CHAPTER

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.
- 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.

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 Chapter 9 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:2011 in Chapter 9.

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, 2011a).

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.

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 Chapter 9. There have been further releases of the standard in late 2003 (SQL:2003), in summer 2008 (SQL:2008), and in late 2011 (SQL:2011).

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 three 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 following 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	(branchNo, street, city, postcode)
Staff	(staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent	(propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)
Client	(clientNo, fName, lName, telNo, prefType, maxRent, eMail)
PrivateOwner	(ownerNo, fName, lName, address, telNo, eMail, password)
Viewing	(clientNo, 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

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-13

6.3.2 Sorting Results (ORDER BY Clause)

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 (leftmost) 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

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

EXAMPLE 6.14 Use of COUNT(DISTINCT)

How many different properties were viewed in May 2013?

```
SELECT COUNT(DISTINCT propertyNo) AS myCount
FROM Viewing
WHERE viewDate BETWEEN '1-May-13' AND '31-May-13';
```

Again, restricting the query to viewings that occurred in May 2013 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.

TABLE 6.14 Result table for Example 6.14.

myCount
2

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)

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

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
B003	SG37	12000.00
B003	SG14	18000.00
B003	SG5	24000.00
B005	SL21	30000.00
B005	SL41	9000.00
B007	SA9	9000.00

COUNT(staffNo)	SUM(salary)
3	54000.00
2	39000.00
1	9000.00

(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

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

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 (=, <, >, <=, > =,< >) 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

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

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;
```

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

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

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

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent1

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

The (Inner) join of these two tables:

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

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

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.branFchNo = b.branchNo AND city = 'London';
```

6.3.9 Combining Result Tables (UNION, INTERSECT, EXCEPT)

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, SMALLINT.

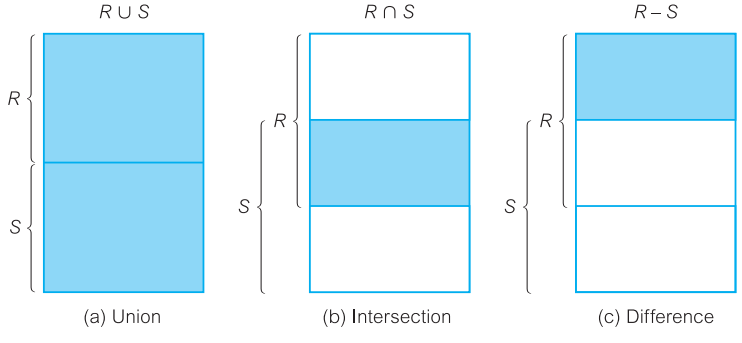

Figure 6.1 Union, Intersection, and Difference set operations.

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.

```
(SELECT city
 FROM Branch
 WHERE city IS NOT NULL)
 UNION
(SELECT city
 FROM PropertyForRent
 WHERE city IS NOT NULL);
```

or

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

TABLE 6.32 Result table for Example 6.32.

city
London
Glasgow
Aberdeen
Bristol

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.

```
(SELECT city
 FROM Branch)
 INTERSECT
(SELECT city
 FROM PropertyForRent);
```

or

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

TABLE 6.33 Result table for Example 6.33.

city
Aberdeen
Glasgow
London

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
FROM Branch b, PropertyForRent p
WHERE b.city = p.city;
```

or

```
SELECT DISTINCT city
   FROM Branch b
   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
 FROM Branch)
 EXCEPT
(SELECT city
 FROM PropertyForRent);
```

or

```
(SELECT *
 FROM Branch)
 EXCEPT CORRESPONDING BY city
(SELECT *
 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.

TABLE 6.34 Result table for Example 6.34.

city
Bristol

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

```
SELECT DISTINCT city
FROM Branch
WHERE city NOT IN (SELECT city
                   FROM PropertyForRent);
```

or

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

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.

- There are three types of subquery: **scalar**, **row**, and **table**. 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. 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.
- If the columns of the result table come from more than one table, a **join** must be used, by specifying more than one table in the FROM clause and typically including a WHERE clause to specify the join column(s). The ISO standard allows **Outer joins** to be defined. It also allows the set operations of *Union, Intersection,* and *Difference* to be used with the **UNION**, **INTERSECT**, and **EXCEPT** commands.
- As well as SELECT, the SQL DML includes the **INSERT** statement to insert a single row of data into a named table or to insert an arbitrary number of rows from one or more other tables using a **subselect**; the **UPDATE** statement to update one or more values in a specified column or columns of a named table; the **DELETE** statement to delete one or more rows from a named table.

Review Questions

6.1 Briefly describe the four basic SQL DML statements and explain their use.

6.2 Explain the importance and application of the WHERE clause in the UPDATE and DELETE statements.

6.3 Explain the function of each of the clauses in the SELECT statement. What restrictions are imposed on these clauses?

6.4 What restrictions apply to the use of the aggregate functions within the SELECT statement? How do nulls affect the aggregate functions?

6.5 How can results from two SQL queries be combined? Differentiate how the INTERSECT and EXCEPT commands work.

6.6 Differentiate between the three types of subqueries. Why is it important to understand the nature of subquery result before you write an SQL statement?

Exercises

For Exercises 6.7–6.28, use the Hotel schema defined at the start of the Exercises at the end of Chapter 4.

Simple queries

6.7 List full details of all hotels.

6.8 List full details of all hotels in London.

6.9 List the names and addresses of all guests living in London, alphabetically ordered by name.

6.10 List all double or family rooms with a price below £40.00 per night, in ascending order of price.

6.11 List the bookings for which no dateTo has been specified.

Aggregate functions

6.12 How many hotels are there?

6.13 What is the average price of a room?

6.14 What is the total revenue per night from all double rooms?

6.15 How many different guests have made bookings for August?

Subqueries and joins

6.16 List the price and type of all rooms at the Grosvenor Hotel.

6.17 List all guests currently staying at the Grosvenor Hotel.

6.18 List the details of all rooms at the Grosvenor Hotel, including the name of the guest staying in the room, if the room is occupied.

6.19 What is the total income from bookings for the Grosvenor Hotel today?

6.20 List the rooms that are currently unoccupied at the Grosvenor Hotel.

6.21 What is the lost income from unoccupied rooms at the Grosvenor Hotel?

Grouping

6.22 List the number of rooms in each hotel.

6.23 List the number of rooms in each hotel in London.

6.24 What is the average number of bookings for each hotel in August?

6.25 What is the most commonly booked room type for each hotel in London?

6.26 What is the lost income from unoccupied rooms at each hotel today?

Populating tables

6.27 Insert rows into each of these tables.

6.28 Update the price of all rooms by 5%.

General

6.29 Investigate the SQL dialect on any DBMS that you are currently using. Determine the system's compliance with the DML statements of the ISO standard. Investigate the functionality of any extensions that the DBMS supports. Are there any functions not supported?

6.30 Demonstrate that queries written using the UNION operator can be rewritten using the OR operator to produce the same result.

6.31 Apply the syntax for inserting data into a table.

Case Study 2

For Exercises 6.32–6.40, use the Projects schema defined in the Exercises at the end of Chapter 5.

6.32 List all employees from BRICS countries in alphabetical order of surname.

6.33 List all the details of employees born between 1980–90.

6.34 List all managers who are female in alphabetical order of surname, and then first name.

6.35 Remove all projects that are managed by the planning department.

6.36 Assume the planning department is going to be merged with the IT department. Update employee records to reflect the proposed change.

6.37 Using the UNION command, list all projects that are managed by the IT and the HR department.

6.38 Produce a report of the total hours worked by each female employee, arranged by department number and alphabetically by employee surname within each department.

6.39 Remove all project from the database which had no employees worked..

6.40 List the total number of employees in each department for those departments with more than 10 employees. Create an appropriate heading for the columns of the results table.

Case Study 3

For Exercises 6.41–6.54, use the Library schema defined in the Exercises at the end of Chapter 5.

6.41 List all book titles.

6.42 List all borrower details.

6.43 List all books titles published between 2010 and 2014.

6.44 Remove all books published before 1950 from the database.

6.45 List all book titles that have never been borrowed by any borrower.

6.46 List all book titles that contain the word 'database' and are available for loan.

6.47 List the names of borrowers with overdue books.

6.48 How many copies of each book title are there?

6.49 How many copies of ISBN "0-321-52306-7" are currently available?

6.50 How many times has the book title with ISBN "0-321-52306-7" been borrowed?

6.51 Produce a report of book titles that have been borrowed by "Peter Bloomfield."

6.52 For each book title with more than three copies, list the names of library members who have borrowed them.

6.53 Produce a report with the details of borrowers who currently have books overdue.

6.54 Produce a report detailing how many times each book title has been borrowed.

CHAPTER

7 SQL: Data Definition

Chapter Objectives

In this chapter you will learn:

- The data types supported by the SQL standard.
- The purpose of the integrity enhancement feature of SQL.
- How to define integrity constraints using SQL, including:
  - required data;
  - domain constraints;
  - entity integrity;
  - referential integrity;
  - general constraints.
- How to use the integrity enhancement feature in the CREATE and ALTER TABLE statements.
- The purpose of views.
- How to create and delete views using SQL.
- How the DBMS performs operations on views.
- Under what conditions views are updatable.
- The advantages and disadvantages of views.
- How the ISO transaction model works.
- How to use the GRANT and REVOKE statements as a level of security.

In the previous chapter we discussed in some detail SQL and, in particular, the SQL data manipulation facilities. In this chapter we continue our presentation of SQL and examine the main SQL data definition facilities.

Structure of this Chapter In Section 7.1 we examine the ISO SQL data types. The 1989 ISO standard introduced an Integrity Enhancement Feature (IEF), which provides facilities for defining referential integrity and other constraints (ISO, 1989). Prior to this standard, it was the responsibility of each application program to ensure compliance with these constraints. The provision of an IEF greatly enhances the functionality of SQL and allows constraint checking to be centralized and standardized. We consider the IEF in Section 7.2 and the main SQL data definition facilities in Section 7.3.

In Section 7.4 we show how views can be created using SQL, and how the DBMS converts operations on views into equivalent operations on the base tables. We also discuss the restrictions that the ISO SQL standard places on views in order for them to be updatable. In Section 7.5, we briefly describe the ISO SQL transaction model.

Views provide a certain degree of database security. SQL also provides a separate access control subsystem, containing facilities to allow users to share database objects or, alternatively, to restrict access to database objects. We discuss the access control subsystem in Section 7.6.

In Chapter 9 we examine in some detail the features that have recently 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 until recently were not available in SQL. As in the previous chapter, we present the features of SQL using examples drawn from the *DreamHome* case study. We use the same notation for specifying the format of SQL statements as defined in Section 6.2.

7.1 The ISO SQL Data Types

In this section we introduce the data types defined in the SQL standard. We start by defining what constitutes a valid identifier in SQL.

7.1.1 SQL Identifiers

SQL identifiers are used to identify objects in the database, such as table names, view names, and columns. The characters that can be used in a user-defined SQL identifier must appear in a **character set**. The ISO standard provides a default character set, which consists of the uppercase letters A . . . Z, the lowercase letters a . . . z, the digits 0 . . . 9, and the underscore (_) character. It is also possible to specify an alternative character set. The following restrictions are imposed on an identifier:

- an identifier can be no longer than 128 characters (most dialects have a much lower limit than this);
- an identifier must start with a letter;
- an identifier cannot contain spaces.

TABLE 7.1 ISO SQL data types.

DATA TYPE	DECLARATIONS				
boolean	BOOLEAN				
character	CHAR	VARCHAR			
bit†	BIT	BIT VARYING			
exact numeric	NUMERIC	DECIMAL	INTEGER	SMALLINT	BIGINT
approximate numeric	FLOAT	REAL	DOUBLE PRECISION		
datetime	DATE	TIME	TIMESTAMP		
interval	INTERVAL				
large objects	CHARACTER LARGE OBJECT		BINARY LARGE OBJECT		

†BIT and BIT VARYING have been removed from the SQL:2003 standard.

7.1.2 SQL Scalar Data Types

Table 7.1 shows the SQL scalar data types defined in the ISO standard. Sometimes, for manipulation and conversion purposes, the data types *character* and *bit* are collectively referred to as **string** data types, and *exact numeric* and *approximate numeric* are referred to as **numeric** data types, as they share similar properties. The SQL standard now also defines both character large objects and binary large objects, although we defer discussion of these data types until Chapter 9.

Boolean data

Boolean data consists of the distinct truth values TRUE and FALSE. Unless prohibited by a NOT NULL constraint, boolean data also supports the UNKNOWN truth value as the NULL value. All boolean data type values and SQL truth values are mutually comparable and assignable. The value TRUE is greater than the value FALSE, and any comparison involving the NULL value or an UNKNOWN truth value returns an UNKNOWN result.

Character data

Character data consists of a sequence of characters from an implementation-defined character set, that is, it is defined by the vendor of the particular SQL dialect. Thus, the exact characters that can appear as data values in a character type column will vary. ASCII and EBCDIC are two sets in common use today. The format for specifying a character data type is:

CHARACTER [VARYING] [length]
CHARACTER can be abbreviated to **CHAR** and
CHARACTER VARYING to **VARCHAR**.

When a character string column is defined, a length can be specified to indicate the maximum number of characters that the column can hold (default length is 1). A character string may be defined as having a **fixed** or **varying** length. If the string

is defined to be a fixed length and we enter a string with fewer characters than this length, the string is padded with blanks on the right to make up the required size. If the string is defined to be of a varying length and we enter a string with fewer characters than this length, only those characters entered are stored, thereby using less space. For example, the branch number column branchNo of the Branch table, which has a fixed length of four characters, is declared as:

branchNo **CHAR**(4)

The column address of the PrivateOwner table, which has a variable number of characters up to a maximum of 30, is declared as:

address **VARCHAR**(30)

Bit data

The bit data type is used to define bit strings, that is, a sequence of binary digits (bits), each having either the value 0 or 1. The format for specifying the bit data type is similar to that of the character data type:

BIT [VARYING] [length]

For example, to hold the fixed length binary string "0011", we declare a column *bitString*, as:

bitString **BIT**(4)

Exact Numeric Data

The exact numeric data type is used to define numbers with an exact representation. The number consists of digits, an optional decimal point, and an optional sign. An exact numeric data type consists of a **precision** and a **scale**. The precision gives the total number of significant decimal digits, that is, the total number of digits, including decimal places but excluding the point itself. The scale gives the total number of decimal places. For example, the exact numeric value –12.345 has precision 5 and scale 3. A special case of exact numeric occurs with integers. There are several ways of specifying an exact numeric data type:

NUMERIC [precision [, scale]]
DECIMAL [precision [, scale]]
INTEGER
SMALLINT
BIGINT

INTEGER can be abbreviated to **INT** and **DECIMAL** to **DEC**

NUMERIC and DECIMAL store numbers in decimal notation. The default scale is always 0; the default precision is implementation-defined. INTEGER is used for large positive or negative whole numbers. SMALLINT is used for small positive or negative whole numbers and BIGINT for very large whole numbers. By specifying this data type, less storage space can be reserved for the data. For example, the maximum absolute value that can be stored with SMALLINT might be 32 767. The column rooms of the PropertyForRent table, which represents the number of rooms in a property, is obviously a small integer and can be declared as:

rooms **SMALLINT**

The column salary of the Staff table can be declared as:

salary **DECIMAL**(7,2)

which can handle a value up to 99,999.99.

Approximate numeric data

The approximate numeric data type is used for defining numbers that do not have an exact representation, such as real numbers. Approximate numeric, or floating point, notation is similar to scientific notation, in which a number is written as a *mantissa* times some power of ten (the *exponent).* For example, 10E3, +5.2E6, −0.2E–4. There are several ways of specifying an approximate numeric data type:

FLOAT [precision]
REAL
DOUBLE PRECISION

The *precision* controls the precision of the mantissa. The precision of REAL and DOUBLE PRECISION is implementation-defined.

Datetime data

The datetime data type is used to define points in time to a certain degree of accuracy. Examples are dates, times, and times of day. The ISO standard subdivides the datetime data type into YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE. The latter two fields specify the hour and minute part of the time zone offset from Universal Coordinated Time (which used to be called Greenwich Mean Time). Three types of datetime data type are supported:

DATE
TIME [timePrecision] [**WITH TIME ZONE**]
TIMESTAMP [timePrecision] [**WITH TIME ZONE**]

DATE is used to store calendar dates using the YEAR, MONTH, and DAY fields. TIME is used to store time using the HOUR, MINUTE, and SECOND fields. TIMESTAMP is used to store date and times. The *timePrecision* is the number of decimal places of accuracy to which the SECOND field is kept. If not specified, TIME defaults to a precision of 0 (that is, whole seconds), and TIMESTAMP defaults to 6 (that is, microseconds). The WITH TIME ZONE keyword controls the presence of the TIMEZONE_HOUR and TIMEZONE_MINUTE fields. For example, the column date of the Viewing table, which represents the date (year, month, day) that a client viewed a property, is declared as:

viewDate **DATE**

Interval data

The interval data type is used to represent periods of time. Every interval data type consists of a contiguous subset of the fields: YEAR, MONTH, DAY, HOUR,

TABLE 7.2 ISO SQL scalar operators.

OPERATOR	MEANING
OCTET_LENGTH	Returns the length of a string in octets (bit length divided by 8). For example, **OCTET_LENGTH**(×'FFFF') returns 2.
CHAR_LENGTH	Returns the length of a string in characters (or octets, if the string is a bit string). For example, **CHAR_LENGTH** ('Beech') returns 5.
CAST	Converts a value expression of one data type into a value in another data type. For example, **CAST**(5.2E6 AS INTEGER).
||	Concatenates two character strings or bit strings. For example, fName || lName.
CURRENT_USER or **USER**	Returns a character string representing the current authorization identifier (informally, the current user name).
SESSION_USER	Returns a character string representing the SQL-session authorization identifier.
SYSTEM_USER	Returns a character string representing the identifier of the user who invoked the current module.
LOWER	Converts uppercase letters to lowercase. For example, **LOWER**(**SELECT** fName **FROM** Staff **WHERE** staffNo = 'SL21') returns 'john'.
UPPER	Converts lower-case letters to upper-case. For example, **UPPER**(**SELECT** fName **FROM** Staff **WHERE** staffNo = 'SL21') returns 'JOHN'.
TRIM	Removes leading (**LEADING**), trailing (**TRAILING**), or both leading and trailing **(BOTH)** characters from a string. For example, **TRIM**(**BOTH** '*' **FROM** '*** Hello World ***') returns 'Hello World'.
POSITION	Returns the position of one string within another string. For example, **POSITION** ('ee' **IN** 'Beech') returns 2.
SUBSTRING	Returns a substring selected from within a string. For example, **SUBSTRING** ('Beech' **FROM** 1 **TO** 3) returns the string 'Bee'.
CASE	Returns one of a specified set of values, based on some condition. For example, **CASE** type **WHEN** 'House' **THEN** 1 **WHEN** 'Flat' **THEN** 2 **ELSE** 0 **END**
CURRENT_DATE	Returns the current date in the time zone that is local to the user.
CURRENT_TIME	Returns the current time in the time zone that is the current default for the session. For example, **CURRENT_TIME**(6) gives time to microseconds precision.
CURRENT_TIMESTAMP	Returns the current date and time in the time zone that is the current default for the session. For example, **CURRENT_TIMESTAMP**(0) gives time to seconds precision.
EXTRACT	Returns the value of a specified field from a datetime or interval value. For example, **EXTRACT**(**YEAR FROM** Registration.dateJoined).
ABS	Operates on a numeric argument and returns its absolute value in the same most specific type. For example, **ABS**(−17.1) returns 17.1.
MOD	Operates on two exact numeric arguments with scale 0 and returns the modulus (remainder) of the first argument divided by the second argument as an exact numeric with scale 0. For example, **MOD**(26, 11) returns 4.
LN	Computes the natural logarithm of its argument. For example, **LN**(65) returns 4.174 (approx).
EXP	Computes the exponential function, that is, *e*, (the base of natural logarithms) raised to the power equal to its argument. For example, **EXP**(2) returns 7.389 (approx).

POWER	Raises its first argument to the power of its second argument. For example, **POWER**(2,3) returns 8.
SQRT	Computes the square root of its argument. For example, **SQRT**(16) returns 4.
FLOOR	Computes the greatest integer less than or equal to its argument. For example, **FLOOR**(15.7) returns 15.
CEIL	Computes the least integer greater than or equal to its argument. For example, **CEIL**(15.7) returns 16.

MINUTE, SECOND. There are two classes of interval data type: **year–month** intervals and **day–time** intervals. The year–month class may contain only the YEAR and/or the MONTH fields; the day–time class may contain only a contiguous selection from DAY, HOUR, MINUTE, SECOND. The format for specifying the interval data type is:

INTERVAL {{startField **TO** endField} singleDatetimeField}
startField = **YEAR | MONTH | DAY | HOUR | MINUTE**
[(intervalLeadingFieldPrecision)]
endField = **YEAR | MONTH | DAY | HOUR | MINUTE | SECOND**
[(fractionalSecondsPrecision)]
singleDatetimeField = startField | **SECOND**
[(intervalLeadingFieldPrecision [, fractionalSecondsPrecision])]

In all cases, *startField* has a leading field precision that defaults to 2. For example:

INTERVAL YEAR(2) **TO MONTH**

represents an interval of time with a value between 0 years 0 months, and 99 years 11 months. In addition,

INTERVAL HOUR TO SECOND(4)

represents an interval of time with a value between 0 hours 0 minutes 0 seconds and 99 hours 59 minutes 59.9999 seconds (the fractional precision of second is 4).

Large objects

A **large object** is a data type that holds a large amount of data, such as a long text file or a graphics file. Three different types of large object data types are defined in SQL:

- Binary Large Object (BLOB), a binary string that does not have a character set or collation association;
- Character Large Object (CLOB) and National Character Large Object (NCLOB), both character strings.

We discuss large objects in more detail in Chapter 9.

Scalar operators

SQL provides a number of built-in scalar operators and functions that can be used to construct a scalar expression, that is, an expression that evaluates to a scalar value. Apart from the obvious arithmetic operators (+, −, *, /), the operators shown in Table 7.2 are available.

7.2 Integrity Enhancement Feature

In this section, we examine the facilities provided by the SQL standard for integrity control. Integrity control consists of constraints that we wish to impose in order to protect the database from becoming inconsistent. We consider five types of integrity constraint (see Section 4.3):

- required data;
- domain constraints;
- entity integrity;
- referential integrity;
- general constraints.

These constraints can be defined in the CREATE and ALTER TABLE statements, as we will explain shortly.

7.2.1 Required Data

Some columns must contain a valid value; they are not allowed to contain nulls. A null is distinct from blank or zero, and is used to represent data that is either not available, missing, or not applicable (see Section 4.3.1). For example, every member of staff must have an associated job position (for example, Manager, Assistant, and so on). The ISO standard provides the **NOT NULL** column specifier in the CREATE and ALTER TABLE statements to provide this type of constraint. When NOT NULL is specified, the system rejects any attempt to insert a null in the column. If NULL is specified, the system accepts nulls. The ISO default is NULL. For example, to specify that the column position of the Staff table cannot be null, we define the column as:

position **VARCHAR**(10) **NOT NULL**

7.2.2 Domain Constraints

Every column has a domain; in other words, a set of legal values (see Section 4.2.1). For example, the sex of a member of staff is either 'M' or 'F', so the domain of the column sex of the Staff table is a single character string consisting of either 'M' or 'F'. The ISO standard provides two mechanisms for specifying domains in the CREATE and ALTER TABLE statements. The first is the **CHECK** clause, which allows a constraint to be defined on a column or the entire table. The format of the CHECK clause is:

CHECK (searchCondition)

In a column constraint, the CHECK clause can reference only the column being defined. Thus, to ensure that the column sex can be specified only as 'M' or 'F', we could define the column as:

sex **CHAR NOT NULL CHECK** (sex **IN** ('M', 'F'))

However, the ISO standard allows domains to be defined more explicitly using the **CREATE DOMAIN** statement:

CREATE DOMAIN DomainName [**AS**] dataType
[**DEFAULT** defaultOption]
[**CHECK** (searchCondition)]

A domain is given a name, *DomainName*, a data type (as described in Section 7.1.2), an optional default value, and an optional CHECK constraint. This is not the complete definition, but it is sufficient to demonstrate the basic concept. Thus, for the previous example, we could define a domain for sex as:

CREATE DOMAIN SexType **AS CHAR**
DEFAULT 'M'
CHECK (**VALUE IN** ('M', 'F'));

This definition creates a domain SexType that consists of a single character with either the value 'M' or 'F'. When defining the column sex, we can now use the domain name SexType in place of the data type CHAR:

sex SexType **NOT NULL**

The *searchCondition* can involve a table lookup. For example, we can create a domain BranchNumber to ensure that the values entered correspond to an existing branch number in the Branch table, using the statement:

CREATE DOMAIN BranchNumber **AS CHAR(4)**
CHECK (**VALUE IN** (**SELECT** branchNo **FROM** Branch));

The preferred method of defining domain constraints is using the CREATE DOMAIN statement. Domains can be removed from the database using the DROP DOMAIN statement:

DROP DOMAIN DomainName **[RESTRICT | CASCADE]**

The drop behavior, RESTRICT or CASCADE, specifies the action to be taken if the domain is currently being used. If RESTRICT is specified and the domain is used in an existing table, view, or assertion definition (see Section 7.2.5), the drop will fail. In the case of CASCADE, any table column that is based on the domain is automatically changed to use the domain's underlying data type, and any constraint or default clause for the domain is replaced by a column constraint or column default clause, if appropriate.

7.2.3 Entity Integrity

The primary key of a table must contain a unique, nonnull value for each row (see Section 4.3.2). For example, each row of the PropertyForRent table has a unique value for the property number propertyNo, which uniquely identifies the property represented by that row. The ISO standard supports entity integrity with the PRIMARY KEY clause in the CREATE and ALTER TABLE statements. For example, to define the primary key of the PropertyForRent table, we include the following clause:

PRIMARY KEY(propertyNo)

To define a composite primary key, we specify multiple column names in the PRIMARY KEY clause, separating each by a comma. For example, to define the primary key of the Viewing table, which consists of the columns clientNo and propertyNo, we include the clause:

PRIMARY KEY(clientNo, propertyNo)

The PRIMARY KEY clause can be specified only once per table. However, it is still possible to ensure uniqueness for any alternate keys in the table using the keyword **UNIQUE**. Every column that appears in a UNIQUE clause must also be declared as NOT NULL. There may be as many UNIQUE clauses per table as required. SQL rejects any INSERT or UPDATE operation that attempts to create a duplicate value within each candidate key (that is, primary key or alternate key). For example, with the Viewing table we could also have written:

```
clientNo        VARCHAR(5)      NOT NULL,
propertyNo      VARCHAR(5)      NOT NULL,
UNIQUE (clientNo, propertyNo)
```

7.2.4 Referential Integrity

A foreign key is a column, or set of columns, that links each row in the child table containing the foreign key to the row of the parent table containing the matching candidate key value. Referential integrity means that, if the foreign key contains a value, that value must refer to an existing, valid row in the parent table (see Section 4.3.3). For example, the branch number column branchNo in the PropertyForRent table links the property to that row in the Branch table where the property is assigned. If the branch number is not null, it must contain a valid value from the column branchNo of the Branch table, or the property is assigned to an invalid branch office.

The ISO standard supports the definition of foreign keys with the FOREIGN KEY clause in the CREATE and ALTER TABLE statements. For example, to define the foreign key branchNo of the PropertyForRent table, we include the clause:

FOREIGN KEY(branchNo) **REFERENCES** Branch

SQL rejects any INSERT or UPDATE operation that attempts to create a foreign key value in a child table without a matching candidate key value in the parent table. The action SQL takes for any UPDATE or DELETE operation that attempts to update or delete a candidate key value in the parent table that has some matching rows in the child table is dependent on the **referential action** specified using the ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause. When the user attempts to delete a row from a parent table, and there are one or more matching rows in the child table, SQL supports four options regarding the action to be taken:

- CASCADE: Delete the row from the parent table and automatically delete the matching rows in the child table. Because these deleted rows may themselves have a candidate key that is used as a foreign key in another table, the foreign key rules for these tables are triggered, and so on in a cascading manner.
- SET NULL: Delete the row from the parent table and set the foreign key value(s) in the child table to NULL. This option is valid only if the foreign key columns do not have the NOT NULL qualifier specified.

- SET DEFAULT: Delete the row from the parent table and set each component of the foreign key in the child table to the specified default value. This option is valid only if the foreign key columns have a DEFAULT value specified (see Section 7.3.2).
- NO ACTION: Reject the delete operation from the parent table. This is the default setting if the ON DELETE rule is omitted.

SQL supports the same options when the candidate key in the parent table is updated. With CASCADE, the foreign key value(s) in the child table are set to the new value(s) of the candidate key in the parent table. In the same way, the updates cascade if the updated column(s) in the child table reference foreign keys in another table.

For example, in the PropertyForRent table the staff number staffNo is a foreign key referencing the Staff table. We can specify a deletion rule such that if a staff record is deleted from the Staff table, the values of the corresponding staffNo column in the PropertyForRent table are set to NULL:

FOREIGN KEY (staffNo) **REFERENCES** Staff **ON DELETE SET NULL**

Similarly, the owner number ownerNo in the PropertyForRent table is a foreign key referencing the PrivateOwner table. We can specify an update rule such that if an owner number is updated in the PrivateOwner table, the corresponding column(s) in the PropertyForRent table are set to the new value:

FOREIGN KEY (ownerNo) **REFERENCES** PrivateOwner **ON UPDATE CASCADE**

7.2.5 General Constraints

Updates to tables may be constrained by enterprise rules governing the real-world transactions that are represented by the updates (see Section 4.3.4). For example, *DreamHome* may have a rule that prevents a member of staff from managing more than 100 properties at the same time. The ISO standard allows general constraints to be specified using the CHECK and UNIQUE clauses of the CREATE and ALTER TABLE statements and the **CREATE ASSERTION** statement. We discussed the CHECK and UNIQUE clauses earlier in this section. The CREATE ASSERTION statement is an integrity constraint that is not directly linked with a table definition. The format of the statement is:

CREATE ASSERTION AssertionName
CHECK (searchCondition)

This statement is very similar to the CHECK clause discussed earlier. However, when a general constraint involves more than one table, it may be preferable to use an ASSERTION rather than duplicate the check in each table or place the constraint in an arbitrary table. For example, to define the general constraint that prevents a member of staff from managing more than 100 properties at the same time, we could write:

```
CREATE ASSERTION StaffNotHandlingTooMuch
    CHECK (NOT EXISTS (SELECT staffNo
                       FROM PropertyForRent
                       GROUP BY staffNo
                       HAVING COUNT(*) > 100))
```

We show how to use these integrity features in the following section when we examine the CREATE and ALTER TABLE statements.

7.3 Data Definition

The SQL DDL allows database objects such as schemas, domains, tables, views, and indexes to be created and destroyed. In this section, we briefly examine how to create and destroy schemas, tables, and indexes. We discuss how to create and destroy views in the next section. The ISO standard also allows the creation of character sets, collations, and translations. However, we will not consider these database objects in this book. The interested reader is referred to Cannan and Otten, 1993.

The main SQL data definition language statements are:

CREATE SCHEMA		DROP SCHEMA
CREATE DOMAIN	ALTER DOMAIN	DROP DOMAIN
CREATE TABLE	ALTER TABLE	DROP TABLE
CREATE VIEW		DROP VIEW

These statements are used to create, change, and destroy the structures that make up the conceptual schema. Although not covered by the SQL standard, the following two statements are provided by many DBMSs:

CREATE INDEX	DROP INDEX

Additional commands are available to the DBA to specify the physical details of data storage; however, we do not discuss these commands here, as they are system-specific.

7.3.1 Creating a Database

The process of creating a database differs significantly from product to product. In multi-user systems, the authority to create a database is usually reserved for the DBA. In a single-user system, a default database may be established when the system is installed and configured and others can be created by the user as and when required. The ISO standard does not specify how databases are created, and each dialect generally has a different approach.

According to the ISO standard, relations and other database objects exist in an **environment**. Among other things, each environment consists of one or more **catalogs**, and each catalog consists of a set of **schemas**. A schema is a named collection of database objects that are in some way related to one another (all the objects in the database are described in one schema or another). The objects in a schema can be tables, views, domains, assertions, collations, translations, and character sets. All the objects in a schema have the same owner and share a number of defaults.

The standard leaves the mechanism for creating and destroying catalogs as implementation-defined, but provides mechanisms for creating and destroying schemas. The schema definition statement has the following (simplified) form:

CREATE SCHEMA [Name | **AUTHORIZATION** CreatorIdentifier]

Therefore, if the creator of a schema SqlTests is Smith, the SQL statement is:

CREATE SCHEMA SqlTests **AUTHORIZATION** Smith;

The ISO standard also indicates that it should be possible to specify within this statement the range of facilities available to the users of the schema, but the details of how these privileges are specified are implementation-dependent.

A schema can be destroyed using the DROP SCHEMA statement, which has the following form:

DROP SCHEMA Name **[RESTRICT | CASCADE]**

If RESTRICT is specified, which is the default if neither qualifier is specified, the schema must be empty or the operation fails. If CASCADE is specified, the operation cascades to drop all objects associated with the schema in the order defined previously. If any of these drop operations fail, the DROP SCHEMA fails. The total effect of a DROP SCHEMA with CASCADE can be very extensive and should be carried out only with extreme caution. It should be noted, however, that the CREATE and DROP SCHEMA statements are not always supported.

7.3.2 Creating a Table (CREATE TABLE)

Having created the database structure, we may now create the table structures for the base relations to be stored in the database. This task is achieved using the CREATE TABLE statement, which has the following basic syntax:

CREATE TABLE TableName
{(columnName dataType [**NOT NULL**] **[UNIQUE]**
[DEFAULT defaultOption] **[CHECK** (searchCondition)] [, . . .]}
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns)] [, . . .]}
{[FOREIGN KEY (listOfForeignKeyColumns)
REFERENCES ParentTableName [(listOfCandidateKeyColumns)]
[MATCH {PARTIAL | FULL}
[ON UPDATE referentialAction]
[ON DELETE referentialAction]] [, . . .]}
{[CHECK (searchCondition)] [, . . .]})

As we discussed in the previous section, this version of the CREATE TABLE statement incorporates facilities for defining referential integrity and other constraints. There is significant variation in the support provided by different dialects for this version of the statement. However, when it is supported, the facilities should be used.

The CREATE TABLE statement creates a table called TableName consisting of one or more columns of the specified *dataType*. The set of permissible data types is described in Section 7.1.2. The optional **DEFAULT** clause can be specified to provide a default value for a particular column. SQL uses this default value whenever an INSERT statement fails to specify a value for the column. Among other values, the *defaultOption* includes literals. The NOT NULL, UNIQUE, and CHECK clauses were discussed in the previous section. The remaining clauses are known as **table constraints** and can optionally be preceded with the clause:

CONSTRAINT ConstraintName

which allows the constraint to be dropped by name using the ALTER TABLE statement (see following).

The **PRIMARY KEY** clause specifies the column or columns that form the primary key for the table. If this clause is available, it should be specified for every table created. By default, NOT NULL is assumed for each column that comprises the primary

key. Only one PRIMARY KEY clause is allowed per table. SQL rejects any INSERT or UPDATE operation that attempts to create a duplicate row within the PRIMARY KEY column(s). In this way, SQL guarantees the uniqueness of the primary key.

The **FOREIGN KEY** clause specifies a foreign key in the (child) table and the relationship it has to another (parent) table. This clause implements referential integrity constraints. The clause specifies the following:

- A *listOfForeignKeyColumns*, the column or columns from the table being created that form the foreign key.
- A REFERENCES subclause, giving the parent table, that is, the table holding the matching candidate key. If the *listOfCandidateKeyColumns* is omitted, the foreign key is assumed to match the primary key of the parent table. In this case, the parent table must have a PRIMARY KEY clause in its CREATE TABLE statement.
- An optional update rule (ON UPDATE) for the relationship that specifies the action to be taken when a candidate key is updated in the parent table that matches a foreign key in the child table. The **referentialAction** can be CASCADE, SET NULL, SET DEFAULT, or NO ACTION. If the ON UPDATE clause is omitted, the default NO ACTION is assumed (see Section 7.2).
- An optional delete rule (ON DELETE) for the relationship that specifies the action to be taken when a row is deleted from the parent table that has a candidate key that matches a foreign key in the child table. The **referentialAction** is the same as for the ON UPDATE rule.
- By default, the referential constraint is satisfied if any component of the foreign key is null or there is a matching row in the parent table. The MATCH option provides additional constraints relating to nulls within the foreign key. If MATCH FULL is specified, the foreign key components must all be null or must all have values. If MATCH PARTIAL is specified, the foreign key components must all be null, or there must be at least one row in the parent table that could satisfy the constraint if the other nulls were correctly substituted. Some authors argue that referential integrity should imply MATCH FULL.

There can be as many FOREIGN KEY clauses as required. The **CHECK** and **CONSTRAINT** clauses allow additional constraints to be defined. If used as a column constraint, the CHECK clause can reference only the column being defined. Constraints are in effect checked after every SQL statement has been executed, although this check can be deferred until the end of the enclosing transaction (see Section 7.5). Example 7.1 demonstrates the potential of this version of the CREATE TABLE statement.

EXAMPLE 7.1 CREATE TABLE

Create the PropertyForRent *table using the available features of the CREATE TABLE statement.*

```
CREATE DOMAIN OwnerNumber AS VARCHAR(5)
        CHECK (VALUE IN (SELECT ownerNo FROM PrivateOwner));
CREATE DOMAIN StaffNumber AS VARCHAR(5)
        CHECK (VALUE IN (SELECT staffNo FROM Staff));
CREATE DOMAIN BranchNumber AS CHAR(4)
        CHECK (VALUE IN (SELECT branchNo FROM Branch));
CREATE DOMAIN PropertyNumber AS VARCHAR(5);
```

```
CREATE DOMAIN Street AS VARCHAR(25);
CREATE DOMAIN City AS VARCHAR(15);
CREATE DOMAIN Postcode AS VARCHAR(8);
CREATE DOMAIN PropertyType AS CHAR(1)
        CHECK(VALUE IN ('B', 'C', 'D', 'E', 'F', 'M', 'S'));
CREATE DOMAIN PropertyRooms AS SMALLINT;
        CHECK(VALUE BETWEEN 1 AND 15);
CREATE DOMAIN PropertyRent AS DECIMAL(6,2)
        CHECK(VALUE BETWEEN 0 AND 9999.99);
CREATE TABLE PropertyForRent(
    propertyNo      PropertyNumber      NOT NULL,
    street          Street              NOT NULL,
    city            City                NOT NULL,
    postcode        PostCode,
    type            PropertyType        NOT NULL DEFAULT 'F',
    rooms           PropertyRooms       NOT NULL DEFAULT 4,
    rent            PropertyRent        NOT NULL DEFAULT 600,
    ownerNo         OwnerNumber         NOT NULL,
    staffNo         StaffNumber
                    CONSTRAINT StaffNotHandlingTooMuch
                      CHECK (NOT EXISTS (SELECT staffNo
                                         FROM PropertyForRent
                                         GROUP BY staffNo
                                         HAVING COUNT(*) > 100)),
    branchNo        BranchNumber        NOT NULL,
    PRIMARY KEY (propertyNo),
    FOREIGN KEY (staffNo) REFERENCES Staff ON DELETE SET NULL
                                           ON UPDATE CASCADE,
    FOREIGN KEY (ownerNo) REFERENCES PrivateOwner ON DELETE NO
                                           ACTION ON UPDATE CASCADE,
    FOREIGN KEY (branchNo) REFERENCES Branch ON DELETE NO
                                           ACTION ON UPDATE CASCADE);
```

A default value of 'F' for 'Flat' has been assigned to the property type column type. A CONSTRAINT for the staff number column has been specified to ensure that a member of staff does not handle too many properties. The constraint checks whether the number of properties the staff member currently handles is more than than 100.

The primary key is the property number, propertyNo. SQL automatically enforces uniqueness on this column. The staff number, staffNo, is a foreign key referencing the Staff table. A deletion rule has been specified such that, if a record is deleted from the Staff table, the corresponding values of the staffNo column in the PropertyForRent table are set to NULL. Additionally, an update rule has been specified such that if a staff number is updated in the Staff table, the corresponding values in the staffNo column in the PropertyForRent table are updated accordingly. The owner number, ownerNo, is a foreign key referencing the PrivateOwner table. A deletion rule of NO ACTION has been specified to prevent deletions from the PrivateOwner table if there are matching ownerNo values in the PropertyForRent table. An update rule of CASCADE has been specified such that, if an owner number is

updated, the corresponding values in the ownerNo column in the PropertyForRent table are set to the new value. The same rules have been specified for the branchNo column. In all FOREIGN KEY constraints, because the *listOfCandidateKeyColumns* has been omitted, SQL assumes that the foreign keys match the primary keys of the respective parent tables.

Note, we have not specified NOT NULL for the staff number column staffNo, because there may be periods of time when there is no member of staff allocated to manage the property (for example, when the property is first registered). However, the other foreign key columns—ownerNo (the owner number) and branchNo (the branch number)—must be specified at all times.

7.3.3 Changing a Table Definition (ALTER TABLE)

The ISO standard provides an ALTER TABLE statement for changing the structure of a table once it has been created. The definition of the ALTER TABLE statement in the ISO standard consists of six options to:

- add a new column to a table;
- drop a column from a table;
- add a new table constraint;
- drop a table constraint;
- set a default for a column;
- drop a default for a column.

The basic format of the statement is:

ALTER TABLE TableName
[ADD [**COLUMN**] columnName dataType **[NOT NULL]** **[UNIQUE]**
[DEFAULT defaultOption**]** **[CHECK** (searchCondition)]]
[DROP [**COLUMN**] columnName **[RESTRICT** | **CASCADE**]]
[ADD [CONSTRAINT [ConstraintName]] tableConstraintDefinition]
[DROP CONSTRAINT ConstraintName **[RESTRICT** | **CASCADE**]]
[ALTER [**COLUMN**] **SET DEFAULT** defaultOption**]**
[ALTER [**COLUMN**] **DROP DEFAULT]**

where the parameters are as defined for the CREATE TABLE statement in the previous section. A *tableConstraintDefinition* is one of the clauses: PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK. The ADD COLUMN clause is similar to the definition of a column in the CREATE TABLE statement. The DROP COLUMN clause specifies the name of the column to be dropped from the table definition, and has an optional qualifier that specifies whether the DROP action is to cascade or not:

- RESTRICT: The DROP operation is rejected if the column is referenced by another database object (for example, by a view definition). This is the default setting.
- CASCADE: The DROP operation proceeds and automatically drops the column from any database objects it is referenced by. This operation cascades, so that if a column is dropped from a referencing object, SQL checks whether *that* column is referenced by any other object and drops it from there if it is, and so on.

EXAMPLE 7.2 ALTER TABLE

(a) Change the Staff table by removing the default of 'Assistant' for the position column and setting the default for the sex column to female ('F').

```
ALTER TABLE Staff
    ALTER position DROP DEFAULT;
ALTER TABLE Staff
    ALTER sex SET DEFAULT 'F';
```

(b) Change the PropertyForRent *table by removing the constraint that staff are not allowed to handle more than 100 properties at a time. Change the* Client *table by adding a new column representing the preferred number of rooms.*

```
ALTER TABLE PropertyForRent
    DROP CONSTRAINT StaffNotHandlingTooMuch;
ALTER TABLE Client
    ADD prefNoRooms PropertyRooms;
```

The ALTER TABLE statement is not available in all dialects of SQL. In some dialects, the ALTER TABLE statement cannot be used to remove an existing column from a table. In such cases, if a column is no longer required, the column could simply be ignored but kept in the table definition. If, however, you wish to remove the column from the table, you must:

- upload all the data from the table;
- remove the table definition using the DROP TABLE statement;
- redefine the new table using the CREATE TABLE statement;
- reload the data back into the new table.

The upload and reload steps are typically performed with special-purpose utility programs supplied with the DBMS. However, it is possible to create a temporary table and use the INSERT . . . SELECT statement to load the data from the old table into the temporary table and then from the temporary table into the new table.

7.3.4 Removing a Table (DROP TABLE)

Over time, the structure of a database will change; new tables will be created and some tables will no longer be needed. We can remove a redundant table from the database using the DROP TABLE statement, which has the format:

```
DROP TABLE TableName [RESTRICT | CASCADE]
```

For example, to remove the PropertyForRent table we use the command:

```
DROP TABLE PropertyForRent;
```

Note, however, that this command removes not only the named table, but also all the rows within it. To simply remove the rows from the table but retain the table structure, use the DELETE statement instead (see Section 6.3.10). The DROP TABLE statement allows you to specify whether the DROP action is to be cascaded:

- RESTRICT: The DROP operation is rejected if there are any other objects that depend for their existence upon the continued existence of the table to be dropped.

- CASCADE: The DROP operation proceeds and SQL automatically drops all dependent objects (and objects dependent on these objects).

The total effect of a DROP TABLE with CASCADE can be very extensive and should be carried out only with extreme caution. One common use of DROP TABLE is to correct mistakes made when creating a table. If a table is created with an incorrect structure, DROP TABLE can be used to delete the newly created table and start again.

7.3.5 Creating an Index (CREATE INDEX)

An index is a structure that provides accelerated access to the rows of a table based on the values of one or more columns (see Appendix F for a discussion of indexes and how they may be used to improve the efficiency of data retrievals). The presence of an index can significantly improve the performance of a query. However, as indexes may be updated by the system every time the underlying tables are updated, additional overheads may be incurred. Indexes are usually created to satisfy particular search criteria after the table has been in use for some time and has grown in size. The creation of indexes is *not* standard SQL. However, most dialects support at least the following capabilities:

CREATE [UNIQUE] INDEX IndexName
ON TableName (columnName [**ASC** | **DESC**] [, . . .])

The specified columns constitute the index key and should be listed in major to minor order. Indexes can be created only on base tables *not* on views. If the UNIQUE clause is used, uniqueness of the indexed column or combination of columns will be enforced by the DBMS. This is certainly required for the primary key and possibly for other columns as well (for example, for alternate keys). Although indexes can be created at any time, we may have a problem if we try to create a unique index on a table with records in it, because the values stored for the indexed column(s) may already contain duplicates. Therefore, it is good practice to create unique indexes, at least for primary key columns, when the base table is created and the DBMS does not automatically enforce primary key uniqueness.

For the Staff and PropertyForRent tables, we may want to create at least the following indexes:

CREATE UNIQUE INDEX StaffNoInd **ON** Staff (staffNo);
CREATE UNIQUE INDEX PropertyNoInd **ON** PropertyForRent (propertyNo);

For each column, we may specify that the order is ascending (ASC) or descending (DESC), with ASC being the default setting. For example, if we create an index on the PropertyForRent table as:

CREATE INDEX RentInd **ON** PropertyForRent (city, rent);

then an index called RentInd is created for the PropertyForRent table. Entries will be in alphabetical order by city and then by rent within each city.

7.3.6 Removing an Index (DROP INDEX)

If we create an index for a base table and later decide that it is no longer needed, we can use the DROP INDEX statement to remove the index from the database. DROP INDEX has the following format:

DROP INDEX IndexName

The following statement will remove the index created in the previous example:

DROP INDEX RentInd;

7.4 Views

Recall from Section 4.4 the definition of a view:

View

The dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a *virtual relation* that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request.

To the database user, a view appears just like a real table, with a set of named columns and rows of data. However, unlike a base table, a view does not necessarily exist in the database as a stored set of data values. Instead, a view is defined as a query on one or more base tables or views. The DBMS stores the definition of the view in the database. When the DBMS encounters a reference to a view, one approach is to look up this definition and translate the request into an equivalent request against the source tables of the view and then perform the equivalent request. This merging process, called **view resolution**, is discussed in Section 7.4.3. An alternative approach, called **view materialization**, stores the view as a temporary table in the database and maintains the currency of the view as the underlying base tables are updated. We discuss view materialization in Section 7.4.8. First, we examine how to create and use views.

7.4.1 Creating a View (CREATE VIEW)

The format of the CREATE VIEW statement is:

CREATE VIEW ViewName [(newColumnName [, . . .])]
AS subselect [**WITH** [**CASCADED** | **LOCAL**] **CHECK OPTION**]

A view is defined by specifying an SQL SELECT statement. A name may optionally be assigned to each column in the view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the *subselect*. If the list of column names is omitted, each column in the view takes the name of the corresponding column in the *subselect* statement. The list of column names must be specified if there is any ambiguity in the name for a column. This may occur if the *subselect* includes calculated columns, and the AS subclause has not been used to name such columns, or it produces two columns with identical names as the result of a join.

The *subselect* is known as the **defining query**. If WITH CHECK OPTION is specified, SQL ensures that if a row fails to satisfy the WHERE clause of the defining query of the view, it is not added to the underlying base table of the view (see Section 7.4.6). It should be noted that to create a view successfully, you must have SELECT privilege on all the tables referenced in the subselect and USAGE privilege on any domains used in referenced columns. These privileges are discussed further in Section 7.6. Although all views are created in the same way, in practice different types of views are used for different purposes. We illustrate the different types of views with examples.

EXAMPLE 7.3 Create a horizontal view

Create a view so that the manager at branch B003 can see the details only for staff who work in his or her branch office.

A horizontal view restricts a user's access to selected rows of one or more tables.

```
CREATE VIEW Manager3Staff
AS SELECT *
    FROM Staff
    WHERE branchNo = 'B003';
```

This creates a view called Manager3Staff with the same column names as the Staff table but containing only those rows where the branch number is B003. (Strictly speaking, the branchNo column is unnecessary and could have been omitted from the definition of the view, as all entries have branchNo = 'B003'.) If we now execute this statement:

```
SELECT * FROM Manager3Staff;
```

we get the result table shown in Table 7.3. To ensure that the branch manager can see only these rows, the manager should not be given access to the base table Staff. Instead, the manager should be given access permission to the view Manager3Staff. This, in effect, gives the branch manager a customized view of the Staff table, showing only the staff at his or her own branch. We discuss access permissions in Section 7.6.

TABLE 7.3 Data for view Manager3Staff.

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003

EXAMPLE 7.4 Create a vertical view

Create a view of the staff details at branch B003 that excludes salary information, so that only managers can access the salary details for staff who work at their branch.

A vertical view restricts a user's access to selected columns of one or more tables.

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Staff
    WHERE branchNo = 'B003';
```

Note that we could rewrite this statement to use the Manager3Staff view instead of the Staff table, thus:

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Manager3Staff;
```

Either way, this creates a view called Staff3 with the same columns as the Staff table, but excluding the salary, DOB, and branchNo columns. If we list this view, we get the result table

shown in Table 7.4. To ensure that only the branch manager can see the salary details, staff at branch B003 should not be given access to the base table Staff or the view Manager3Staff. Instead, they should be given access permission to the view Staff3, thereby denying them access to sensitive salary data.

Vertical views are commonly used where the data stored in a table is used by various users or groups of users. They provide a private table for these users composed only of the columns they need.

TABLE 7.4 Data for view Staff3.

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

EXAMPLE 7.5 Grouped and joined views

Create a view of staff who manage properties for rent, which includes the branch number they work at, their staff number, and the number of properties they manage (see Example 6.27).

```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
    FROM Staff s, PropertyForRent p
    WHERE s.staffNo = p.staffNo
    GROUP BY s.branchNo, s.staffNo;
```

This example gives the data shown in Table 7.5. It illustrates the use of a subselect containing a GROUP BY clause (giving a view called a **grouped view**), and containing multiple tables (giving a view called a **joined view).** One of the most frequent reasons for using views is to simplify multi-table queries. Once a joined view has been defined, we can often use a simple single-table query against the view for queries that would otherwise require a multi-table join. Note that we have to name the columns in the definition of the view because of the use of the unqualified aggregate function COUNT in the subselect.

TABLE 7.5 Data for view StaffPropCnt.

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

7.4.2 Removing a View (DROP VIEW)

A view is removed from the database with the DROP VIEW statement:

```
DROP VIEW ViewName [RESTRICT | CASCADE]
```

DROP VIEW causes the definition of the view to be deleted from the database. For example, we could remove the Manager3Staff view using the following statement:

```
DROP VIEW Manager3Staff;
```

If CASCADE is specified, DROP VIEW deletes all related dependent objects; in other words, all objects that reference the view. This means that DROP VIEW also deletes any views that are defined on the view being dropped. If RESTRICT is specified and there are any other objects that depend for their existence on the continued existence of the view being dropped, the command is rejected. The default setting is RESTRICT.

7.4.3 View Resolution

Having considered how to create and use views, we now look more closely at how a query on a view is handled. To illustrate the process of **view resolution**, consider the following query, which counts the number of properties managed by each member of staff at branch office B003. This query is based on the StaffPropCnt view of Example 7.5:

```
SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = 'B003'
ORDER BY staffNo;
```

View resolution merges the example query with the defining query of the StaffPropCnt view as follows:

(1) The view column names in the SELECT list are translated into their corresponding column names in the defining query. This gives:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
```

(2) View names in the FROM clause are replaced with the corresponding FROM lists of the defining query:

```
FROM Staff s, PropertyForRent p
```

(3) The WHERE clause from the user query is combined with the WHERE clause of the defining query using the logical operator AND, thus:

```
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
```

(4) The GROUP BY and HAVING clauses are copied from the defining query. In this example, we have only a GROUP BY clause:

```
GROUP BY s.branchNo, s.staffNo
```

(5) Finally, the ORDER BY clause is copied from the user query with the view column name translated into the defining query column name:

```
ORDER BY s.staffNo
```

(6) The final merged query becomes:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
FROM Staff s, PropertyForRent p
```

```
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
GROUP BY s.branchNo, s.staffNo
ORDER BY s.staffNo;
```

This gives the result table shown in Table 7.6.

TABLE 7.6 Result table after view resolution.

staffNo	cnt
SG14	1
SG37	2

7.4.4 Restrictions on Views

The ISO standard imposes several important restrictions on the creation and use of views, although there is considerable variation among dialects.

- If a column in the view is based on an aggregate function, then the column may appear only in SELECT and ORDER BY clauses of queries that access the view. In particular, such a column may not be used in a WHERE clause and may not be an argument to an aggregate function in any query based on the view. For example, consider the view StaffPropCnt of Example 7.5, which has a column cnt based on the aggregate function COUNT. The following query would fail:

  ```
  SELECT COUNT(cnt)
  FROM StaffPropCnt;
  ```

  because we are using an aggregate function on the column cnt, which is itself based on an aggregate function. Similarly, the following query would also fail:

  ```
  SELECT *
  FROM StaffPropCnt
  WHERE cnt > 2;
  ```

  because we are using the view column, cnt, derived from an aggregate function, on the left-hand side of a WHERE clause.
- A grouped view may never be joined with a base table or a view. For example, the StaffPropCnt view is a grouped view, so any attempt to join this view with another table or view fails.

7.4.5 View Updatability

All updates to a base table are immediately reflected in all views that encompass that base table. Similarly, we may expect that if a view is updated, the base table(s) will reflect that change. However, consider again the view StaffPropCnt of Example 7.5. Consider what would happen if we tried to insert a record that showed that at branch B003, staff member SG5 manages two properties, using the following insert statement:

```
INSERT INTO StaffPropCnt
VALUES ('B003', 'SG5', 2);
```

We have to insert two records into the PropertyForRent table showing which properties staff member SG5 manages. However, we do not know which properties they

are; all we know is that this member of staff manages two properties. In other words, we do not know the corresponding primary key values for the PropertyForRent table. If we change the definition of the view and replace the count with the actual property numbers as follows:

```
CREATE VIEW StaffPropList (branchNo, staffNo, propertyNo)
AS SELECT s.branchNo, s.staffNo, p.propertyNo
    FROM Staff s, PropertyForRent p
    WHERE s.staffNo = p.staffNo;
```

and we try to insert the record:

```
INSERT INTO StaffPropList
VALUES ('B003', 'SG5', 'PG19');
```

there is still a problem with this insertion, because we specified in the definition of the PropertyForRent table that all columns except postcode and staffNo were not allowed to have nulls (see Example 7.1). However, as the StaffPropList view excludes all columns from the PropertyForRent table except the property number, we have no way of providing the remaining nonnull columns with values.

The ISO standard specifies the views that must be updatable in a system that conforms to the standard. The definition given in the ISO standard is that a view is updatable if and only if:

- DISTINCT is not specified; that is, duplicate rows must not be eliminated from the query results.
- Every element in the SELECT list of the defining query is a column name (rather than a constant, expression, or aggregate function) and no column name appears more than once.
- The FROM clause specifies only one table; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must satisfy these conditions. This, therefore, excludes any views based on a join, union (UNION), intersection (INTERSECT), or difference (EXCEPT).
- The WHERE clause does not include any nested SELECTs that reference the table in the FROM clause.
- There is no GROUP BY or HAVING clause in the defining query.

In addition, every row that is added through the view must not violate the integrity constraints of the base table. For example, if a new row is added through a view, columns that are not included in the view are set to null, but this must not violate a NOT NULL integrity constraint in the base table. The basic concept behind these restrictions is as follows:

Updatable view — For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

7.4.6 WITH CHECK OPTION

Rows exist in a view, because they satisfy the WHERE condition of the defining query. If a row is altered such that it no longer satisfies this condition, then it will

disappear from the view. Similarly, new rows will appear within the view when an insert or update on the view causes them to satisfy the WHERE condition. The rows that enter or leave a view are called **migrating rows**.

Generally, the WITH CHECK OPTION clause of the CREATE VIEW statement prohibits a row from migrating out of the view. The optional qualifiers LOCAL/CASCADED are applicable to view hierarchies, that is, a view that is derived from another view. In this case, if WITH LOCAL CHECK OPTION is specified, then any row insert or update on this view, and on any view directly or indirectly defined on this view, must not cause the row to disappear from the view, unless the row also disappears from the underlying derived view/table. If the WITH CASCADED CHECK OPTION is specified (the default setting), then any row insert or update on this view and on any view directly or indirectly defined on this view must not cause the row to disappear from the view.

This feature is so useful that it can make working with views more attractive than working with the base tables. When an INSERT or UPDATE statement on the view violates the WHERE condition of the defining query, the operation is rejected. This behavior enforces constraints on the database and helps preserve database integrity. The WITH CHECK OPTION can be specified only for an updatable view, as defined in the previous section.

EXAMPLE 7.6 WITH CHECK OPTION

Consider again the view created in Example 7.3:

```
CREATE VIEW Manager3Staff
AS SELECT *
    FROM Staff
    WHERE branchNo = 'B003'
WITH CHECK OPTION;
```

with the virtual table shown in Table 7.3. If we now attempt to update the branch number of one of the rows from B003 to B005, for example:

```
UPDATE Manager3Staff
SET branchNo = 'B005'
WHERE staffNo = 'SG37';
```

then the specification of the WITH CHECK OPTION clause in the definition of the view prevents this from happening, as it would cause the row to migrate from this horizontal view. Similarly, if we attempt to insert the following row through the view:

```
INSERT INTO Manager3Staff
VALUES('SL15', 'Mary', 'Black', 'Assistant', 'F', DATE'1967-06-21', 8000, 'B002');
```

then the specification of WITH CHECK OPTION would prevent the row from being inserted into the underlying Staff table and immediately disappearing from this view (as branch B002 is not part of the view).

Now consider the situation where Manager3Staff is defined not on Staff directly but on another view of Staff:

```
CREATE VIEW LowSalary
AS SELECT *
    FROM Staff
    WHERE salary > 9000;

CREATE VIEW HighSalary
AS SELECT *
    FROM LowSalary
    WHERE salary > 10000
WITH LOCAL CHECK OPTION;

CREATE VIEW Manager3Staff
AS SELECT *
    FROM HighSalary
    WHERE branchNo = 'B003';
```

If we now attempt the following update on Manager3Staff:

```
UPDATE Manager3Staff
SET salary = 9500
WHERE staffNo = 'SG37';
```

then this update would fail: although the update would cause the row to disappear from the view HighSalary, the row would not disappear from the table LowSalary that HighSalary is derived from. However, if instead the update tried to set the salary to 8000, then the update would succeed, as the row would no longer be part of LowSalary. Alternatively, if the view HighSalary had specified WITH CASCADED CHECK OPTION, then setting the salary to either 9500 or 8000 would be rejected, because the row would disappear from HighSalary. Therefore, to ensure that anomalies like this do not arise, each view should normally be created using the WITH CASCADED CHECK OPTION.

7.4.7 Advantages and Disadvantages of Views

Restricting some users' access to views has potential advantages over allowing users direct access to the base tables. Unfortunately, views in SQL also have disadvantages. In this section we briefly review the advantages and disadvantages of views in SQL as summarized in Table 7.7.

TABLE 7.7 Summary of advantages/disadvantages of views in SQL.

ADVANTAGES	DISADVANTAGES
Data independence	Update restriction
Currency	Structure restriction
Improved security	Performance
Reduced complexity	
Convenience	
Customization	
Data integrity	

Advantages

In the case of a DBMS running on a standalone PC, views are usually a convenience, defined to simplify database requests. However, in a multi-user DBMS, views play a central role in defining the structure of the database and enforcing security and integrity. The major advantages of views are described next.

Data independence A view can present a consistent, unchanging picture of the structure of the database, even if the underlying source tables are changed (for example, if columns added or removed, relationships changed, tables split,

restructured, or renamed). If columns are added or removed from a table, and these columns are not required by the view, then the definition of the view need not change. If an existing table is rearranged or split up, a view may be defined so that users can continue to see the old table. In the case of splitting a table, the old table can be recreated by defining a view from the join of the new tables, provided that the split is done in such a way that the original table can be reconstructed. We can ensure that this is possible by placing the primary key in both of the new tables. Thus, if we originally had a Client table of the following form:

Client (clientNo, fName, lName, telNo, prefType, maxRent, eMail)

we could reorganize it into two new tables:

ClientDetails (clientNo, fName, lName, telNo, eMail)
ClientReqts (clientNo, prefType, maxRent)

Users and applications could still access the data using the old table structure, which would be recreated by defining a view called Client as the natural join of ClientDetails and ClientReqts, with clientNo as the join column:

```
CREATE VIEW Client
AS SELECT cd.clientNo, fName, lName, telNo, prefType, maxRent, eMail
    FROM ClientDetails cd, ClientReqts cr
    WHERE cd.clientNo = cr.clientNo;
```

Currency Changes to any of the base tables in the defining query are immediately reflected in the view.

Improved security Each user can be given the privilege to access the database only through a small set of views that contain the data appropriate for that user, thus restricting and controlling each user's access to the database.

Reduced complexity A view can simplify queries, by drawing data from several tables into a single table, thereby transforming multi-table queries into single-table queries.

Convenience Views can provide greater convenience to users as users are presented with only that part of the database that they need to see. This also reduces the complexity from the user's point of view.

Customization Views provide a method to customize the appearance of the database, so that the same underlying base tables can be seen by different users in different ways.

Data integrity If the WITH CHECK OPTION clause of the CREATE VIEW statement is used, then SQL ensures that no row that fails to satisfy the WHERE clause of the defining query is ever added to any of the underlying base table(s) through the view, thereby ensuring the integrity of the view.

Disadvantages

Although views provide many significant benefits, there are also some disadvantages with SQL views.

Update restriction In Section 7.4.5 we showed that, in some cases, a view cannot be updated.

Structure restriction The structure of a view is determined at the time of its creation. If the defining query was of the form SELECT * FROM . . . , then the * refers to the columns of the base table present when the view is created. If columns are subsequently added to the base table, then these columns will not appear in the view, unless the view is dropped and recreated.

Performance There is a performance penalty to be paid when using a view. In some cases, this will be negligible; in other cases, it may be more problematic. For example, a view defined by a complex, multi-table query may take a long time to process, as the view resolution must join the tables together *every time the view is accessed*. View resolution requires additional computer resources. In the next section we briefly discuss an alternative approach to maintaining views that attempts to overcome this disadvantage.

7.4.8 View Materialization

In Section 7.4.3 we discussed one approach to handling queries based on a view, in which the query is modified into a query on the underlying base tables. One disadvantage with this approach is the time taken to perform the view resolution, particularly if the view is accessed frequently. An alternative approach, called **view materialization**, is to store the view as a temporary table in the database when the view is first queried. Thereafter, queries based on the materialized view can be much faster than recomputing the view each time. The speed difference may be critical in applications where the query rate is high and the views are complex, so it is not practical to recompute the view for every query.

Materialized views are useful in new applications such as data warehousing, replication servers, data visualization, and mobile systems. Integrity constraint checking and query optimization can also benefit from materialized views. The difficulty with this approach is maintaining the currency of the view while the base table(s) are being updated. The process of updating a materialized view in response to changes to the underlying data is called **view maintenance**. The basic aim of view maintenance is to apply only those changes necessary to the view to keep it current. As an indication of the issues involved, consider the following view:

```
CREATE VIEW StaffPropRent (staffNo)
AS SELECT DISTINCT staffNo
    FROM PropertyForRent
    WHERE branchNo = 'B003' AND rent > 400;
```

TABLE 7.8 Data for view StaffPropRent.

staffNo
SG37
SG14

with the data shown in Table 7.8. If we were to insert a row into the PropertyForRent table with a rent ≤ 400, then the view would be unchanged. If we were to insert the row ('PG24', . . . , 550, 'CO40', 'SG19', 'B003') into the PropertyForRent table, then the row should also appear within the materialized view. However, if we were to insert the row ('PG54', . . . , 450, 'CO89', 'SG37', 'B003') into the PropertyForRent table, then no new row need be added to the materialized view, because there is a row for SG37 already. Note that in these three cases the decision whether to insert

the row into the materialized view can be made without access to the underlying PropertyForRent table.

If we now wished to delete the new row ('PG24', . . . , 550, 'CO40', 'SG19', 'B003') from the PropertyForRent table, then the row should also be deleted from the materialized view. However, if we wished to delete the new row ('PG54', . . . , 450, 'CO89', 'SG37', 'B003') from the PropertyForRent table, then the row corresponding to SG37 should not be deleted from the materialized view, owing to the existence of the underlying base row corresponding to property PG21. In these two cases, the decision on whether to delete or retain the row in the materialized view requires access to the underlying base table PropertyForRent. For a more complete discussion of materialized views, the interested reader is referred to Gupta and Mumick, 1999.

7.5 Transactions

The ISO standard defines a transaction model based on two SQL statements: COMMIT and ROLLBACK. Most, but not all, commercial implementations of SQL conform to this model, which is based on IBM's DB2 DBMS. A transaction is a logical unit of work consisting of one or more SQL statements that is guaranteed to be atomic with respect to recovery. The standard specifies that an SQL transaction automatically begins with a **transaction-initiating** SQL statement executed by a user or program (for example, SELECT, INSERT, UPDATE). Changes made by a transaction are not visible to other concurrently executing transactions until the transaction completes. A transaction can complete in one of four ways:

- A COMMIT statement ends the transaction successfully, making the database changes permanent. A new transaction starts after COMMIT with the next transaction-initiating statement.
- A ROLLBACK statement aborts the transaction, backing out any changes made by the transaction. A new transaction starts after ROLLBACK with the next transaction-initiating statement.
- For programmatic SQL (see Appendix I), successful program termination ends the final transaction successfully, even if a COMMIT statement has not been executed.
- For programmatic SQL, abnormal program termination aborts the transaction.

SQL transactions cannot be nested (see Section 22.4). The SET TRANSACTION statement allows the user to configure certain aspects of the transaction. The basic format of the statement is:

SET TRANSACTION
[READ ONLY | READ WRITE] |
[ISOLATION LEVEL READ UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SERIALIZABLE]

The READ ONLY and READ WRITE qualifiers indicate whether the transaction is read-only or involves both read and write operations. The default is READ WRITE if neither qualifier is specified (unless the isolation level is READ UNCOMMITTED). Perhaps confusingly, READ ONLY allows a transaction to issue INSERT, UPDATE, and DELETE statements against temporary tables (but only temporary tables).

TABLE 7.9 Violations of serializability permitted by isolation levels.

ISOLATION LEVEL	DIRTY READ	NONREPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

The *isolation level* indicates the degree of interaction that is allowed from other transactions during the execution of the transaction. Table 7.9 shows the violations of serializability allowed by each isolation level against the following three preventable phenomena:

- *Dirty read.* A transaction reads data that has been written by another as yet uncommitted transaction.
- *Nonrepeatable read.* A transaction rereads data that it has previously read, but another committed transaction has modified or deleted the data in the intervening period.
- *Phantom read.* A transaction executes a query that retrieves a set of rows satisfying a certain search condition. When the transaction re-executes the query at a later time, additional rows are returned that have been inserted by another committed transaction in the intervening period.

Only the SERIALIZABLE isolation level is safe, that is, generates serializable schedules. The remaining isolation levels require a mechanism to be provided by the DBMS that can be used by the programmer to ensure serializability. Chapter 22 provides additional information on transactions and serializability.

7.5.1 Immediate and Deferred Integrity Constraints

In some situations, we do not want integrity constraints to be checked immediately—that is, after every SQL statement has been executed—but instead at transaction commit. A constraint may be defined as INITIALLY IMMEDIATE or INITIALLY DEFERRED, indicating which mode the constraint assumes at the start of each transaction. In the former case, it is also possible to specify whether the mode can be changed subsequently using the qualifier [NOT] DEFERRABLE. The default mode is INITIALLY IMMEDIATE.

The SET CONSTRAINTS statement is used to set the mode for specified constraints for the current transaction. The format of this statement is:

SET CONSTRAINTS
{**ALL** | constraintName [, . . .]} {**DEFERRED | IMMEDIATE**}

7.6 Discretionary Access Control

In Section 2.4 we stated that a DBMS should provide a mechanism to ensure that only authorized users can access the database. Modern DBMSs typically provide one or both of the following authorization mechanisms:

- *Discretionary access control.* Each user is given appropriate access rights (or *privileges)* on specific database objects. Typically users obtain certain privileges when they create an object and can pass some or all of these privileges to other users at their discretion. Although flexible, this type of authorization mechanism can be circumvented by a devious unauthorized user tricking an authorized user into revealing sensitive data.
- *Mandatory access control.* Each database object is assigned a certain *classification level* (for example, Top Secret, Secret, Confidential, Unclassified) and each *subject* (for example, users or programs) is given a designated *clearance level.* The classification levels form a strict ordering (Top Secret > Secret > Confidential > Unclassified) and a subject requires the necessary clearance to read or write a database object. This type of multilevel security mechanism is important for certain government, military, and corporate applications. The most commonly used mandatory access control model is known as Bell–LaPadula (Bell and La Padula, 1974), which we discuss further in Chapter 20.

SQL supports only discretionary access control through the GRANT and REVOKE statements. The mechanism is based on the concepts **of authorization identifiers**, **ownership**, and **privileges**, as we now discuss.

Authorization identifiers and ownership

An authorization identifier is a normal SQL identifier that is used to establish the identity of a user. Each database user is assigned an authorization identifier by the DBA. Usually, the identifier has an associated password, for obvious security reasons. Every SQL statement that is executed by the DBMS is performed on behalf of a specific user. The authorization identifier is used to determine which database objects the user may reference and what operations may be performed on those objects.

Each object that is created in SQL has an owner. The owner is identified by the authorization identifier defined in the AUTHORIZATION clause of the schema to which the object belongs (see Section 7.3.1). The owner is initially the only person who may know of the existence of the object and, consequently, perform any operations on the object.

Privileges

Privileges are the actions that a user is permitted to carry out on a given base table or view. The privileges defined by the ISO standard are:

- SELECT—the privilege to retrieve data from a table;
- INSERT—the privilege to insert new rows into a table;
- UPDATE—the privilege to modify rows of data in a table;
- DELETE—the privilege to delete rows of data from a table;
- REFERENCES—the privilege to reference columns of a named table in integrity constraints;
- USAGE—the privilege to use domains, collations, character sets, and translations. We do not discuss collations, character sets, and translations in this book; the interested reader is referred to Cannan and Otten, 1993.

The INSERT and UPDATE privileges can be restricted to specific columns of the table, allowing changes to these columns but disallowing changes to any other

column. Similarly, the REFERENCES privilege can be restricted to specific columns of the table, allowing these columns to be referenced in constraints, such as check constraints and foreign key constraints, when creating another table, but disallowing others from being referenced.

When a user creates a table using the CREATE TABLE statement, he or she automatically becomes the owner of the table and receives full privileges for the table. Other users initially have no privileges on the newly created table. To give them access to the table, the owner must explicitly grant them the necessary privileges using the GRANT statement.

When a user creates a view with the CREATE VIEW statement, he or she automatically becomes the owner of the view, but does not necessarily receive full privileges on the view. To create the view, a user must have SELECT privilege on all the tables that make up the view and REFERENCES privilege on the named columns of the view. However, the view owner gets INSERT, UPDATE, and DELETE privileges only if he or she holds these privileges for every table in the view.

7.6.1 Granting Privileges to Other Users (GRANT)

The GRANT statement is used to grant privileges on database objects to specific users. Normally the GRANT statement is used by the owner of a table to give other users access to the data. The format of the GRANT statement is:

```
GRANT      {PrivilegeList | ALL PRIVILEGES}
ON         ObjectName
TO         {AuthorizationIdList | PUBLIC}
[WITH GRANT OPTION]
```

PrivilegeList consists of one or more of the following privileges, separated by commas:

```
SELECT
DELETE
INSERT        [(columnName [, . . . ])]
UPDATE        [(columnName [, . . . ])]
REFERENCES    [(columnName [, . . . ])]
USAGE
```

For convenience, the GRANT statement allows the keyword ALL PRIVILEGES to be used to grant all privileges to a user instead of having to specify the six privileges individually. It also provides the keyword PUBLIC to allow access to be granted to all present and future authorized users, not just to the users currently known to the DBMS. *ObjectName* can be the name of a base table, view, domain, character set, collation, or translation.

The WITH GRANT OPTION clause allows the user(s) in *AuthorizationIdList* to pass the privileges they have been given for the named object on to other users. If these users pass a privilege on specifying WITH GRANT OPTION, the users receiving the privilege may in turn grant it to still other users. If this keyword is not specified, the receiving user(s) will not be able to pass the privileges on to other users. In this way, the owner of the object maintains very tight control over who has permission to use the object and what forms of access are allowed.

EXAMPLE 7.7 GRANT all privileges

Give the user with authorization identifier Manager *all privileges on the* Staff *table.*

```
GRANT ALL PRIVILEGES
ON Staff
TO Manager WITH GRANT OPTION;
```

The user identified as Manager can now retrieve rows from the Staff table, and also insert, update, and delete data from this table. Manager can also reference the Staff table, and all the Staff columns in any table that he or she creates subsequently. We also specified the keyword WITH GRANT OPTION, so that Manager can pass these privileges on to other users.

EXAMPLE 7.8 GRANT specific privileges

Give users Personnel *and* Director *the privileges SELECT and UPDATE on column* salary *of the* Staff *table.*

```
GRANT SELECT, UPDATE (salary)
ON Staff
TO Personnel, Director;
```

We have omitted the keyword WITH GRANT OPTION, so that users Personnel and Director cannot pass either of these privileges on to other users.

EXAMPLE 7.9 GRANT specific privileges to PUBLIC

Give all users the privilege SELECT on the Branch *table.*

```
GRANT SELECT
ON Branch
TO PUBLIC;
```

The use of the keyword PUBLIC means that all users (now and in the future) are able to retrieve all the data in the Branch table. Note that it does not make sense to use WITH GRANT OPTION in this case: as every user has access to the table, there is no need to pass the privilege on to other users.

7.6.2 Revoking Privileges from Users (REVOKE)

The REVOKE statement is used to take away privileges that were granted with the GRANT statement. A REVOKE statement can take away all or some of the privileges that were previously granted to a user. The format of the statement is:

```
REVOKE [GRANT OPTION FOR] {PrivilegeList | ALL PRIVILEGES}
ON      ObjectName
FROM    {AuthorizationIdList | PUBLIC} [RESTRICT | CASCADE]
```

The keyword ALL PRIVILEGES refers to all the privileges granted to a user by the user revoking the privileges. The optional GRANT OPTION FOR clause allows privileges passed on via the WITH GRANT OPTION of the GRANT statement to be revoked separately from the privileges themselves.

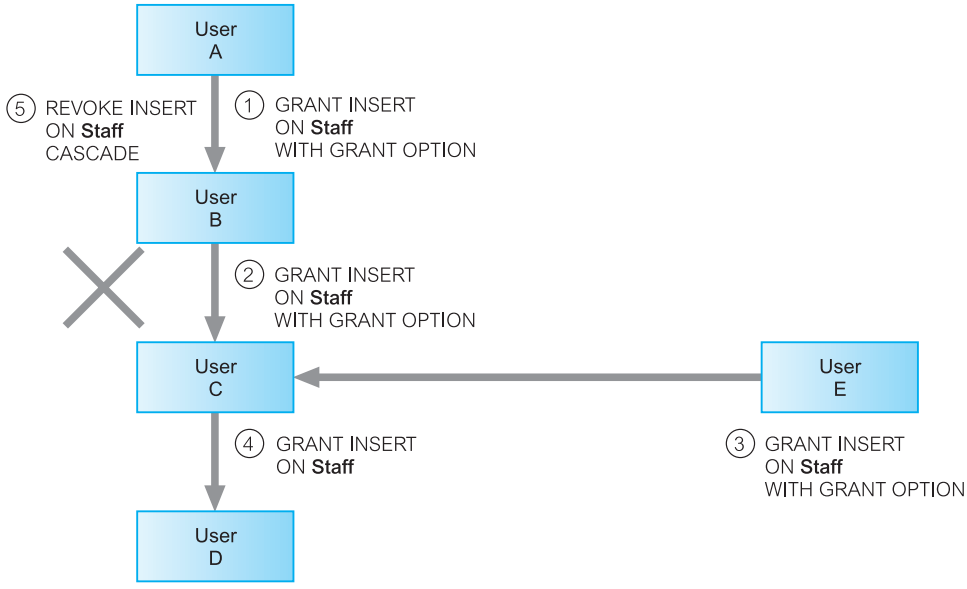

Figure 7.1 Effects of REVOKE.

The RESTRICT and CASCADE qualifiers operate exactly as in the DROP TABLE statement (see Section 7.3.3). Because privileges are required to create certain objects, revoking a privilege can remove the authority that allowed the object to be created (such an object is said to be **abandoned**). The REVOKE statement fails if it results in an abandoned object, such as a view, unless the CASCADE keyword has been specified. If CASCADE is specified, an appropriate DROP statement is issued for any abandoned views, domains, constraints, or assertions.

The privileges that were granted to this user by other users are not affected by this REVOKE statement. Therefore, if another user has granted the user the privilege being revoked, the other user's grant still allows the user to access the table. For example, in Figure 7.1 User A grants User B INSERT privilege on the Staff table WITH GRANT OPTION (step 1). User B passes this privilege on to User C (step 2). Subsequently, User C gets the same privilege from User E (step 3). User C then passes the privilege on to User D (step 4). When User A revokes the INSERT privilege from User B (step 5), the privilege cannot be revoked from User C, because User C has also received the privilege from User E. If User E had not given User C this privilege, the revoke would have cascaded to User C and User D.

EXAMPLE 7.10 REVOKE specific privileges from PUBLIC

Revoke the privilege SELECT on the Branch *table from all users.*

```
REVOKE SELECT
ON Branch
FROM PUBLIC;
```

EXAMPLE 7.11 REVOKE specific privileges from named user

Revoke all privileges you have given to Director *on the* Staff *table.*

```
REVOKE ALL PRIVILEGES
ON Staff
FROM Director;
```

This is equivalent to REVOKE SELECT . . . , as this was the only privilege that has been given to Director.

Chapter Summary

- The ISO standard provides eight base data types: boolean, character, bit, exact numeric, approximate numeric, datetime, interval, and character/binary large objects.
- The SQL DDL statements allow database objects to be defined. The CREATE and DROP SCHEMA statements allow schemas to be created and destroyed; the CREATE, ALTER, and DROP TABLE statements allow tables to be created, modified, and destroyed; the CREATE and DROP INDEX statements allow indexes to be created and destroyed.
- The ISO SQL standard provides clauses in the **CREATE** and **ALTER TABLE** statements to define **integrity constraints** that handle required data, domain constraints, entity integrity, referential integrity, and general constraints. **Required data** can be specified using NOT NULL. **Domain constraints** can be specified using the CHECK clause or by defining domains using the CREATE DOMAIN statement. **Primary keys** should be defined using the PRIMARY KEY clause and **alternate keys** using the combination of NOT NULL and UNIQUE. **Foreign keys** should be defined using the FOREIGN KEY clause and update and delete rules using the subclauses ON UPDATE and ON DELETE. **General constraints** can be defined using the CHECK and UNIQUE clauses. General constraints can also be created using the CREATE ASSERTION statement.
- A **view** is a virtual table representing a subset of columns and/or rows and/or column expressions from one or more base tables or views. A view is created using the CREATE VIEW statement by specifying a **defining query**. It may not necessarily be a physically stored table, but may be recreated each time it is referenced.
- Views can be used to simplify the structure of the database and make queries easier to write. They can also be used to protect certain columns and/or rows from unauthorized access. Not all views are updatable.
- **View resolution** merges the query on a view with the definition of the view producing a query on the underlying base table(s). This process is performed each time the DBMS has to process a query on a view. An alternative approach, called **view materialization**, stores the view as a temporary table in the database when the view is first queried. Thereafter, queries based on the materialized view can be much faster than recomputing the view each time. One disadvantage with materialized views is maintaining the currency of the temporary table.
- The COMMIT statement signals successful completion of a transaction and all changes to the database are made permanent. The ROLLBACK statement signals that the transaction should be aborted and all changes to the database are undone.
- SQL access control is built around the concepts of authorization identifiers, ownership, and privileges. **Authorization identifiers** are assigned to database users by the DBA and identify a user. Each object

that is created in SQL has an **owner**. The owner can pass **privileges** on to other users using the GRANT statement and can revoke the privileges passed on using the REVOKE statement. The privileges that can be passed on are USAGE, SELECT, DELETE, INSERT, UPDATE, and REFERENCES; INSERT, UPDATE, and REFERENCES can be restricted to specific columns. A user can allow a receiving user to pass privileges on using the WITH GRANT OPTION clause and can revoke this privilege using the GRANT OPTION FOR clause.

Review Questions

7.1 What are the main SQL DDL statements?

7.2 Discuss the functionality and importance of the Integrity Enhancement Feature (IFF).

7.3 What are the privileges commonly granted to database users?

7.4 Discuss the advantages and disadvantages of views.

7.5 Discuss the ways by which a transaction can complete.

7.6 What restrictions are necessary to ensure that a view is updatable?

7.7 What is a materialized view and what are the advantages of a maintaining a materialized view rather than using the view resolution process?

7.8 Describe the difference between discretionary and mandatory access control. What type of control mechanism does SQL support?

7.9 Describe how the access control mechanisms of SQL work.

Exercises

Answer the following questions using the relational schema from the Exercises at the end of Chapter 4:

7.10 Create the **Hotel** table using the integrity enhancement features of SQL.

7.11 Now create the **Room**, **Booking**, and **Guest** tables using the integrity enhancement features of SQL with the following constraints:
- **(a)** **type** must be one of Single, Double, or Family.
- **(b)** **price** must be between £10 and £100.
- **(c)** **roomNo** must be between 1 and 100.
- **(d)** **dateFrom** and **dateTo** must be greater than today's date.
- **(e)** The same room cannot be double-booked.
- **(f)** The same guest cannot have overlapping bookings.

7.12 Create a separate table with the same structure as the **Booking** table to hold archive records. Using the INSERT statement, copy the records from the **Booking** table to the archive table relating to bookings before 1 January 2013. Delete all bookings before 1 January 2013 from the **Booking** table.

7.13 Assume that all hotels are loaded. Create a view containing the cheapest hotels in the world.

7.14 Create a view containing the guests who are from BRICS countries.

7.15 Give the users **Manager** and **Director** full access to these views, with the privilege to pass the access on to other users.

7.16 Give the user **Accounts** SELECT access to these views. Now revoke the access from this user.

7.17 Consider the following view defined on the Hotel schema:

```
CREATE VIEW HotelBookingCount (hotelNo, bookingCount)
AS SELECT h.hotelNo, COUNT(*)
    FROM Hotel h, Room r, Booking b
    WHERE h.hotelNo = r.hotelNo AND r.roomNo = b.roomNo
    GROUP BY h.hotelNo;
```

For each of the following queries, state whether the query is valid, and for the valid ones, show how each of the queries would be mapped on to a query on the underlying base tables.

(a)
```
SELECT *
FROM HotelBookingCount;
```
(b)
```
SELECT hotelNo
FROM HotelBookingCount
WHERE hotelNo ='H001';
```
(c)
```
SELECT MIN(bookingCount)
FROM HotelBookingCount;
```
(d)
```
SELECT COUNT(*)
FROM HotelBookingCount;
```
(e)
```
SELECT hotelNo
FROM HotelBookingCount
WHERE bookingCount > 1000;
```
(f)
```
SELECT hotelNo
FROM HotelBookingCount
ORDER BY bookingCount;
```

7.19 Assume that we also have a table for suppliers:

```
Supplier (supplierNo. partNo, price)
```

and a view SupplierParts, which contains the distinct part numbers that are supplied by at least one supplier:

```
CREATE VIEW SupplierParts (partNo)
AS SELECT DISTINCT partNo
    FROM Supplier s, Part p
    WHERE s.partNo = p.partNo;
```

Discuss how you would maintain this as a materialized view and under what circumstances you would be able to maintain the view without having to access the underlying base tables Part and Supplier.

7.20 Analyze three different DBMSs of your choice. Identify objects that are available in the system catalog. Compare and contrast the object organization, name scheme, and the ways used to retrieve object description.

7.21 Create the *DreamHome* rental database schema defined in Section 4.2.6 and insert the tuples shown in Figure 4.3.

7.22 Use the view you created in exercise 7.13 to discuss how you would improve the performance of the SQL command.

7.23 You are contracted to investigate queries with degraded performance to improve them. Based on the schemas created in previous exercises, discuss the criteria to decide for or against indexing.

Case Study 2

For Exercises 7.24 to 7.40, use the Projects schema defined in the Exercises at the end of Chapter 5.

7.24 Create the Projects schema using the integrity enhancement features of SQL with the following constraints:
(a) sex must be one of the single characters 'M' or 'F'.
(b) position must be one of 'Manager', 'Team Leader', 'Analyst', or 'Software Developer'.
(c) hoursWorked must be an integer value between 0 and 40.

7.25 Create a view consisting of projects managed by female managers and ordered by project number.

7.26 Create a view consisting of the attributes empNo, fName, lName, projName, and hoursWorked attributes.

7.27 Consider the following view defined on the Projects schema:

```
CREATE VIEW EmpProject(empNo, projNo, totalHours)
AS SELECT w.empNo, w.projNo, SUM(hoursWorked)
FROM Employee e, Project p, WorksOn w
WHERE e.empNo = w.empNo AND p.projNo = w.projNo
GROUP BY w.empNo, w.projNo;
```

(a)
```
SELECT*
FROM EmpProject;
```
(b)
```
SELECT projNo
FROM EmpProject
WHERE projNo = 'SCCS';
```
(c)
```
SELECT COUNT(projNo)
FROM EmpProject
WHERE empNo = 'E1';
```
(d)
```
SELECT empNo, totalHours
FROM EmpProject
GROUP BY empNo;
```

General

7.28 Consider the following table:

Part (partNo, contract, partCost)

which represents the cost negotiated under each contract for a part (a part may have a different price under each contract). Now consider the following view ExpensiveParts, which contains the distinct part numbers for parts that cost more than £1000:

```
CREATE VIEW ExpensiveParts (partNo)
AS SELECT DISTINCT partNo
    FROM Part
    WHERE partCost > 1000;
```

Discuss how you would maintain this as a materialized view and under what circumstances you would be able to maintain the view without having to access the underlying base table Part.