome* case study shown in Figure 4.3, which consists of the following tables:



Branch	(<u>branchNo</u> , street, city, postcode)
Staff	(<u>staffNo</u> , fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent	(<u>propertyNo</u> , street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)
Client	(<u>clientNo</u> , fName, lName, telNo, prefType, maxRent, eMail)
PrivateOwner	(<u>ownerNo</u> , fName, lName, address, telNo, eMail, password)
Viewing	(<u>clientNo</u> , propertyNo, viewDate, comment)

Literals

Before we discuss the SQL DML statements, it is necessary to understand the concept of **literals**. Literals are **constants** that are used in SQL statements. There are different forms of literals for every data type supported by SQL (see Section 7.1.1). However, for simplicity, we can distinguish between literals that are enclosed in single quotes and those that are not. All nonnumeric data values must be enclosed in single quotes; all numeric data values must **not** be enclosed in single quotes. For example, we could use literals to insert data into a table:

```
INSERT INTO PropertyForRent(propertyNo, street, city, postcode, type, rooms, rent,
                           ownerNo, staffNo, branchNo)
VALUES ('PA14', '16 Holhead', 'Aberdeen', 'AB7 5SU', 'House', 6, 650.00,
       'CO46', 'SA9', 'B007');
```

The value in column *rooms* is an integer literal and the value in column *rent* is a decimal number literal; they are not enclosed in single quotes. All other columns are character strings and are enclosed in single quotes.

6.3.1 Simple Queries

The purpose of the SELECT statement is to retrieve and display data from one or more database tables. It is an extremely powerful command, capable of performing the equivalent of the relational algebra's *Selection*, *Projection*, and *Join* operations in a single statement (see Section 5.1). SELECT is the most frequently used SQL command and has the following general form:

SELECT	[DISTINCT ALL] {*} [columnExpression AS newName] [, . . .]
FROM	TableName [alias] [, . . .]
[WHERE]	condition
[GROUP BY]	columnList [HAVING condition]
[ORDER BY]	columnList

columnExpression represents a column name or an expression, *TableName* is the name of an existing database table or view that you have access to, and *alias* is an optional abbreviation for *TableName*. The sequence of processing in a SELECT statement is:

FROM	specifies the table or tables to be used
WHERE	filters the rows subject to some condition
GROUP BY	forms groups of rows with the same column value
HAVING	filters the groups subject to some condition
SELECT	specifies which columns are to appear in the output
ORDER BY	specifies the order of the output

The order of the clauses in the SELECT statement *cannot* be changed. The only two mandatory clauses are the first two: SELECT and FROM; the remainder are optional. The SELECT operation is **closed**: the result of a query on a table is another table (see Section 5.1). There are many variations of this statement, as we now illustrate.

Retrieve all rows

EXAMPLE 6.1 Retrieve all columns, all rows

List full details of all staff.

Because there are no restrictions specified in this query, the WHERE clause is unnecessary and all columns are required. We write this query as:

```
SELECT staffNo, fName, lName, position, sex, DOB, salary, branchNo
  FROM Staff;
```

Because many SQL retrievals require all columns of a table, there is a quick way of expressing “all columns” in SQL, using an asterisk (*) in place of the column names. The following statement is an equivalent and shorter way of expressing this query:

```
SELECT *
  FROM Staff;
```

The result table in either case is shown in Table 6.1.

TABLE 6.1 Result table for Example 6.1.

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000.00	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000.00	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000.00	B005

EXAMPLE 6.2 Retrieve specific columns, all rows

Produce a list of salaries for all staff, showing only the staff number, the first and last names, and the salary details.

```
SELECT staffNo, fName, lName, salary  
FROM Staff;
```

In this example a new table is created from `Staff` containing only the designated columns `staffNo`, `fName`, `lName`, and `salary`, in the specified order. The result of this operation is shown in Table 6.2. Note that, unless specified, the rows in the result table may not be sorted. Some DBMSs do sort the result table based on one or more columns (for example, Microsoft Office Access would sort this result table based on the primary key `staffNo`). We describe how to sort the rows of a result table in the next section.

TABLE 6.2 Result table for Example 6.2.

staffNo	fName	lName	salary
SL21	John	White	30000.00
SG37	Ann	Beech	12000.00
SG14	David	Ford	18000.00
SA9	Mary	Howe	9000.00
SG5	Susan	Brand	24000.00
SL41	Julie	Lee	9000.00

EXAMPLE 6.3 Use of DISTINCT

List the property numbers of all properties that have been viewed.

```
SELECT propertyNo  
FROM Viewing;
```

The result table is shown in Table 6.3(a). Notice that there are several duplicates, because unlike the relational algebra Projection operation (see Section 5.1.1), SELECT does not

TABLE 6.3(a) Result table for Example 6.3 with duplicates.

propertyNo
PA14
PG4
PG4
PA14
PG36

eliminate duplicates when it projects over one or more columns. To eliminate the duplicates, we use the DISTINCT keyword. Rewriting the query as:

```
SELECT DISTINCT propertyNo
  FROM Viewing;
```

we get the result table shown in Table 6.3(b) with the duplicates eliminated.

TABLE 6.3(b) Result table for Example 6.3 with duplicates eliminated.

propertyNo

PA14

PG4

PG36

EXAMPLE 6.4 Calculated fields

Produce a list of monthly salaries for all staff, showing the staff number, the first and last names, and the salary details.

```
SELECT staffNo, fName, lName, salary/12
  FROM Staff;
```

This query is almost identical to Example 6.2, with the exception that monthly salaries are required. In this case, the desired result can be obtained by simply dividing the salary by 12, giving the result table shown in Table 6.4.

This is an example of the use of a **calculated field** (sometimes called a **computed** or **derived field**). In general, to use a calculated field, you specify an SQL expression in the SELECT list. An SQL expression can involve addition, subtraction, multiplication, and division, and parentheses can be used to build complex expressions. More than one table column can be used in a calculated column; however, the columns referenced in an arithmetic expression must have a numeric type.

The fourth column of this result table has been output as *col4*. Normally, a column in the result table takes its name from the corresponding column of the database table from which it has been retrieved. However, in this case, SQL does not know how to label the column. Some dialects give the column a name corresponding to its position

TABLE 6.4 Result table for Example 6.4.

staffNo	fName	lName	col4
SL21	John	White	2500.00
SG37	Ann	Beech	1000.00
SG14	David	Ford	1500.00
SA9	Mary	Howe	750.00
SG5	Susan	Brand	2000.00
SL41	Julie	Lee	750.00

in the table (for example, col4); some may leave the column name blank or use the expression entered in the SELECT list. The ISO standard allows the column to be named using an AS clause. In the previous example, we could have written:

```
SELECT staffNo, fName, lName, salary/12 AS monthlySalary
  FROM Staff;
```

In this case, the column heading of the result table would be monthlySalary rather than col4.

Row selection (WHERE clause)

The previous examples show the use of the SELECT statement to retrieve all rows from a table. However, we often need to restrict the rows that are retrieved. This can be achieved with the WHERE clause, which consists of the keyword WHERE followed by a search condition that specifies the rows to be retrieved. The five basic search conditions (*or predicates*, using the ISO terminology) are as follows:

- *Comparison* Compare the value of one expression to the value of another expression.
- *Range* Test whether the value of an expression falls within a specified range of values.
- *Set membership* Test whether the value of an expression equals one of a set of values.
- *Pattern match* Test whether a string matches a specified pattern.
- *Null* Test whether a column has a null (unknown) value.

The WHERE clause is equivalent to the relational algebra Selection operation discussed in Section 5.1.1. We now present examples of each of these types of search conditions.

EXAMPLE 6.5 Comparison search condition

List all staff with a salary greater than £10,000.

```
SELECT staffNo, fName, lName, position, salary
  FROM Staff
 WHERE salary > 10000;
```

Here, the table is Staff and the predicate is salary > 10000. The selection creates a new table containing only those Staff rows with a salary greater than £10,000. The result of this operation is shown in Table 6.5.

TABLE 6.5 Result table for Example 6.5.

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG37	Ann	Beech	Assistant	12000.00
SG14	David	Ford	Supervisor	18000.00
SG5	Susan	Brand	Manager	24000.00

In SQL, the following simple comparison operators are available:

=	equals
<>	is not equal to (ISO standard) != is not equal to (allowed in some dialects)
<	is less than <= is less than or equal to
>	is greater than >= is greater than or equal to

More complex predicates can be generated using the logical operators **AND**, **OR**, and **NOT**, with parentheses (if needed or desired) to show the order of evaluation. The rules for evaluating a conditional expression are:

- an expression is evaluated left to right;
- subexpressions in brackets are evaluated first;
- NOTs are evaluated before ANDs and ORs;
- ANDs are evaluated before ORs.

The use of parentheses is always recommended, in order to remove any possible ambiguities.

EXAMPLE 6.6 Compound comparison search condition

List the addresses of all branch offices in London or Glasgow.

```
SELECT *
FROM Branch
WHERE city = 'London' OR city = 'Glasgow';
```

In this example the logical operator **OR** is used in the WHERE clause to find the branches in London (`city = 'London'`) or in Glasgow (`city = 'Glasgow'`). The result table is shown in Table 6.6.

TABLE 6.6 Result table for Example 6.6.

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B003	163 Main St	Glasgow	G11 9QX
B002	56 Clover Dr	London	NW10 6EU

EXAMPLE 6.7 Range search condition (BETWEEN/NOT BETWEEN)

List all staff with a salary between £20,000 and £30,000.

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary BETWEEN 20000 AND 30000;
```

The **BETWEEN** test includes the endpoints of the range, so any members of staff with a salary of £20,000 or £30,000 would be included in the result. The result table is shown in Table 6.7.

TABLE 6.7 Result table for Example 6.7.

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG5	Susan	Brand	Manager	24000.00

© CourseSmart

There is also a negated version of the range test (NOT BETWEEN) that checks for values outside the range. The BETWEEN test does not add much to the expressive power of SQL, because it can be expressed equally well using two comparison tests. We could have expressed the previous query as:

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary >= 20000 AND salary <= 30000;
```

However, the BETWEEN test is a simpler way to express a search condition when considering a range of values.

EXAMPLE 6.8 Set membership search condition (IN/NOT IN)

List all managers and supervisors.

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE position IN ('Manager', 'Supervisor');
```

The set membership test (IN) tests whether a data value matches one of a list of values, in this case either 'Manager' or 'Supervisor'. The result table is shown in Table 6.8.

TABLE 6.8 Result table for Example 6.8.

staffNo	fName	lName	position
SL21	John	White	Manager
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager

There is a negated version (NOT IN) that can be used to check for data values that do not lie in a specific list of values. Like BETWEEN, the IN test does not add much to the expressive power of SQL. We could have expressed the previous query as:

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE position = 'Manager' OR position = 'Supervisor';
```

However, the IN test provides a more efficient way of expressing the search condition, particularly if the set contains many values.

EXAMPLE 6.9 Pattern match search condition (LIKE/NOT LIKE)

Find all owners with the string ‘Glasgow’ in their address.

For this query, we must search for the string ‘Glasgow’ appearing somewhere within the address column of the PrivateOwner table. SQL has two special pattern-matching symbols:

- The % percent character represents any sequence of zero or more characters (*wildcard*).
- The _ underscore character represents any single character.

All other characters in the pattern represent themselves. For example:

- address LIKE ‘H%’ means the first character must be *H*, but the rest of the string can be anything.
- address LIKE ‘H___’ means that there must be exactly four characters in the string, the first of which must be an *H*.
- address LIKE ‘%e’ means any sequence of characters, of length at least 1, with the last character an *e*.
- address LIKE ‘%Glasgow%’ means a sequence of characters of any length containing *Glasgow*.
- address NOT LIKE ‘H%’ means the first character cannot be an *H*.

If the search string can include the pattern-matching character itself, we can use an **escape character** to represent the pattern-matching character. For example, to check for the string ‘15%', we can use the predicate:

LIKE ‘15#%’ ESCAPE ‘#’

Using the pattern-matching search condition of SQL, we can find all owners with the string “Glasgow” in their address using the following query, producing the result table shown in Table 6.9:

```
SELECT ownerNo, fName, lName, address, telNo
  FROM PrivateOwner
 WHERE address LIKE '%Glasgow%';
```

Note that some RDBMSs, such as Microsoft Office Access, use the wildcard characters * and ? instead of % and _.

TABLE 6.9 Result table for Example 6.9.

ownerNo	fName	lName	address	telNo
CO87	Carol	Farrel	6 Achray St, Glasgow G32 9DX	0141-357-7419
CO40	Tina	Murphy	63 Well St, Glasgow G42	0141-943-1728
CO93	Tony	Shaw	12 Park Pl, Glasgow G4 0QR	0141-225-7025

EXAMPLE 6.10 NULL search condition (IS NULL/IS NOT NULL)

List the details of all viewings on property PG4 where a comment has not been supplied.

From the Viewing table of Figure 4.3, we can see that there are two viewings for property PG4: one with a comment, the other without a comment. In this simple example,

you may think that the latter row could be accessed by using one of the search conditions:

(propertyNo = 'PG4' AND comment = '')

or

(propertyNo = 'PG4' AND comment < > 'too remote')

However, neither of these conditions would work. A null comment is considered to have an unknown value, so we cannot test whether it is equal or not equal to another string. If we tried to execute the SELECT statement using either of these compound conditions, we would get an empty result table. Instead, we have to test for null explicitly using the special keyword IS NULL:

```
SELECT clientNo, viewDate
FROM Viewing
WHERE propertyNo = 'PG4' AND comment IS NULL;
```

The result table is shown in Table 6.10. The negated version (IS NOT NULL) can be used to test for values that are not null.

TABLE 6.10 Result table for Example 6.10.

clientNo	viewDate
CR56	26-May-08

6.3.2 Sorting Results (ORDER BY Clause)

Esta subsección no

In general, the rows of an SQL query result table are not arranged in any particular order (although some DBMSs may use a default ordering based, for example, on a primary key). However, we can ensure the results of a query are sorted using the ORDER BY clause in the SELECT statement. The ORDER BY clause consists of a list of **column identifiers** that the result is to be sorted on, separated by commas. A column identifier may be either a column name or a column number[†] that identifies an element of the SELECT list by its position within the list, 1 being the first (left-most) element in the list, 2 the second element in the list, and so on. Column numbers could be used if the column to be sorted on is an expression and no AS clause is specified to assign the column a name that can subsequently be referenced. The ORDER BY clause allows the retrieved rows to be ordered in ascending (ASC) or descending (DESC) order on any column or combination of columns, regardless of whether that column appears in the result. However, some dialects insist that the ORDER BY elements appear in the SELECT list. In either case, the ORDER BY clause must always be the last clause of the SELECT statement.

[†]Column numbers are a deprecated feature of the ISO standard and should not be used.

EXAMPLE 6.11 Single-column ordering

Produce a list of salaries for all staff, arranged in descending order of salary.

```
SELECT staffNo, fName, lName, salary
FROM Staff
ORDER BY salary DESC;
```

This example is very similar to Example 6.2. The difference in this case is that the output is to be arranged in descending order of salary. This is achieved by adding the ORDER BY clause to the end of the SELECT statement, specifying salary as the column to be sorted, and DESC to indicate that the order is to be descending. In this case, we get the result table shown in Table 6.11. Note that we could have expressed the ORDER BY clause as: ORDER BY 4 DESC, with the 4 relating to the fourth column name in the SELECT list, namely salary.

TABLE 6.11 Result table for Example 6.11.

staffNo	fName	lName	salary
SL21	John	White	30000.00
SG5	Susan	Brand	24000.00
SG14	David	Ford	18000.00
SG37	Ann	Beech	12000.00
SA9	Mary	Howe	9000.00
SL41	Julie	Lee	9000.00

It is possible to include more than one element in the ORDER BY clause. The **major sort key** determines the overall order of the result table. In Example 6.11, the major sort key is salary. If the values of the major sort key are unique, there is no need for additional keys to control the sort. However, if the values of the major sort key are not unique, there may be multiple rows in the result table with the same value for the major sort key. In this case, it may be desirable to order rows with the same value for the major sort key by some additional sort key. If a second element appears in the ORDER BY clause, it is called a **minor sort key**.

EXAMPLE 6.12 Multiple column ordering

Produce an abbreviated list of properties arranged in order of property type.

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type;
```

In this case we get the result table shown in Table 6.12(a).

There are four flats in this list. As we did not specify any minor sort key, the system arranges these rows in any order it chooses. To arrange the properties in order of rent, we specify a minor order, as follows:

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type, rent DESC;
```

TABLE 6.12(a) Result table for Example 6.12 with one sort key.

propertyNo	type	rooms	rent
PL94	Flat	4	400
PG4	Flat	3	350
PG36	Flat	3	375
PG16	Flat	4	450
PA14	House	6	650
PG21	House	5	600

Now, the result is ordered first by property type, in ascending alphabetic order (ASC being the default setting), and within property type, in descending order of rent. In this case, we get the result table shown in Table 6.12(b).

TABLE 6.12(b) Result table for Example 6.12 with two sort keys.

propertyNo	type	rooms	rent
PG16	Flat	4	450
PL94	Flat	4	400
PG36	Flat	3	375
PG4	Flat	3	350
PA14	House	6	650
PG21	House	5	600

The ISO standard specifies that nulls in a column or expression sorted with ORDER BY should be treated as either less than all nonnull values or greater than all nonnull values. The choice is left to the DBMS implementor.

6.3.3 Using the SQL Aggregate Functions

Esta subsección no

As well as retrieving rows and columns from the database, we often want to perform some form of summation or **aggregation** of data, similar to the totals at the bottom of a report. The ISO standard defines five **aggregate functions**:

- COUNT – returns the number of values in a specified column
- SUM – returns the sum of the values in a specified column
- AVG – returns the average of the values in a specified column
- MIN – returns the smallest value in a specified column
- MAX – returns the largest value in a specified column

These functions operate on a single column of a table and return a single value. COUNT, MIN, and MAX apply to both numeric and nonnumeric fields, but SUM and AVG may be used on numeric fields only. Apart from COUNT(*), each function eliminates nulls first and operates only on the remaining nonnull values. COUNT(*) is a special use of COUNT that counts all the rows of a table, regardless of whether nulls or duplicate values occur.

If we want to eliminate duplicates before the function is applied, we use the keyword DISTINCT before the column name in the function. The ISO standard allows the keyword ALL to be specified if we do not want to eliminate duplicates, although ALL is assumed if nothing is specified. DISTINCT has no effect with the MIN and MAX functions. However, it may have an effect on the result of SUM or AVG, so consideration must be given to whether duplicates should be included or excluded in the computation. In addition, DISTINCT can be specified only once in a query.

It is important to note that an aggregate function can be used only in the SELECT list and in the HAVING clause (see Section 6.3.4). It is incorrect to use it elsewhere. If the SELECT list includes an aggregate function and no GROUP BY clause is being used to group data together (see Section 6.3.4), then no item in the SELECT list can include any reference to a column unless that column is the argument to an aggregate function. For example, the following query is illegal:

```
SELECT staffNo, COUNT(salary)
  FROM Staff;
```

because the query does not have a GROUP BY clause and the column staffNo in the SELECT list is used outside an aggregate function.

EXAMPLE 6.13 Use of COUNT(*)

How many properties cost more than £350 per month to rent?

```
SELECT COUNT(*) AS myCount
  FROM PropertyForRent
 WHERE rent > 350;
```

Restricting the query to properties that cost more than £350 per month is achieved using the WHERE clause. The total number of properties satisfying this condition can then be found by applying the aggregate function COUNT. The result table is shown in Table 6.13.

TABLE 6.13
Result table for
Example 6.13.

myCount
5

TABLE 6.14
Result table for
Example 6.14.

myCount
2

EXAMPLE 6.14 Use of COUNT(DISTINCT)

How many different properties were viewed in May 2008?

```
SELECT COUNT(DISTINCT propertyNo) AS myCount
  FROM Viewing
 WHERE viewDate BETWEEN '1-May-08' AND '31-May-08';
```

Again, restricting the query to viewings that occurred in May 2008 is achieved using the WHERE clause. The total number of viewings satisfying this condition can then be found by applying the aggregate function COUNT. However, as the same property may be viewed many times, we have to use the DISTINCT keyword to eliminate duplicate properties. The result table is shown in Table 6.14.

EXAMPLE 6.15 Use of COUNT and SUM

Find the total number of Managers and the sum of their salaries.

```
SELECT COUNT(staffNo) AS myCount, SUM(salary) AS mySum
FROM Staff
WHERE position = 'Manager';
```

Restricting the query to Managers is achieved using the WHERE clause. The number of Managers and the sum of their salaries can be found by applying the COUNT and the SUM functions respectively to this restricted set. The result table is shown in Table 6.15.

TABLE 6.15 Result table for Example 6.15.

myCount	mySum
2	54000.00

EXAMPLE 6.16 Use of MIN, MAX, AVG

Find the minimum, maximum, and average staff salary.

```
SELECT MIN(salary) AS myMin, MAX(salary) AS myMax, AVG(salary) AS myAvg
FROM Staff;
```

In this example, we wish to consider all staff and therefore do not require a WHERE clause. The required values can be calculated using the MIN, MAX, and AVG functions based on the salary column. The result table is shown in Table 6.16.

TABLE 6.16 Result table for Example 6.16.

myMin	myMax	myAvg
9000.00	30000.00	17000.00

6.3.4 Grouping Results (GROUP BY Clause) Esta subsección no

The previous summary queries are similar to the totals at the bottom of a report. They condense all the detailed data in the report into a single summary row of data. However, it is often useful to have subtotals in reports. We can use the GROUP BY clause of the SELECT statement to do this. A query that includes the GROUP BY clause is called a **grouped query**, because it groups the data from the SELECT table(s) and produces a single summary row for each group. The columns named in the GROUP BY clause are called the **grouping columns**. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is used, each item in the SELECT list must be **single-valued per group**. In addition, the SELECT clause may contain only:

- column names;
- aggregate functions;

- constants;
- an expression involving combinations of these elements.

All column names in the SELECT list must appear in the GROUP BY clause unless the name is used only in an aggregate function. The contrary is not true: there may be column names in the GROUP BY clause that do not appear in the SELECT list. When the WHERE clause is used with GROUP BY, the WHERE clause is applied first, then groups are formed from the remaining rows that satisfy the search condition.

The ISO standard considers two nulls to be equal for purposes of the GROUP BY clause. If two rows have nulls in the same grouping columns and identical values in all the nonnull grouping columns, they are combined into the same group.

EXAMPLE 6.17 Use of GROUP BY

Find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNo, COUNT(staffNo) AS myCount, SUM(salary) AS mySum
  FROM Staff
 GROUP BY branchNo
 ORDER BY branchNo;
```

It is not necessary to include the column names staffNo and salary in the GROUP BY list, because they appear only in the SELECT list within aggregate functions. On the other hand, branchNo is not associated with an aggregate function and so must appear in the GROUP BY list. The result table is shown in Table 6.17.

TABLE 6.17 Result table for Example 6.17.

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00
B007	1	9000.00

CourseSmart

Conceptually, SQL performs the query as follows:

- (1) SQL divides the staff into groups according to their respective branch numbers. Within each group, all staff have the same branch number. In this example, we get three groups:

branchNo	staffNo	salary	COUNT(staffNo)	SUM(salary)
B003	SG37	12000.00	3	54000.00
B003	SG14	18000.00		
B003	SG5	24000.00		
B005	SL21	30000.00	2	39000.00
B005	SL41	9000.00		
B007	SA9	9000.00	1	9000.00

CourseSmart

- (2) For each group, SQL computes the number of staff members and calculates the sum of the values in the salary column to get the total of their salaries. SQL generates a single summary row in the query result for each group.
- (3) Finally, the result is sorted in ascending order of branch number, branchNo.

The SQL standard allows the SELECT list to contain nested queries (see Section 6.3.5). Therefore, we could also express the previous query as:

```
SELECT branchNo, (SELECT COUNT(staffNo) AS myCount
                  FROM Staff s
                 WHERE s.branchNo = b.branchNo),
       (SELECT SUM(salary) AS mySum
          FROM Staff s
         WHERE s.branchNo = b.branchNo)
    FROM Branch b
   ORDER BY branchNo;
```

With this version of the query, however, the two aggregate values are produced for each branch office in Branch; in some cases possibly with zero values.

Restricting groupings (HAVING clause)

The HAVING clause is designed for use with the GROUP BY clause to restrict the groups that appear in the final result table. Although similar in syntax, HAVING and WHERE serve different purposes. The WHERE clause filters individual rows going into the final result table, whereas HAVING filters groups going into the final result table. The ISO standard requires that column names used in the HAVING clause must also appear in the GROUP BY list or be contained within an aggregate function. In practice, the search condition in the HAVING clause always includes at least one aggregate function; otherwise the search condition could be moved to the WHERE clause and applied to individual rows. (Remember that aggregate functions cannot be used in the WHERE clause.)

The HAVING clause is not a necessary part of SQL—any query expressed using a HAVING clause can always be rewritten without the HAVING clause.

EXAMPLE 6.18 Use of HAVING

For each branch office with more than one member of staff, find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNo, COUNT(staffNo) AS myCount, SUM(salary) AS mySum
  FROM Staff
 GROUP BY branchNo
 HAVING COUNT(staffNo) > 1
 ORDER BY branchNo;
```

This is similar to the previous example, with the additional restriction that we want to consider only those groups (that is, branches) with more than one member of staff. This restriction applies to the groups, so the HAVING clause is used. The result table is shown in Table 6.18.

TABLE 6.18 Result table for Example 6.18.

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00

© CourseSmart

6.3.5 Subqueries

In this section we examine the use of a complete SELECT statement embedded within another SELECT statement. The results of this **inner** SELECT statement (or **subselect**) are used in the **outer** statement to help determine the contents of the final result. A sub-select can be used in the WHERE and HAVING clauses of an outer SELECT statement, where it is called a **subquery** or **nested query**. Subselects may also appear in INSERT, UPDATE, and DELETE statements (see Section 6.3.10). There are three types of subquery:

- A *scalar subquery* returns a single column and a single row, that is, a single value. In principle, a scalar subquery can be used whenever a single value is needed. Example 6.19 uses a scalar subquery.
- A *row subquery* returns multiple columns, but only a single row. A row subquery can be used whenever a row value constructor is needed, typically in predicates.
- A *table subquery* returns one or more columns and multiple rows. A table subquery can be used whenever a table is needed, for example, as an operand for the IN predicate.

EXAMPLE 6.19 Using a subquery with equality

List the staff who work in the branch at '163 Main St'.

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo = (SELECT branchNo
                  FROM Branch
                  WHERE street = '163 Main St');
```

The inner SELECT statement (SELECT branchNo FROM Branch . . .) finds the branch number that corresponds to the branch with street name '163 Main St' (there will be only one such branch number, so this is an example of a scalar subquery). Having obtained this branch number, the outer SELECT statement then retrieves the details of all staff who work at this branch. In other words, the inner SELECT returns a result table containing a single value 'B003', corresponding to the branch at '163 Main St', and the outer SELECT becomes:

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo = 'B003';
```

The result table is shown in Table 6.19.

TABLE 6.19 Result table for Example 6.19.

staffNo	fName	lName	position
SG37	Ann	Beech	Assistant
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager

© CourseSmart

We can think of the subquery as producing a temporary table with results that can be accessed and used by the outer statement. A subquery can be used immediately following a relational operator ($=$, $<$, $>$, \leq , \geq , $<>$) in a WHERE clause, or a HAVING clause. The subquery itself is always enclosed in parentheses.

EXAMPLE 6.20 Using a subquery with an aggregate function

List all staff whose salary is greater than the average salary, and show by how much their salary is greater than the average.

```
SELECT staffNo, fName, lName, position,
       salary - (SELECT AVG(salary) FROM Staff) AS salDiff
  FROM Staff
 WHERE salary > (SELECT AVG(salary) FROM Staff);
```

First, note that we cannot write 'WHERE salary > AVG(salary)', because aggregate functions cannot be used in the WHERE clause. Instead, we use a subquery to find the average salary, and then use the outer SELECT statement to find those staff with a salary greater than this average. In other words, the subquery returns the average salary as £17,000. Note also the use of the scalar subquery in the SELECT list to determine the difference from the average salary. The outer query is reduced then to:

```
SELECT staffNo, fName, lName, position, salary - 17000 AS salDiff
  FROM Staff
 WHERE salary > 17000;
```

The result table is shown in Table 6.20.

TABLE 6.20 Result table for Example 6.20.

staffNo	fName	lName	position	salDiff
SL21	John	White	Manager	13000.00
SG14	David	Ford	Supervisor	1000.00
SG5	Susan	Brand	Manager	7000.00

© CourseSmart

The following rules apply to subqueries:

- (1) The ORDER BY clause may not be used in a subquery (although it may be used in the outermost SELECT statement).

- (2) The subquery SELECT list must consist of a single column name or expression, except for subqueries that use the keyword EXISTS (see Section 6.3.8).
- (3) By default, column names in a subquery refer to the table name in the FROM clause of the subquery. It is possible to refer to a table in a FROM clause of an outer query by qualifying the column name (see following).
- (4) When a subquery is one of the two operands involved in a comparison, the subquery must appear on the right-hand side of the comparison. For example, it would be incorrect to express the previous example as:

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE (SELECT AVG(salary) FROM Staff) < salary;
```

because the subquery appears on the left-hand side of the comparison with salary.

EXAMPLE 6.21 Nested subqueries: use of IN

List the properties that are handled by staff who work in the branch at '163 Main St'.

```
SELECT propertyNo, street, city, postcode, type, rooms, rent
FROM PropertyForRent
WHERE staffNo IN (SELECT staffNo
FROM Staff
WHERE branchNo = (SELECT branchNo
FROM Branch
WHERE street = '163 Main St'));
```

Working from the innermost query outwards, the first query selects the number of the branch at '163 Main St'. The second query then selects those staff who work at this branch number. In this case, there may be more than one such row found, and so we cannot use the equality condition (=) in the outermost query. Instead, we use the IN keyword. The outermost query then retrieves the details of the properties that are managed by each member of staff identified in the middle query. The result table is shown in Table 6.21.

TABLE 6.21 Result table for Example 6.21.

propertyNo	street	city	postcode	type	rooms	rent
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375
PG21	18 Dale Rd	Glasgow	G12	House	5	600

6.3.6 ANY and ALL

Esta subsección no

The keywords ANY and ALL may be used with subqueries that produce a single column of numbers. If the subquery is preceded by the keyword ALL, the condition will be true only if it is satisfied by all values produced by the subquery. If the subquery is preceded by the keyword ANY, the condition will be true if it is satisfied by any (one or more) values produced by the subquery. If the subquery is

empty, the ALL condition returns true, the ANY condition returns false. The ISO standard also allows the qualifier SOME to be used in place of ANY.

EXAMPLE 6.22 Use of ANY/SOME

Find all staff whose salary is larger than the salary of at least one member of staff at branch B003.

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary > SOME (SELECT salary
                      FROM Staff
                      WHERE branchNo = 'B003');
```

Although this query can be expressed using a subquery that finds the minimum salary of the staff at branch B003 and then an outer query that finds all staff whose salary is greater than this number (see Example 6.20), an alternative approach uses the SOME/ANY keyword. The inner query produces the set {12000, 18000, 24000} and the outer query selects those staff whose salaries are greater than any of the values in this set (that is, greater than the minimum value, 12000). This alternative method may seem more natural than finding the minimum salary in a subquery. In either case, the result table is shown in Table 6.22.

TABLE 6.22 Result table for Example 6.22.

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG14	David	Ford	Supervisor	18000.00
SG5	Susan	Brand	Manager	24000.00

EXAMPLE 6.23 Use of ALL

Find all staff whose salary is larger than the salary of every member of staff at branch B003.

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary > ALL (SELECT salary
                      FROM Staff
                      WHERE branchNo = 'B003');
```

This example is very similar to the previous example. Again, we could use a subquery to find the maximum salary of staff at branch B003 and then use an outer query to find all staff whose salary is greater than this number. However, in this example we use the ALL keyword. The result table is shown in Table 6.23.

TABLE 6.23 Result table for Example 6.23.

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00



6.3.7 Multi-table Queries

All the examples we have considered so far have a major limitation: the columns that are to appear in the result table must all come from a single table. In many cases, this is insufficient to answer common queries that users will have. To combine columns from several tables into a result table, we need to use a **join** operation. The SQL join operation combines information from two tables by forming pairs of related rows from the two tables. The row pairs that make up the joined table are those where the matching columns in each of the two tables have the same value.

If we need to obtain information from more than one table, the choice is between using a subquery and using a join. If the final result table is to contain columns from different tables, then we must use a join. To perform a join, we simply include more than one table name in the FROM clause, using a comma as a separator, and typically including a WHERE clause to specify the join column(s). It is also possible to use an **alias** for a table named in the FROM clause. In this case, the alias is separated from the table name with a space. An alias can be used to qualify a column name whenever there is ambiguity regarding the source of the column name. It can also be used as a shorthand notation for the table name. If an alias is provided, it can be used anywhere in place of the table name.

EXAMPLE 6.24 Simple join

List the names of all clients who have viewed a property, along with any comments supplied.

```
SELECT c.clientNo, fName, lName, propertyNo, comment
  FROM Client c, Viewing v
 WHERE c.clientNo = v.clientNo; eSmart
```

We want to display the details from both the Client table and the Viewing table, and so we have to use a join. The SELECT clause lists the columns to be displayed. Note that it is necessary to qualify the client number, clientNo, in the SELECT list: clientNo could come from either table, and we have to indicate which one. (We could also have chosen the clientNo column from the Viewing table.) The qualification is achieved by prefixing the column name with the appropriate table name (or its alias). In this case, we have used *c* as the alias for the Client table.

To obtain the required rows, we include those rows from both tables that have identical values in the clientNo columns, using the search condition (*c.clientNo = v.clientNo*). We call these two columns the **matching columns** for the two tables. This is equivalent to the relational algebra Equijoin operation discussed in Section 5.1.3. The result table is shown in Table 6.24.

TABLE 6.24 Result table for Example 6.24.

clientNo	fName	lName	propertyNo	comment
CR56	Aline	Stewart	PG36	
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR62	Mary	Tregear	PA14	no dining room
CR76	John	Kay	PG4	too remote

The most common multi-table queries involve two tables that have a one-to-many (1:*) (or a parent/child) relationship (see Section 12.6.2). The previous query involving clients and viewings is an example of such a query. Each viewing (child) has an associated client (parent), and each client (parent) can have many associated viewings (children). The pairs of rows that generate the query results are parent/child row combinations. In Section 4.2.5 we described how primary key and foreign keys create the parent/child relationship in a relational database: the table containing the primary key is the parent table and the table containing the foreign key is the child table. To use the parent/child relationship in an SQL query, we specify a search condition that compares the primary key and the foreign key. In Example 6.24, we compared the primary key in the Client table, c.clientNo, with the foreign key in the Viewing table, v.clientNo.

The SQL standard provides the following alternative ways to specify this join:

```
FROM Client c JOIN Viewing v ON c.clientNo = v.clientNo
FROM Client JOIN Viewing USING clientNo
FROM Client NATURAL JOIN Viewing
```

In each case, the FROM clause replaces the original FROM and WHERE clauses. However, the first alternative produces a table with two identical clientNo columns; the remaining two produce a table with a single clientNo column.

EXAMPLE 6.25 Sorting a join

For each branch office, list the staff numbers and names of staff who manage properties and the properties that they manage.

```
SELECT s.branchNo, s.staffNo, fName, lName, propertyNo
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
ORDER BY s.branchNo, s.staffNo, propertyNo;
```

In this example, we need to join the Staff and PropertyForRent tables, based on the primary key/foreign key attribute (staffNo). To make the results more readable, we have ordered the output using the branch number as the major sort key and the staff number and property number as the minor keys. The result table is shown in Table 6.25.

TABLE 6.25 Result table for Example 6.25.

branchNo	staffNo	fName	lName	propertyNo
B003	SG14	David	Ford	PG16
B003	SG37	Ann	Beech	PG21
B003	SG37	Ann	Beech	PG36
B005	SL41	Julie	Lee	PL94
B007	SA9	Mary	Howe	PA14

EXAMPLE 6.26 Three-table join

For each branch, list the staff numbers and names of staff who manage properties, including the city in which the branch is located and the properties that the staff manage.

```
SELECT b.branchNo, b.city, s.staffNo, fName, lName, propertyNo
FROM Branch b, Staff s, PropertyForRent p
WHERE b.branchNo = s.branchNo AND s.staffNo = p.staffNo
ORDER BY b.branchNo, s.staffNo, propertyNo;
```

The result table requires columns from three tables: Branch (branchNo and city), Staff (staffNo, fName and lName), and PropertyForRent (propertyNo), so a join must be used. The Branch and Staff details are joined using the condition (b.branchNo = s.branchNo) to link each branch to the staff who work there. The Staff and PropertyForRent details are joined using the condition (s.staffNo = p.staffNo) to link staff to the properties they manage. The result table is shown in Table 6.26.

TABLE 6.26 Result table for Example 6.26.

branchNo	city	staffNo	fName	lName	propertyNo
B003	Glasgow	SG14	David	Ford	PG16
B003	Glasgow	SG37	Ann	Beech	PG21
B003	Glasgow	SG37	Ann	Beech	PG36
B005	London	SL41	Julie	Lee	PL94
B007	Aberdeen	SA9	Mary	Howe	PA14

Note again that the SQL standard provides alternative formulations for the FROM and WHERE clauses, for example:

```
FROM (Branch b JOIN Staff s USING branchNo) AS bs
JOIN PropertyForRent p USING staffNo
```

EXAMPLE 6.27 Multiple grouping columns

Find the number of properties handled by each staff member, along with the branch number of the member of staff.

```
SELECT s.branchNo, s.staffNo, COUNT(*) AS myCount
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo
ORDER BY s.branchNo, s.staffNo;
```

To list the required numbers, we first need to find out which staff actually manage properties. This can be found by joining the Staff and PropertyForRent tables on the staffNo column, using the FROM/WHERE clauses. Next, we need to form groups consisting of the branch number and staff number, using the GROUP BY clause. Finally, we sort the output using the ORDER BY clause. The result table is shown in Table 6.27(a).

TABLE 6.27(a) Result table for Example 6.27.

branchNo	staffNo	myCount
B003	SG14	1
B003	SG37	2
B005	SL41	1
B007	SA9	1

@ CourseSmart

Computing a join

A join is a subset of a more general combination of two tables known as the **Cartesian product** (see Section 5.1.2). The Cartesian product of two tables is another table consisting of all possible pairs of rows from the two tables. The columns of the product table are all the columns of the first table followed by all the columns of the second table. If we specify a two-table query without a WHERE clause, SQL produces the Cartesian product of the two tables as the query result. In fact, the ISO standard provides a special form of the SELECT statement for the Cartesian product:

```
SELECT [DISTINCT | ALL] {*} | columnList
FROM TableName1 CROSS JOIN TableName2
```

Consider again Example 6.24, where we joined the Client and Viewing tables using the matching column, clientNo. Using the data from Figure 4.3, the Cartesian product of these two tables would contain 20 rows (4 clients * 5 viewings = 20 rows). It is equivalent to the query used in Example 6.24 without the WHERE clause.

Conceptually, the procedure for generating the results of a SELECT with a join is as follows:

- (1) Form the Cartesian product of the tables named in the FROM clause.
- (2) If there is a WHERE clause, apply the search condition to each row of the product table, retaining those rows that satisfy the condition. In terms of the relational algebra, this operation yields a **restriction** of the Cartesian product.
- (3) For each remaining row, determine the value of each item in the SELECT list to produce a single row in the result table.
- (4) If SELECT DISTINCT has been specified, eliminate any duplicate rows from the result table. In the relational algebra, Steps 3 and 4 are equivalent to a **projection** of the restriction over the columns mentioned in the SELECT list.
- (5) If there is an ORDER BY clause, sort the result table as required.

We will discuss query processing in more detail in Chapter 23.

Outer joins Esta subsección no

The join operation combines data from two tables by forming pairs of related rows where the matching columns in each table have the same value. If one row of a

table is unmatched, the row is omitted from the result table. This has been the case for the joins we examined earlier. The ISO standard provides another set of join operators called **outer joins** (see Section 5.1.3). The Outer join retains rows that do not satisfy the join condition. To understand the Outer join operators, consider the following two simplified Branch and PropertyForRent tables, which we refer to as Branch1 and PropertyForRent1, respectively:

Branch1		PropertyForRent1	
branchNo	bCity	propertyNo	pCity
B003	Glasgow	PA14	Aberdeen
B004	Bristol	PL94	London
B002	London	PG4	Glasgow

The (Inner) join of these two tables:

```
© CourseSmart
SELECT b.*, p.*
FROM Branch1 b, PropertyForRent1 p
WHERE b.bCity = p.pCity;
```

produces the result table shown in Table 6.27(b).

© CourseSmart

TABLE 6.27(b) Result table for inner join of the Branch1 and PropertyForRent1 tables.

branchNo	bCity	propertyNo	pCity
B003	Glasgow	PG4	Glasgow
B002	London	PL94	London

The result table has two rows where the cities are the same. In particular, note that there is no row corresponding to the branch office in Bristol and there is no row corresponding to the property in Aberdeen. If we want to include the unmatched rows in the result table, we can use an Outer join. There are three types of Outer join: **Left**, **Right**, and **Full Outer** joins. We illustrate their functionality in the following examples.

EXAMPLE 6.28 Left Outer join

List all branch offices and any properties that are in the same city.

The Left Outer join of these two tables:

```
SELECT b.*, p.*
FROM Branch1 b LEFT JOIN PropertyForRent1 p ON b.bCity = p.pCity;
```

produces the result table shown in Table 6.28. In this example the Left Outer join includes not only those rows that have the same city, but also those rows of the first (left) table that are unmatched with rows from the second (right) table. The columns from the second table are filled with NULLs.

TABLE 6.28 Result table for Example 6.28.

branchNo	bCity	propertyNo	pCity
B003	Glasgow	PG4	Glasgow
B004	Bristol	NULL	NULL
B002	London	PL94	London

EXAMPLE 6.29 Right Outer join

List all properties and any branch offices that are in the same city.

The Right Outer join of these two tables:

```
SELECT b.*, p.*  
FROM Branch1 b RIGHT JOIN PropertyForRent1 p ON b.bCity = p.pCity;
```

produces the result table shown in Table 6.29. In this example, the Right Outer join includes not only those rows that have the same city, but also those rows of the second (right) table that are unmatched with rows from the first (left) table. The columns from the first table are filled with NULLs.

TABLE 6.29 Result table for Example 6.29.

branchNo	bCity	propertyNo	pCity
NULL	NULL	PA14	Aberdeen
B003	Glasgow	PG4	Glasgow
B002	London	PL94	London

EXAMPLE 6.30 Full Outer join

List the branch offices and properties that are in the same city along with any unmatched branches or properties.

The Full Outer join of these two tables:

```
SELECT b.*, p.*  
FROM Branch1 b FULL JOIN PropertyForRent1 p ON b.bCity = p.pCity;
```

produces the result table shown in Table 6.30. In this case, the Full Outer join includes not only those rows that have the same city, but also those rows that are unmatched in both tables. The unmatched columns are filled with NULLs.

TABLE 6.30 Result table for Example 6.30.

branchNo	bCity	propertyNo	pCity
NULL	NULL	PA14	Aberdeen
B003	Glasgow	PG4	Glasgow
B004	Bristol	NULL	NULL
B002	London	PL94	London

6.3.8 EXISTS and NOT EXISTS

The keywords EXISTS and NOT EXISTS are designed for use only with subqueries. They produce a simple true/false result. EXISTS is true if and only if there exists at least one row in the result table returned by the subquery; it is false if the subquery returns an empty result table. NOT EXISTS is the opposite of EXISTS. Because EXISTS and NOT EXISTS check only for the existence or nonexistence of rows in the subquery result table, the subquery can contain any number of columns. For simplicity, it is common for subqueries following one of these keywords to be of the form:

(SELECT * FROM . . .)

EXAMPLE 6.31 Query using EXISTS

Find all staff who work in a London branch office.

```
SELECT staffNo, fName, lName, position
FROM Staff s
WHERE EXISTS (SELECT *
               FROM Branch b
               WHERE s.branchNo = b.branchNo AND city = 'London');
```

This query could be rephrased as ‘Find all staff such that there exists a Branch row containing his/her branch number, branchNo, and the branch city equal to London’. The test for inclusion is the existence of such a row. If it exists, the subquery evaluates to true. The result table is shown in Table 6.31.

TABLE 6.31 Result table for Example 6.31.

staffNo	fName	lName	position
SL21	John	White	Manager
SL41	Julie	Lee	Assistant

Note that the first part of the search condition `s.branchNo = b.branchNo` is necessary to ensure that we consider the correct branch row for each member of staff. If we omitted this part of the query, we would get all staff rows listed out because the subquery (`SELECT * FROM Branch WHERE city = 'London'`) would always be true and the query would be reduced to:

```
SELECT staffNo, fName, lName, position FROM Staff WHERE true;
```

which is equivalent to:

```
SELECT staffNo, fName, lName, position FROM Staff;
```

We could also have written this query using the join construct:

```
SELECT staffNo, fName, lName, position  
FROM Staff s, Branch b  
WHERE s.branchNo = b.branchNo AND city = 'London';
```

6.3.9 Combining Result Tables (UNION, INTERSECT, EXCEPT)

Esta subsección no

In SQL, we can use the normal set operations of *Union*, *Intersection*, and *Difference* to combine the results of two or more queries into a single result table:

- The **Union** of two tables, A and B, is a table containing all rows that are in either the first table A or the second table B or both.
- The **Intersection** of two tables, A and B, is a table containing all rows that are common to both tables A and B.
- The **Difference** of two tables, A and B, is a table containing all rows that are in table A but are not in table B.

The set operations are illustrated in Figure 6.1. There are restrictions on the tables that can be combined using the set operations, the most important one being that the two tables have to be **union-compatible**; that is, they have the same structure. This implies that the two tables must contain the same number of columns, and that their corresponding columns have the same data types and lengths. It is the user's responsibility to ensure that data values in corresponding columns come from the same *domain*. For example, it would not be sensible to combine a column containing the age of staff with the number of rooms in a property, even though both columns may have the same data type: for example, `SMLINT`.

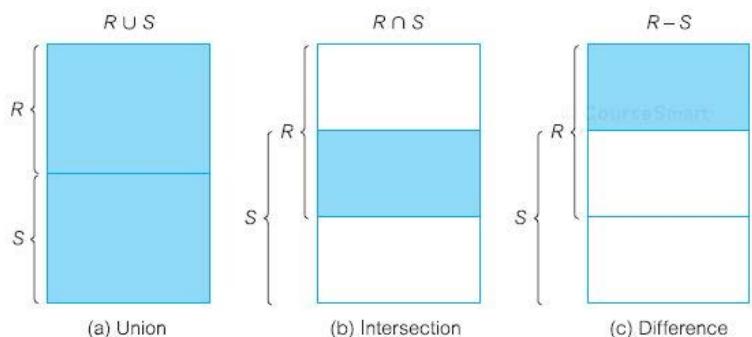


Figure 6.1
Union,
Intersection,
and
Difference set
operations.

The three set operators in the ISO standard are called UNION, INTERSECT, and EXCEPT. The format of the set operator clause in each case is:

operator [ALL] [CORRESPONDING [BY {column1 [, . . .]}]]

If CORRESPONDING BY is specified, then the set operation is performed on the named column(s); if CORRESPONDING is specified but not the BY clause, the set operation is performed on the columns that are common to both tables. If ALL is specified, the result can include duplicate rows. Some dialects of SQL do not support INTERSECT and EXCEPT; others use MINUS in place of EXCEPT.

EXAMPLE 6.32 Use of UNION

Construct a list of all cities where there is either a branch office or a property.

TABLE 6.32

Result table for Example 6.32.

city

London

Glasgow

Aberdeen

Bristol

```
(SELECT city          or (SELECT *
   FROM Branch        FROM Branch
   WHERE city IS NOT NULL) WHERE city IS NOT NULL)
UNION
(SELECT city          UNION CORRESPONDING BY city
   FROM PropertyForRent (SELECT *
   WHERE city IS NOT NULL) WHERE city IS NOT NULL);
   
```

This query is executed by producing a result table from the first query and a result table from the second query, and then merging both tables into a single result table consisting of all the rows from both result tables with the duplicate rows removed. The final result table is shown in Table 6.32.

EXAMPLE 6.33 Use of INTERSECT

Construct a list of all cities where there is both a branch office and a property.

TABLE 6.33

Result table for Example 6.33.

city

Aberdeen

Glasgow

London

```
(SELECT city          or (SELECT *
   FROM Branch        FROM Branch)
   INTERSECT          INTERSECT CORRESPONDING BY city
   (SELECT city        (SELECT *
   FROM PropertyForRent) FROM PropertyForRent);
   
```

This query is executed by producing a result table from the first query and a result table from the second query, and then creating a single result table consisting of those rows that are common to both result tables. The final result table is shown in Table 6.33.

We could rewrite this query without the INTERSECT operator, for example:

```
SELECT DISTINCT b.city      or SELECT DISTINCT city
   FROM Branch b, PropertyForRent p   FROM Branch b
   WHERE b.city = p.city;           WHERE EXISTS (SELECT *
                                                 FROM PropertyForRent p
                                                 WHERE b.city = p.city);
   
```

The ability to write a query in several equivalent forms illustrates one of the disadvantages of the SQL language.

EXAMPLE 6.34 Use of EXCEPT

Construct a list of all cities where there is a branch office but no properties.

```
(SELECT city          or (SELECT *
  FROM Branch)      FROM Branch)
EXCEPT              EXCEPT CORRESPONDING BY city
(SELECT city          (SELECT *
  FROM PropertyForRent);   FROM PropertyForRent);
```

This query is executed by producing a result table from the first query and a result table from the second query, and then creating a single result table consisting of those rows that appear in the first result table but not in the second one. The final result table is shown in Table 6.34.

We could rewrite this query without the EXCEPT operator, for example:

```
SELECT DISTINCT city      or SELECT DISTINCT city
  FROM Branch           FROM Branch b
 WHERE city NOT IN (SELECT city
                      WHERE NOT EXISTS
                        (SELECT *
                         FROM PropertyForRent p
                         WHERE b.city = p.city);
                      FROM PropertyForRent);
```

TABLE 6.34
Result table for Example 6.34.

city
Bristol

6.3.10 Database Updates

SQL is a complete data manipulation language that can be used for modifying the data in the database as well as querying the database. The commands for modifying the database are not as complex as the SELECT statement. In this section, we describe the three SQL statements that are available to modify the contents of the tables in the database:

- INSERT: adds new rows of data to a table
- UPDATE: modifies existing data in a table
- DELETE: removes rows of data from a table

Adding data to the database (INSERT)

There are two forms of the INSERT statement. The first allows a single row to be inserted into a named table and has the following format:

```
INSERT INTO TableName [(columnList)]
VALUES (dataValueList)
```

TableName may be either a base table or an updatable view (see Section 7.4), and *columnList* represents a list of one or more column names separated by commas. The *columnList* is optional; if omitted, SQL assumes a list of all columns in their original CREATE TABLE order. If specified, then any columns that are omitted from the list must have been declared as NULL columns when the table was created,

unless the DEFAULT option was used when creating the column (see Section 7.3.2). The *dataValueList* must match the *columnList* as follows:

- The number of items in each list must be the same.
- There must be a direct correspondence in the position of items in the two lists, so that the first item in *dataValueList* applies to the first item in *columnList*, the second item in *dataValueList* applies to the second item in *columnList*, and so on.
- The data type of each item in *dataValueList* must be compatible with the data type of the corresponding column.

EXAMPLE 6.35 INSERT . . . VALUES

Insert a new row into the Staff table supplying data for all columns.

```
INSERT INTO Staff
VALUES ('SG16', 'Alan', 'Brown', 'Assistant', 'M', DATE '1957-05-25', 8300, 'B003');
```

As we are inserting data into each column in the order the table was created, there is no need to specify a column list. Note that character literals such as 'Alan' must be enclosed in single quotes.

EXAMPLE 6.36 INSERT using defaults

Insert a new row into the Staff table supplying data for all mandatory columns: staffNo, fName, lName, position, salary, and branchNo.

```
INSERT INTO Staff (staffNo, fName, lName, position, salary, branchNo)
VALUES ('SG44', 'Anne', 'Jones', 'Assistant', 8100, 'B003');
```

As we are inserting data into only certain columns, we must specify the names of the columns that we are inserting data into. The order for the column names is not significant, but it is more normal to specify them in the order they appear in the table. We could also express the INSERT statement as:

```
INSERT INTO Staff
VALUES ('SG44', 'Anne', 'Jones', 'Assistant', NULL, NULL, 8100, 'B003');
```

In this case, we have explicitly specified that the columns sex and DOB should be set to NULL.

The second form of the INSERT statement allows multiple rows to be copied from one or more tables to another, and has the following format:

```
INSERT INTO TableName [(columnList)]
SELECT . . .
```

TableName and *columnList* are defined as before when inserting a single row. The SELECT clause can be any valid SELECT statement. The rows inserted into the named table are identical to the result table produced by the subselect. The same restrictions that apply to the first form of the INSERT statement also apply here.

EXAMPLE 6.37 INSERT . . . SELECT

Assume that there is a table StaffPropCount that contains the names of staff and the number of properties they manage:

```
StaffPropCount(staffNo, fName, lName, propCount)
```

Populate the StaffPropCount table using details from the Staff and PropertyForRent tables.

```
INSERT INTO StaffPropCount
(SELECT s.staffNo, fName, lName, COUNT(*)
 FROM Staff s, PropertyForRent p
 WHERE s.staffNo = p.staffNo
 GROUP BY s.staffNo, fName, lName)
 UNION
 (SELECT staffNo, fName, lName, 0
 FROM Staff s
 WHERE NOT EXISTS (SELECT *
                   FROM PropertyForRent p
                   WHERE p.staffNo = s.staffNo));
```

This example is complex, because we want to count the number of properties that staff manage. If we omit the second part of the UNION, then we get a list of only those staff who currently manage at least one property; in other words, we exclude those staff who currently do not manage any properties. Therefore, to include the staff who do not manage any properties, we need to use the UNION statement and include a second SELECT statement to add in such staff, using a 0 value for the count attribute. The StaffPropCount table will now be as shown in Table 6.35.

Note that some dialects of SQL may not allow the use of the UNION operator within a subselect for an INSERT.

TABLE 6.35 Result table for Example 6.37.

staffNo	fName	lName	propCount
SG14	David	Ford	1
SL21	John	White	0
SG37	Ann	Beech	2
SA9	Mary	Howe	1
SG5	Susan	Brand	0
SL41	Julie	Lee	1

Modifying data in the database (UPDATE)

The UPDATE statement allows the contents of existing rows in a named table to be changed. The format of the command is:

```
UPDATE TableName
SET columnName1 = dataValue1 [, columnName2 = dataValue2 . . . ]
[WHERE searchCondition]
```

TableName can be the name of a base table or an updatable view (see Section 7.4). The SET clause specifies the names of one or more columns that are to be updated. The WHERE clause is optional; if omitted, the named columns are updated for *all* rows in the table. If a WHERE clause is specified, only those rows that satisfy the *searchCondition* are updated. The new *dataValue(s)* must be compatible with the data type(s) for the corresponding column(s).

EXAMPLE 6.38 UPDATE all rows

Give all staff a 3% pay increase.

```
UPDATE Staff
SET salary = salary*1.03;
```

As the update applies to all rows in the Staff table, the WHERE clause is omitted.

EXAMPLE 6.39 UPDATE specific rows

Give all Managers a 5% pay increase.

```
UPDATE Staff
SET salary = salary*1.05
WHERE position = 'Manager';
```

The WHERE clause finds the rows that contain data for Managers and the update `salary = salary*1.05` is applied only to these particular rows.

EXAMPLE 6.40 UPDATE multiple columns

Promote David Ford (`staffNo = 'SG14'`) to Manager and change his salary to £18,000.

```
UPDATE Staff
SET position = 'Manager', salary = 18000
WHERE staffNo = 'SG14';
```

Deleting data from the database (DELETE)

The DELETE statement allows rows to be deleted from a named table. The format of the command is:

```
DELETE FROM TableName
[WHERE searchCondition]
```

As with the INSERT and UPDATE statements, *TableName* can be the name of a base table or an updatable view (see Section 7.4). The *searchCondition* is optional; if omitted, *all* rows are deleted from the table. This does not delete the table itself—to delete the table contents and the table definition, the DROP TABLE statement must be used instead (see Section 7.3.3). If a *searchCondition* is specified, only those rows that satisfy the condition are deleted.

EXAMPLE 6.41 DELETE specific rows

Delete all viewings that relate to property PG4.

```
DELETE FROM Viewing
WHERE propertyNo = 'PG4';
```

The WHERE clause finds the rows for property PG4 and the delete operation is applied only to these particular rows.

EXAMPLE 6.42 DELETE all rows

Delete all rows from the Viewing table.

```
DELETE FROM Viewing;
```

No WHERE clause has been specified, so the delete operation applies to all rows in the table. This query removes all rows from the table, leaving only the table definition, so that we are still able to insert data into the table at a later stage.

Chapter Summary

- SQL is a nonprocedural language consisting of standard English words such as SELECT, INSERT, and DELETE that can be used by professionals and non-professionals alike. It is both the formal and de facto standard language for defining and manipulating relational databases.
- The **SELECT** statement is the most important statement in the language and is used to express a query. It combines the three fundamental relational algebra operations of *Selection*, *Projection*, and *Join*. Every SELECT statement produces a query result table consisting of one or more columns and zero or more rows.
- The SELECT clause identifies the columns and/or calculated data to appear in the result table. All column names that appear in the SELECT clause must have their corresponding tables or views listed in the FROM clause.
- The WHERE clause selects rows to be included in the result table by applying a search condition to the rows of the named table(s). The ORDER BY clause allows the result table to be sorted on the values in one or more columns. Each column can be sorted in ascending or descending order. If specified, the ORDER BY clause must be the last clause in the SELECT statement.
- SQL supports five aggregate functions (COUNT, SUM, AVG, MIN, and MAX) that take an entire column as an argument and compute a single value as the result. It is illegal to mix aggregate functions with column names in a SELECT clause, unless the GROUP BY clause is used.
- The GROUP BY clause allows summary information to be included in the result table. Rows that have the same value for one or more columns can be grouped together and treated as a unit for using the aggregate functions. In this case, the aggregate functions take each group as an argument and compute a single value for each group as the result. The HAVING clause acts as a WHERE clause for groups, restricting the groups that appear in the final result table. However, unlike the WHERE clause, the HAVING clause can include aggregate functions.
- A **subselect** is a complete SELECT statement embedded in another query. A subselect may appear within the WHERE or HAVING clauses of an outer SELECT statement, where it is called a **subquery** or **nested query**. Conceptually, a subquery produces a temporary table whose contents can be accessed by the outer query. A subquery can be embedded in another subquery.