Part 2

The Relational Model and Languages

Chapter 3 The Relational Model

Chapter Objectives

In this chapter you will learn:

- The origins of the relational model.
- The terminology of the relational model.
- How tables are used to represent data.
- The connection between mathematical relations and relations in the relational model.
- Properties of database relations.
- How to identify candidate, primary, alternate, and foreign keys.
- The meaning of entity integrity and referential integrity.
- The purpose and advantages of views in relational systems.

The Relational Database Management System (RDBMS) has become the dominant data-processing software in use today, with estimated new licence sales of between US$6 billion and US$10 billion per year (US$25 billion with tools sales included). This software represents the second generation of DBMSs and is based on the relational data model proposed by E. F. Codd (1970). In the relational model, all data is logically structured within relations (tables). Each **relation** has a name and is made up of named **attributes** (columns) of data. Each **tuple** (row) contains one value per attribute. A great strength of the relational model is this simple logical structure. Yet, behind this simple structure is a sound theoretical foundation that is lacking in the first generation of DBMSs (the network and hierarchical DBMSs).

We devote a significant amount of this book to the RDBMS, in recognition of the importance of these systems. In this chapter, we discuss the terminology and basic structural concepts of the relational data model. In the next chapter, we examine the relational languages that can be used for update and data retrieval.

Structure of this Chapter

To put our treatment of the RDBMS into perspective, in Section 3.1 we provide a brief history of the relational model. In Section 3.2 we discuss the underlying concepts and terminology of the relational model. In Section 3.3 we discuss the relational integrity rules, including entity integrity and referential integrity. In Section 3.4 we introduce the concept of views, which are important features of relational DBMSs although, strictly speaking, not a concept of the relational model *per se*.

Looking ahead, in Chapters 5 and 6 we examine SQL (Structured Query Language), the formal and *de facto* standard language for RDBMSs, and in Chapter 7 we examine QBE (Query-By-Example), another highly popular visual query language for RDBMSs. In Chapters 15–18 we present a complete methodology for relational database design. In Appendix D, we examine Codd's twelve rules, which form a yardstick against which RDBMS products can be identified. The examples in this chapter are drawn from the *DreamHome* case study, which is described in detail in Section 10.4 and Appendix A.

3.1 Brief History of the Relational Model

The relational model was first proposed by E. F. Codd in his seminal paper 'A relational model of data for large shared data banks' (Codd, 1970). This paper is now generally accepted as a landmark in database systems, although a set-oriented model had been proposed previously (Childs, 1968). The relational model's objectives were specified as follows:

- To allow a high degree of data independence. Application programs must not be affected by modifications to the internal data representation, particularly by changes to file organizations, record orderings, or access paths.
- To provide substantial grounds for dealing with data semantics, consistency, and redundancy problems. In particular, Codd's paper introduced the concept of **normalized** relations, that is, relations that have no repeating groups. (The process of normalization is discussed in Chapters 13 and 14.)
- To enable the expansion of set-oriented data manipulation languages.

Although interest in the relational model came from several directions, the most significant research may be attributed to three projects with rather different perspectives. The first of these, at IBM's San José Research Laboratory in California, was the prototype relational DBMS System R, which was developed during the late 1970s (Astrahan *et al.*, 1976). This project was designed to prove the practicality of the relational model by providing an implementation of its data structures and operations. It also proved to be an excellent source of information about implementation concerns such as transaction management, concurrency control, recovery techniques, query optimization, data security and integrity, human factors, and user interfaces, and led to the publication of many research papers and to the development of other prototypes. In particular, the System R project led to two major developments:

- the development of a structured query language called SQL (pronounced 'S-Q-L', or sometimes 'See-Quel'), which has since become the formal International Organization for Standardization (ISO) and *de facto* standard language for relational DBMSs;
- the production of various commercial relational DBMS products during the late 1970s and the 1980s: for example, DB2 and SQL/DS from IBM and Oracle from Oracle Corporation.

The second project to have been significant in the development of the relational model was the INGRES (Interactive Graphics Retrieval System) project at the University of California at Berkeley, which was active at about the same time as the System R project. The INGRES project involved the development of a prototype RDBMS, with the research concentrating on the same overall objectives as the System R project. This research led to an academic version of INGRES, which contributed to the general appreciation of relational concepts, and spawned the commercial products INGRES from Relational Technology Inc. (now Advantage Ingres Enterprise Relational Database from Computer Associates) and the Intelligent Database Machine from Britton Lee Inc.

The third project was the Peterlee Relational Test Vehicle at the IBM UK Scientific Centre in Peterlee (Todd, 1976). This project had a more theoretical orientation than the System R and INGRES projects and was significant, principally for research into such issues as query processing and optimization, and functional extension.

Commercial systems based on the relational model started to appear in the late 1970s and early 1980s. Now there are several hundred RDBMSs for both mainframe and PC environments, even though many do not strictly adhere to the definition of the relational model. Examples of PC-based RDBMSs are Office Access and Visual FoxPro from Microsoft, InterBase and JDataStore from Borland, and R:Base from R:BASE Technologies.

Owing to the popularity of the relational model, many non-relational systems now provide a relational user interface, irrespective of the underlying model. Computer Associates' IDMS, the principal network DBMS, has become Advantage CA-IDMS, supporting a relational view of data. Other mainframe DBMSs that support some relational features are Computer Corporation of America's Model 204 and Software AG's ADABAS.

Some extensions to the relational model have also been proposed; for example, extensions to:

- capture more closely the meaning of data (for example, Codd, 1979);
- support object-oriented concepts (for example, Stonebraker and Rowe, 1986);
- support deductive capabilities (for example, Gardarin and Valduriez, 1989).

We discuss some of these extensions in Chapters 25–28 on Object DBMSs.

3.2 Terminology

The relational model is based on the mathematical concept of a **relation**, which is physically represented as a **table**. Codd, a trained mathematician, used terminology taken from mathematics, principally set theory and predicate logic. In this section we explain the terminology and structural concepts of the relational model.

3.2.1 Relational Data Structure

Relation A relation is a table with columns and rows.

An RDBMS requires only that the database be perceived by the user as tables. Note, however, that this perception applies only to the logical structure of the database: that is, the external and conceptual levels of the ANSI-SPARC architecture discussed in Section 2.1. It does not apply to the physical structure of the database, which can be implemented using a variety of storage structures (see Appendix C).

Attribute An attribute is a named column of a relation.

In the relational model, **relations** are used to hold information about the objects to be represented in the database. A relation is represented as a two-dimensional table in which the rows of the table correspond to individual records and the table columns correspond to **attributes**. Attributes can appear in any order and the relation will still be the same relation, and therefore convey the same meaning.

For example, the information on branch offices is represented by the Branch relation, with columns for attributes branchNo (the branch number), street, city, and postcode. Similarly, the information on staff is represented by the Staff relation, with columns for attributes staffNo (the staff number), fName, lName, position, sex, DOB (date of birth), salary, and branchNo (the number of the branch the staff member works at). Figure 3.1 shows instances of the Branch and Staff relations. As you can see from this example, a column contains values of a single attribute; for example, the branchNo columns contain only numbers of existing branch offices.

Domain A domain is the set of allowable values for one or more attributes.

Domains are an extremely powerful feature of the relational model. Every attribute in a relation is defined on a **domain**. Domains may be distinct for each attribute, or two or more attributes may be defined on the same domain. Figure 3.2 shows the domains for some of the attributes of the Branch and Staff relations. Note that, at any given time, typically there will be values in a domain that do not currently appear as values in the corresponding attribute.

The domain concept is important because it allows the user to define in a central place the meaning and source of values that attributes can hold. As a result, more information is available to the system when it undertakes the execution of a relational operation, and operations that are semantically incorrect can be avoided. For example, it is not sensible to compare a street name with a telephone number, even though the domain definitions for both these attributes are character strings. On the other hand, the monthly rental on a property and the number of months a property has been leased have different domains (the first a monetary value, the second an integer value), but it is still a legal operation to

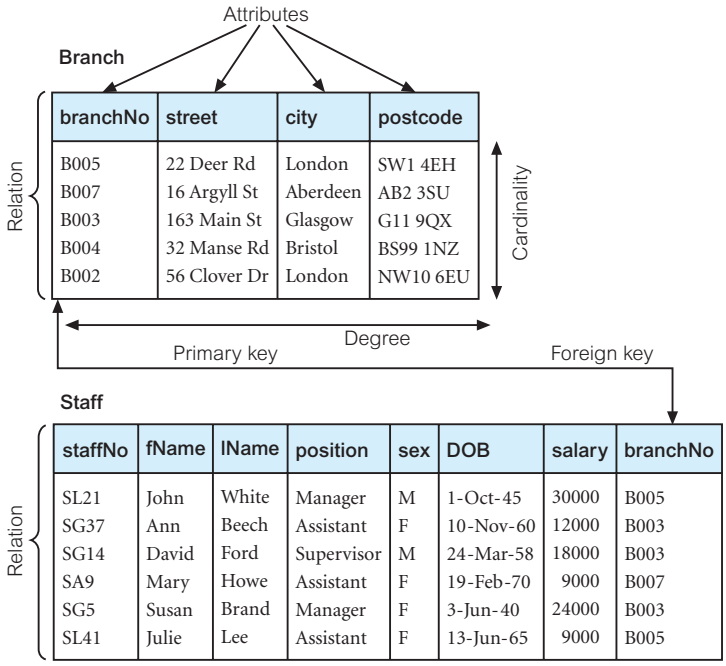


Figure 3.1
Instances of the Branch and Staff relations.

Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

Figure 3.2
Domains for some attributes of the Branch and Staff relations.

multiply two values from these domains. As these two examples illustrate, a complete implementation of domains is not straightforward and, as a result, many RDBMSs do not support them fully.

Tuple A tuple is a row of a relation.

The elements of a relation are the rows or **tuples** in the table. In the Branch relation, each row contains four values, one for each attribute. Tuples can appear in any order and the relation will still be the same relation, and therefore convey the same meaning.

The structure of a relation, together with a specification of the domains and any other restrictions on possible values, is sometimes called its **intension**, which is usually fixed unless the meaning of a relation is changed to include additional attributes. The tuples are called the **extension** (or **state**) of a relation, which changes over time.

Degree The degree of a relation is the number of attributes it contains.

The Branch relation in Figure 3.1 has four attributes or degree four. This means that each row of the table is a four-tuple, containing four values. A relation with only one attribute would have degree one and be called a **unary** relation or one-tuple. A relation with two attributes is called **binary**, one with three attributes is called **ternary**, and after that the term ***n*-ary** is usually used. The degree of a relation is a property of the *intension* of the relation.

Cardinality The cardinality of a relation is the number of tuples it contains.

By contrast, the number of tuples is called the **cardinality** of the relation and this changes as tuples are added or deleted. The cardinality is a property of the *extension* of the relation and is determined from the particular instance of the relation at any given moment. Finally, we have the definition of a relational database.

Relational database A collection of normalized relations with distinct relation names.

A relational database consists of relations that are appropriately structured. We refer to this appropriateness as *normalization*. We defer the discussion of normalization until Chapters 13 and 14.

Alternative terminology

The terminology for the relational model can be quite confusing. We have introduced two sets of terms. In fact, a third set of terms is sometimes used: a relation may be referred to as a **file**, the tuples as **records**, and the attributes as **fields**. This terminology stems from the fact that, physically, the RDBMS may store each relation in a file. Table 3.1 summarizes the different terms for the relational model.

Table 3.1 Alternative terminology for relational model terms.

Formal terms	Alternative 1	Alternative 2
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

Mathematical Relations 3.2.2

To understand the true meaning of the term *relation*, we have to review some concepts from mathematics. Suppose that we have two sets, D_1 and D_2, where $D_1 = \{2, 4\}$ and $D_2 = \{1, 3, 5\}$. The **Cartesian product** of these two sets, written $D_1 \times D_2$, is the set of all ordered pairs such that the first element is a member of D_1 and the second element is a member of D_2. An alternative way of expressing this is to find all combinations of elements with the first from D_1 and the second from D_2. In our case, we have:

$$D_1 \times D_2 = \{(2, 1), (2, 3), (2, 5), (4, 1), (4, 3), (4, 5)\}$$

Any subset of this Cartesian product is a relation. For example, we could produce a relation R such that:

$$R = \{(2, 1), (4, 1)\}$$

We may specify which ordered pairs will be in the relation by giving some condition for their selection. For example, if we observe that R includes all those ordered pairs in which the second element is 1, then we could write R as:

$$R = \{(x, y) \mid x \in D_1, y \in D_2, \text{ and } y = 1\}$$

Using these same sets, we could form another relation S in which the first element is always twice the second. Thus, we could write S as:

$$S = \{(x, y) \mid x \in D_1, y \in D_2, \text{ and } x = 2y\}$$

or, in this instance,

$$S = \{(2, 1)\}$$

since there is only one ordered pair in the Cartesian product that satisfies this condition. We can easily extend the notion of a relation to three sets. Let D_1, D_2, and D_3 be three sets. The Cartesian product $D_1 \times D_2 \times D_3$ of these three sets is the set of all ordered triples such that the first element is from D_1, the second element is from D_2, and the third element is from D_3. Any subset of this Cartesian product is a relation. For example, suppose we have:

$$D_1 = \{1, 3\} \quad D_2 = \{2, 4\} \quad D_3 = \{5, 6\}$$

$$D_1 \times D_2 \times D_3 = \{(1,2,5), (1,2,6), (1,4,5), (1,4,6), (3,2,5), (3,2,6), (3,4,5), (3,4,6)\}$$

Any subset of these ordered triples is a relation. We can extend the three sets and define a general relation on n domains. Let $D_1, D_2, \ldots, D_n$ be n sets. Their Cartesian product is defined as:

$$D_1 \times D_2 \times \ldots \times D_n = \{(d_1, d_2, \ldots, d_n) \mid d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n\}$$

and is usually written as:

$$\bigtimes_{i=1}^{n} D_i$$

Any set of n-tuples from this Cartesian product is a relation on the n sets. Note that in defining these relations we have to specify the sets, or **domains**, from which we choose values.

3.2.3 Database Relations

Applying the above concepts to databases, we can define a relation schema.

Relation schema A named relation defined by a set of attribute and domain name pairs.

Let $A_1, A_2, \ldots, A_n$ be attributes with domains $D_1, D_2, \ldots, D_n$. Then the set $\{A_1{:}D_1, A_2{:}D_2, \ldots, A_n{:}D_n\}$ is a relation schema. A relation R defined by a relation schema S is a set of mappings from the attribute names to their corresponding domains. Thus, relation R is a set of n-tuples:

$$(A_1{:}d_1, A_2{:}d_2, \ldots, A_n{:}d_n) \text{ such that } d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n$$

Each element in the n-tuple consists of an attribute and a value for that attribute. Normally, when we write out a relation as a table, we list the attribute names as column headings and write out the tuples as rows having the form $(d_1, d_2, \ldots, d_n)$, where each value is taken from the appropriate domain. In this way, we can think of a relation in the relational model as any subset of the Cartesian product of the domains of the attributes. A table is simply a physical representation of such a relation.

In our example, the Branch relation shown in Figure 3.1 has attributes branchNo, street, city, and postcode, each with its corresponding domain. The Branch relation is any subset of the Cartesian product of the domains, or any set of four-tuples in which the first element is from the domain BranchNumbers, the second is from the domain StreetNames, and so on. One of the four-tuples is:

{(B005, 22 Deer Rd, London, SW1 4EH)}

or more correctly:

{(branchNo: B005, street: 22 Deer Rd, city: London, postcode: SW1 4EH)}

We refer to this as a **relation instance**. The Branch table is a convenient way of writing out all the four-tuples that form the relation at a specific moment in time, which explains why table rows in the relational model are called tuples. In the same way that a relation has a schema, so too does the relational database.

Relational database schema A set of relation schemas, each with a distinct name.

If $R_1, R_2, \ldots, R_n$ are a set of relation schemas, then we can write the *relational database schema*, or simply *relational schema*, R, as:

$$R = \{R_1, R_2, \ldots, R_n\}$$

Properties of Relations 3.2.4

A relation has the following properties:

- the relation has a name that is distinct from all other relation names in the relational schema;
- each cell of the relation contains exactly one atomic (single) value;
- each attribute has a distinct name;
- the values of an attribute are all from the same domain;
- each tuple is distinct; there are no duplicate tuples;
- the order of attributes has no significance;
- the order of tuples has no significance, theoretically. (However, in practice, the order may affect the efficiency of accessing tuples.)

To illustrate what these restrictions mean, consider again the Branch relation shown in Figure 3.1. Since each cell should contain only one value, it is illegal to store two postcodes for a single branch office in a single cell. In other words, relations do not contain repeating groups. A relation that satisfies this property is said to be **normalized** or in **first normal form**. (Normal forms are discussed in Chapters 13 and 14.)

The column names listed at the tops of columns correspond to the attributes of the relation. The values in the branchNo attribute are all from the BranchNumbers domain; we should not allow a postcode value to appear in this column. There can be no duplicate tuples in a relation. For example, the row (B005, 22 Deer Rd, London, SW1 4EH) appears only once.

Provided an attribute name is moved along with the attribute values, we can interchange columns. The table would represent the same relation if we were to put the city attribute before the postcode attribute, although for readability it makes more sense to keep the address elements in the normal order. Similarly, tuples can be interchanged, so the records of branches B005 and B004 can be switched and the relation will still be the same.

Most of the properties specified for relations result from the properties of mathematical relations:

- When we derived the Cartesian product of sets with simple, single-valued elements such as integers, each element in each tuple was single-valued. Similarly, each cell of a relation contains exactly one value. However, a mathematical relation need not be normalized. Codd chose to disallow repeating groups to simplify the relational data model.
- In a relation, the possible values for a given position are determined by the set, or domain, on which the position is defined. In a table, the values in each column must come from the same attribute domain.
- In a set, no elements are repeated. Similarly, in a relation, there are no duplicate tuples.
- Since a relation is a set, the order of elements has no significance. Therefore, in a relation the order of tuples is immaterial.

However, in a mathematical relation, the order of elements in a tuple is important. For example, the ordered pair (1, 2) is quite different from the ordered pair (2, 1). This is not

the case for relations in the relational model, which specifically requires that the order of attributes be immaterial. The reason is that the column headings define which attribute the value belongs to. This means that the order of column headings in the intension is immaterial, but once the structure of the relation is chosen, the order of elements within the tuples of the extension must match the order of attribute names.

3.2.5 Relational Keys

As stated above, there are no duplicate tuples within a relation. Therefore, we need to be able to identify one or more attributes (called **relational keys**) that uniquely identifies each tuple in a relation. In this section, we explain the terminology used for relational keys.

Superkey An attribute, or set of attributes, that uniquely identifies a tuple within a relation.

A superkey uniquely identifies each tuple within a relation. However, a superkey may contain additional attributes that are not necessary for unique identification, and we are interested in identifying superkeys that contain only the minimum number of attributes necessary for unique identification.

Candidate key A superkey such that no proper subset is a superkey within the relation.

A candidate key, *K*, for a relation *R* has two properties:

- **uniqueness** – in each tuple of *R*, the values of *K* uniquely identify that tuple;
- **irreducibility** – no proper subset of *K* has the uniqueness property.

There may be several candidate keys for a relation. When a key consists of more than one attribute, we call it a **composite key**. Consider the Branch relation shown in Figure 3.1. Given a value of city, we can determine several branch offices (for example, London has two branch offices). This attribute cannot be a candidate key. On the other hand, since *DreamHome* allocates each branch office a unique branch number, then given a branch number value, branchNo, we can determine at most one tuple, so that branchNo is a candidate key. Similarly, postcode is also a candidate key for this relation.

Now consider a relation Viewing, which contains information relating to properties viewed by clients. The relation comprises a client number (clientNo), a property number (propertyNo), a date of viewing (viewDate) and, optionally, a comment (comment). Given a client number, clientNo, there may be several corresponding viewings for different properties. Similarly, given a property number, propertyNo, there may be several clients who viewed this property. Therefore, clientNo by itself or propertyNo by itself cannot be selected as a candidate key. However, the combination of clientNo and propertyNo identifies at most one tuple, so, for the Viewing relation, clientNo and propertyNo together form the (composite) candidate key. If we need to cater for the possibility that a client may view a property more

than once, then we could add viewDate to the composite key. However, we assume that this is not necessary.

Note that an instance of a relation cannot be used to prove that an attribute or combination of attributes is a candidate key. The fact that there are no duplicates for the values that appear at a particular moment in time does not guarantee that duplicates are not possible. However, the presence of duplicates in an instance can be used to show that some attribute combination is not a candidate key. Identifying a candidate key requires that we know the 'real world' meaning of the attribute(s) involved so that we can decide whether duplicates are possible. Only by using this semantic information can we be certain that an attribute combination is a candidate key. For example, from the data presented in Figure 3.1, we may think that a suitable candidate key for the Staff relation would be lName, the employee's surname. However, although there is only a single value of 'White' in this instance of the Staff relation, a new member of staff with the surname 'White' may join the company, invalidating the choice of lName as a candidate key.

Primary key	The candidate key that is selected to identify tuples uniquely within the relation.

Since a relation has no duplicate tuples, it is always possible to identify each row uniquely. This means that a relation always has a primary key. In the worst case, the entire set of attributes could serve as the primary key, but usually some smaller subset is sufficient to distinguish the tuples. The candidate keys that are not selected to be the primary key are called **alternate keys**. For the Branch relation, if we choose branchNo as the primary key, postcode would then be an alternate key. For the Viewing relation, there is only one candidate key, comprising clientNo and propertyNo, so these attributes would automatically form the primary key.

Foreign key	An attribute, or set of attributes, within one relation that matches the candidate key of some (possibly the same) relation.

When an attribute appears in more than one relation, its appearance usually represents a relationship between tuples of the two relations. For example, the inclusion of branchNo in both the Branch and Staff relations is quite deliberate and links each branch to the details of staff working at that branch. In the Branch relation, branchNo is the primary key. However, in the Staff relation the branchNo attribute exists to match staff to the branch office they work in. In the Staff relation, branchNo is a foreign key. We say that the attribute branchNo in the Staff relation **targets** the primary key attribute branchNo in the **home relation**, Branch. These common attributes play an important role in performing data manipulation, as we see in the next chapter.

Representing Relational Database Schemas 3.2.6

A relational database consists of any number of normalized relations. The relational schema for part of the *DreamHome* case study is:

Figure 3.3
Instance of the *DreamHome* rental database.

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Client

clientNo	fName	lName	telNo	prefType	maxRent
CR76	John	Kay	0207-774-5632	Flat	425
CR56	Aline	Stewart	0141-848-1825	Flat	350
CR74	Mike	Ritchie	01475-392178	House	750
CR62	Mary	Tregear	01224-196720	Flat	600

PrivateOwner

ownerNo	fName	lName	address	telNo
CO46	Joe	Keogh	2 Fergus Dr, Aberdeen AB2 7SX	01224-861212
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

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

Registration

clientNo	branchNo	staffNo	dateJoined
CR76	B005	SL41	2-Jan-04
CR56	B003	SG37	11-Apr-03
CR74	B003	SG37	16-Nov-02
CR62	B007	SA9	7-Mar-03

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)
PrivateOwner	(ownerNo, fName, lName, address, telNo)
Viewing	(clientNo, propertyNo, viewDate, comment)
Registration	(clientNo, branchNo, staffNo, dateJoined)

The common convention for representing a relation schema is to give the name of the relation followed by the attribute names in parentheses. Normally, the primary key is underlined.

The *conceptual model*, or *conceptual schema*, is the set of all such schemas for the database. Figure 3.3 shows an instance of this relational schema.

Integrity Constraints 3.3

In the previous section we discussed the structural part of the relational data model. As stated in Section 2.3, a data model has two other parts: a manipulative part, defining the types of operation that are allowed on the data, and a set of integrity constraints, which ensure that the data is accurate. In this section we discuss the relational integrity constraints and in the next chapter we discuss the relational manipulation operations.

We have already seen an example of an integrity constraint in Section 3.2.1: since every attribute has an associated domain, there are constraints (called **domain constraints**) that form restrictions on the set of values allowed for the attributes of relations. In addition, there are two important **integrity rules**, which are constraints or restrictions that apply to all instances of the database. The two principal rules for the relational model are known as **entity integrity** and **referential integrity**. Other types of integrity constraint are **multiplicity**, which we discuss in Section 11.6, and **general constraints**, which we introduce in Section 3.3.4. Before we define entity and referential integrity, it is necessary to understand the concept of nulls.

Nulls 3.3.1

Null Represents a value for an attribute that is currently unknown or is not applicable for this tuple.

A null can be taken to mean the logical value ‘unknown’. It can mean that a value is not applicable to a particular tuple, or it could merely mean that no value has yet been supplied. Nulls are a way to deal with incomplete or exceptional data. However, a null is not the same as a zero numeric value or a text string filled with spaces; zeros and spaces are values, but a null represents the absence of a value. Therefore, nulls should be treated differently from other values. Some authors use the term ‘null value’, however as a null is not a value but represents the absence of a value, the term ‘null value’ is deprecated.

For example, in the Viewing relation shown in Figure 3.3, the comment attribute may be undefined until the potential renter has visited the property and returned his or her comment to the agency. Without nulls, it becomes necessary to introduce false data to represent this state or to add additional attributes that may not be meaningful to the user. In our example, we may try to represent a null comment with the value '–1'. Alternatively, we may add a new attribute hasCommentBeenSupplied to the Viewing relation, which contains a Y (Yes) if a comment has been supplied, and N (No) otherwise. Both these approaches can be confusing to the user.

Nulls can cause implementation problems, arising from the fact that the relational model is based on first-order predicate calculus, which is a two-valued or Boolean logic – the only values allowed are true or false. Allowing nulls means that we have to work with a higher-valued logic, such as three- or four-valued logic (Codd, 1986, 1987, 1990).

The incorporation of nulls in the relational model is a contentious issue. Codd later regarded nulls as an integral part of the model (Codd, 1990). Others consider this approach to be misguided, believing that the missing information problem is not fully understood, that no fully satisfactory solution has been found and, consequently, that the incorporation of nulls in the relational model is premature (see, for example, Date, 1995).

We are now in a position to define the two relational integrity rules.

3.3.2 Entity Integrity

The first integrity rule applies to the primary keys of base relations. For the present, we define a base relation as a relation that corresponds to an entity in the conceptual schema (see Section 2.1). We provide a more precise definition in Section 3.4.

Entity integrity In a base relation, no attribute of a primary key can be null.

By definition, a primary key is a minimal identifier that is used to identify tuples uniquely. This means that no subset of the primary key is sufficient to provide unique identification of tuples. If we allow a null for any part of a primary key, we are implying that not all the attributes are needed to distinguish between tuples, which contradicts the definition of the primary key. For example, as branchNo is the primary key of the Branch relation, we should not be able to insert a tuple into the Branch relation with a null for the branchNo attribute. As a second example, consider the composite primary key of the Viewing relation, comprising the client number (clientNo) and the property number (propertyNo). We should not be able to insert a tuple into the Viewing relation with either a null for the clientNo attribute, or a null for the propertyNo attribute, or nulls for both attributes.

If we were to examine this rule in detail, we would find some anomalies. First, why does the rule apply only to primary keys and not more generally to candidate keys, which also identify tuples uniquely? Secondly, why is the rule restricted to base relations? For example, using the data of the Viewing relation shown in Figure 3.3, consider the query, 'List all comments from viewings'. This will produce a unary relation consisting of the attribute comment. By definition, this attribute must be a primary key, but it contains nulls

(corresponding to the viewings on PG36 and PG4 by client CR56). Since this relation is not a base relation, the model allows the primary key to be null. There have been several attempts to redefine this rule (see, for example, Codd, 1988; Date, 1990).

Referential Integrity 3.3.3

The second integrity rule applies to foreign keys.

Referential integrity	If a foreign key exists in a relation, either the foreign key value must match a candidate key value of some tuple in its home relation or the foreign key value must be wholly null.

For example, branchNo in the Staff relation is a foreign key targeting the branchNo attribute in the home relation, Branch. It should not be possible to create a staff record with branch number B025, for example, unless there is already a record for branch number B025 in the Branch relation. However, we should be able to create a new staff record with a null branch number, to cater for the situation where a new member of staff has joined the company but has not yet been assigned to a particular branch office.

General Constraints 3.3.4

General constraints	Additional rules specified by the users or database administrators of a database that define or constrain some aspect of the enterprise.

It is also possible for users to specify additional constraints that the data must satisfy. For example, if an upper limit of 20 has been placed upon the number of staff that may work at a branch office, then the user must be able to specify this general constraint and expect the DBMS to enforce it. In this case, it should not be possible to add a new member of staff at a given branch to the Staff relation if the number of staff currently assigned to that branch is 20. Unfortunately, the level of support for general constraints varies from system to system. We discuss the implementation of relational integrity in Chapters 6 and 17.

Views 3.4

In the three-level ANSI-SPARC architecture presented in Chapter 2, we described an external view as the structure of the database as it appears to a particular user. In the relational model, the word 'view' has a slightly different meaning. Rather than being the entire external model of a user's view, a view is a **virtual** or **derived relation**: a relation that does not necessarily exist in its own right, but may be dynamically derived from one or more **base relations**. Thus, an external model can consist of both base (conceptual-level) relations and views derived from the base relations. In this section, we briefly discuss

views in relational systems. In Section 6.4 we examine views in more detail and show how they can be created and used within SQL.

3.4.1 Terminology

The relations we have been dealing with so far in this chapter are known as base relations.

Base relation	A named relation corresponding to an entity in the conceptual schema, whose tuples are physically stored in the database.

We can define views in terms of base relations:

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.

A view is a relation that appears to the user to exist, can be manipulated as if it were a base relation, but does not necessarily exist in storage in the sense that the base relations do (although its definition is stored in the system catalog). The contents of a view are defined as a query on one or more base relations. Any operations on the view are automatically translated into operations on the relations from which it is derived. Views are **dynamic**, meaning that changes made to the base relations that affect the view are immediately reflected in the view. When users make permitted changes to the view, these changes are made to the underlying relations. In this section, we describe the purpose of views and briefly examine restrictions that apply to updates made through views. However, we defer treatment of how views are defined and processed until Section 6.4.

3.4.2 Purpose of Views

The view mechanism is desirable for several reasons:

- It provides a powerful and flexible security mechanism by hiding parts of the database from certain users. Users are not aware of the existence of any attributes or tuples that are missing from the view.
- It permits users to access data in a way that is customized to their needs, so that the same data can be seen by different users in different ways, at the same time.
- It can simplify complex operations on the base relations. For example, if a view is defined as a combination (join) of two relations (see Section 4.1), users may now perform more simple operations on the view, which will be translated by the DBMS into equivalent operations on the join.

A view should be designed to support the external model that the user finds familiar. For example:

- A user might need Branch tuples that contain the names of managers as well as the other attributes already in Branch. This view is created by combining the Branch relation with a restricted form of the Staff relation where the staff position is 'Manager'.
- Some members of staff should see Staff tuples without the salary attribute.
- Attributes may be renamed or the order of attributes changed. For example, the user accustomed to calling the branchNo attribute of branches by the full name Branch Number may see that column heading.
- Some members of staff should see only property records for those properties that they manage.

Although all these examples demonstrate that a view provides *logical data independence* (see Section 2.1.5), views allow a more significant type of logical data independence that supports the reorganization of the conceptual schema. For example, if a new attribute is added to a relation, existing users can be unaware of its existence if their views are defined to exclude it. If an existing relation is rearranged or split up, a view may be defined so that users can continue to see their original views. We will see an example of this in Section 6.4.7 when we discuss the advantages and disadvantages of views in more detail.

Updating Views 3.4.3

All updates to a base relation should be immediately reflected in all views that reference that base relation. Similarly, if a view is updated, then the underlying base relation should reflect the change. However, there are restrictions on the types of modification that can be made through views. We summarize below the conditions under which most systems determine whether an update is allowed through a view:

- Updates are allowed through a view defined using a simple query involving a single base relation and containing either the primary key or a candidate key of the base relation.
- Updates are not allowed through views involving multiple base relations.
- Updates are not allowed through views involving aggregation or grouping operations.

Classes of views have been defined that are **theoretically not updatable**, **theoretically updatable**, and **partially updatable**. A survey on updating relational views can be found in Furtado and Casanova (1985).

Chapter Summary

- The Relational Database Management System (RDBMS) has become the dominant data-processing software in use today, with estimated new licence sales of between US$6 billion and US$10 billion per year (US$25 billion with tools sales included). This software represents the second generation of DBMSs and is based on the relational data model proposed by E. F. Codd.
- A mathematical **relation** is a subset of the Cartesian product of two or more sets. In database terms, a relation is any subset of the Cartesian product of the domains of the attributes. A relation is normally written as a set of *n*-tuples, in which each element is chosen from the appropriate domain.
- Relations are physically represented as **tables**, with the rows corresponding to individual tuples and the columns to attributes.
- The structure of the relation, with domain specifications and other constraints, is part of the **intension** of the database, while the relation with all its tuples written out represents an **instance** or **extension** of the database.
- Properties of database relations are: each cell contains exactly one atomic value, attribute names are distinct, attribute values come from the same domain, attribute order is immaterial, tuple order is immaterial, and there are no duplicate tuples.
- The **degree** of a relation is the number of attributes, while the **cardinality** is the number of tuples. A **unary** relation has one attribute, a **binary** relation has two, a **ternary** relation has three, and an ***n*-ary** relation has *n* attributes.
- A **superkey** is an attribute, or set of attributes, that identifies tuples of a relation uniquely, while a **candidate key** is a minimal superkey. A **primary key** is the candidate key chosen for use in identification of tuples. A relation must always have a primary key. A **foreign key** is an attribute, or set of attributes, within one relation that is the candidate key of another relation.
- A **null** represents a value for an attribute that is unknown at the present time or is not applicable for this tuple.
- **Entity integrity** is a constraint that states that in a base relation no attribute of a primary key can be null. **Referential integrity** states that foreign key values must match a candidate key value of some tuple in the home relation or be wholly null. Apart from relational integrity, integrity constraints include, required data, domain, and multiplicity constraints; other integrity constraints are called **general constraints**.
- A **view** in the relational model is a **virtual** or **derived relation** that is dynamically created from the underlying base relation(s) when required. Views provide security and allow the designer to customize a user's model. Not all views are updatable.

Review Questions

3.1 Discuss each of the following concepts in the context of the relational data model:
(a) relation
(b) attribute
(c) domain
(d) tuple
(e) intension and extension
(f) degree and cardinality.

3.2 Describe the relationship between mathematical relations and relations in the relational data model.

3.3 Describe the differences between a relation and a relation schema. What is a relational database schema?

3.4 Discuss the properties of a relation.

3.5 Discuss the differences between the candidate keys and the primary key of a relation. Explain what is meant by a foreign key. How do foreign keys of relations relate to candidate keys? Give examples to illustrate your answer.

3.6 Define the two principal integrity rules for the relational model. Discuss why it is desirable to enforce these rules.

3.7 What is a view? Discuss the difference between a view and a base relation.

Exercises

The following tables form part of a database held in a relational DBMS:

Hotel	(hotelNo, hotelName, city)
Room	(roomNo, hotelNo, type, price)
Booking	(hotelNo, guestNo, dateFrom, dateTo, roomNo)
Guest	(guestNo, guestName, guestAddress)

where Hotel contains hotel details and hotelNo is the primary key;
Room contains room details for each hotel and (roomNo, hotelNo) forms the primary key;
Booking contains details of bookings and (hotelNo, guestNo, dateFrom) forms the primary key;
Guest contains guest details and guestNo is the primary key.

3.8 Identify the foreign keys in this schema. Explain how the entity and referential integrity rules apply to these relations.

3.9 Produce some sample tables for these relations that observe the relational integrity rules. Suggest some general constraints that would be appropriate for this schema.

3.10 Analyze the RDBMSs that you are currently using. Determine the support the system provides for primary keys, alternate keys, foreign keys, relational integrity, and views.

3.11 Implement the above schema in one of the RDBMSs you currently use. Implement, where possible, the primary, alternate and foreign keys, and appropriate relational integrity constraints.

Chapter

4 Relational Algebra and Relational Calculus

Chapter Objectives

In this chapter you will learn:

- The meaning of the term 'relational completeness'.
- How to form queries in the relational algebra.
- How to form queries in the tuple relational calculus.
- How to form queries in the domain relational calculus.
- The categories of relational Data Manipulation Languages (DMLs).

In the previous chapter we introduced the main structural components of the relational model. As we discussed in Section 2.3, another important part of a data model is a manipulation mechanism, or *query language*, to allow the underlying data to be retrieved and updated. In this chapter we examine the query languages associated with the relational model. In particular, we concentrate on the relational algebra and the relational calculus as defined by Codd (1971) as the basis for relational languages. Informally, we may describe the relational algebra as a (high-level) procedural language: it can be used to tell the DBMS how to build a new relation from one or more relations in the database. Again, informally, we may describe the relational calculus as a non-procedural language: it can be used to formulate the definition of a relation in terms of one or more database relations. However, formally the relational algebra and relational calculus are equivalent to one another: for every expression in the algebra, there is an equivalent expression in the calculus (and vice versa).

Both the algebra and the calculus are formal, non-user-friendly languages. They have been used as the basis for other, higher-level Data Manipulation Languages (DMLs) for relational databases. They are of interest because they illustrate the basic operations required of any DML and because they serve as the standard of comparison for other relational languages.

The relational calculus is used to measure the selective power of relational languages. A language that can be used to produce any relation that can be derived using the relational calculus is said to be **relationally complete**. Most relational query languages are relationally complete but have more expressive power than the relational algebra or relational calculus because of additional operations such as calculated, summary, and ordering functions.

Structure of this Chapter

In Section 4.1 we examine the relational algebra and in Section 4.2 we examine two forms of the relational calculus: tuple relational calculus and domain relational calculus. In Section 4.3 we briefly discuss some other relational languages. We use the *DreamHome* rental database instance shown in Figure 3.3 to illustrate the operations.

In Chapters 5 and 6 we examine SQL (Structured Query Language), the formal and *de facto* standard language for RDBMSs, which has constructs based on the tuple relational calculus. In Chapter 7 we examine QBE (Query-By-Example), another highly popular visual query language for RDBMSs, which is in part based on the domain relational calculus.

4.1 The Relational Algebra

The relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation(s). Thus, both the operands and the results are relations, and so the output from one operation can become the input to another operation. This allows expressions to be nested in the relational algebra, just as we can nest arithmetic operations. This property is called **closure**: relations are closed under the algebra, just as numbers are closed under arithmetic operations.

The relational algebra is a relation-at-a-time (or set) language in which all tuples, possibly from several relations, are manipulated in one statement without looping. There are several variations of syntax for relational algebra commands and we use a common symbolic notation for the commands and present it informally. The interested reader is referred to Ullman (1988) for a more formal treatment.

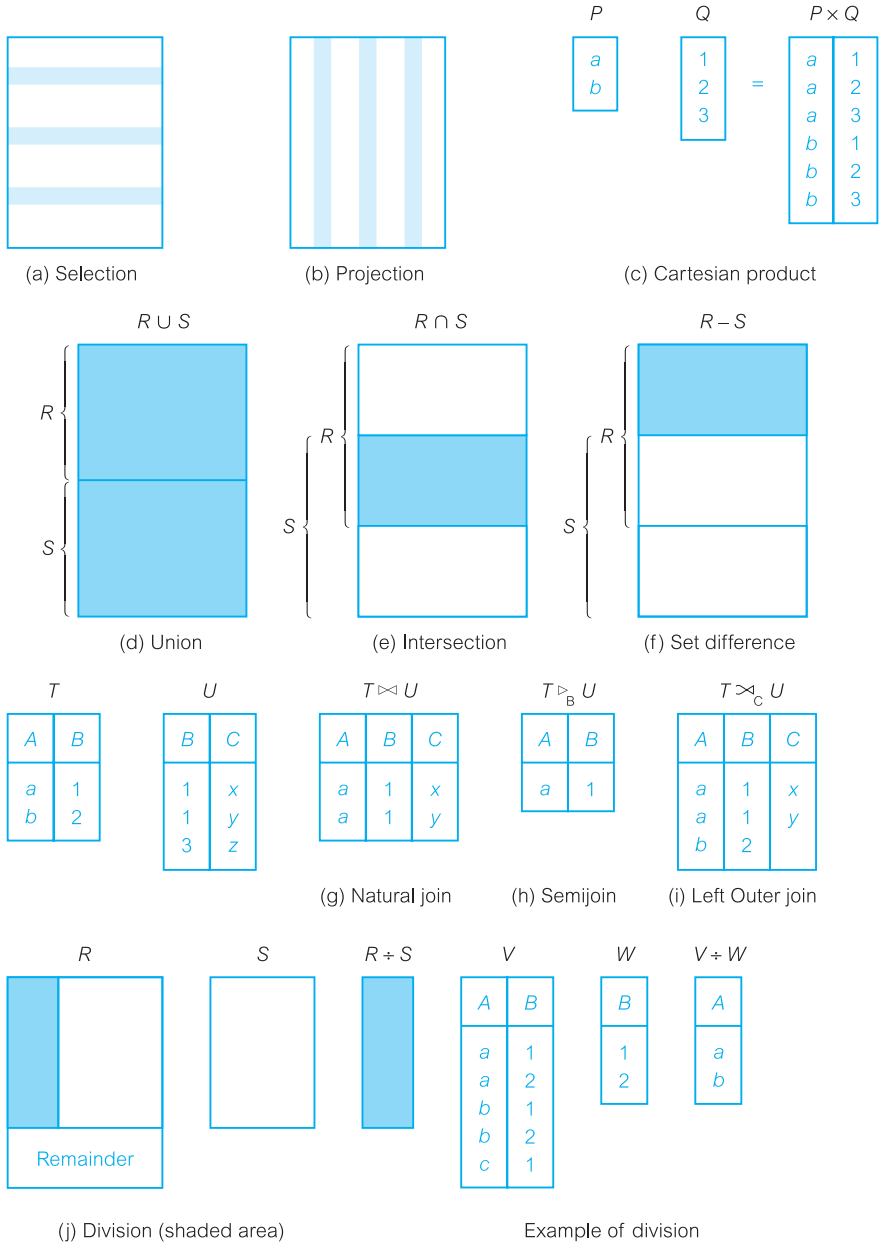
There are many variations of the operations that are included in relational algebra. Codd (1972a) originally proposed eight operations, but several others have been developed. The five fundamental operations in relational algebra, *Selection*, *Projection*, *Cartesian product*, *Union*, and *Set difference*, perform most of the data retrieval operations that we are interested in. In addition, there are also the *Join*, *Intersection*, and *Division* operations, which can be expressed in terms of the five basic operations. The function of each operation is illustrated in Figure 4.1.

The Selection and Projection operations are **unary** operations, since they operate on one relation. The other operations work on pairs of relations and are therefore called **binary** operations. In the following definitions, let R and S be two relations defined over the attributes $A = (a_1, a_2, \ldots, a_N)$ and $B = (b_1, b_2, \ldots, b_M)$, respectively.

4.1.1 Unary Operations

We start the discussion of the relational algebra by examining the two unary operations: Selection and Projection.

Figure 4.1
Illustration showing the function of the relational algebra operations.



Selection (or Restriction)

$\sigma_{predicate}(R)$ The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition (*predicate*).

Example 4.1 Selection operation

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

$\sigma_{salary > 10000}(Staff)$

Here, the input relation is Staff and the predicate is salary > 10000. The Selection operation defines a relation containing only those Staff tuples with a salary greater than £10,000. The result of this operation is shown in Figure 4.2. More complex predicates can be generated using the logical operators ∧ (AND), ∨ (OR) and ~ (NOT).

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Figure 4.2 Selecting salary > 10000 from the Staff relation.

Projection

$\Pi_{a_1, \ldots, a_n}(R)$	The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

Example 4.2 Projection operation

Produce a list of salaries for all staff, showing only the staffNo, fName, lName, *and* salary *details.*

$\Pi_{staffNo, fName, lName, salary}(Staff)$

In this example, the Projection operation defines a relation that contains only the designated Staff attributes staffNo, fName, lName, and salary, in the specified order. The result of this operation is shown in Figure 4.3.

staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

Figure 4.3 Projecting the Staff relation over the staffNo, fName, lName, and salary attributes.

4.1.2 Set Operations

The Selection and Projection operations extract information from only one relation. There are obviously cases where we would like to combine information from several relations. In the remainder of this section, we examine the binary operations of the relational algebra, starting with the set operations of Union, Set difference, Intersection, and Cartesian product.

Union

R ∪ S The union of two relations R and S defines a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.

If R and S have I and J tuples, respectively, their union is obtained by concatenating them into one relation with a maximum of $(I + J)$ tuples. Union is possible only if the schemas of the two relations match, that is, if they have the same number of attributes with each pair of corresponding attributes having the same domain. In other words, the relations must be **union-compatible**. Note that attributes names are not used in defining union-compatibility. In some cases, the Projection operation may be used to make two relations union-compatible.

Example 4.3 Union operation

List all cities where there is either a branch office or a property for rent.

$$\Pi_{city}(\text{Branch}) \cup \Pi_{city}(\text{PropertyForRent})$$

To produce union-compatible relations, we first use the Projection operation to project the Branch and PropertyForRent relations over the attribute city, eliminating duplicates where necessary. We then use the Union operation to combine these new relations to produce the result shown in Figure 4.4.

city
London
Aberdeen
Glasgow
Bristol

Figure 4.4 Union based on the city attribute from the Branch and PropertyForRent relations.

Set difference

R – S The Set difference operation defines a relation consisting of the tuples that are in relation R, but not in S. R and S must be union-compatible.

Example 4.4 Set difference operation

List all cities where there is a branch office but no properties for rent.

$$\Pi_{city}(\text{Branch}) - \Pi_{city}(\text{PropertyForRent})$$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city. We then use the Set difference operation to combine these new relations to produce the result shown in Figure 4.5.

city
Bristol

Figure 4.5 Set difference based on the city attribute from the Branch and PropertyForRent relations.

Intersection

R ∩ S The Intersection operation defines a relation consisting of the set of all tuples that are in both R and S. R and S must be union-compatible.

Example 4.5 Intersection operation

List all cities where there is both a branch office and at least one property for rent.

$$\Pi_{city}(\text{Branch}) \cap \Pi_{city}(\text{PropertyForRent})$$

As in the previous example, we produce union-compatible relations by projecting the Branch and PropertyForRent relations over the attribute city. We then use the Intersection operation to combine these new relations to produce the result shown in Figure 4.6.

city
Aberdeen
London
Glasgow

Figure 4.6 Intersection based on city attribute from the Branch and PropertyForRent relations.

Note that we can express the Intersection operation in terms of the Set difference operation:

$$R \cap S = R - (R - S)$$

Cartesian product

R × S The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

The Cartesian product operation multiplies two relations to define another relation consisting of all possible pairs of tuples from the two relations. Therefore, if one relation has I tuples and N attributes and the other has J tuples and M attributes, the Cartesian product relation will contain $(I * J)$ tuples with $(N + M)$ attributes. It is possible that the two relations may have attributes with the same name. In this case, the attribute names are prefixed with the relation name to maintain the uniqueness of attribute names within a relation.

Example 4.6 Cartesian product operation

List the names and comments of all clients who have viewed a property for rent.

The names of clients are held in the Client relation and the details of viewings are held in the Viewing relation. To obtain the list of clients and the comments on properties they have viewed, we need to combine these two relations:

$$(\Pi_{\text{clientNo, fName, lName}}(\text{Client})) \times (\Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing}))$$

This result of this operation is shown in Figure 4.7. In its present form, this relation contains more information than we require. For example, the first tuple of this relation contains different clientNo values. To obtain the required list, we need to carry out a Selection operation on this relation to extract those tuples where Client.clientNo = Viewing.clientNo. The complete operation is thus:

$$\sigma_{\text{Client.clientNo = Viewing.clientNo}}((\Pi_{\text{clientNo, fName, lName}}(\text{Client})) \times (\Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing})))$$

The result of this operation is shown in Figure 4.8.

Figure 4.7
Cartesian product of reduced Client and Viewing relations.

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR56	PG4	
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	
CR74	Mike	Ritchie	CR56	PA14	too small
CR74	Mike	Ritchie	CR76	PG4	too remote
CR74	Mike	Ritchie	CR56	PG4	
CR74	Mike	Ritchie	CR62	PA14	no dining room
CR74	Mike	Ritchie	CR56	PG36	
CR62	Mary	Tregear	CR56	PA14	too small
CR62	Mary	Tregear	CR76	PG4	too remote
CR62	Mary	Tregear	CR56	PG4	
CR62	Mary	Tregear	CR62	PA14	no dining room
CR62	Mary	Tregear	CR56	PG36	

Figure 4.8
Restricted Cartesian product of reduced Client and Viewing relations.

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

Decomposing complex operations

The relational algebra operations can be of arbitrary complexity. We can decompose such operations into a series of smaller relational algebra operations and give a name to the results of intermediate expressions. We use the assignment operation, denoted by ←, to name the results of a relational algebra operation. This works in a similar manner to the assignment operation in a programming language: in this case, the right-hand side of the operation is assigned to the left-hand side. For instance, in the previous example we could rewrite the operation as follows:

$$\text{TempViewing(clientNo, propertyNo, comment)} \leftarrow \Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing})$$
$$\text{TempClient(clientNo, fName, lName)} \leftarrow \Pi_{\text{clientNo, fName, lName}}(\text{Client})$$
$$\text{Comment(clientNo, fName, lName, vclientNo, propertyNo, comment)} \leftarrow$$
$$\text{TempClient} \times \text{TempViewing}$$
$$\text{Result} \leftarrow \sigma_{\text{clientNo} = \text{vclientNo}}(\text{Comment})$$

Alternatively, we can use the Rename operation ρ (rho), which gives a name to the result of a relational algebra operation. Rename allows an optional name for each of the attributes of the new relation to be specified.

$\rho_S(E)$ or $\rho_{S(a_1, a_2, \ldots, a_n)}(E)$	The Rename operation provides a new name S for the expression E, and optionally names the attributes as $a_1, a_2, \ldots, a_n$.

Join Operations 4.1.3

Typically, we want only combinations of the Cartesian product that satisfy certain conditions and so we would normally use a **Join operation** instead of the Cartesian product operation. The Join operation, which combines two relations to form a new relation, is one of the essential operations in the relational algebra. Join is a derivative of Cartesian product, equivalent to performing a Selection operation, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Join is one of the most difficult operations to implement efficiently in an RDBMS and is one of the reasons why relational systems have intrinsic performance problems. We examine strategies for implementing the Join operation in Section 21.4.3.

There are various forms of Join operation, each with subtle differences, some more useful than others:

- Theta join
- Equijoin (a particular type of Theta join)
- Natural join
- Outer join
- Semijoin.

Theta join (θ-join)

$R \bowtie_F S$ The Theta join operation defines a relation that contains tuples satisfying the predicate *F* from the Cartesian product of R and S. The predicate *F* is of the form $R.a_i\ \theta\ S.b_i$ where θ may be one of the comparison operators ($<, \leq, >, \geq, =, \neq$).

We can rewrite the Theta join in terms of the basic Selection and Cartesian product operations:

$$R \bowtie_F S = \sigma_F(R \times S)$$

As with Cartesian product, the degree of a Theta join is the sum of the degrees of the operand relations R and S. In the case where the predicate *F* contains only equality (=), the term **Equijoin** is used instead. Consider again the query of Example 4.6.

Example 4.7 Equijoin operation

List the names and comments of all clients who have viewed a property for rent.

In Example 4.6 we used the Cartesian product and Selection operations to obtain this list. However, the same result is obtained using the Equijoin operation:

$$(\Pi_{clientNo,\ fName,\ lName}(Client)) \bowtie_{Client.clientNo\ =\ Viewing.clientNo} (\Pi_{clientNo,\ propertyNo,\ comment}(Viewing))$$

or

$$Result \leftarrow TempClient \bowtie_{TempClient.clientNo\ =\ TempViewing.clientNo} TempViewing$$

The result of these operations was shown in Figure 4.8.

Natural join

$R \bowtie S$ The Natural join is an Equijoin of the two relations R and S over all common attributes *x*. One occurrence of each common attribute is eliminated from the result.

The Natural join operation performs an Equijoin over all the attributes in the two relations that have the same name. The degree of a Natural join is the sum of the degrees of the relations R and S less the number of attributes in *x*.

Example 4.8 Natural join operation

List the names and comments of all clients who have viewed a property for rent.

In Example 4.7 we used the Equijoin to produce this list, but the resulting relation had two occurrences of the join attribute clientNo. We can use the Natural join to remove one occurrence of the clientNo attribute:

$$(\Pi_{\text{clientNo, fName, lName}}(\text{Client})) \bowtie (\Pi_{\text{clientNo, propertyNo, comment}}(\text{Viewing}))$$

or

$$\text{Result} \leftarrow \text{TempClient} \bowtie \text{TempViewing}$$

The result of this operation is shown in Figure 4.9.

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

Figure 4.9
Natural join of restricted Client and Viewing relations.

Outer join

Often in joining two relations, a tuple in one relation does not have a matching tuple in the other relation; in other words, there is no matching value in the join attributes. We may want tuples from one of the relations to appear in the result even when there are no matching values in the other relation. This may be accomplished using the Outer join.

R ⟕ S The (left) Outer join is a join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation. Missing values in the second relation are set to null.

The Outer join is becoming more widely available in relational systems and is a specified operator in the SQL standard (see Section 5.3.7). The advantage of an Outer join is that information is preserved, that is, the Outer join preserves tuples that would have been lost by other types of join.

Example 4.9 Left Outer join operation

Produce a status report on property viewings.

In this case, we want to produce a relation consisting of the properties that have been viewed with comments and those that have not been viewed. This can be achieved using the following Outer join:

$$(\Pi_{\text{propertyNo, street, city}}(\text{PropertyForRent})) \mathbin{⟕} \text{Viewing}$$

The resulting relation is shown in Figure 4.10. Note that properties PL94, PG21, and PG16 have no viewings, but these tuples are still contained in the result with nulls for the attributes from the Viewing relation.

Figure 4.10
Left (natural) Outer join of PropertyForRent and Viewing relations.

propertyNo	street	city	clientNo	viewDate	comment
PA14	16 Holhead	Aberdeen	CR56	24-May-04	too small
PA14	16 Holhead	Aberdeen	CR62	14-May-04	no dining room
PL94	6 Argyll St	London	**null**	**null**	**null**
PG4	6 Lawrence St	Glasgow	CR76	20-Apr-04	too remote
PG4	6 Lawrence St	Glasgow	CR56	26-May-04	
PG36	2 Manor Rd	Glasgow	CR56	28-Apr-04	
PG21	18 Dale Rd	Glasgow	**null**	**null**	**null**
PG16	5 Novar Dr	Glasgow	**null**	**null**	**null**

Strictly speaking, Example 4.9 is a **Left** (**natural**) **Outer join** as it keeps every tuple in the left-hand relation in the result. Similarly, there is a **Right Outer join** that keeps every tuple in the right-hand relation in the result. There is also a **Full Outer join** that keeps all tuples in both relations, padding tuples with nulls when no matching tuples are found.

Semijoin

R ▷$_F$ S The Semijoin operation defines a relation that contains the tuples of R that participate in the join of R with S.

The Semijoin operation performs a join of the two relations and then projects over the attributes of the first operand. One advantage of a Semijoin is that it decreases the number of tuples that need to be handled to form the join. It is particularly useful for computing joins in distributed systems (see Sections 22.4.2 and 23.6.2). We can rewrite the Semijoin using the Projection and Join operations:

$$R \triangleright_F S = \Pi_A(R \bowtie_F S) \qquad \text{A is the set of all attributes for R}$$

This is actually a Semi-Theta join. There are variants for Semi-Equijoin and Semi-Natural join.

Example 4.10 Semijoin operation

List complete details of all staff who work at the branch in Glasgow.

If we are interested in seeing only the attributes of the Staff relation, we can use the following Semijoin operation, producing the relation shown in Figure 4.11.

$$\text{Staff} \triangleright_{\text{Staff.branchNo} = \text{Branch branchNo.}} (\sigma_{\text{city} = \text{'Glasgow'}} (\text{Branch}))$$

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24- Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

Figure 4.11
Semijoin of Staff and Branch relations.

4.1.4 Division Operation

The Division operation is useful for a particular type of query that occurs quite frequently in database applications. Assume relation R is defined over the attribute set A and relation S is defined over the attribute set B such that $B \subseteq A$ (B is a subset of A). Let $C = A - B$, that is, C is the set of attributes of R that are not attributes of S. We have the following definition of the Division operation.

R ÷ S The Division operation defines a relation over the attributes C that consists of the set of tuples from R that match the combination of **every** tuple in S.

We can express the Division operation in terms of the basic operations:

$$T_1 \leftarrow \Pi_C(R)$$
$$T_2 \leftarrow \Pi_C((T_1 \times S) - R)$$
$$T \leftarrow T_1 - T_2$$

Example 4.11 Division operation

Identify all clients who have viewed all properties with three rooms.

We can use the Selection operation to find all properties with three rooms followed by the Projection operation to produce a relation containing only these property numbers. We can then use the following Division operation to obtain the new relation shown in Figure 4.12.

$$(\Pi_{\text{clientNo, propertyNo}}(\text{Viewing})) \div (\Pi_{\text{propertyNo}}(\sigma_{\text{rooms} = 3}(\text{PropertyForRent})))$$

Figure 4.12 Result of the Division operation on the Viewing and PropertyForRent relations.

$\Pi_{clientNo,propertyNo}$(Viewing)

clientNo	propertyNo
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

$\Pi_{propertyNo}(\sigma_{rooms=3}$(PropertyForRent))

propertyNo
PG4
PG36

RESULT

clientNo
CR56

4.1.5 Aggregation and Grouping Operations

As well as simply retrieving certain tuples and attributes of one or more relations, we often want to perform some form of summation or **aggregation** of data, similar to the totals at the bottom of a report, or some form of **grouping** of data, similar to subtotals in a report. These operations cannot be performed using the basic relational algebra operations considered above. However, additional operations have been proposed, as we now discuss.

Aggregate operations

$\mathfrak{I}_{AL}(R)$	Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.

The main aggregate functions are:

- COUNT – returns the number of values in the associated attribute.
- SUM – returns the sum of the values in the associated attribute.
- AVG – returns the average of the values in the associated attribute.
- MIN – returns the smallest value in the associated attribute.
- MAX – returns the largest value in the associated attribute.

Example 4.12 Aggregate operations

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

We can use the aggregate function COUNT to produce the relation R shown in Figure 4.13(a) as follows:

$$\rho_R(\text{myCount})\ \mathfrak{I}_{\text{COUNT propertyNo}}(\sigma_{\text{rent} > 350}(\text{PropertyForRent}))$$

(b) Find the minimum, maximum, and average staff salary.

We can use the aggregate functions, MIN, MAX, and AVERAGE, to produce the relation R shown in Figure 4.13(b) as follows:

myCount
5

(a)

myMin	myMax	myAverage
9000	30000	17000

(b)

Figure 4.13 Result of the Aggregate operations: (a) finding the number of properties whose rent is greater than £350; (b) finding the minimum, maximum, and average staff salary.

$$\rho_R(\text{myMin, myMax, myAverage})\ \Im_{\text{MIN salary, MAX salary, AVERAGE salary}}(\text{Staff})$$

Grouping operation

$_{GA}\Im_{AL}(R)$ Groups the tuples of relation R by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. AL contains one or more (<aggregate_function>, <attribute>) pairs. The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

The general form of the grouping operation is as follows:

$$a_1, a_2, \ldots, a_n\ \Im_{<A_p\ a_p>, <A_q\ a_q>, \ldots, <A_z\ a_z>}(R)$$

where R is any relation, $a_1, a_2, \ldots, a_n$ are attributes of R on which to group, $a_p, a_q, \ldots, a_z$ are other attributes of R, and $A_p, A_q, \ldots, A_z$ are aggregate functions. The tuples of R are partitioned into groups such that:

- all tuples in a group have the same value for $a_1, a_2, \ldots, a_n$;
- tuples in different groups have different values for $a_1, a_2, \ldots, a_n$.

We illustrate the use of the grouping operation with the following example.

Example 4.13 Grouping operation

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

We first need to group tuples according to the branch number, branchNo, and then use the aggregate functions COUNT and SUM to produce the required relation. The relational algebra expression is as follows:

$$\rho_R(\text{branchNo, myCount, mySum})\ _{\text{branchNo}}\Im_{\text{COUNT staffNo, SUM salary}}(\text{Staff})$$

The resulting relation is shown in Figure 4.14.

branchNo	myCount	mySum
B003	3	54000
B005	2	39000
B007	1	9000

Figure 4.14 Result of the grouping operation to find the number of staff working in each branch and the sum of their salaries.

4.1.6 Summary of the Relational Algebra Operations

The relational algebra operations are summarized in Table 4.1.

Table 4.1 Operations in the relational algebra.

Operation	Notation	Function
Selection	$\sigma_{predicate}(R)$	Produces a relation that contains only those tuples of R that satisfy the specified *predicate*.
Projection	$\Pi_{a_1, \ldots, a_n}(R)$	Produces a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.
Union	$R \cup S$	Produces a relation that contains all the tuples of R, or S, or both R and S, duplicate tuples being eliminated. R and S must be union-compatible.
Set difference	$R - S$	Produces a relation that contains all the tuples in R that are not in S. R and S must be union-compatible.
Intersection	$R \cap S$	Produces a relation that contains all the tuples in both R and S. R and S must be union-compatible.
Cartesian product	$R \times S$	Produces a relation that is the concatenation of every tuple of relation R with every tuple of relation S.
Theta join	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate F from the Cartesian product of R and S.
Equijoin	$R \bowtie_F S$	Produces a relation that contains tuples satisfying the predicate F (which only contains equality comparisons) from the Cartesian product of R and S.
Natural join	$R \bowtie S$	An Equijoin of the two relations R and S over all common attributes *x*. One occurrence of each common attribute is eliminated.
(Left) Outer join	$R \mathbin{⟕} S$	A join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation.
Semijoin	$R \triangleright_F S$	Produces a relation that contains the tuples of R that participate in the join of R with S satisfying the predicate F.
Division	$R \div S$	Produces a relation that consists of the set of tuples from R defined over the attributes C that match the combination of **every** tuple in S, where C is the set of attributes that are in R but not in S.
Aggregate	$\Im_{AL}(R)$	Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more (<aggregate_function>, <attribute>) pairs.
Grouping	${}_{GA}\Im_{AL}(R)$	Groups the tuples of relation R by the grouping attributes, GA, and then applies the aggregate function list AL to define a new relation. AL contains one or more (<aggregate_function>, <attribute>) pairs. The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

4.2 The Relational Calculus

A certain order is always explicitly specified in a relational algebra expression and a strategy for evaluating the query is implied. In the relational calculus, there is no description of how to evaluate a query; a relational calculus query specifies *what* is to be retrieved rather than *how* to retrieve it.

The relational calculus is not related to differential and integral calculus in mathematics, but takes its name from a branch of symbolic logic called **predicate calculus**. When applied to databases, it is found in two forms: **tuple** relational calculus, as originally proposed by Codd (1972a), and **domain** relational calculus, as proposed by Lacroix and Pirotte (1977).

In first-order logic or predicate calculus, a **predicate** is a truth-valued function with arguments. When we substitute values for the arguments, the function yields an expression, called a **proposition**, which can be either true or false. For example, the sentences, 'John White is a member of staff' and 'John White earns more than Ann Beech' are both propositions, since we can determine whether they are true or false. In the first case, we have a function, 'is a member of staff', with one argument (John White); in the second case, we have a function, 'earns more than', with two arguments (John White and Ann Beech).

If a predicate contains a variable, as in 'x is a member of staff', there must be an associated **range** for x. When we substitute some values of this range for x, the proposition may be true; for other values, it may be false. For example, if the range is the set of all people and we replace x by John White, the proposition 'John White is a member of staff' is true. If we replace x by the name of a person who is not a member of staff, the proposition is false.

If P is a predicate, then we can write the set of all x such that P is true for x, as:

$$\{x \mid P(x)\}$$

We may connect predicates by the logical connectives $\wedge$ (AND), $\vee$ (OR), and $\sim$ (NOT) to form compound predicates.

4.2.1 Tuple Relational Calculus

In the tuple relational calculus we are interested in finding tuples for which a predicate is true. The calculus is based on the use of **tuple variables**. A tuple variable is a variable that 'ranges over' a named relation: that is, a variable whose only permitted values are tuples of the relation. (The word 'range' here does not correspond to the mathematical use of range, but corresponds to a mathematical domain.) For example, to specify the range of a tuple variable S as the Staff relation, we write:

Staff(S)

To express the query 'Find the set of all tuples S such that $F(S)$ is true', we can write:

$$\{S \mid F(S)\}$$

F is called a **formula** (**well-formed formula**, or **wff** in mathematical logic). For example, to express the query 'Find the staffNo, fName, lName, position, sex, DOB, salary, and branchNo of all staff earning more than £10,000', we can write:

$$\{S \mid Staff(S) \wedge S.salary > 10000\}$$

S.salary means the value of the salary attribute for the tuple variable *S*. To retrieve a particular attribute, such as salary, we would write:

$$\{S.salary \mid Staff(S) \wedge S.salary > 10000\}$$

The existential and universal quantifiers

There are two **quantifiers** we can use with formulae to tell how many instances the predicate applies to. The **existential quantifier** $\exists$ ('there exists') is used in formulae that must be true for at least one instance, such as:

$$Staff(S) \wedge (\exists B)\,(Branch(B) \wedge (B.branchNo = S.branchNo) \wedge B.city = \text{'London'})$$

This means, 'There exists a Branch tuple that has the same branchNo as the branchNo of the current Staff tuple, S, and is located in London'. The **universal quantifier** $\forall$ ('for all') is used in statements about every instance, such as:

$$(\forall B)\,(B.city \neq \text{'Paris'})$$

This means, 'For all Branch tuples, the address is not in Paris'. We can apply a generalization of De Morgan's laws to the existential and universal quantifiers. For example:

$$(\exists X)(F(X)) \equiv \sim(\forall X)(\sim(F(X)))$$

$$(\forall X)(F(X)) \equiv \sim(\exists X)(\sim(F(X)))$$

$$(\exists X)(F_1(X) \wedge F_2(X)) \equiv \sim(\forall X)(\sim(F_1(X)) \vee \sim(F_2(X)))$$

$$(\forall X)(F_1(X) \wedge F_2(X)) \equiv \sim(\exists X)(\sim(F_1(X)) \vee \sim(F_2(X)))$$

Using these equivalence rules, we can rewrite the above formula as:

$$\sim(\exists B)\,(B.city = \text{'Paris'})$$

which means, 'There are no branches with an address in Paris'.

Tuple variables that are qualified by $\forall$ or $\exists$ are called **bound variables**, otherwise the tuple variables are called **free variables**. The only free variables in a relational calculus expression should be those on the left side of the bar (|). For example, in the following query:

$$\{S.fName, S.lName \mid Staff(S) \wedge (\exists B)\,(Branch(B) \wedge (B.branchNo = S.branchNo) \wedge B.city = \text{'London'})\}$$

S is the only free variable and S is then bound successively to each tuple of Staff.

Expressions and formulae

As with the English alphabet, in which some sequences of characters do not form a correctly structured sentence, so in calculus not every sequence of formulae is acceptable. The formulae should be those sequences that are unambiguous and make sense. An expression in the tuple relational calculus has the following general form:

$$\{S_1.a_1, S_2.a_2, \ldots, S_n.a_n \mid F(S_1, S_2, \ldots, S_m)\} \qquad m \geq n$$

where $S_1, S_2, \ldots, S_n, \ldots, S_m$ are tuple variables, each a_i is an attribute of the relation over which S_i ranges, and F is a formula. A (well-formed) formula is made out of one or more *atoms*, where an atom has one of the following forms:

- $R(S_i)$, where S_i is a tuple variable and R is a relation.
- $S_i.a_1\ \theta\ S_j.a_2$, where S_i and S_j are tuple variables, a_1 is an attribute of the relation over which S_i ranges, a_2 is an attribute of the relation over which S_j ranges, and θ is one of the comparison operators ($<, \leq, >, \geq, =, \neq$); the attributes a_1 and a_2 must have domains whose members can be compared by θ.
- $S_i.a_1\ \theta\ c$, where S_i is a tuple variable, a_1 is an attribute of the relation over which S_i ranges, c is a constant from the domain of attribute a_1, and θ is one of the comparison operators.

We recursively build up formulae from atoms using the following rules:

- An atom is a formula.
- If F_1 and F_2 are formulae, so are their conjunction $F_1 \wedge F_2$, their disjunction $F_1 \vee F_2$, and the negation $\sim F_1$.
- If F is a formula with free variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

Example 4.14 Tuple relational calculus

(a) List the names of all managers who earn more than £25,000.

$$\{S.fName, S.lName \mid Staff(S) \wedge S.position = \text{'Manager'} \wedge S.salary > 25000\}$$

(b) List the staff who manage properties for rent in Glasgow.

$$\{S \mid Staff(S) \wedge (\exists P)\ (PropertyForRent(P) \wedge (P.staffNo = S.staffNo) \wedge P.city = \text{'Glasgow'})\}$$

The staffNo attribute in the PropertyForRent relation holds the staff number of the member of staff who manages the property. We could reformulate the query as: 'For each member of staff whose details we want to list, there exists a tuple in the relation PropertyForRent for that member of staff with the value of the attribute city in that tuple being Glasgow.'

Note that in this formulation of the query, there is no indication of a strategy for executing the query – the DBMS is free to decide the operations required to fulfil the request and the execution order of these operations. On the other hand, the equivalent

relational algebra formulation would be: 'Select tuples from PropertyForRent such that the city is Glasgow and perform their join with the Staff relation', which has an implied order of execution.

(c) List the names of staff who currently do not manage any properties.

$$\{S.fName, S.lName \mid Staff(S) \wedge (\sim(\exists P)\ (PropertyForRent(P) \wedge (S.staffNo = P.staffNo)))\}$$

Using the general transformation rules for quantifiers given above, we can rewrite this as:

$$\{S.fName, S.lName \mid Staff(S) \wedge ((\forall P)\ (\sim PropertyForRent(P) \vee \sim(S.staffNo = P.staffNo)))\}$$

(d) List the names of clients who have viewed a property for rent in Glasgow.

$$\{C.fName, C.lName \mid Client(C) \wedge ((\exists V)\ (\exists P)\ (Viewing(V) \wedge PropertyForRent(P) \wedge (C.clientNo = V.clientNo) \wedge (V.propertyNo = P.propertyNo) \wedge P.city = \text{'Glasgow'}))\}$$

To answer this query, note that we can rephrase 'clients who have viewed a property in Glasgow' as 'clients for whom there exists some viewing of some property in Glasgow'.

(e) List all cities where there is either a branch office or a property for rent.

$$\{T.city \mid (\exists B)\ (Branch(B) \wedge B.city = T.city) \vee (\exists P)\ (PropertyForRent(P) \wedge P.city = T.city)\}$$

Compare this with the equivalent relational algebra expression given in Example 4.3.

(f) List all the cities where there is a branch office but no properties for rent.

$$\{B.city \mid Branch(B) \wedge (\sim(\exists P)\ (PropertyForRent(P) \wedge B.city = P.city))\}$$

Compare this with the equivalent relational algebra expression given in Example 4.4.

(g) List all the cities where there is both a branch office and at least one property for rent.

$$\{B.city \mid Branch(B) \wedge ((\exists P)\ (PropertyForRent(P) \wedge B.city = P.city))\}$$

Compare this with the equivalent relational algebra expression given in Example 4.5.

Safety of expressions

Before we complete this section, we should mention that it is possible for a calculus expression to generate an infinite set. For example:

{S | ~ Staff(S)}

would mean the set of all tuples that are not in the Staff relation. Such an expression is said to be **unsafe**. To avoid this, we have to add a restriction that all values that appear in the result must be values in the *domain* of the expression E, denoted *dom(E)*. In other words, the domain of E is the set of all values that appear explicitly in E or that appear in one or more relations whose names appear in E. In this example, the domain of the expression is the set of all values appearing in the Staff relation.

An expression is *safe* if all values that appear in the result are values from the domain of the expression. The above expression is not safe since it will typically include tuples from outside the Staff relation (and so outside the domain of the expression). All other examples of tuple relational calculus expressions in this section are safe. Some authors have avoided this problem by using range variables that are defined by a separate RANGE statement. The interested reader is referred to Date (2000).

Domain Relational Calculus 4.2.2

In the tuple relational calculus, we use variables that range over tuples in a relation. In the domain relational calculus, we also use variables but in this case the variables take their values from *domains* of attributes rather than tuples of relations. An expression in the domain relational calculus has the following general form:

$$\{d_1, d_2, \ldots, d_n \mid F(d_1, d_2, \ldots, d_m)\} \qquad m \geq n$$

where $d_1, d_2, \ldots, d_n, \ldots, d_m$ represent domain variables and $F(d_1, d_2, \ldots, d_m)$ represents a formula composed of atoms, where each atom has one of the following forms:

- $R(d_1, d_2, \ldots, d_n)$, where R is a relation of degree n and each d_i is a domain variable.
- $d_i \, \theta \, d_j$, where d_i and d_j are domain variables and θ is one of the comparison operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$); the domains d_i and d_j must have members that can be compared by θ.
- $d_i \, \theta \, c$, where d_i is a domain variable, c is a constant from the domain of d_i, and θ is one of the comparison operators.

We recursively build up formulae from atoms using the following rules:

- An atom is a formula.
- If F_1 and F_2 are formulae, so are their conjunction $F_1 \wedge F_2$, their disjunction $F_1 \vee F_2$, and the negation $\sim F_1$.
- If F is a formula with domain variable X, then $(\exists X)(F)$ and $(\forall X)(F)$ are also formulae.

Example 4.15 Domain relational calculus

In the following examples, we use the following shorthand notation:

$(\exists d_1, d_2, \ldots, d_n)$ in place of $(\exists d_1), (\exists d_2), \ldots, (\exists d_n)$

(a) Find the names of all managers who earn more than £25,000.

$$\{fN, lN \mid (\exists sN, posn, sex, DOB, sal, bN)\ (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) \wedge posn = \text{'Manager'} \wedge sal > 25000)\}$$

If we compare this query with the equivalent tuple relational calculus query in Example 4.12(a), we see that each attribute is given a (variable) name. The condition Staff(sN, fN, . . . , bN) ensures that the domain variables are restricted to be attributes of the same tuple. Thus, we can use the formula posn = 'Manager', rather than Staff.position = 'Manager'. Also note the difference in the use of the existential quantifier. In the tuple relational calculus, when we write ∃posn for some tuple variable posn, we bind the variable to the relation Staff by writing Staff(posn). On the other hand, in the domain relational calculus posn refers to a domain value and remains unconstrained until it appears in the subformula Staff(sN, fN, lN, posn, sex, DOB, sal, bN) when it becomes constrained to the position values that appear in the Staff relation.

For conciseness, in the remaining examples in this section we quantify only those domain variables that actually appear in a condition (in this example, posn and sal).

(b) List the staff who manage properties for rent in Glasgow.

$$\{sN, fN, lN, posn, sex, DOB, sal, bN \mid (\exists sN1, cty)\ (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) \wedge PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1) \wedge (sN = sN1) \wedge cty = \text{'Glasgow'})\}$$

This query can also be written as:

$$\{sN, fN, lN, posn, sex, DOB, sal, bN \mid (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) \wedge PropertyForRent(pN, st, \text{'Glasgow'}, pc, typ, rms, rnt, oN, sN, bN1))\}$$

In this version, the domain variable cty in PropertyForRent has been replaced with the constant 'Glasgow' and the same domain variable sN, which represents the staff number, has been repeated for Staff and PropertyForRent.

(c) List the names of staff who currently do not manage any properties for rent.

$$\{fN, lN \mid (\exists sN)\ (Staff(sN, fN, lN, posn, sex, DOB, sal, bN) \wedge (\sim(\exists sN1)\ (PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1) \wedge (sN = sN1))))\}$$

(d) List the names of clients who have viewed a property for rent in Glasgow.

$$\{fN, lN \mid (\exists cN, cN1, pN, pN1, cty)\ (Client(cN, fN, lN, tel, pT, mR) \wedge Viewing(cN1, pN1, dt, cmt) \wedge PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN, bN) \wedge (cN = cN1) \wedge (pN = pN1) \wedge cty = \text{'Glasgow'})\}$$

(e) List all cities where there is either a branch office or a property for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \vee PropertyForRent(pN, st1, cty, pc1, typ, rms, rnt, oN, sN, bN1))\}$$

(f) List all the cities where there is a branch office but no properties for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \wedge (\sim(\exists cty1)\ (PropertyForRent(pN, st1, cty1, pc1, typ, rms, rnt, oN, sN, bN1) \wedge (cty = cty1))))\}$$

(g) List all the cities where there is both a branch office and at least one property for rent.

$$\{cty \mid (Branch(bN, st, cty, pc) \wedge (\exists cty1)\ (PropertyForRent(pN, st1, cty1, pc1, typ, rms, rnt, oN, sN, bN1) \wedge (cty = cty1)))\}$$

These queries are **safe**. When the domain relational calculus is restricted to safe expressions, it is equivalent to the tuple relational calculus restricted to safe expressions, which in turn is equivalent to the relational algebra. This means that for every relational algebra expression there is an equivalent expression in the relational calculus, and for every tuple or domain relational calculus expression there is an equivalent relational algebra expression.

4.3 Other Languages

Although the relational calculus is hard to understand and use, it was recognized that its non-procedural property is exceedingly desirable, and this resulted in a search for other easy-to-use non-procedural techniques. This led to another two categories of relational languages: transform-oriented and graphical.

Transform-oriented languages are a class of non-procedural languages that use relations to transform input data into required outputs. These languages provide easy-to-use structures for expressing what is desired in terms of what is known. SQUARE (Boyce *et al.*, 1975), SEQUEL (Chamberlin *et al.*, 1976), and SEQUEL's offspring, SQL, are all transform-oriented languages. We discuss SQL in Chapters 5 and 6.

Graphical languages provide the user with a picture or illustration of the structure of the relation. The user fills in an example of what is wanted and the system returns the required data in that format. QBE (Query-By-Example) is an example of a graphical language (Zloof, 1977). We demonstrate the capabilities of QBE in Chapter 7.

Another category is **fourth-generation languages** (4GLs), which allow a complete customized application to be created using a limited set of commands in a user-friendly, often menu-driven environment (see Section 2.2). Some systems accept a form of *natural language*, a restricted version of natural English, sometimes called a **fifth-generation language** (5GL), although this development is still at an early stage.

Chapter Summary

- The **relational algebra** is a (high-level) procedural language: it can be used to tell the DBMS how to build a new relation from one or more relations in the database. The **relational calculus** is a non-procedural language: it can be used to formulate the definition of a relation in terms of one or more database relations. However, formally the relational algebra and relational calculus are equivalent to one another: for every expression in the algebra, there is an equivalent expression in the calculus (and vice versa).
- The relational calculus is used to measure the selective power of relational languages. A language that can be used to produce any relation that can be derived using the relational calculus is said to be **relationally complete**. Most relational query languages are relationally complete but have more expressive power than the relational algebra or relational calculus because of additional operations such as calculated, summary, and ordering functions.
- The five fundamental operations in relational algebra, *Selection*, *Projection*, *Cartesian product*, *Union*, and *Set difference*, perform most of the data retrieval operations that we are interested in. In addition, there are also the *Join*, *Intersection*, and *Division* operations, which can be expressed in terms of the five basic operations.
- The **relational calculus** is a formal non-procedural language that uses predicates. There are two forms of the relational calculus: tuple relational calculus and domain relational calculus.
- In the **tuple relational calculus**, we are interested in finding tuples for which a predicate is true. A tuple variable is a variable that 'ranges over' a named relation: that is, a variable whose only permitted values are tuples of the relation.
- In the **domain relational calculus**, domain variables take their values from domains of attributes rather than tuples of relations.
- The relational algebra is logically equivalent to a safe subset of the relational calculus (and vice versa).
- Relational data manipulation languages are sometimes classified as **procedural** or **non-procedural**, **transform-oriented**, **graphical**, **fourth-generation**, or **fifth-generation**.

Review Questions

4.1 What is the difference between a procedural and a non-procedural language? How would you classify the relational algebra and relational calculus?

4.2 Explain the following terms:
(a) relationally complete
(b) closure of relational operations.

4.3 Define the five basic relational algebra operations. Define the Join, Intersection, and Division operations in terms of these five basic operations.

4.4 Discuss the differences between the five Join operations: Theta join, Equijoin, Natural join, Outer join, and Semijoin. Give examples to illustrate your answer.

4.5 Compare and contrast the tuple relational calculus with domain relational calculus. In particular, discuss the distinction between tuple and domain variables.

4.6 Define the structure of a (well-formed) formula in both the tuple relational calculus and domain relational calculus.

4.7 Explain how a relational calculus expression can be unsafe. Illustrate your answer with an example. Discuss how to ensure that a relational calculus expression is safe.

Exercises

For the following exercises, use the Hotel schema defined at the start of the Exercises at the end of Chapter 3.

4.8 Describe the relations that would be produced by the following relational algebra operations:

(a) $\Pi_{hotelNo}(\sigma_{price > 50}(Room))$
(b) $\sigma_{Hotel.hotelNo = Room.hotelNo}(Hotel \times Room)$
(c) $\Pi_{hotelName}(Hotel \bowtie_{Hotel.hotelNo = Room.hotelNo}(\sigma_{price > 50}(Room)))$
(d) $Guest \rtimes (\sigma_{dateTo \geq '1\text{-}Jan\text{-}2002'}(Booking))$
(e) $Hotel \triangleright_{Hotel.hotelNo = Room.hotelNo}(\sigma_{price > 50}(Room))$
(f) $\Pi_{guestName, hotelNo}(Booking \bowtie_{Booking.guestNo = Guest.guestNo} Guest) \div \Pi_{hotelNo}(\sigma_{city = 'London'}(Hotel))$

4.9 Provide the equivalent tuple relational calculus and domain relational calculus expressions for each of the relational algebra queries given in Exercise 4.8.

4.10 Describe the relations that would be produced by the following tuple relational calculus expressions:

(a) $\{H.hotelName \mid Hotel(H) \wedge H.city = \text{'London'}\}$
(b) $\{H.hotelName \mid Hotel(H) \wedge (\exists R)\,(Room(R) \wedge H.hotelNo = R.hotelNo \wedge R.price > 50)\}$
(c) $\{H.hotelName \mid Hotel(H) \wedge (\exists B)\,(\exists G)\,(Booking(B) \wedge Guest(G) \wedge H.hotelNo = B.hotelNo \wedge B.guestNo = G.guestNo \wedge G.guestName = \text{'John Smith'})\}$
(d) $\{H.hotelName, G.guestName, B1.dateFrom, B2.dateFrom \mid Hotel(H) \wedge Guest(G) \wedge Booking(B1) \wedge Booking(B2) \wedge H.hotelNo = B1.hotelNo \wedge G.guestNo = B1.guestNo \wedge B2.hotelNo = B1.hotelNo \wedge B2.guestNo = B1.guestNo \wedge B2.dateFrom \neq B1.dateFrom\}$

4.11 Provide the equivalent domain relational calculus and relational algebra expressions for each of the tuple relational calculus expressions given in Exercise 4.10.

4.12 Generate the relational algebra, tuple relational calculus, and domain relational calculus expressions for the following queries:

(a) List all hotels.
(b) List all single rooms with a price below £20 per night.
(c) List the names and cities of all guests.
(d) List the price and type of all rooms at the Grosvenor Hotel.
(e) List all guests currently staying at the Grosvenor Hotel.
(f) 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.
(g) List the guest details (guestNo, guestName, and guestAddress) of all guests staying at the Grosvenor Hotel.

4.13 Using relational algebra, create a view of all rooms in the Grosvenor Hotel, excluding price details. What are the advantages of this view?

4.14 Analyze the RDBMSs that you are currently using. What types of relational language does the system provide? For each of the languages provided, what are the equivalent operations for the eight relational algebra operations defined in Section 4.1?

Chapter 5 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 3 and 4 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), which was subsequently adopted in 1987 as an international standard by the International Organization for Standardization (ISO, 1987). More than one hundred Database Management Systems now support SQL, running on various hardware platforms from PCs to mainframes.

Owing to the current importance of SQL, we devote three chapters of this book to examining the language in detail, providing a comprehensive treatment for both technical and non-technical 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 5.1 we introduce SQL and discuss why the language is so important to database applications. In Section 5.2 we introduce the notation used in this book to specify the structure of an SQL statement. In Section 5.3 we discuss how to retrieve data from relations using SQL, and how to insert, update, and delete data from relations.

Looking ahead, in Chapter 6 we examine other features of the language, including data definition, views, transactions, and access control. In Section 28.4 we examine in some detail the features that have been added to the SQL specification to support object-oriented data management, referred to as SQL:1999 or SQL3. In Appendix E 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 4 provide a foundation for a large part of the SQL standard and it may be useful to refer back 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 4. The examples in this chapter use the *DreamHome* rental database instance shown in Figure 3.3.

Introduction to SQL 5.1

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.

Objectives of SQL 5.1.1

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 SQL:1999, 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 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 E. We also discuss SQL:1999 and SQL:2003 in Chapter 28.

SQL is a relatively easy language to learn:

- It is a non-procedural 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, 1999a).

5.1.2 History of SQL

As stated in Chapter 3, the history of the relational model (and indirectly SQL) started with the publication of the seminal paper by E. F. Codd, while working 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 pre-dates

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, the American National Standards Institute 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 was formalized, commonly referred to as SQL:1999 (ISO, 1999a). This release contains additional features to support object-oriented data management, which we examine in Section 28.4. A further release, SQL:2003, was produced in late 2003.

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:2003 has a set of features called **Core SQL** that a vendor must implement to claim **conformance** with the SQL:2003 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.

5.1.3 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 X/OPEN 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 US 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).

5.1.4 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 3. 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.

5.2 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 spelt *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 lower case. 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 chapter, we use the following extended form of the Backus Naur Form (BNF) notation to define SQL statements:

- upper-case letters are used to represent reserved words and must be spelt exactly as shown;
- lower-case 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.

5.3 Data Manipulation

This section looks at the SQL DML statements, namely:

- 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 3.3, 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)
PrivateOwner	(ownerNo, fName, lName, address, telNo)
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 6.1.1). However, for simplicity, we can distinguish between literals that are enclosed in single quotes and those that are not. All non-numeric 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.

5.3.1 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 4.1). 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 4.1). There are many variations of this statement, as we now illustrate.

Retrieve all rows

Example 5.1 Retrieve all columns, all rows

List full details of all staff.

Since 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;
```

Since 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 5.1.

Table 5.1 Result table for Example 5.1.

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 5.2 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 5.2. 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 5.2 Result table for Example 5.2.

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 5.3 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 5.3(a). Notice that there are several duplicates because, unlike the relational algebra Projection operation (see Section 4.1.1), SELECT does not 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 5.3(b) with the duplicates eliminated.

Table 5.3(a) Result table for Example 5.3 with duplicates.

propertyNo
PA14
PG4
PG4
PA14
PG36

Table 5.3(b) Result table for Example 5.3 with duplicates eliminated.

propertyNo
PA14
PG4
PG36

Example 5.4 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 5.2, 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 5.4.

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 in the table

Table 5.4 Result table for Example 5.4.

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

(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 above 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 4.1.1. We now present examples of each of these types of search conditions.

Example 5.5 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 5.5.

Table 5.5 Result table for Example 5.5.

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 5.6 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 5.6.

Table 5.6 Result table for Example 5.6.

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 5.7 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 5.7.

Table 5.7 Result table for Example 5.7.

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 above 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 5.8 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 5.8.

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 above query as:

Table 5.8 Result table for Example 5.8.

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

```
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 5.9 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:

- % percent character represents any sequence of zero or more characters (*wildcard*).
- _ 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 5.9:

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

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

Table 5.9 Result table for Example 5.9.

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 5.10 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 3.3, 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 5.10. The negated version (IS NOT NULL) can be used to test for values that are not null.

Table 5.10 Result table for Example 5.10.

clientNo	viewDate
CR56	26-May-04

Sorting Results (ORDER BY Clause) 5.3.2

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

Example 5.11 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 5.2. 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 5.11. 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 5.11 Result table for Example 5.11.

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

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

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 5.11, 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 5.12 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 5.12(a).

Table 5.12(a) Result table for Example 5.12 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

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;
```

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 5.12(b).

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

Table 5.12(b) Result table for Example 5.12 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

Using the SQL Aggregate Functions 5.3.3

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 non-numeric 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 non-null values. COUNT(*) is a special use of COUNT, which 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 5.3.4). 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 5.3.4), 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 5.13 Use of COUNT(*)

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

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

Table 5.13 Result table for Example 5.13.

myCount
5

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 5.13.

Example 5.14 Use of COUNT(DISTINCT)

How many different properties were viewed in May 2004?

```
SELECT COUNT(DISTINCT propertyNo) AS myCount
FROM Viewing
WHERE viewDate BETWEEN '1-May-04' AND '31-May-04';
```

Table 5.14 Result table for Example 5.14.

myCount
2

Again, restricting the query to viewings that occurred in May 2004 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 5.14.

Example 5.15 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';
```

Table 5.15 Result table for Example 5.15.

myCount	mySum
2	54000.00

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 5.15.

Example 5.16 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 5.16.

Table 5.16 Result table for Example 5.16.

myMin	myMax	myAvg
9000.00	30000.00	17000.00

5.3.4 Grouping Results (GROUP BY Clause)

The above 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**. Further, the SELECT clause may contain only:

- column names;
- aggregate functions;
- constants;
- an expression involving combinations of the above.

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 non-null grouping columns, they are combined into the same group.

Example 5.17 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;
```

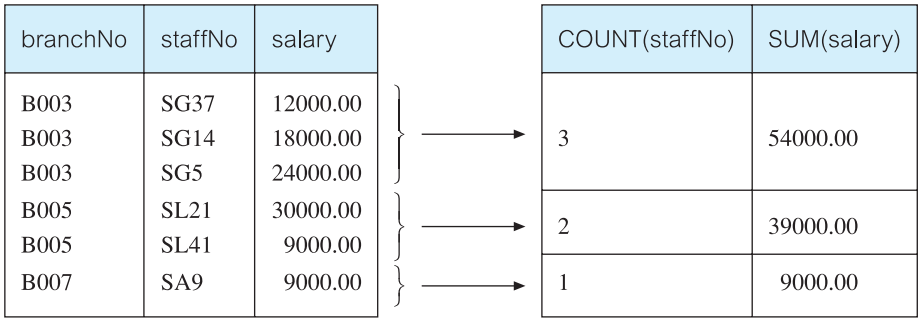
It is not necessary to include the column names staffNo and salary in the GROUP BY list because they appear only in the SELECT list within aggregate functions. On the other hand, branchNo is not associated with an aggregate function and so must appear in the GROUP BY list. The result table is shown in Table 5.17.

Table 5.17 Result table for Example 5.17.

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 5.3.5). Therefore, we could also express the above 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 5.18 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 and so the HAVING clause is used. The result table is shown in Table 5.18.

Table 5.18 Result table for Example 5.18.

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00

5.3.5 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 subselect 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 5.3.10). 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 5.19 uses a scalar subquery.
- A *row subquery* returns multiple columns, but again 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 5.19 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 5.19.

Table 5.19 Result table for Example 5.19.

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 5.20 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 5.20.

Table 5.20 Result table for Example 5.20.

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 5.3.8).

(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 below).

(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 last 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 5.21 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 5.21.

Table 5.21 Result table for Example 5.21.

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

5.3.6 ANY and ALL

The words 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 only be true 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 5.22 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');
```

While 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 5.20), 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 5.22.

Table 5.22 Result table for Example 5.22.

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

Example 5.23 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 is very similar to the last 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 5.23.

Table 5.23 Result table for Example 5.23.

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00

5.3.7 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 not sufficient. 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 5.24 Simple join

List the names of all clients who have viewed a property along with any comment 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 equally well 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 4.1.3. The result table is shown in Table 5.24.

Table 5.24 Result table for Example 5.24.

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 11.6.2). 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 3.2.5 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 5.24, 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 5.25 Sorting a join

For each branch office, list the 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;
```

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 5.25.

Table 5.25 Result table for Example 5.25.

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 5.26 Three-table join

For each branch, list the 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, Staff, and PropertyForRent, 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 5.26.

Table 5.26 Result table for Example 5.26.

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 5.27 Multiple grouping columns

Find the number of properties handled by each staff member.

```
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 5.27(a).

Table 5.27(a) Result table for Example 5.27.

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 4.1.2). 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 5.24, where we joined the Client and Viewing tables using the matching column, clientNo. Using the data from Figure 3.3, 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 5.24 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.

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 above. The ISO standard provides another set of join operators called **outer joins** (see Section 4.1.3). 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 5.27(b).

Table 5.27(b) Result table for inner join of 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 5.28 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 5.28. 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 5.28 Result table for Example 5.28.

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

Example 5.29 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 5.29. 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 5.29 Result table for Example 5.29.

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

Example 5.30 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 5.30. 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 5.30 Result table for Example 5.30.

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

5.3.8 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. Since EXISTS and NOT EXISTS check only for the existence or non-existence 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 5.31 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 5.31.

Table 5.31 Result table for Example 5.31.

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

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

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

which is equivalent to:

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

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

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

5.3.9 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 5.1. 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.

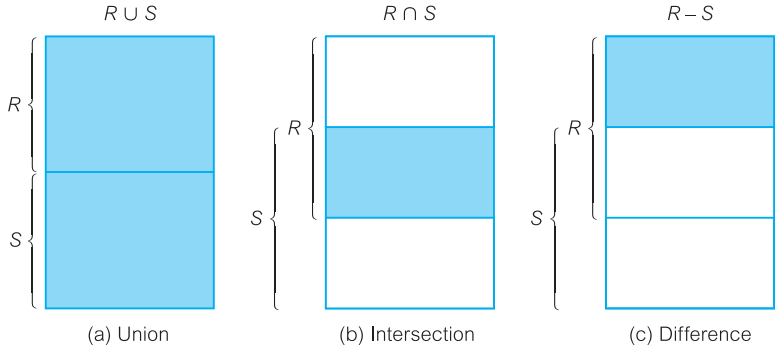


Figure 5.1 Union, intersection, and difference set operations.

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 5.32 Use of UNION

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

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

This query is executed by producing a result table from the first query and a result table from the second query, and then merging both tables into a single result table consisting of all the rows from both result tables with the duplicate rows removed. The final result table is shown in Table 5.32.

Table 5.32 Result table for Example 5.32.

city
London
Glasgow
Aberdeen
Bristol

Example 5.33 Use of INTERSECT

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

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

This query is executed by producing a result table from the first query and a result table from the second query, and then creating a single result table consisting of those rows that are common to both result tables. The final result table is shown in Table 5.33.

Table 5.33 Result table for Example 5.33.

city
Aberdeen
Glasgow
London

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 5.34 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 5.34.

Table 5.34 Result table for Example 5.34.

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);
```

Database Updates 5.3.10

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 6.4), 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 6.3.2). 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 5.35 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 5.36 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 only into 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 5.37 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 only a list of 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 5.35.

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

Table 5.35 Result table for Example 5.37.

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 6.4). 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 5.38 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 5.39 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 5.40 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 6.4). 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 6.3.3). If a *searchCondition* is specified, only those rows that satisfy the condition are deleted.

Example 5.41 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 5.42 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 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 non-procedural language, consisting of standard English words such as SELECT, INSERT, 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 again 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

5.1 What are the two major components of SQL and what function do they serve?

5.2 What are the advantages and disadvantages of SQL?

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

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

5.5 Explain how the GROUP BY clause works. What is the difference between the WHERE and HAVING clauses?

5.6 What is the difference between a subquery and a join? Under what circumstances would you not be able to use a subquery?

Exercises

For Exercises 5.7–5.28, use the Hotel schema defined at the start of the Exercises at the end of Chapter 3.

Simple queries

5.7 List full details of all hotels.

5.8 List full details of all hotels in London.

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

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

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

Aggregate functions

5.12 How many hotels are there?

5.13 What is the average price of a room?

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

5.15 How many different guests have made bookings for August?

Subqueries and joins

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

5.17 List all guests currently staying at the Grosvenor Hotel.

5.18 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.

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

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

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

Grouping

5.22 List the number of rooms in each hotel.

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

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

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

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

Populating tables

5.27 Insert rows into each of these tables.

5.28 Update the price of all rooms by 5%.

General

5.29 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 the DBMS supports. Are there any functions not supported?

5.30 Show that a query using the HAVING clause has an equivalent formulation without a HAVING clause.

5.31 Show that SQL is relationally complete.

Chapter

6 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 the Structured Query Language (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 6.1 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 Integrity Enhancement Feature in Section 6.2 and the main SQL data definition facilities in Section 6.3.

In Section 6.4 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 6.5, 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, restrict access to database objects. We discuss the access control subsystem in Section 6.6.

In Section 28.4 we examine in some detail the features that have recently been added to the SQL specification to support object-oriented data management, often covering SQL:1999 and SQL:2003. In Appendix E 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 5.2.

6.1 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.

6.1.1 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 upper-case letters A . . . Z, the lower-case 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 6.1 ISO SQL data types.

Data type	Declarations			
boolean	BOOLEAN			
character	CHAR	VARCHAR		
bit†	BIT	BIT VARYING		
exact numeric	NUMERIC	DECIMAL	INTEGER	SMALLINT
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.

SQL Scalar Data Types 6.1.2

Table 6.1 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:2003 standard also defines both character large objects and binary large objects, although we defer discussion of these data types until Section 28.4.

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)

6.1.3 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

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. 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, 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, 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; and:

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).

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 6.2 are available.

Table 6.2 ISO SQL scalar operators.

Operator	Meaning
BIT_LENGTH	Returns the length of a string in bits. For example, **BIT_LENGTH**(X'FFFF') returns 16.
OCTET_LENGTH	Returns the length of a string in octets (bit length divided by 8). For example, **OCTET_LENGTH**(X'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 upper-case letters to lower-case. 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).

6.2 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 3.3):

- 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 see shortly.

6.2.1 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 3.3.1). 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**

6.2.2 Domain Constraints

Every column has a domain, in other words a set of legal values (see Section 3.2.1). 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 only be specified 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 6.1.2), 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 above example, we could define a domain for sex as:

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

This 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 6.2.5), 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.

6.2.3 Entity Integrity

The primary key of a table must contain a unique, non-null value for each row. 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 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)
```

6.2.4 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 3.3.3). 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. Since 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 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 is valid only if the foreign key columns have a DEFAULT value specified (see Section 6.3.2).
- 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**

General Constraints 6.2.5

Updates to tables may be constrained by enterprise rules governing the real-world transactions that are represented by the updates. 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 have already 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 above. 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.

6.3 Data Definition

The SQL Data Definition Language (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 them here as these commands are system specific.

6.3.1 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 above. 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. The CREATE and DROP SCHEMA statements are not yet widely implemented.

Creating a Table (CREATE TABLE) 6.3.2

Having created the database structure, we may now create the table structures for the base relations to be located in the database. This 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 6.1.2. 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 below).

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 6.2).
- 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 6.5). Example 6.1 demonstrates the potential of this version of the CREATE TABLE statement.

Example 6.1 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 that the number of properties the staff member currently handles is not greater 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.

6.3.3 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]
```

Here 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 6.2 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.

6.3.4 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 5.3.10). The DROP TABLE statement allows you to specify whether the DROP action is to be cascaded or not:

- 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.

Creating an Index (CREATE INDEX) 6.3.5

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 C 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, since 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.

6.3.6 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 format:

DROP INDEX IndexName

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

DROP INDEX RentInd;

6.4 Views

Recall from Section 3.4 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 6.4.3. 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 6.4.8. First, we examine how to create and use views.

6.4.1 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 6.4.6). 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 6.6. Although all views are created in the same way, in practice different types of view are used for different purposes. We illustrate the different types of view with examples.

Example 6.3 Create a horizontal view

Create a view so that the manager at branch B003 can see only the details 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 the statement:

```
SELECT * FROM Manager3Staff;
```

we would get the result table shown in Table 6.3. 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 6.6.

Table 6.3 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 6.4 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 would get the result table shown in Table 6.4. 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 6.4 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 6.5 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 5.27).

```
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 gives the data shown in Table 6.5. This example 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 6.5 Data for view StaffPropCnt.

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

6.4.2 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 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.

6.4.3 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 that 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 6.5:

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

View resolution merges the above 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 6.6.

Table 6.6 Result table after view resolution.

staffNo	cnt
SG14	1
SG37	2

6.4.4 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 6.5, 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 in 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 that any attempt to join this view with another table or view fails.

6.4.5 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 then the base table(s) will reflect that change. However, consider again the view StaffPropCnt of Example 6.5. 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:

```
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');
```

then 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 6.1). However, as the StaffPropList view excludes all columns from the PropertyForRent table except the property number, we have no way of providing the remaining non-null 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.

WITH CHECK OPTION 6.4.6

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 cause 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 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 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 6.6 WITH CHECK OPTION

Consider again the view created in Example 6.3:

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

with the virtual table shown in Table 6.3. 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 this 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.

6.4.7 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 6.7.

Table 6.7 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. The major advantages of views are described below.

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, 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 form:

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

we could reorganize it into two new tables:

ClientDetails (clientNo, fName, lName, telNo)
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
    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 6.4.5 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.

6.4.8 View Materialization

In Section 6.4.3 we discussed one approach to handling queries based on a view, where 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 that 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;
```

with the data shown in Table 6.8. 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.

Table 6.8 Data for view StaffPropRent.

staffNo
SG37
SG14

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).

6.5 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 E), 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 20.4). 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).

The *isolation level* indicates the degree of interaction that is allowed from other transactions during the execution of the transaction. Table 6.9 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 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

Table 6.9 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

can be used by the programmer to ensure serializability. Chapter 20 provides additional information on transactions and serializability.

Immediate and Deferred Integrity Constraints 6.5.1

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}
```

Discretionary Access Control 6.6

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, 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 LaPadula, 1974), which we discuss further in Chapter 19.

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 Database Administrator (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 6.3.1). 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.

Granting Privileges to Other Users (GRANT) 6.6.1

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 6.7 GRANT all privileges

Give the user with authorization identifier Manager *full privileges to 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 6.8 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 6.9 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.

6.6.2 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.

The RESTRICT and CASCADE qualifiers operate exactly as in the DROP TABLE statement (see Section 6.3.3). Since 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 6.1 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.

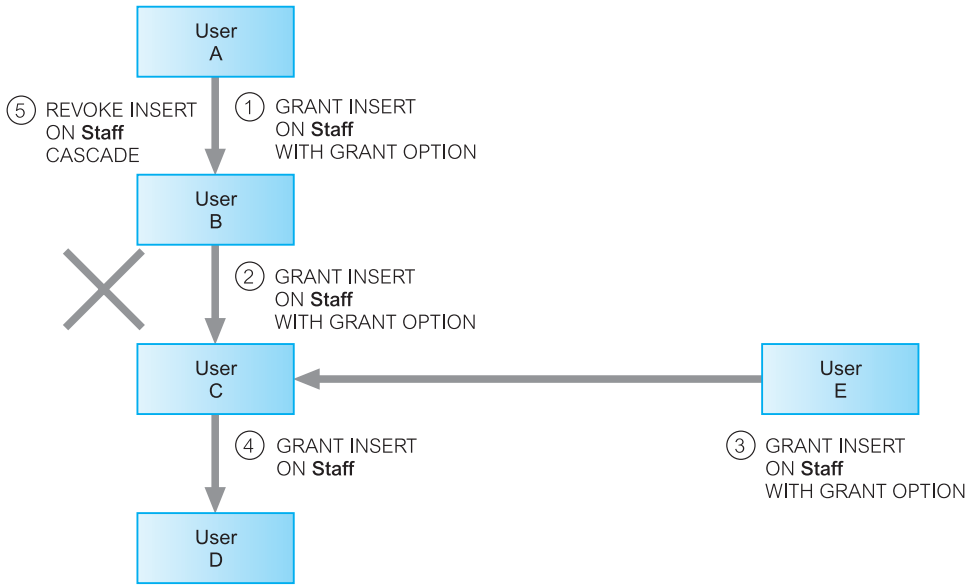


Figure 6.1
Effects of REVOKE.

Example 6.10 REVOKE specific privileges from PUBLIC

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

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

Example 6.11 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; the latter three 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

6.1 Describe the eight base data types in SQL.
6.2 Discuss the functionality and importance of the Integrity Enhancement Feature (IEF).
6.3 Discuss each of the clauses of the CREATE TABLE statement.
6.4 Discuss the advantages and disadvantages of views.
6.5 Describe how the process of view resolution works.
6.6 What restrictions are necessary to ensure that a view is updatable?
6.7 What is a materialized view and what are the advantages of a maintaining a materialized view rather than using the view resolution process?
6.8 Describe the difference between discretionary and mandatory access control. What type of control mechanism does SQL support?
6.9 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 3:

6.10 Create the Hotel table using the integrity enhancement features of SQL.
6.11 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.

6.12 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 2003. Delete all bookings before 1 January 2003 from the Booking table.

6.13 Create a view containing the hotel name and the names of the guests staying at the hotel.

6.14 Create a view containing the account for each guest at the Grosvenor Hotel.

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

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

6.17 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;
```

General

6.18 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.

6.19 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.

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

6.21 Create the *DreamHome* rental database schema defined in Section 3.2.6 and insert the tuples shown in Figure 3.3.

6.22 Using the schema you have created above, run the SQL queries given in the examples in Chapter 5.

6.23 Create the schema for the Hotel schema given at the start of the exercises for Chapter 3 and insert some sample tuples. Now run the SQL queries that you produced for Exercises 5.7–5.28.

Chapter 7 Query-By-Example

Chapter Objectives

In this chapter you will learn:

- The main features of Query-By-Example (QBE).
- The types of query provided by the Microsoft Office Access DBMS QBE facility.
- How to use QBE to build queries to select fields and records.
- How to use QBE to target single or multiple tables.
- How to perform calculations using QBE.
- How to use advanced QBE facilities including parameter, find matched, find unmatched, crosstab, and autolookup queries.
- How to use QBE action queries to change the content of tables.

In this chapter, we demonstrate the major features of the Query-By-Example (QBE) facility using the Microsoft Office Access 2003 DBMS. QBE represents a visual approach for accessing data in a database through the use of query templates (Zloof, 1977). We use QBE by entering example values directly into a query template to represent what the access to the database is to achieve, such as the answer to a query.

QBE was developed originally by IBM in the 1970s to help users in their retrieval of data from a database. Such was the success of QBE that this facility is now provided, in one form or another, by the most popular DBMSs including Microsoft Office Access. The Office Access QBE facility is easy to use and has very powerful capabilities. We can use QBE to ask questions about the data held in one or more tables and to specify the fields we want to appear in the answer. We can select records according to specific or non-specific criteria and perform calculations on the data held in tables. We can also use QBE to perform useful operations on tables such as inserting and deleting records, modifying the values of fields, or creating new fields and tables. In this chapter we use simple examples to demonstrate these facilities. We use the sample tables shown in Figure 3.3 of the *DreamHome* case study, which is described in detail in Section 10.4 and Appendix A.

When we create a query using QBE, in the background Microsoft Office Access constructs the equivalent SQL statement. SQL is a language used in the querying, updating, and management of relational databases. In Chapters 5 and 6 we presented a comprehensive overview of the SQL standard. We display the equivalent Microsoft Office Access

SQL statement alongside every QBE example discussed in this chapter. However, we do not discuss the SQL statements in any detail but refer the interested reader to Chapters 5 and 6.

Although this chapter uses Microsoft Office Access to demonstrate QBE, in Section 8.1 we present a general overview of the other facilities of Microsoft Office Access 2003 DBMS. Also, in Chapters 17 and 18 we illustrate by example the physical database design methodology presented in this book, using Microsoft Office Access as one of the target DBMSs.

Structure of this Chapter

In Section 7.1 we present an overview of the types of QBE queries provided by Microsoft Office Access 2003, and in Section 7.2, we demonstrate how to build simple select queries using the QBE grid. In Section 7.3 we illustrate the use of advanced QBE queries (such as crosstab and autolookup), and finally in Section 7.4 we examine action queries (such as update and make-table).

7.1 Introduction to Microsoft Office Access Queries

When we create or open a database using Microsoft Office Access, the Database window is displayed showing the objects (such as tables, forms, queries, and reports) in the database. For example, when we open the *DreamHome* database, we can view the tables in this database, as shown in Figure 7.1.

To ask a question about data in a database, we design a query that tells Microsoft Office Access what data to retrieve. The most commonly used queries are called *select queries*. With select queries, we can view, analyze, or make changes to the data. We can view data from a single table or from multiple tables. When a select query is run, Microsoft Office Access collects the retrieved data in a *dynaset*. A dynaset is a dynamic view of the data from one or more tables, selected and sorted as specified by the query. In other words, a dynaset is an updatable set of records defined by a table or a query that we can treat as an object.

As well as select queries, we can also create many other types of useful queries using Microsoft Office Access. Table 7.1 presents a summary of the types of query provided by Microsoft Office Access 2003. These queries are discussed in more detailed in the following sections, with the exception of SQL-specific queries.

When we create a new query, Microsoft Office Access displays the New Query dialog box shown in Figure 7.2. From the options shown in the dialog box, we can start from scratch with a blank object and build the new query ourselves by choosing Design View or use one of the listed Office Access Wizards to help build the query.

A Wizard is like a database expert who asks questions about the query we want and then builds the query based on our responses. As shown in Figure 7.2, we can use Wizards

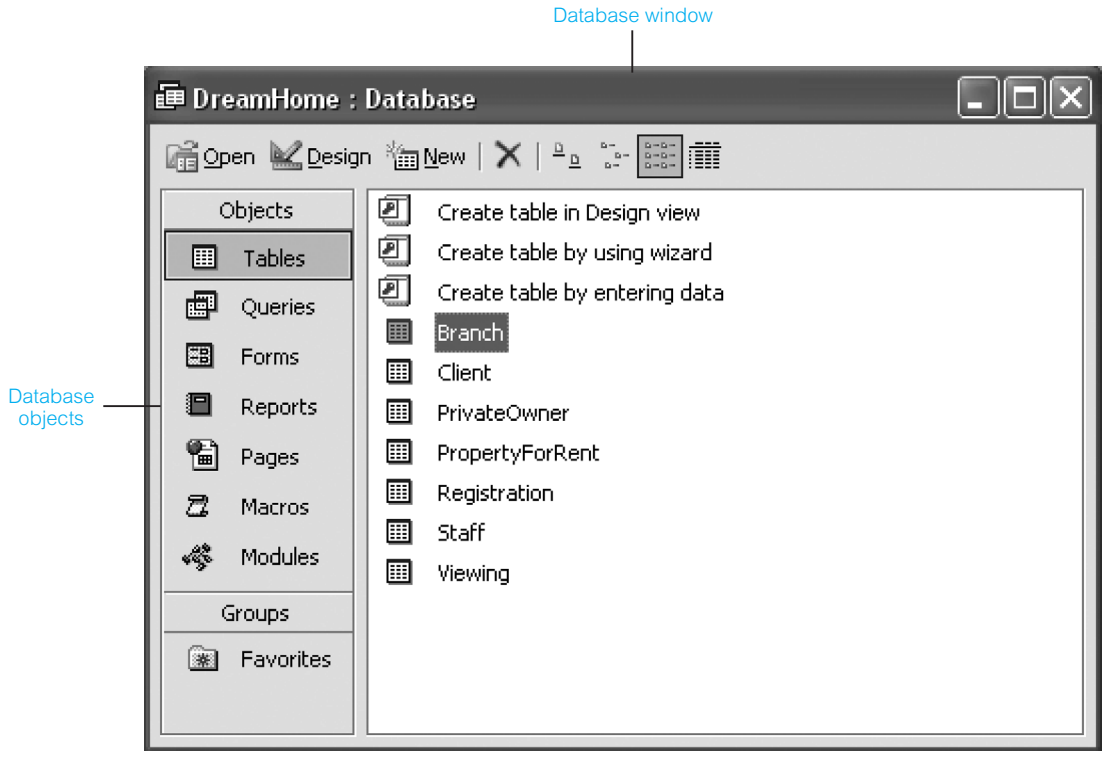


Figure 7.1
Microsoft Office Access Database window of the tables in the *DreamHome* database.

Table 7.1 Summary of Microsoft Office Access 2003 query types.

Query type	Description
Select query	Asks a question or defines a set of criteria about the data in one or more tables.
Totals (Aggregate) query	Performs calculations on groups of records.
Parameter query	Displays one or more predefined dialog boxes that prompts the user for the parameter value(s).
Find Matched query	Finds duplicate records in a single table.
Find Unmatched query	Finds distinct records in related tables.
Crosstab query	Allows large amounts of data to be summarized and presented in a compact spreadsheet.
Autolookup query	Automatically fills in certain field values for a new record.
Action query (including delete, append, update, and make-table queries)	Makes changes to many records in just one operation. Such changes include the ability to delete, append, or make changes to records in a table and also to create a new table.
SQL query (including union, pass-through, data definition, and subqueries)	Used to modify the queries described above and to set the properties of forms and reports. Must be used to create SQL-specific queries such as union, data definition, subqueries (see Chapters 5 and 6), and pass-through queries. Pass-through queries send commands to a SQL database such as Microsoft or Sybase SQL Server.

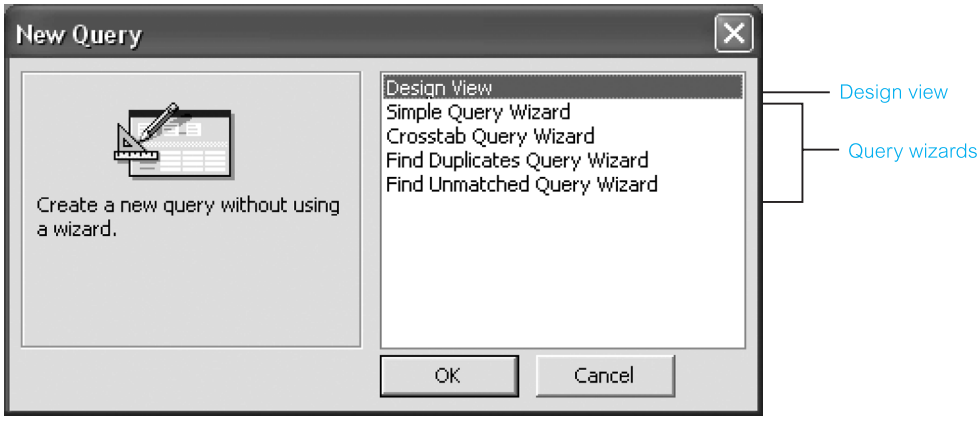


Figure 7.2
Microsoft Office Access New Query dialog box.

to help build simple select queries, crosstab queries, or queries that find duplicates or unmatched records within tables. Unfortunately, Query Wizards are of limited use when we want to build more complex select queries or other useful types of query such as parameter queries, autolookup queries, or action queries.

7.2 Building Select Queries Using QBE

A **select query** is the most common type of query. It retrieves data from one or more tables and displays the results in a *datasheet* where we can update the records (with some restrictions). A datasheet displays data from the table(s) in columns and rows, similar to a spreadsheet. A select query can also group records and calculate sums, counts, averages, and other types of total.

As stated in the previous section, simple select statements can be created using the Simple Query Wizard. However, in this section we demonstrate the building of simple select queries from scratch using Design View, without the use of the Wizards. After reading this section, the interested reader may want to experiment with the available Wizards to determine their usefulness.

When we begin to build the query from scratch, the Select Query window opens and displays a dialog box, which in our example lists the tables and queries in the *DreamHome* database. We then select the tables and/or queries that contain the data that we want to add to the query.

The Select Query window is a graphical Query-By-Example (QBE) tool. Because of its graphical features, we can use a mouse to select, drag, or manipulate objects in the window to define an example of the records we want to see. We specify the fields and records we want to include in the query in the QBE grid.

When we create a query using the QBE design grid, behind the scenes Microsoft Office Access constructs the equivalent SQL statement. We can view or edit the SQL statement in SQL view. Throughout this chapter, we display the equivalent SQL statement for

every query built using the QBE grid or with the help of a Wizard (as demonstrated in later sections of this chapter). Note that many of the Microsoft Office Access SQL statements displayed throughout this chapter do not comply with the SQL standard presented in Chapters 5 and 6.

7.2.1 Specifying Criteria

Criteria are restrictions we place on a query to identify the specific fields or records we want to work with. For example, to view only the property number, city, type, and rent of all properties in the PropertyForRent table, we construct the QBE grid shown in Figure 7.3(a). When this select query is run, the retrieved data is displayed as a datasheet of the selected fields of the PropertyForRent table, as shown in Figure 7.3(b). The equivalent SQL statement for the QBE grid shown in Figure 7.3(a) is given in Figure 7.3(c).

Note that in Figure 7.3(a) we show the complete Select Query window with the target table, namely PropertyForRent, displayed above the QBE grid. In some of the examples that follow, we show only the QBE grid where the target table(s) can be easily inferred from the fields displayed in the grid.

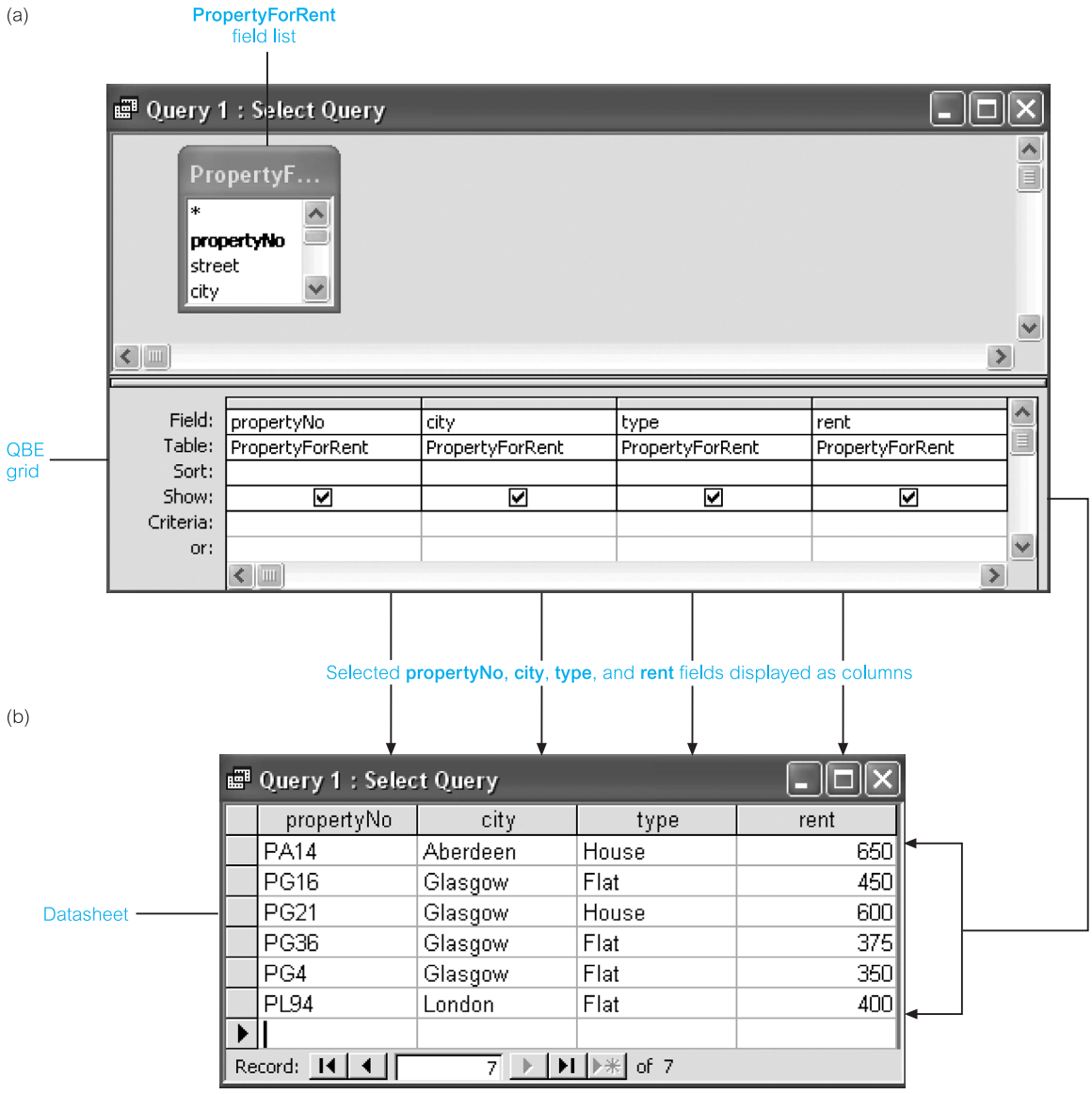
We can add additional criteria to the query shown in Figure 7.3(a) to view only properties in Glasgow. To do this, we specify criteria that limits the results to records whose city field contains the value 'Glasgow' by entering this value in the *Criteria* cell for the city field of the QBE grid. We can enter additional criteria for the same field or different fields. When we enter expressions in more than one Criteria cell, Microsoft Office Access combines them using either:

- the ***And* operator**, if the expressions are in different cells in the same row, which means only the records that meet the criteria in all the cells will be returned;
- the ***Or* operator**, if the expressions are in different rows of the design grid, which means records that meet criteria in any of the cells will be returned.

For example, to view properties in Glasgow with a rent between £350 and £450, we enter 'Glasgow' into the Criteria cell of the city field and enter the expression 'Between 350 And 450' in the Criteria cell of the rent field. The construction of this QBE grid is shown in Figure 7.4(a) and the resulting datasheet containing the records that satisfy the criteria is shown in Figure 7.4(b). The equivalent SQL statement for the QBE grid is shown in Figure 7.4(c).

Suppose that we now want to alter this query to also view all properties in Aberdeen. We enter 'Aberdeen' into the o*r* row below 'Glasgow' in the city field. The construction of this QBE grid is shown in Figure 7.5(a) and the resulting datasheet containing the records that satisfy the criteria is shown in Figure 7.5(b). The equivalent SQL statement for the QBE grid is given in Figure 7.5(c). Note that in this case, the records retrieved by this query satisfy the criteria 'Glasgow' in the city field *And* 'Between 350 And 450' in the rent field *Or* alternatively only 'Aberdeen' in the city field.

We can use *wildcard* characters or the *LIKE* operator to specify a value we want to find and we either know only part of the value or want to find values that start with a specific

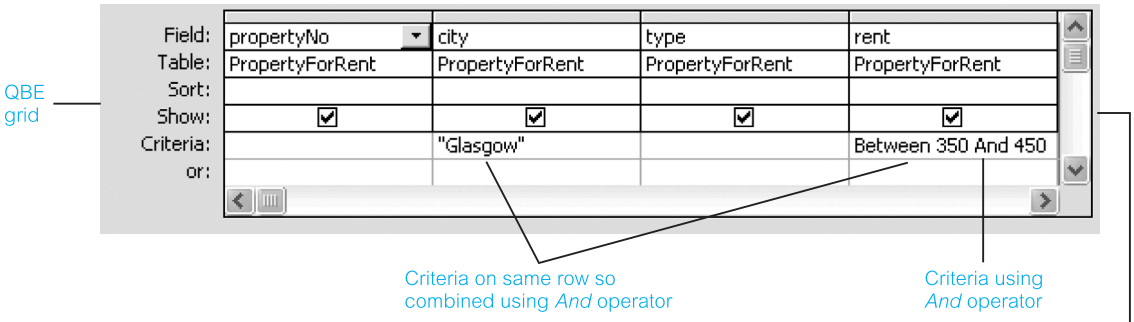


(c)

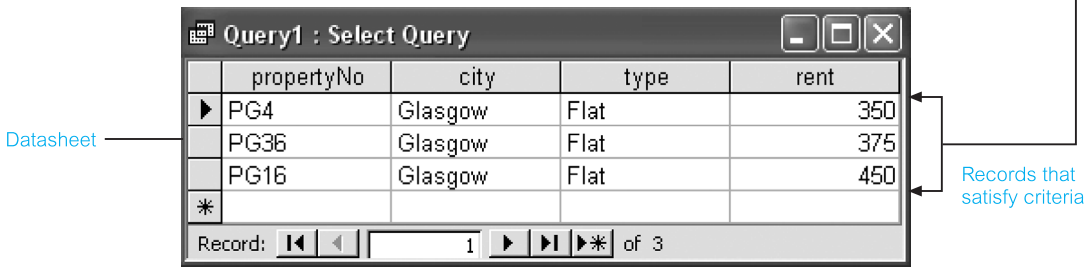
```
SELECT PropertyForRent.propertyNo, PropertyForRent.city, PropertyForRent.type, PropertyForRent.rent
FROM PropertyForRent;
```

Figure 7.3 (a) QBE grid to retrieve the propertyNo, city, type, and rent fields of the PropertyForRent table; (b) resulting datasheet; (c) equivalent SQL statement.

(a)



(b)



(c)

```
SELECT PropertyForRent.propertyNo, PropertyForRent.city, PropertyForRent.type, PropertyForRent.rent
FROM PropertyForRent
WHERE (((PropertyForRent.city)="Glasgow") AND ((PropertyForRent.rent) Between 350 And 450));
```

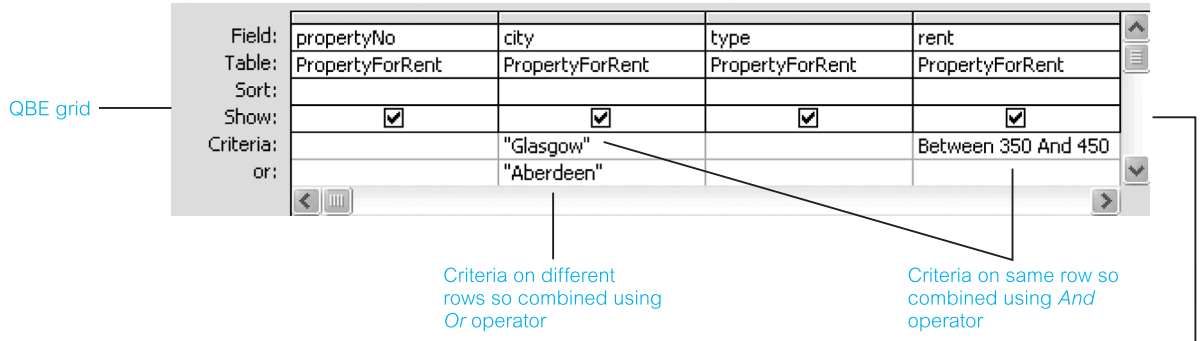
Figure 7.4 (a) QBE grid of select query to retrieve the properties in Glasgow with a rent between £350 and £450; (b) resulting datasheet; (c) equivalent SQL statement.

letter or match a certain pattern. For example, if we want to search for properties in Glasgow but we are unsure of the exact spelling for 'Glasgow', we can enter 'LIKE Glasgo' into the Criteria cell of the city field. Alternatively, we can use wildcard characters to perform the same search. For example, if we were unsure about the number of characters in the correct spelling of 'Glasgow', we could enter 'Glasg*' as the criteria. The wildcard (*) specifies an unknown number of characters. On the other hand, if we did know the number of characters in the correct spelling of 'Glasgow', we could enter 'Glasg??'. The wildcard (?) specifies a single unknown character.

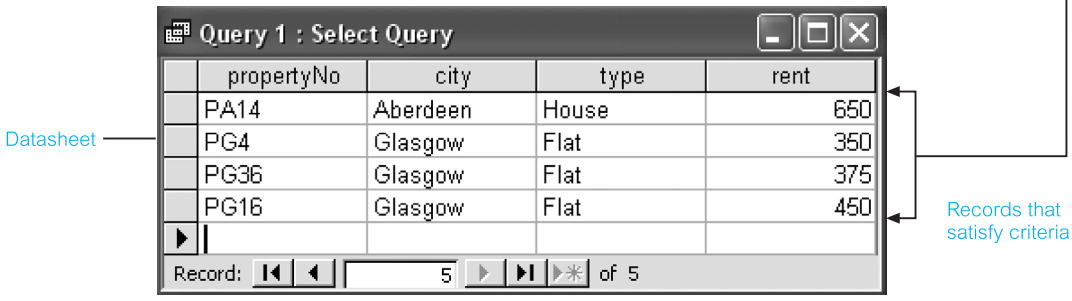
7.2.2 Creating Multi-Table Queries

In a database that is correctly normalized, related data may be stored in several tables. It is therefore essential that in answering a query, the DBMS is capable of joining related data stored in different tables.

(a)



(b)



(c)

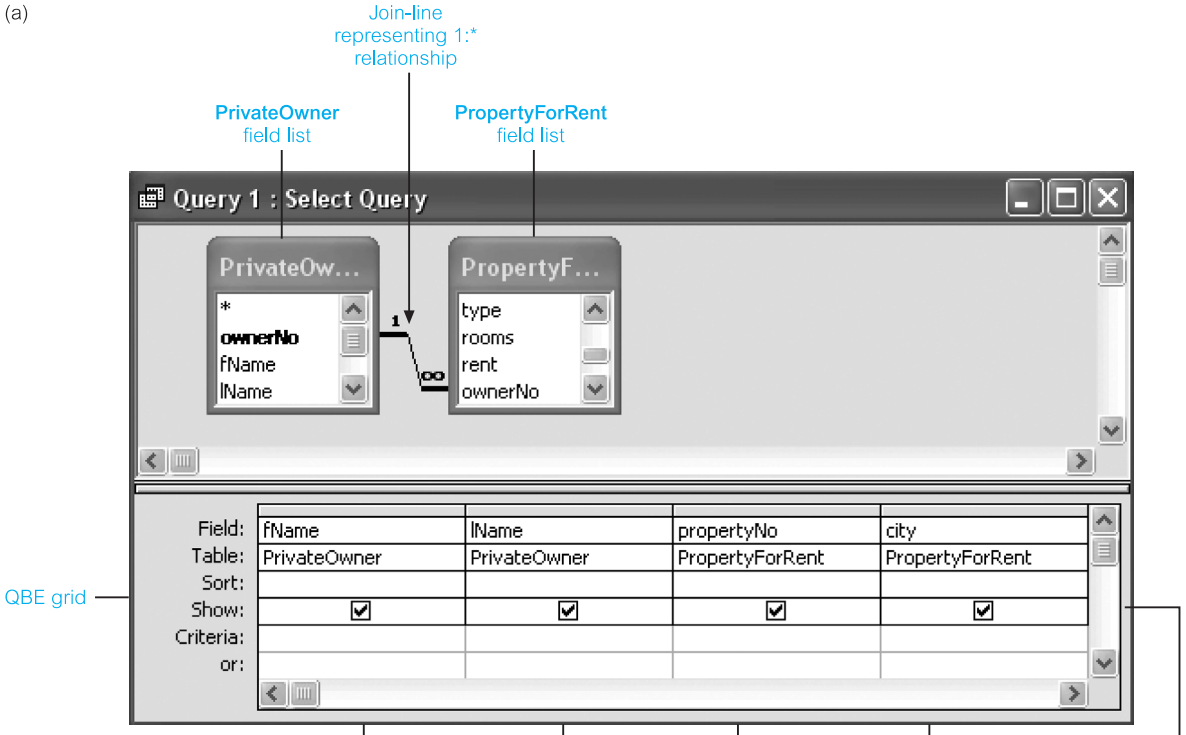
```
SELECT PropertyForRent.propertyNo, PropertyForRent.city, PropertyForRent.type, PropertyForRent.rent
FROM PropertyForRent
WHERE (((PropertyForRent.city)="Glasgow") AND ((PropertyForRent.rent) Between 350 And 450)) OR
(((PropertyForRent.city)="Aberdeen"));
```

Figure 7.5
(a) QBE grid of select query to retrieve the properties in Glasgow with a rent between £350 and £450 and all properties in Aberdeen; (b) resulting datasheet; (c) equivalent SQL statement.

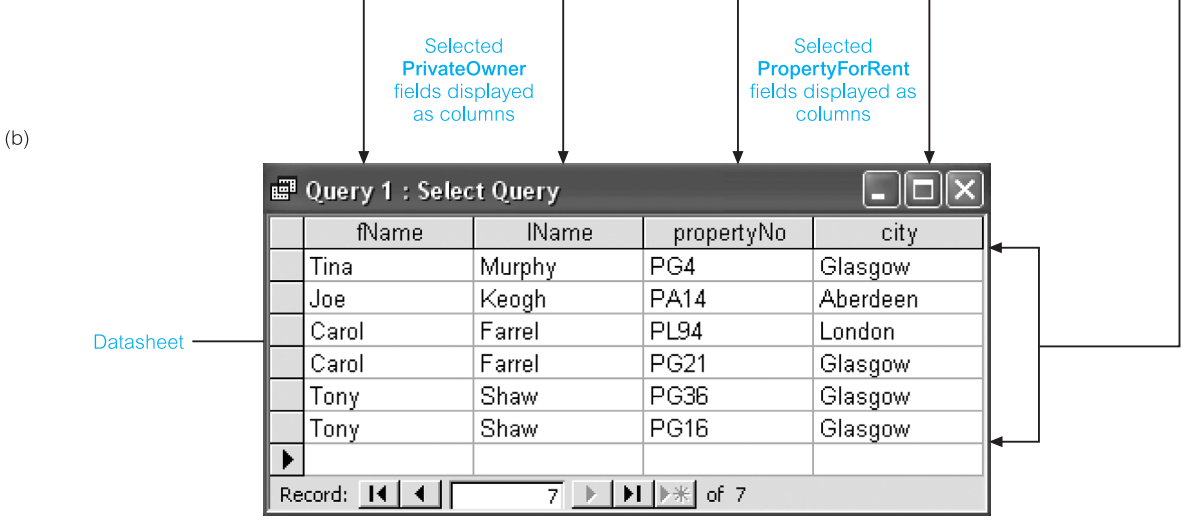
To bring together the data that we need from multiple tables, we create a **multi-table select query** with the tables and/or queries that contain the data we require in the QBE grid. For example, to view the first and last names of owners and the property number and city of their properties, we construct the QBE grid shown in Figure 7.6(a). The target tables for this query, namely PrivateOwner and PropertyForRent, are displayed above the grid. The PrivateOwner table provides the fName and lName fields and the PropertyForRent table provides the propertyNo and city fields. When this query is run the resulting datasheet is displayed, as in Figure 7.6(b). The equivalent SQL statement for the QBE grid is given in Figure 7.6(c). The multi-table query shown in Figure 7.6 is an example of an **Inner (natural) join**, which we discussed in detail in Sections 4.1.3 and 5.3.7.

When we add more than one table or query to a select query, we need to make sure that the field lists are joined to each other with a *join line* so that Microsoft Office Access knows how to join the tables. In Figure 7.6(a), note that Microsoft Office Access displays a '1' above the join line to show which table is on the 'one' side of a one-to-many relationship and an infinity symbol '∞' to show which table is on the 'many' side. In our example, 'one' owner has 'many' properties for rent.

(a)



(b)



(c)

```
SELECT PrivateOwner.fName, PrivateOwner.lName, PropertyForRent.propertyNo, PropertyForRent.city
FROM PrivateOwner INNER JOIN PropertyForRent ON PrivateOwner.ownerNo =
PropertyForRent.ownerNo;
```

Figure 7.6 (a) QBE grid of multi-table query to retrieve the first and last names of owners and the property number and city of their properties; (b) resulting datasheet; (c) equivalent SQL statement.

Microsoft Office Access automatically displays a join line between tables in the QBE grid if they contain a common field. However, the join line is only shown with symbols if a relationship has been previously established between the tables. We describe how to set up relationships between tables in Chapter 8. In the example shown in Figure 7.6, the ownerNo field is the common field in the PrivateOwner and PropertyForRent tables. For the join to work, the two fields must contain matching data in related records.

Microsoft Office Access will not automatically join tables if the related data is in fields with different names. However, we can identify the common fields in the two tables by joining the tables in the QBE grid when we create the query.

7.2.3 Calculating Totals

It is often useful to ask questions about groups of data such as:

- What is the total number of properties for rent in each city?
- What is the average salary for staff?
- How many viewings has each property for rent had since the start of this year?

We can perform calculations on groups of records using **totals queries** (also called aggregate queries). Microsoft Office Access provides various types of aggregate function including Sum, Avg, Min, Max, and Count. To access these functions, we change the query type to Totals, which results in the display of an additional row called *Total* in the QBE grid. When a totals query is run, the resulting datasheet is a *snapshot*, a set of records that is not updatable.

As with other queries, we may also want to specify criteria in a query that includes totals. For example, suppose that we want to view the total number of properties for rent in each city. This requires that the query first groups the properties according to the city field using *Group By* and then performs the totals calculation using *Count* for each group. The construction of the QBE grid to perform this calculation is shown in Figure 7.7(a) and the resulting datasheet in Figure 7.7(b). The equivalent SQL statement is given in Figure 7.7(c).

For some calculations it is necessary to create our own expressions. For example, suppose that we want to calculate the yearly rent for each property in the PropertyForRent table retrieving only the propertyNo, city, and type fields. The yearly rent is calculated as twelve times the monthly rent for each property. We enter 'Yearly Rent: [rent]*12' into a new field of the QBE grid, as shown in Figure 7.8(a). The 'Yearly Rent:' part of the expression provides the name for the new field and '[rent]*12' calculates a yearly rent value for each property using the monthly values in the rent field. The resulting datasheet for this select query is shown in Figure 7.8(b) and the equivalent SQL statement in Figure 7.8(c).

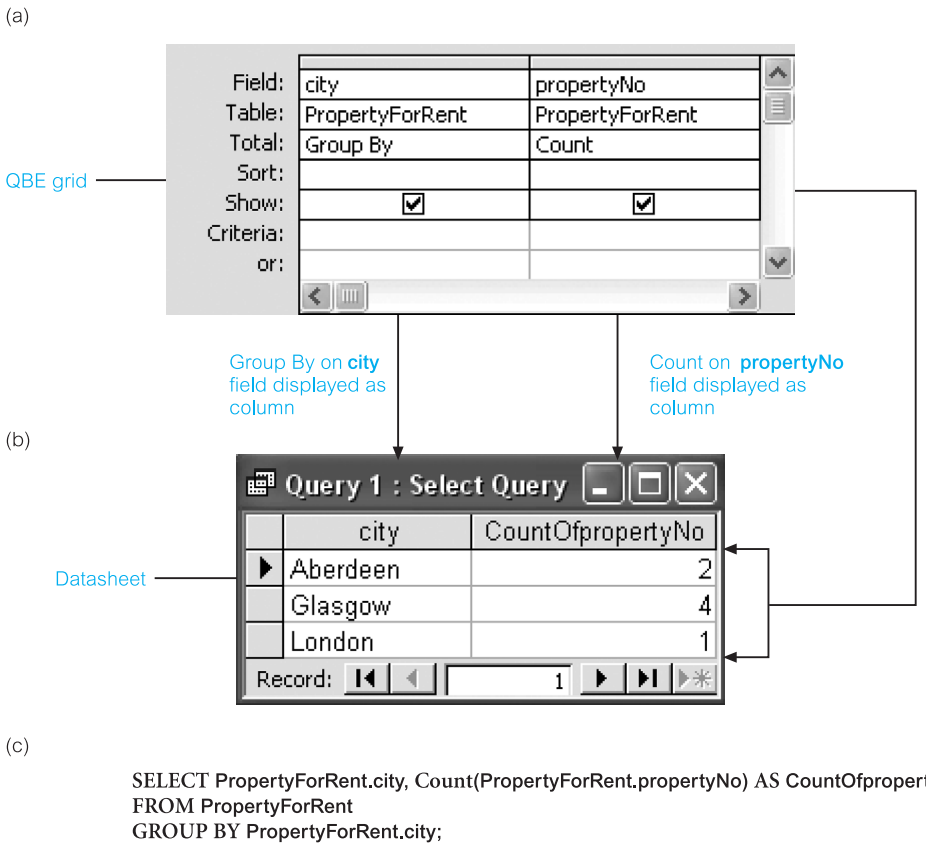


Figure 7.7
(a) QBE grid of totals query to calculate the number of properties for rent in each city; (b) resulting datasheet; (c) equivalent SQL statement.

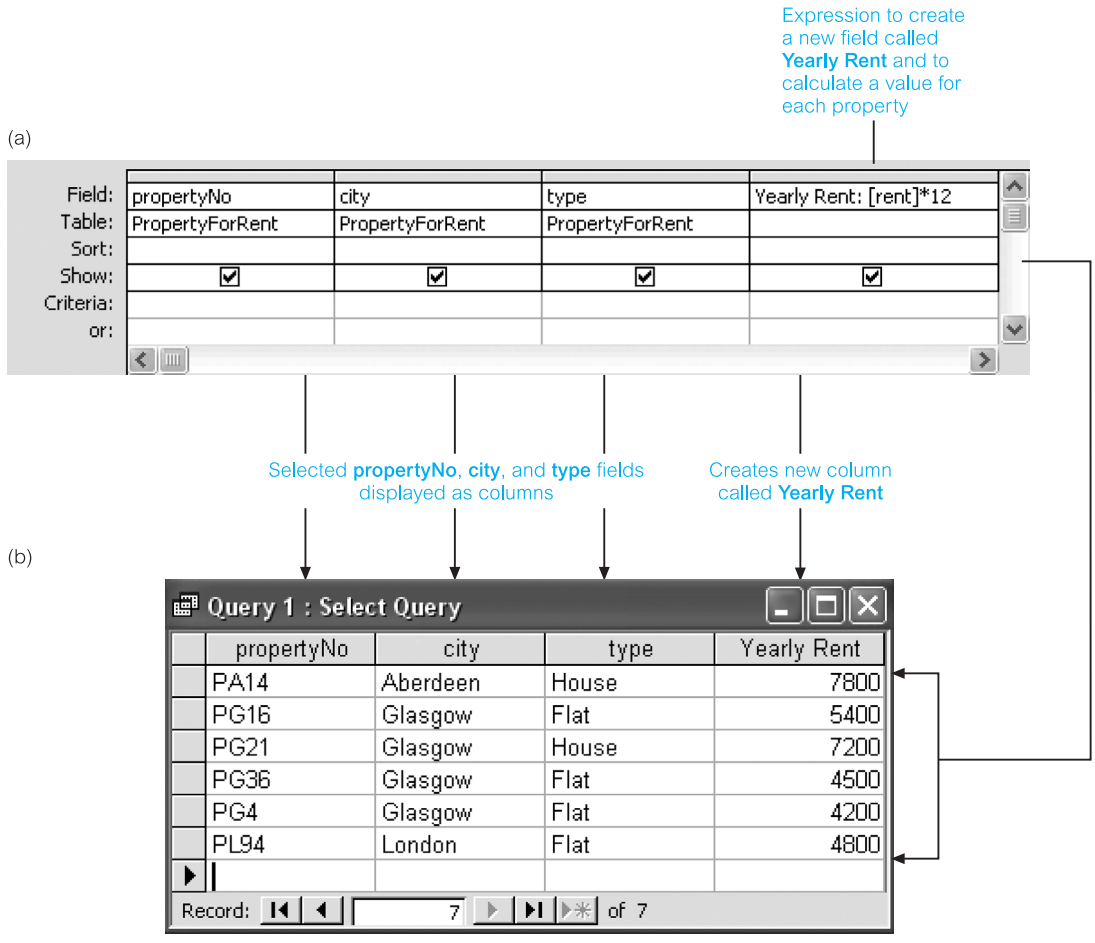
7.3 Using Advanced Queries

Microsoft Office Access provides a range of advanced queries. In this section, we describe some of the most useful examples of those queries including:

- parameter queries;
- crosstab queries;
- Find Duplicates queries;
- Find Unmatched queries.

7.3.1 Parameter Query

A **parameter query** displays one or more predefined dialog boxes that prompt the user for the parameter value(s) (criteria). Parameter queries are created by entering a prompt enclosed in square brackets in the Criteria cell for each field we want to use as a parameter. For example, suppose that we want to amend the select query shown in Figure 7.6(a) to first prompt for the owner's first and last name before retrieving the property number and city



(c)

```
SELECT PropertyForRent.propertyNo, PropertyForRent.city, PropertyForRent.type, [rent]*12 AS [Yearly Rent]
FROM PropertyForRent;
```

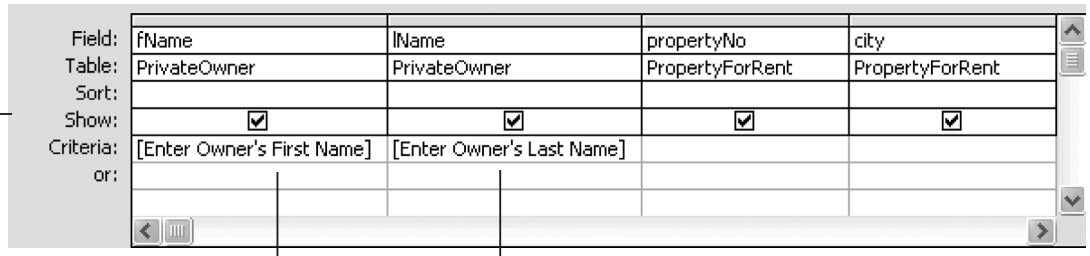
Figure 7.8 (a) QBE grid of select query to calculate the yearly rent for each property; (b) resulting datasheet; (c) equivalent SQL statement.

of his or her properties. The QBE grid for this parameter query is shown in Figure 7.9(a). To retrieve the property details for an owner called 'Carol Farrel', we enter the appropriate values into the first and second dialog boxes as shown in Figure 7.9(b), which results in the display of the resulting datasheet shown in Figure 7.9(c). The equivalent SQL statement is given in Figure 7.9(d).

Crosstab Query 7.3.2

A **crosstab query** can be used to summarize data in a compact spreadsheet format. This format enables users of large amounts of summary data to more easily identify trends and

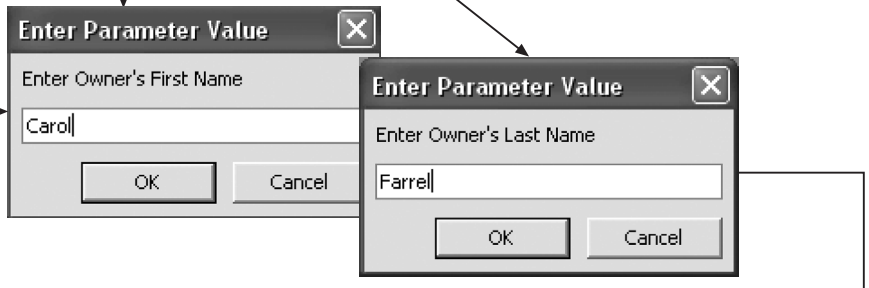
(a)



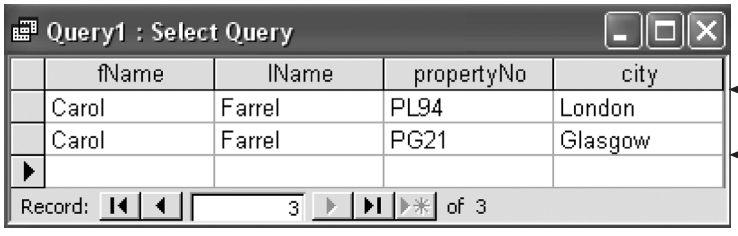
Expression to create prompt for **fName** field

Expression to create prompt for **lName** field

(b)



(c)



Records that satisfy criteria

(d)

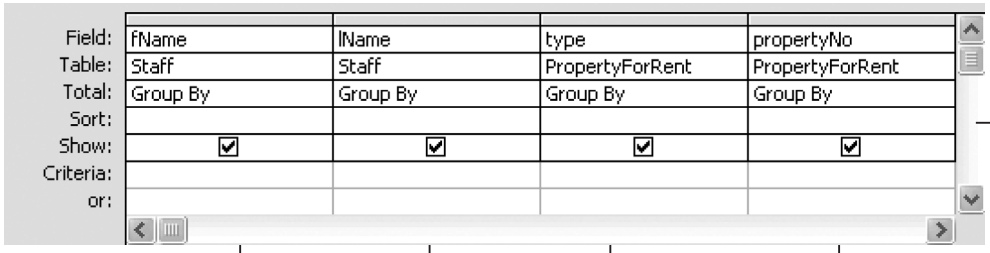
```
SELECT PrivateOwner.fName, PrivateOwner.lName, PropertyForRent.propertyNo, PropertyForRent.city
FROM PrivateOwner INNER JOIN PropertyForRent ON PrivateOwner.ownerNo = PropertyForRent.ownerNo
WHERE (((PrivateOwner.fName)=[Enter Owner's First Name]) AND ((PrivateOwner.lName)=[Enter
Owner's Last Name]));
```

Figure 7.9
(a) QBE grid of example parameter query; (b) dialog boxes for first and last name of owner; (c) resulting datasheet; (d) equivalent SQL statement.

to make comparisons. When a crosstab query is run, it returns a snapshot. We can create a crosstab query using the CrossTab Query Wizard or build the query from scratch using the QBE grid. Creating a crosstab query is similar to creating a query with totals, but we must specify the fields to be used as row headings, column headings, and the fields that are to supply the values.

For example, suppose that we want to know for each member of staff the total number of properties that he or she manages for each type of property. For the purposes of this

(a)



Group By on **fName**, **lName**, and **type** fields displayed as columns

Count on **propertyNo** field displayed as column

(b)

Query 1 : Select Query

fName	lName	type	CountOfpropertyNo
Ann	Beech	Bungalow	43
Ann	Beech	Cottage	4
Ann	Beech	Flat	45
Ann	Beech	Mid-Terrace	26
Ann	Beech	Semi-Detached	33
David	Ford	Bungalow	7
David	Ford	Cottage	2
David	Ford	Flat	14
David	Ford	Semi-Detached	42
Mary	Howe	Bungalow	45
Mary	Howe	Cottage	4
Mary	Howe	Flat	31
Mary	Howe	Mid-Terrace	2
Mary	Howe	Semi-Detached	7

Record: 1 of 14

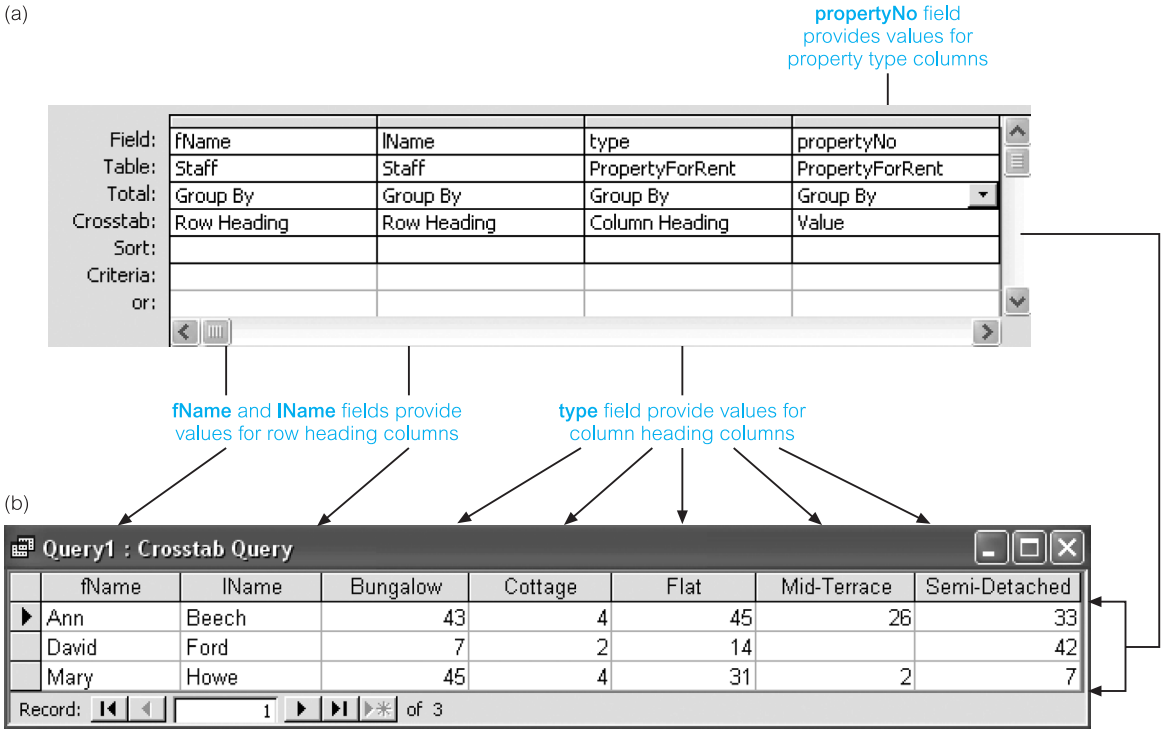
(c)

```
SELECT Staff.fName, Staff.lName, PropertyForRent.type, Count(PropertyForRent.propertyNo) AS
CountOfpropertyNo
FROM Staff INNER JOIN PropertyForRent ON Staff.staffNo = PropertyForRent.staffNo
```

Figure 7.10
(a) QBE grid of example totals query; (b) resulting datasheet; (c) equivalent SQL statement.

example, we have appended additional property records into the PropertyForRent table to more clearly demonstrate the value of crosstab queries. To answer this question, we first design a totals query, as shown in Figure 7.10(a), which creates the datasheet shown in Figure 7.10(b). The equivalent SQL statement for the totals query is given in Figure 7.10(c). Note that the layout of the resulting datasheet makes it difficult to make comparisons between staff.

(a)



(b)

Query1 : Crosstab Query

fName	lName	Bungalow	Cottage	Flat	Mid-Terrace	Semi-Detached
Ann	Beech	43	4	45	26	33
David	Ford	7	2	14		42
Mary	Howe	45	4	31	2	7

Record: 1 of 3

(c)

```
TRANSFORM Count(PropertyForRent.propertyNo) AS CountOfpropertyNo
SELECT Staff.fName, Staff.lName
FROM Staff INNER JOIN PropertyForRent ON Staff.staffNo = PropertyForRent.staffNo
GROUP BY Staff.fName, Staff.lName
PIVOT PropertyForRent.type;
```

Figure 7.11
(a) QBE grid of example crosstab query; (b) resulting datasheet; (c) equivalent SQL statement.

To convert the select query into a crosstab query, we change the type of query to Crosstab, which results in the addition of the *Crosstab* row in the QBE grid. We then identify the fields to be used for row headings, column headings, and to supply the values, as shown in Figure 7.11(a). When we run this query, the datasheet is displayed in a more compact layout, as illustrated in Figure 7.11(b). In this format, we can easily compare figures between staff. The equivalent SQL statement for the crosstab query is given in Figure 7.11(c). The TRANSFORM statement is not supported by standard SQL but is an extension of Microsoft Office Access SQL.

7.3.3 Find Duplicates Query

The **Find Duplicates Query Wizard** shown in Figure 7.2 can be used to determine if there are duplicate records in a table or determine which records in a table share the same value. For example, it is possible to search for duplicate values in the fName and lName fields to

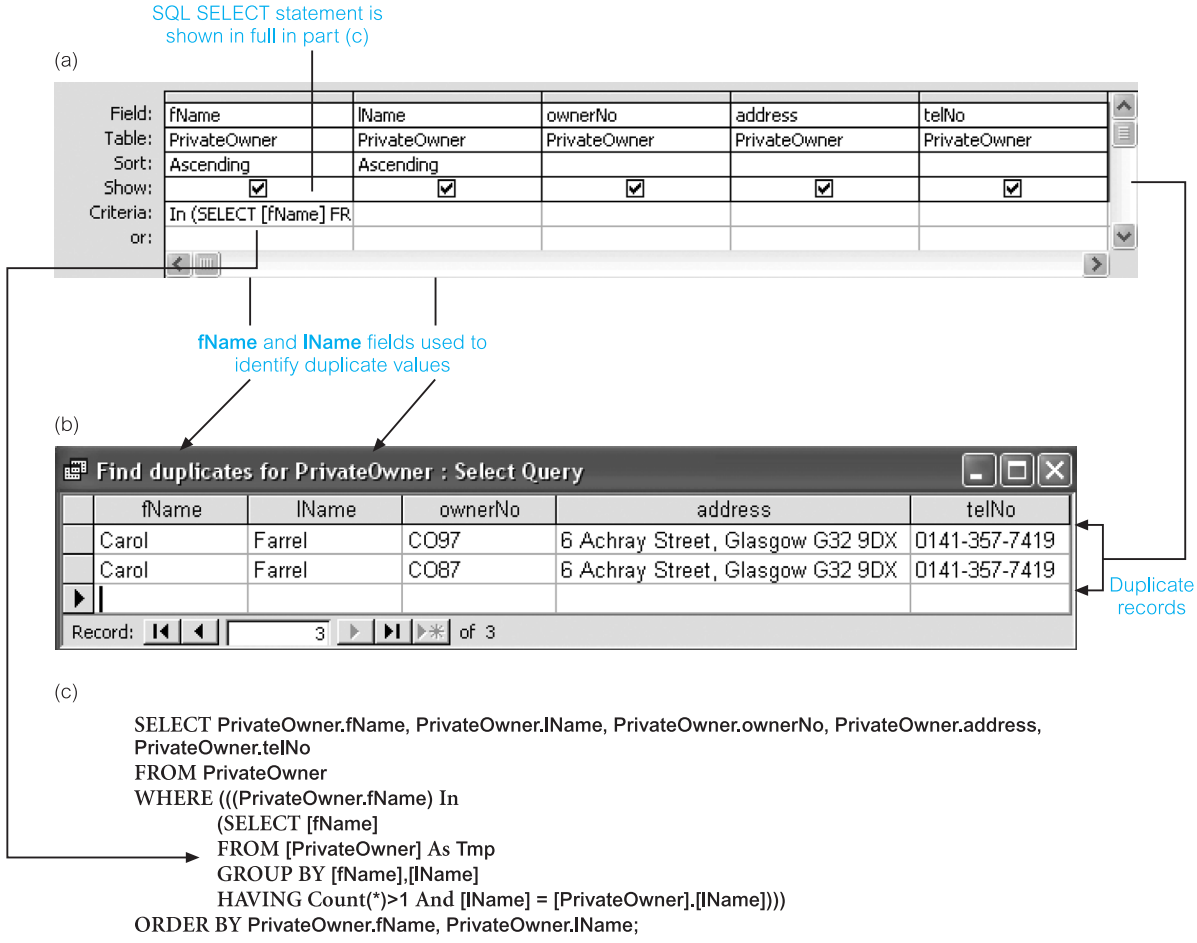


Figure 7.12
(a) QBE for example Find Duplicates query; (b) resulting datasheet; (c) equivalent SQL statement.

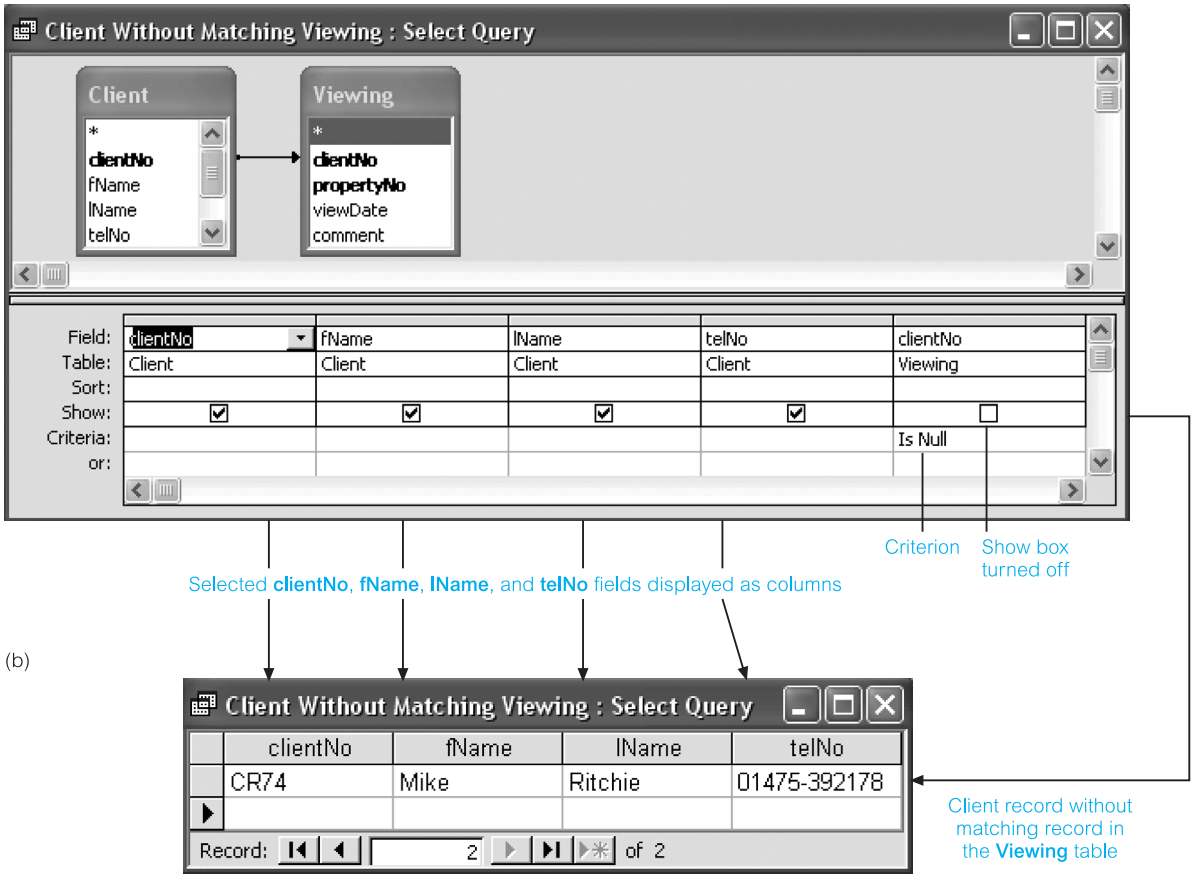
determine if we have duplicate records for the same property owners, or to search for duplicate values in a city field to see which owners are in the same city.

Suppose that we have inadvertently created a duplicate record for the property owner called 'Carol Farrel' and given this record a unique owner number. The database therefore contains two records with different owner numbers, representing the same owner. We can use the Find Duplicates Query Wizard to identify the duplicated property owner records using (for simplicity) only the values in the fName and lName fields. As discussed earlier, the Wizard simply constructs the query based on our answers. Before viewing the results of the query we can view the QBE grid for the Find Duplicates query shown in Figure 7.12(a). The resulting datasheet for the Find Duplicates query is shown in 7.12(b) displaying the two records representing the same property owner called 'Carol Farrel'. The equivalent SQL statement is given in Figure 7.12(c). Note that this SQL statement displays in full the inner SELECT SQL statement that is partially visible in the Criteria row of the fName field shown in Figure 7.12(a).

7.3.4 Find Unmatched Query

The **Find Unmatched Query Wizard** shown in Figure 7.2 can be used to find records in one table that do not have related records in another table. For example, we can find clients who have not viewed properties for rent by comparing the records in the Client and Viewing tables. The Wizard constructs the query based on our answers. Before viewing the results of the query, we can view the QBE grid for the Find Unmatched query, as shown in Figure 7.13(a). The resulting datasheet for the Find Unmatched query is shown in 7.13(b) indicating that there are no records in the Viewing table that relate to 'Mike Ritchie' in the Client table. Note that the *Show box* of the clientNo field in the QBE grid is not ticked

(a)



(c)

```
SELECT Client.clientNo, Client.fName, Client.lName, Client.telNo
FROM Client LEFT JOIN Viewing ON Client.clientNo = Viewing.clientNo
WHERE (((Viewing.clientNo) Is Null));
```

Figure 7.13 (a) QBE grid of example Find Unmatched query; (b) resulting datasheet; (c) equivalent SQL statement.

as this field is not required in the datasheet. The equivalent SQL statement for the QBE grid is given in Figure 7.13(c). The Find Unmatched query is an example of a **Left Outer join**, which we discussed in detail in Sections 4.1.3 and 5.3.7.

Autolookup Query 7.3.5

An **autolookup query** can be used to automatically fill in certain field values for a new record. When we enter a value in the join field in the query or in a form based on the query, Microsoft Office Access looks up and fills in existing data related to that value. For example, if we know the value in the join field (staffNo) between the PropertyForRent table and the Staff table, we can enter the staff number and have Microsoft Office Access enter the rest of the data for that member of staff. If no matching data is found, Microsoft Office Access displays an error message.

To create an autolookup query, we add two tables that have a one-to-many relationship and select fields for the query into the QBE grid. The join field must be selected from the 'many' side of the relationship. For example, in a query that includes fields from the PropertyForRent and Staff tables, we drag the staffNo field (foreign key) from the PropertyForRent table to the design grid. The QBE grid for this autolookup query is shown in Figure 7.14(a). Figure 7.14(b) displays a datasheet based on this query that allows us to enter the property number, street, and city for a new property record. When we enter the staff number of the member of staff responsible for the management of the property, for example 'SA9', Microsoft Office Access looks up the Staff table and automatically fills in the first and last name of the member of staff, which in this case is 'Mary Howe'. Figure 7.14(c) displays the equivalent SQL statement for the QBE grid of the autolookup query.

Changing the Content of Tables Using Action Queries 7.4

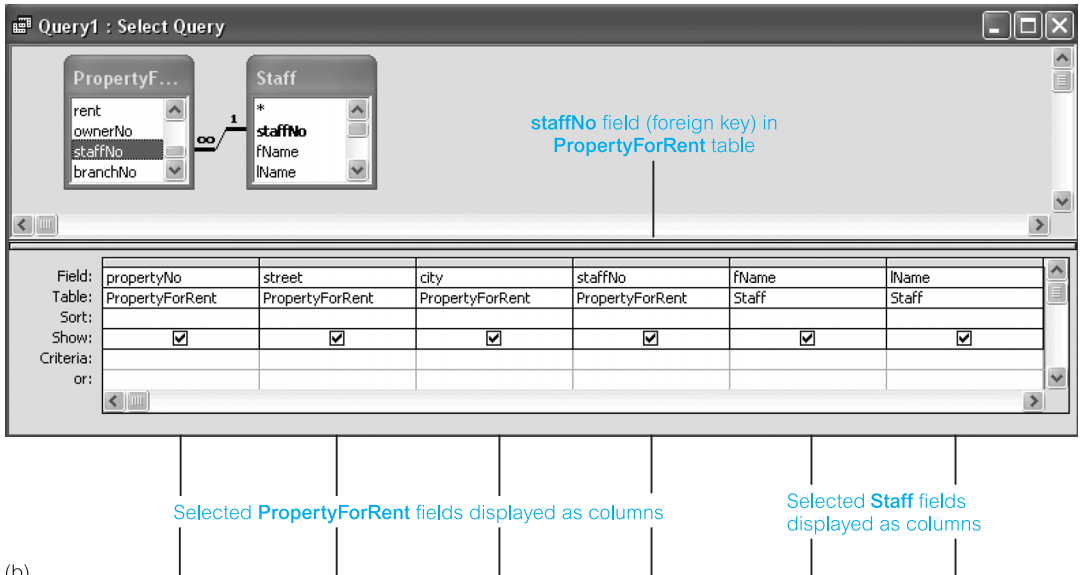
When we create a query, Microsoft Office Access creates a select query unless we choose a different type from the Query menu. When we run a select query, Microsoft Office Access displays the resulting datasheet. As the datasheet is updatable, we can make changes to the data; however, we must make the changes record by record.

If we require a large number of similar changes, we can save time by using an **action query**. An action query allows us to make changes to many records at the same time. There are four types of action query: make-table, delete, update, and append.

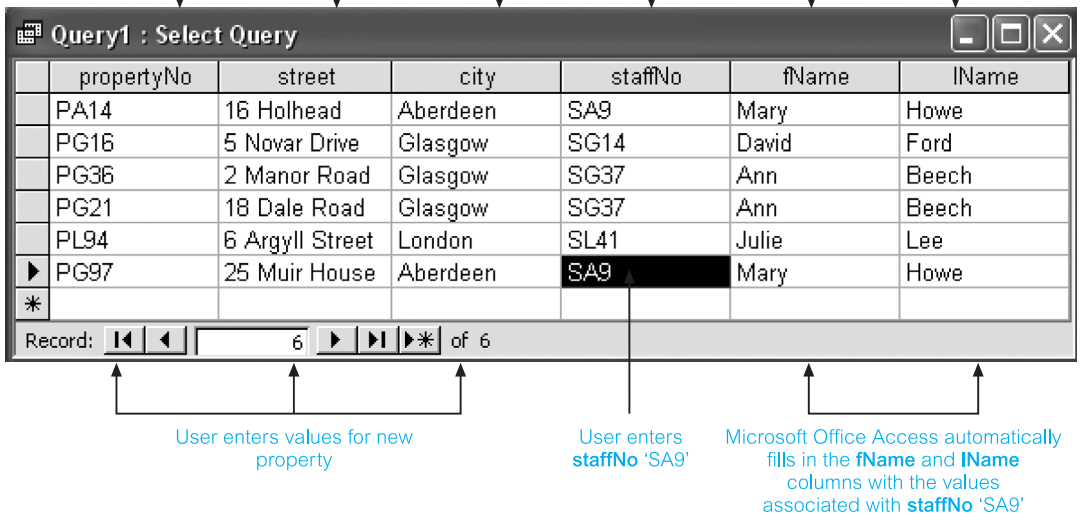
Make-Table Action Query 7.4.1

The **make-table action query** creates a new table from all or part of the data in one or more tables. The newly created table can be saved to the currently opened database or exported to another database. Note that the data in the new table does not inherit the field properties including the primary key from the original table, which needs to be set

(a)



(b)



(c)

```
SELECT PropertyForRent.propertyNo, PropertyForRent.street, PropertyForRent.city,
PropertyForRent.staffNo, Staff.fName, Staff.lName
FROM Staff INNER JOIN PropertyForRent ON Staff.staffNo = PropertyForRent.staffNo;
```

Figure 7.14 (a) QBE grid of example autolookup query; (b) datasheet based on autolookup query; (c) equivalent SQL statement.

manually. Make-table queries are useful for several reasons including the ability to archive historic data, create snapshot reports, and to improve the performance of forms and reports based on multi-table queries.

Suppose we want to create a new table called StaffCut, containing only the staffNo, fName, lName, position, and salary fields of the original Staff table. We first design a query to target the required fields of the Staff table. We then change the query type in Design View to Make-Table and a dialog box is displayed. The dialog box prompts for the name and location of the new table, as shown in Figure 7.15(a). Figure 7.15(b) displays the QBE grid for this make-table action query. When we run the query, a warning message asks whether we want to continue with the make-table operation, as shown in Figure 7.15(c). If we continue, the new table StaffCut is created, as shown in Figure 7.15(d). Figure 7.15(e) displays the equivalent SQL statement for this make-table action query.

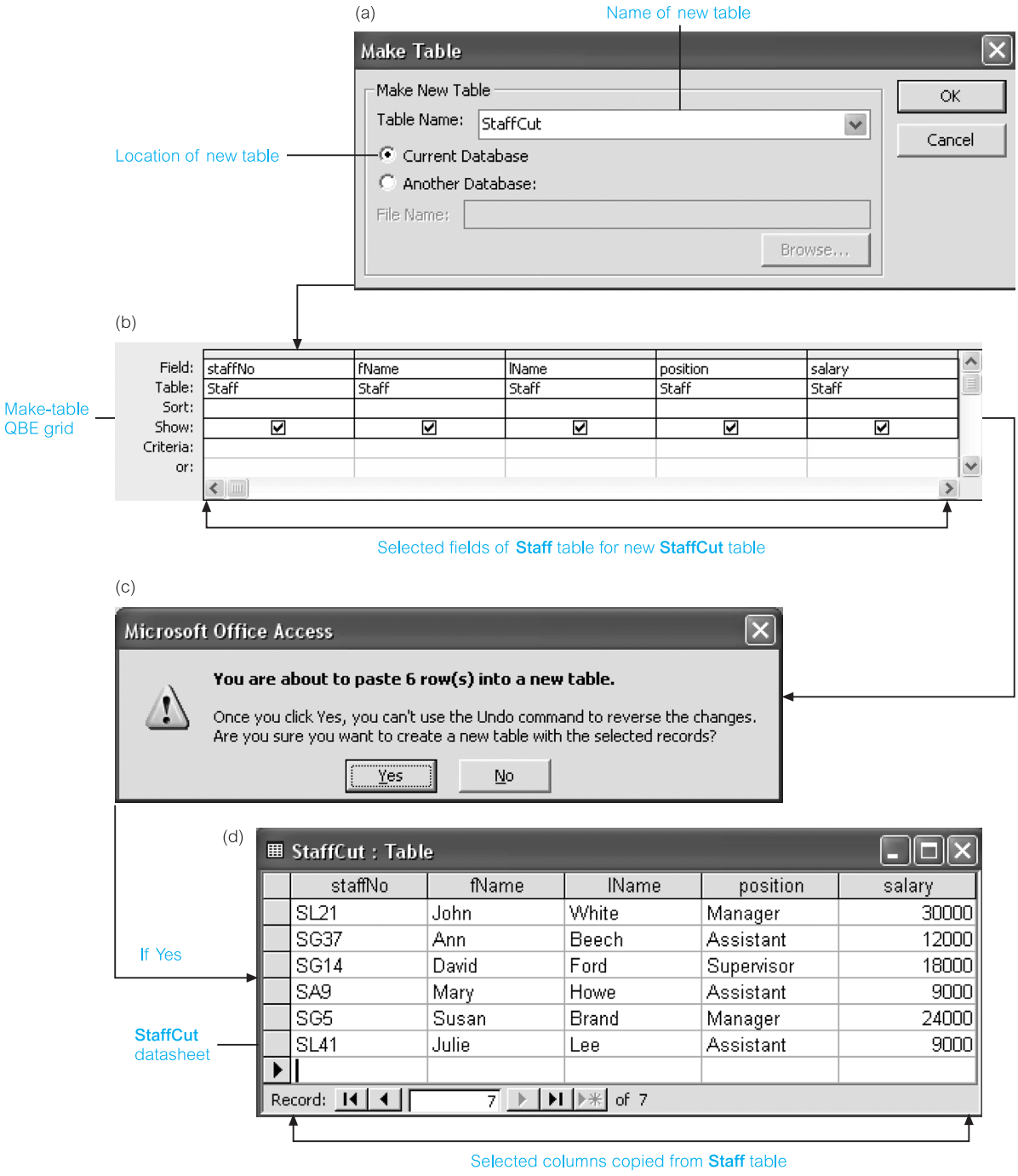
7.4.2 Delete Action Query

The **delete action query** deletes a group of records from one or more tables. We can use a single delete query to delete records from a single table, from multiple tables in a one-to-one relationship, or from multiple tables in a one-to-many relationship with referential integrity set to allow cascading deletes.

For example, suppose that we want to delete all properties for rent in Glasgow and the associated viewings records. To perform this deletion, we first create a query that targets the appropriate records in the PropertyForRent table. We then change the query type in Design View to Delete. The QBE grid for this delete action query is shown in Figure 7.16(a). As the PropertyForRent and Viewing tables have a one-to-many relationship with referential integrity set to the Cascade Delete Related Records option, all the associated viewings records for the properties in Glasgow will also be deleted. When we run the delete action query, a warning message asks whether or not we want to continue with the deletion, as shown in Figure 7.16(b). If we continue, the selected records are deleted from the PropertyForRent table and the related records from the Viewing table, as shown in Figure 7.16(c). Figure 7.16(d) displays the equivalent SQL statement for this delete action query.

7.4.3 Update Action Query

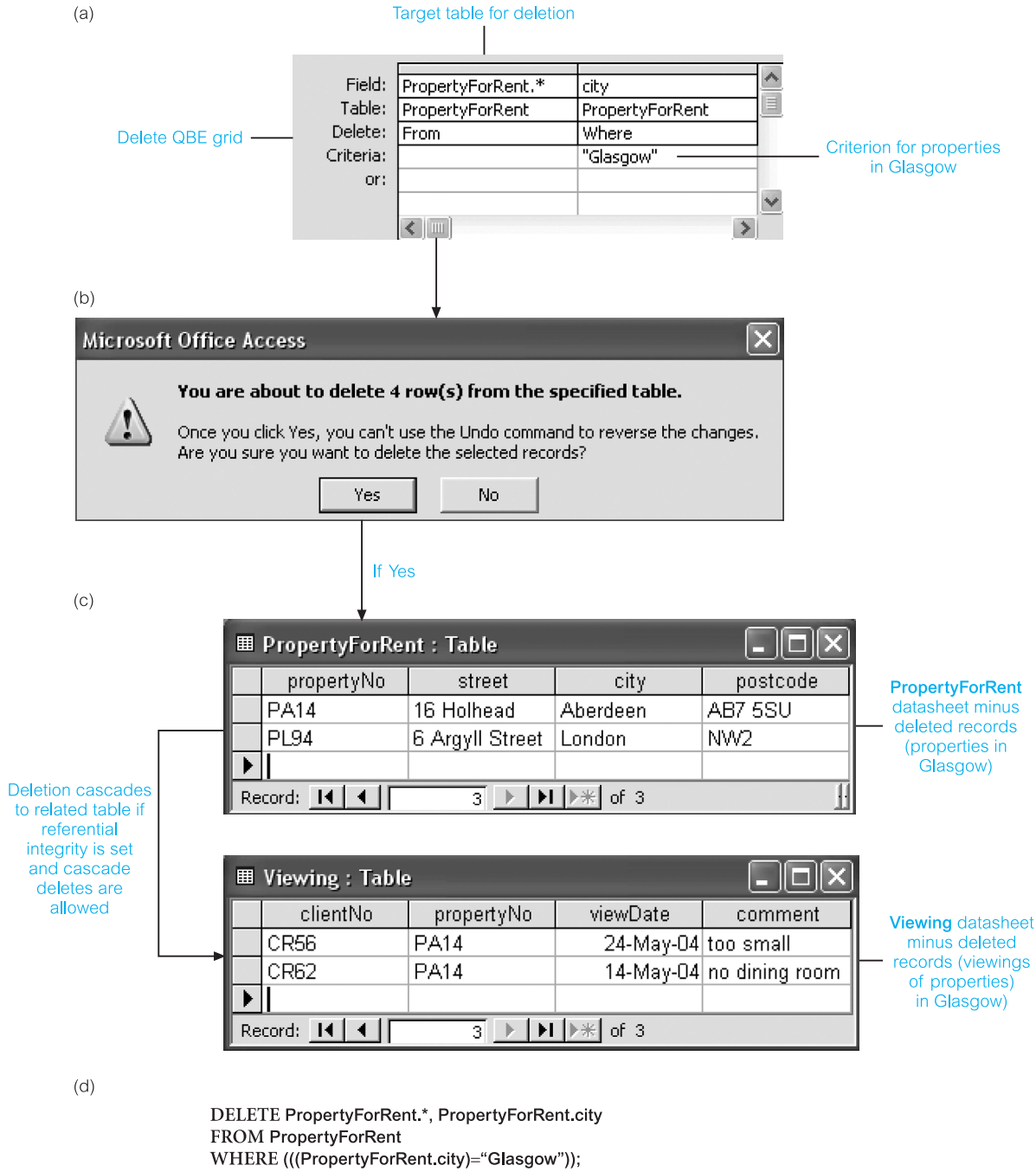
An **update action query** makes global changes to a group of records in one or more tables. For example, suppose we want to increase the rent of all properties by 10%. To perform this update, we first create a query that targets the PropertyForRent table. We then change the query type in Design View to Update. We enter the expression '[Rent]*1.1' in the *Update To* cell for the rent field, as shown in Figure 7.17(a). When we run the query, a warning message asks whether or not we want to continue with the update, as shown in Figure 7.17(b). If we continue, the rent field of PropertyForRent table is updated, as shown in Figure 7.17(c). Figure 7.17(d) displays the equivalent SQL statement for this update action query.



(e)

```
SELECT Staff.staffNo, Staff.fName, Staff.lName, Staff.position, Staff.salary INTO StaffCut
FROM Staff;
```

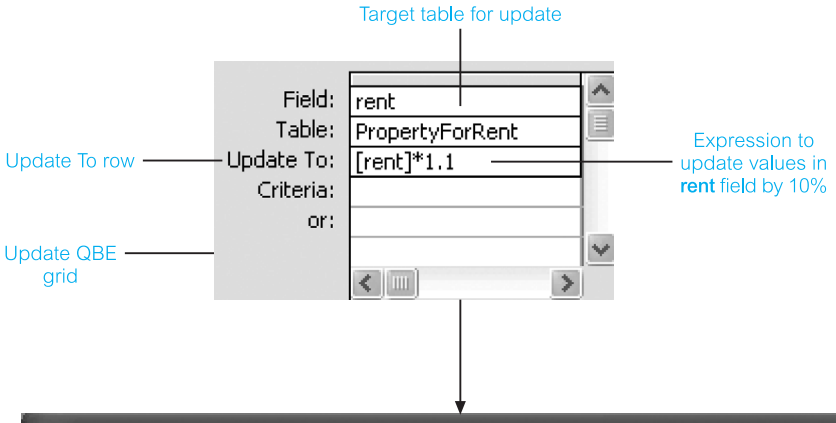
Figure 7.15 (a) Make-Table dialog box; (b) QBE grid of example make-table query; (c) warning message; (d) resulting datasheet; (e) equivalent SQL statement.



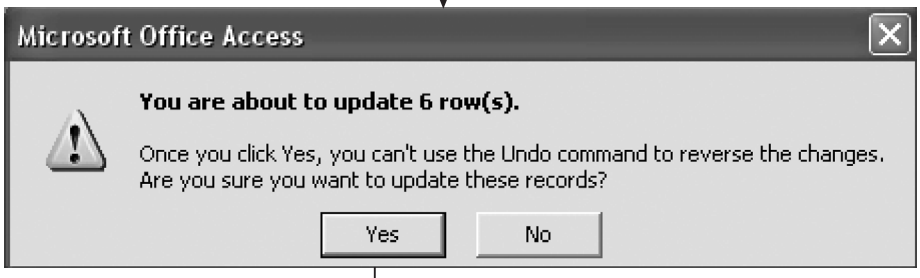

```
DELETE PropertyForRent.*, PropertyForRent.city
FROM PropertyForRent
WHERE (((PropertyForRent.city)="Glasgow"));
```

Figure 7.16 (a) QBE grid of example delete action query; (b) warning message; (c) resulting PropertyForRent and Viewing datasheets with records deleted; (d) equivalent SQL statement.

(a)



(b)



If Yes

(c)

Query1 : Select Query

propertyNo	street	city	rent
PA14	16 Holhead	Aberdeen	715
PG16	5 Novar Drive	Glasgow	495
PG21	18 Dale Road	Glasgow	660
PG36	2 Manor Road	Glasgow	412
PG4	6 Lawrence Street	Glasgow	385
PL94	6 Argyll Street	London	440

Record: 7 of 7

Values updated in **rent** column by 10%

(d)

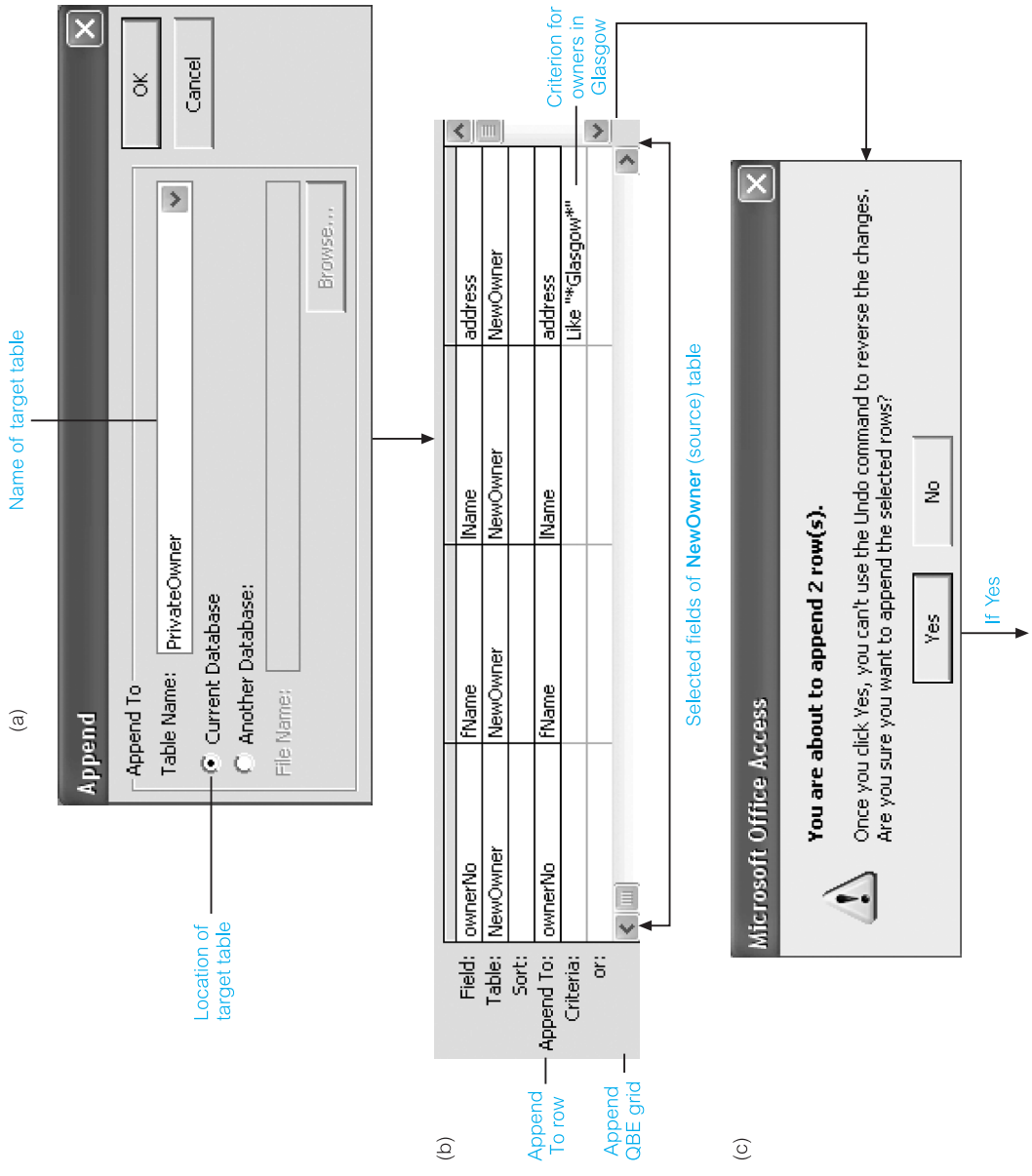
```
UPDATE PropertyForRent SET PropertyForRent.rent = [rent]*1.1;
```

Figure 7.17 (a) QBE grid of example update action query; (b) warning message; (c) resulting datasheet; (d) equivalent SQL statement.

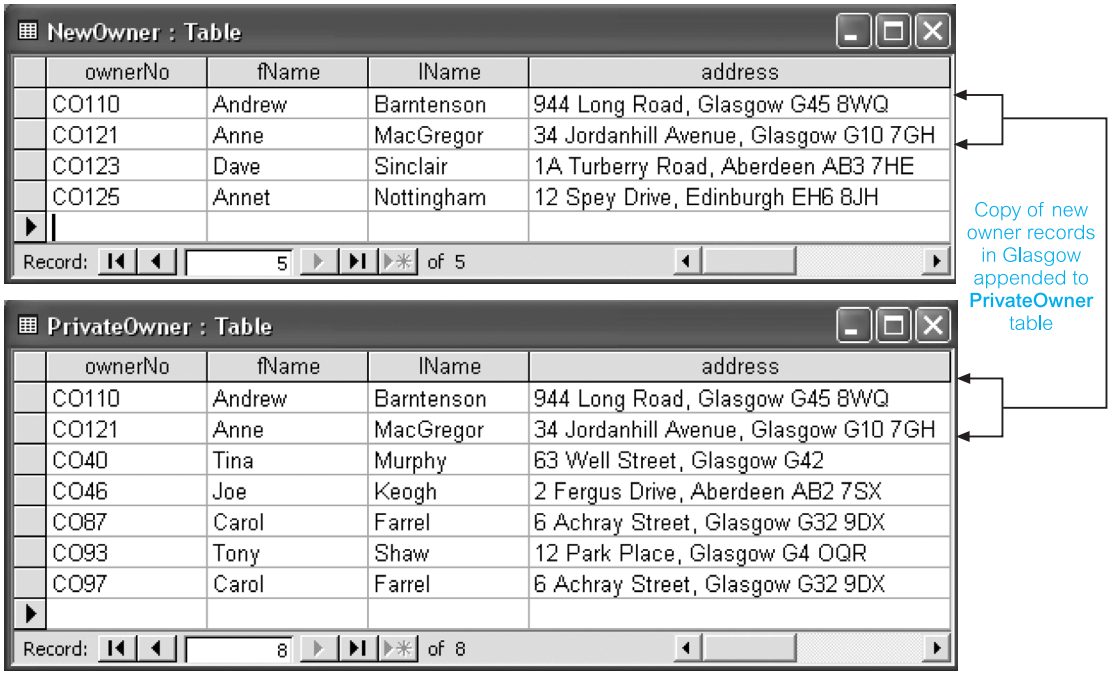
7.4.4 Append Action Query

We use an **append action query** to insert records from one or more source tables into a single target table. We can append records to a table in the same database or in another database. Append queries are also useful when we want to append fields based on criteria or even when some of the fields do not exist in the other table. For example, suppose that we want to insert the details of new owners of property for rent into the PrivateOwner table. Assume that the details of these new owners are contained in a table called NewOwner with only the ownerNo, fName, lName, and the address fields. Furthermore, we want to append only new owners located in Glasgow into the PrivateOwner table. In this example, the PrivateOwner table is the target table and the NewOwner table is the source table.

To create an append action query, we first design a query that targets the appropriate records of the NewOwner table. We change the type of query to Append and a dialog box is displayed, which prompts for the name and location of the target table, as shown in Figure 7.18(a). The QBE grid for this append action query is shown in Figure 7.18(b). When we run the query, a warning message asks whether we want to continue with the append operation, as shown in Figure 7.18(c). If we continue, the two records of owners located in Glasgow in the NewOwner table are appended to the PrivateOwner table, as given in Figure 7.18(d). The equivalent SQL statement for the append action query is shown in Figure 7.18(e).



(d)



(e)

```
INSERT INTO PrivateOwner (ownerNo, fName, lName, address)
SELECT NewOwner.ownerNo, NewOwner.fName, NewOwner.lName, NewOwner.address
FROM NewOwner
WHERE (((NewOwner.address) Like "*Glasgow*"));
```

Figure 7.18 (a) Append dialog box; (b) QBE grid of example append action query; (c) warning message; (d) the NewOwner table and the PrivateOwner table with the newly appended records; (e) equivalent SQL statement.

Exercises

7.1 Create the sample tables of the *DreamHome* case study shown in Figure 3.3 and carry out the exercises demonstrated in this chapter, using (where possible) the QBE facility of your DBMS.

7.2 Create the following additional select QBE queries for the sample tables of the *DreamHome* case study, using (where possible) the QBE facility of your DBMS.

(a) Retrieve the branch number and address for all branch offices.
(b) Retrieve the staff number, position, and salary for all members of staff working at branch office B003.
(c) Retrieve the details of all flats in Glasgow.
(d) Retrieve the details of all female members of staff who are older than 25 years old.
(e) Retrieve the full name and telephone of all clients who have viewed flats in Glasgow.
(f) Retrieve the total number of properties, according to property type.
(g) Retrieve the total number of staff working at each branch office, ordered by branch number.

7.3 Create the following additional advanced QBE queries for the sample tables of the *DreamHome* case study, using (where possible) the QBE facility of your DBMS.

(a) Create a parameter query that prompts for a property number and then displays the details of that property.
(b) Create a parameter query that prompts for the first and last names of a member of staff and then displays the details of the property that the member of staff is responsible for.
(c) Add several more records into the PropertyForRent tables to reflect the fact that property owners 'Carol Farrel' and 'Tony Shaw' now own many properties in several cities. Create a select query to display for each owner, the number of properties that he or she owns in each city. Now, convert the select query into a crosstab query and assess whether the display is more or less useful when comparing the number of properties owned by each owner in each city.
(d) Introduce an error into your Staff table by entering an additional record for the member of staff called 'David Ford' with a new staff number. Use the Find Duplicates query to identify this error.
(e) Use the Find Unmatched query to identify those members of staff who are not assigned to manage property.
(f) Create an autolookup query that fills in the details of an owner, when a new property record is entered into the PropertyForRent table and the owner of the property already exists in the database.

7.4 Use action queries to carry out the following tasks on the sample tables of the *DreamHome* cases study, using (where possible) the QBE facility of your DBMS.

(a) Create a cut-down version of the PropertyForRent table called PropertyGlasgow, which has the propertyNo, street, postcode, and type fields of the original table and contains only the details of properties in Glasgow.
(b) Remove all records of property viewings that do not have an entry in the comment field.
(c) Update the salary of all members of staff, except Managers, by 12.5%.
(d) Create a table called NewClient that contains the details of new clients. Append this data into the original Client table.

7.5 Using the sample tables of the *DreamHome* case study, create equivalent QBE queries for the SQL examples given in Chapter 5.

Chapter 8 Commercial RDBMSs: Office Access and Oracle

Chapter Objectives

In this chapter you will learn:

- About Microsoft Office Access 2003:
  - the DBMS architecture;
  - how to create base tables and relationships;
  - how to create general constraints;
  - how to use forms and reports;
  - how to use macros.
- About Oracle9*i*:
  - the DBMS architecture;
  - how to create base tables and relationships;
  - how to create general constraints;
  - how to use PL/SQL;
  - how to create and use stored procedures and functions;
  - how to create and use triggers;
  - how to create forms and reports;
  - support for grid computing.

As we mentioned in Chapter 3, the Relational Database Management System (RDBMS) has become the dominant data-processing software in use today, with estimated new licence sales of between US$6 billion and US$10 billion per year (US$25 billion with tools sales included). There are many hundreds of RDBMSs on the market. For many users, the process of selecting the best DBMS package can be a difficult task, and in the next chapter we present a summary of the main features that should be considered when selecting a DBMS package. In this chapter, we consider two of the most widely used RDBMSs: Microsoft Office Access and Oracle. In each case, we use the terminology of the particular DBMS (which does not conform to the formal relational terminology we introduced in Chapter 3).

8.1 Microsoft Office Access 2003

Microsoft Office Access is the mostly widely used relational DBMS for the Microsoft Windows environment. It is a typical PC-based DBMS capable of storing, sorting, and retrieving data for a variety of applications. Access provides a Graphical User Interface (GUI) to create tables, queries, forms, and reports, and tools to develop customized database applications using the Microsoft Office Access macro language or the Microsoft Visual Basic for Applications (VBA) language. In addition, Office Access provides programs, called **Wizards**, to simplify many of the processes of building a database application by taking the user through a series of question-and-answer dialog boxes. It also provides **Builders** to help the user build syntactically correct expressions, such as those required in SQL statements and macros. Office Access supports much of the SQL standard presented in Chapters 5 and 6, and the Microsoft Open Database Connectivity (ODBC) standard, which provides a common interface for accessing heterogeneous SQL databases, such as Oracle and Informix. We discuss ODBC in more detail in Appendix E. To start the presentation of Microsoft Office Access, we first introduce the objects that can be created to help develop a database application.

8.1.1 Objects

The user interacts with Microsoft Office Access and develops a database application using a number of objects:

- *Tables* The base tables that make up the database. Using the Microsoft terminology, a table is organized into columns (called *fields*) and rows (called *records*).
- *Queries* Allow the user to view, change, and analyze data in different ways. Queries can also be stored and used as the source of records for forms, reports, and data access pages. We examined queries in some detail in the previous chapter.
- *Forms* Can be used for a variety of purposes such as to create a data entry form to enter data into a table.
- *Reports* Allow data in the database to be presented in an effective way in a customized printed format.
- *Pages* A (data access) page is a special type of Web page designed for viewing and working with data (stored in a Microsoft Office Access database or a Microsoft SQL Server database) from the Internet or an intranet. The data access page may also include data from other sources, such as Microsoft Excel.
- *Macros* A set of one or more actions each of which performs a particular operation, such as opening a form or printing a report. Macros can help automate common tasks such as printing a report when a user clicks a button.
- *Modules* A collection of VBA declarations and procedures that are stored together as a unit.

Before we discuss these objects in more detail, we first examine the architecture of Microsoft Office Access.

Microsoft Office Access Architecture 8.1.2

Microsoft Office Access can be used as a standalone system on a single PC or as a multi-user system on a PC network. Since the release of Access 2000, there is a choice of two data engines[†] in the product: the original Jet engine and the new Microsoft SQL Server Desktop Engine (MSDE, previously the Microsoft Data Engine), which is compatible with Microsoft's backoffice SQL Server. The Jet engine stores all the application data, such as tables, indexes, queries, forms, and reports, in a single Microsoft database (.mdb) file, based on the ISAM (Indexed Sequential Access Method) organization (see Appendix C). MSDE is based on the same data engine as SQL Server, enabling users to write one application that scales from a PC running Windows 95 to multiprocessor clusters running Windows Server 2003. MSDE also provides a migration path to allow users to subsequently upgrade to SQL Server. However, unlike SQL Server, MSDE has a 2 gigabyte database size limit.

Microsoft Office Access, like SQL Server, divides the data stored in its table structures into 2 kilobyte data pages, corresponding to the size of a conventional DOS fixed-disk file cluster. Each page contains one or more records. A record cannot span more than a single page, although Memo and OLE Object fields can be stored in pages separate from the rest of the record. Office Access uses variable-length records as the standard method of storage and allows records to be ordered by the use of an index, such as a primary key. Using variable length, each record occupies only the space required to store its actual data.

A header is added to each page to create a linked list of data pages. The header contains a pointer to the page that precedes it and another pointer to the page that follows. If no indexes are in use, new data is added to the last page of the table until the page is full, and then another page is added at the end. One advantage of data pages with their own header is that a table's data pages can be kept in ISAM order by altering the pointers in the page header, and not the structure of the file itself.

Multi-user support

Microsoft Office Access provides four main ways of working with a database that is shared among users on a network:

- *File-server solutions* An Office Access database is placed on a network so that multiple users can share it. In this case, each workstation runs a copy of the Office Access application.
- *Client–server solutions* In earlier versions of Office Access, the only way to achieve this was to create linked tables that used an ODBC driver to link to a database such as SQL Server. Since Access 2000, an *Access Project (.adp) File* can also be created, which can store forms, reports, macros, and VBA modules locally and can connect to a remote SQL Server database using OLE DB (Object Linking and Embedding for Databases) to display and work with tables, views, relationships, and stored procedures. As mentioned above, MSDE can also be used to achieve this type of solution.

[†] A 'data engine' or 'database engine' is the core process that a DBMS uses to store and maintain data.

- *Database replication solutions* These allow data or database design changes to be shared between copies of an Office Access database in different locations without having to redistribute copies of the entire database. Replication involves producing one or more copies, called *replicas*, of a single original database, called the *Design Master*. Together, the Design Master and its replicas are called a *replica set*. By performing a process called *synchronization*, changes to objects and data are distributed to all members of the replica set. Changes to the design of objects can only be made in the Design Master, but changes to data can be made from any member of the replica set. We discuss replication in Chapter 24.
- *Web-based database solutions* A browser displays one or more data access pages that dynamically link to a shared Office Access or SQL Server database. These pages have to be displayed by Internet Explorer 5 or later. We discuss this solution in Section 29.10.5.

When a database resides on a file server, the operating system's locking primitives are used to lock pages when a table record is being updated. In a multi-user environment, Jet uses a locking database (.ldb) file to store information on which records are locked and which user has them locked. The locking database file is created when a database is opened for shared access. We discuss locking in detail in Section 20.2.

8.1.3 Table Definition

Microsoft Office Access provides five ways to create a blank (empty) table:

- Use the Database Wizard to create in one operation all the tables, forms, and reports that are required for the entire database. The Database Wizard creates a new database, although this particular wizard cannot be used to add new tables, forms, or reports to an existing database.
- Use the Table Wizard to choose the fields for the table from a variety of predefined tables such as business contacts, household inventory, or medical records.
- Enter data directly into a blank table (called a **datasheet**). When the new datasheet is saved, Office Access will analyze the data and automatically assign the appropriate data type and format for each field.
- Use Design View to specify all table details from scratch.
- Use the CREATE TABLE statement in SQL View.

Creating a blank table in Microsoft Office Access using SQL

In Section 6.3.2 we examined the SQL CREATE TABLE statement that allows users to create a table. Microsoft Office Access 2003 does not fully comply with the SQL standard and the Office Access CREATE TABLE statement has no support for the DEFAULT and CHECK clauses. However, default values and certain enterprise constraints can still be specified outside SQL, as we see shortly. In addition, the data types are slightly different from the SQL standard, as shown in Table 8.1. In Example 6.1 in Chapter 6 we showed how to create the PropertyForRent table in SQL. Figure 8.1 shows the SQL View with the equivalent statement in Office Access.

Table 8.1 Microsoft Office Access data types.

Data type	Use	Size
Text	Text or text/numbers. Also numbers that do not require calculations, such as telephone numbers. Corresponds to the SQL character data type (see Section 6.1.2).	Up to 255 characters
Memo	Lengthy text and numbers, such as notes or descriptions.	Up to 65,536 characters
Number	Numeric data to be used for mathematical calculations, except calculations involving money (use Currency type). Corresponds to the SQL exact numeric and approximate numeric data type (see Section 6.1.2).	1, 2, 4, or 8 bytes (16 bytes for Replication ID)
Date/Time	Dates and times. Corresponds to the SQL datetime data type (see Section 6.1.2).	8 bytes
Currency	Currency values. Use the Currency data type to prevent rounding off during calculations.	8 bytes
Autonumber	Unique sequential (incrementing by 1) or random numbers automatically inserted when a record is added.	4 bytes (16 bytes for Replication ID)
Yes/No	Fields that will contain only one of two values, such as Yes/No, True/False, On/Off. Corresponds to the SQL bit data type (see Section 6.1.2).	1 bit
OLE Object	Objects (such as Microsoft Word documents, Microsoft Excel spreadsheets, pictures, sounds, or other binary data), created in other programs using the OLE protocol, which can be linked to, or embedded in, a Microsoft Office Access table.	Up to 1 gigabyte
Hyperlink	Field that will store hyperlinks.	Up to 64,000 characters
Lookup Wizard	Creates a field that allows the user to choose a value from another table or from a list of values using a combo box. Choosing this option in the data type list starts a wizard to define this.	Same size as the primary key that forms the lookup field (typically 4 bytes)

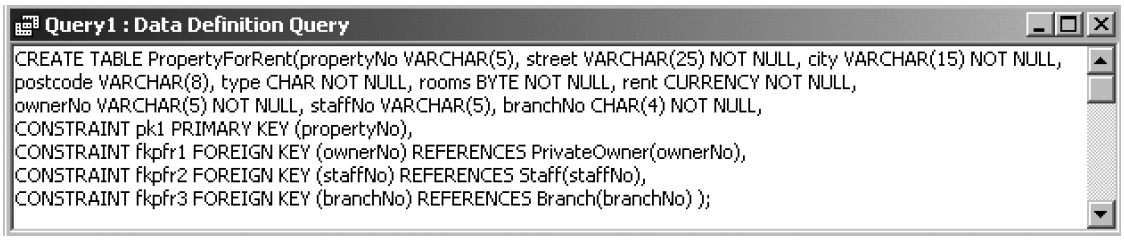


Figure 8.1 SQL View showing creation of the PropertyForRent table.

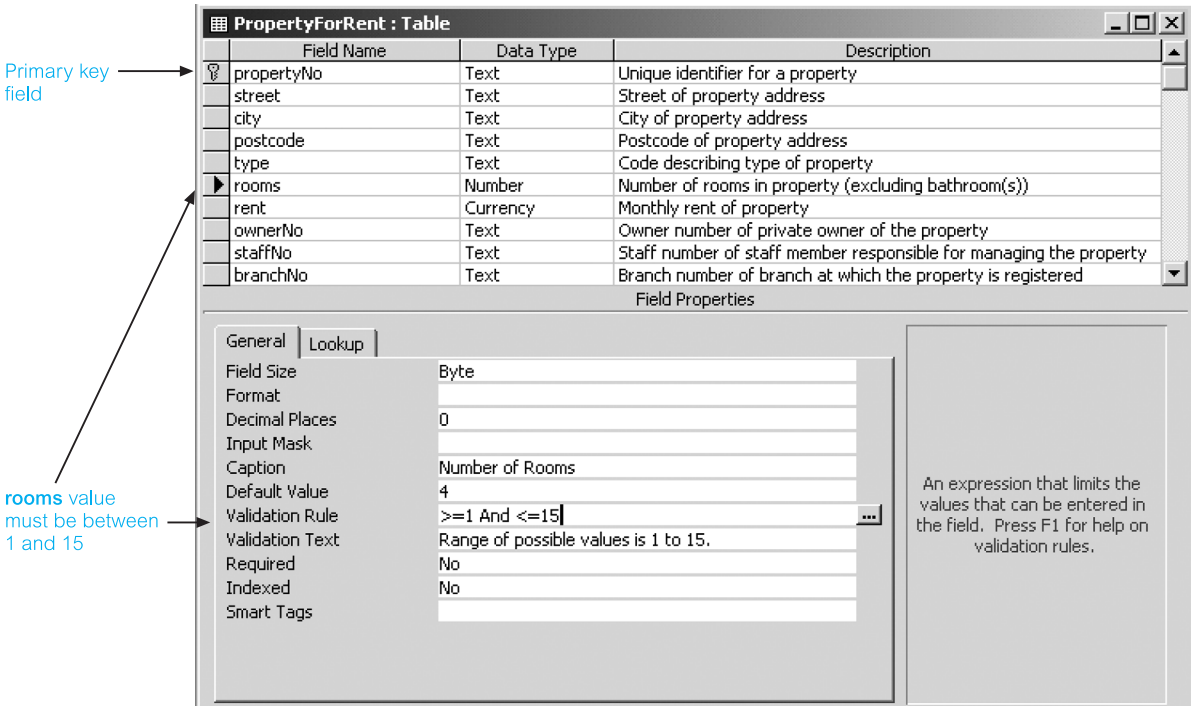


Figure 8.2 Design View showing creation of the PropertyForRent table.

Creating a blank table in Microsoft Office Access using Design View

Figure 8.2 shows the creation of the PropertyForRent table in Design View. Regardless of which method is used to create a table, table Design View can be used at any time to customize the table further, such as adding new fields, setting default values, or creating input masks.

Microsoft Office Access provides facilities for adding constraints to a table through the Field Properties section of the table Design View. Each field has a set of properties that are used to customize how data in a field is stored, managed, or displayed. For example, we can control the maximum number of characters that can be entered into a Text field by setting its *Field Size* property. The data type of a field determines the properties that are available for that field. Setting field properties in Design View ensures that the fields have consistent settings when used at a later stage to build forms and reports. We now briefly discuss each of the field properties.

Field Size property

The Field Size property is used to set the maximum size for data that can be stored in a field of type Text, Number, and AutoNumber. For example, the Field Size property of the propertyNo field (Text) is set to 5 characters, and the Field Size property for the rooms field (Number) is set to Byte to store whole numbers from 0 to 255, as shown in Figure 8.2. In addition to Byte, the valid values for the Number data type are:

- Integer – 16-bit integer (values between –32,768 and 32,767);
- Long integer – 32 bit integer;

- Single – floating point 32-bit representation;
- Double – floating point 64-bit representation;
- Replication ID – 128-bit identifier, unique for each record, even in a distributed system;
- Decimal – floating point number with a precision and scale.

Format property

The Format property is used to customize the way that numbers, dates, times, and text are displayed and printed. Microsoft Office Access provides a range of formats for the display of different data types. For example, a field with a Date/Time data type can display dates in various formats including Short Date, Medium Date, and Long Date. The date 1st November 1933 can be displayed as 01/11/33 (Short Date), 01-Nov-33 (Medium Date), or 1 November 1933 (Long Date).

Decimal Places property

The Decimal Places property is used to specify the number of decimal places to be used when displaying numbers (this does not actually affect the number of decimal places used to store the number).

Input Mask property

Input masks assist the process of data entry by controlling the format of the data as it is entered into the table. A mask determines the type of character allowed for each position of a field. Input masks can simplify data entry by automatically entering special formatted characters when required and generating error messages when incorrect entries are attempted. Microsoft Office Access provides a range of input mask characters to control data entry. For example, the values to be entered into the propertyNo field have a specific format: the first character is 'P' for property, the second character is an upper-case letter and the third, fourth, and fifth characters are numeric. The fourth and fifth characters are optional and are used only when required (for example, property numbers include PA9, PG21, PL306). The input mask used in this case is '\P>L099':

- '\' causes the character that follows to be displayed as the literal character (for example, \P is displayed as just P);
- '>L' causes the letter that follows P to be converted to upper case;
- '0' specifies that a digit must follow and '9' specifies optional entry for a digit or space.

Caption property

The Caption property is used to provide a fuller description of a field name or useful information to the user through captions on objects in various views. For example, if we enter 'Property Number' into the Caption property of the propertyNo field, the column heading 'Property Number' will be displayed for the table in Datasheet View and not the field name, 'propertyNo'.

Default Value property

To speed up and reduce possible errors in data entry, we can assign default values to specify a value that is automatically entered in a field when a new record is created. For

example, the average number of rooms in a single property is four, therefore we set '4' as the default value for the rooms field, as shown in Figure 8.2.

Validation Rule/Validation Text properties

The Validation Rule property is used to specify constraints for data entered into a field. When data is entered that violates the Validation Rule setting, the Validation Text property is used to specify the warning message that is displayed. Validation rules can also be used to set a range of allowable values for numeric or date fields. This reduces the amount of errors that may occur when records are being entered into the table. For example, the number of rooms in a property ranges from a minimum of 1 to a maximum of 15. The validation rule and text for the rooms field are shown in Figure 8.2.

Required property

Required fields must hold a value in every record. If this property is set to 'Yes', we must enter a value in the required field and the value cannot be null. Therefore, setting the Required property is equivalent to the NOT NULL constraint in SQL (see Section 6.2.1). Primary key fields should always be implemented as required fields.

Allow Zero Length property

The Allow Zero Length property is used to specify whether a zero-length string ("") is a valid entry in a field (for Text, Memo, and Hyperlink fields). If we want Microsoft Office Access to store a zero-length string instead of null when we leave a field blank, we set both the Allow Zero Length and Required properties to 'Yes'. The Allow Zero Length property works independently of the Required property. The Required property determines only whether null is valid for the field. If the Allow Zero Length property is set to 'Yes', a zero-length string will be a valid value for the field regardless of the setting of the Required property.

Indexed property

The Indexed property is used to set a single-field index. An index is a structure used to help retrieve data more quickly and efficiently (just as the index in this book allows a particular section to be found more quickly). An index speeds up queries on the indexed fields as well as sorting and grouping operations. The Indexed property has the following values:

No	no index (the default)
Yes (Duplicates OK)	the index allows duplicates
Yes (No Duplicates)	the index does not allow duplicates

For the *DreamHome* database, we discuss which fields to index in Step 5.3 in Chapter 17.

Unicode Compression property

Unicode is a character encoding standard that represents each character as two bytes, enabling almost all of the written languages in the world to be represented using a single character set. For a Latin character (a character of a western European language such as English, Spanish, or German) the first byte is 0. Thus, for Text, Memo, and Hypertext fields more storage space is required than in earlier versions of Office Access, which did not use Unicode. To overcome this, the default value of the Unicode Compression property for

these fields is 'Yes' (for compression), so that any character whose first byte is 0 is compressed when it is stored and uncompressed when it is retrieved. The Unicode Compression property can also be set to 'No' (for no compression). Note that data in a Memo field is not compressed unless it requires 4096 bytes or less of storage space after compression.

IME Mode/IME Sentence Mode properties

An Input Method Editor (IME) is a program that allows entry of East Asian text (traditional Chinese, simplified Chinese, Japanese, or Korean), converting keystrokes into complex East Asian characters. In essence, the IME is treated as an alternative type of keyboard layout. The IME interprets keystrokes as characters and then gives the user an opportunity to insert the correct interpretation. The IME Mode property applies to all East Asian languages, and IME Sentence Mode property applies to Japanese only.

Smart tags property

Smart tags allow actions to be performed within Office Access that would normally require the user to open another program. Smart tags can be associated with the fields of a table or query, or with the controls of a form, report, or data access page. The Smart Tags Action button appears when the field or control is activated and the button can be clicked to see what actions are available. For example, for a person's name the smart tag could allow an e-mail to be generated; for a date, the smart tag could allow a meeting to be scheduled. Microsoft provides some standard tags but custom smart tags can be built using any programming language that can create a Component Object Model (COM) add-in.

Relationships and Referential Integrity Definition 8.1.4

As we saw in Figure 8.1, relationships can be created in Microsoft Office Access using the SQL CREATE TABLE statement. Relationships can also be created in the Relationships window. To create a relationship, we display the tables that we want to create the relationship between, and then drag the primary key field of the parent table to the foreign key field of the child table. At this point, Office Access will display a window allowing specification of the referential integrity constraints.

Figure 8.3(a) shows the referential integrity dialog box that is displayed while creating the one-to-many (1:*) relationship Staff *Manages* PropertyForRent, and Figure 8.3(b) shows the Relationships window after the relationship has been created. Two things to note about setting referential integrity constraints in Microsoft Office Access are:

(1) A one-to-many (1:*) relationship is created if only one of the related fields is a primary key or has a unique index; a 1:1 relationship is created if both the related fields are primary keys or have unique indexes.

(2) There are only two referential integrity actions for update and delete that correspond to NO ACTION and CASCADE (see Section 6.2.4). Therefore, if other actions are required, consideration must be given to modifying these constraints to fit in with the constraints available in Office Access, or to implementing these constraints in application code.

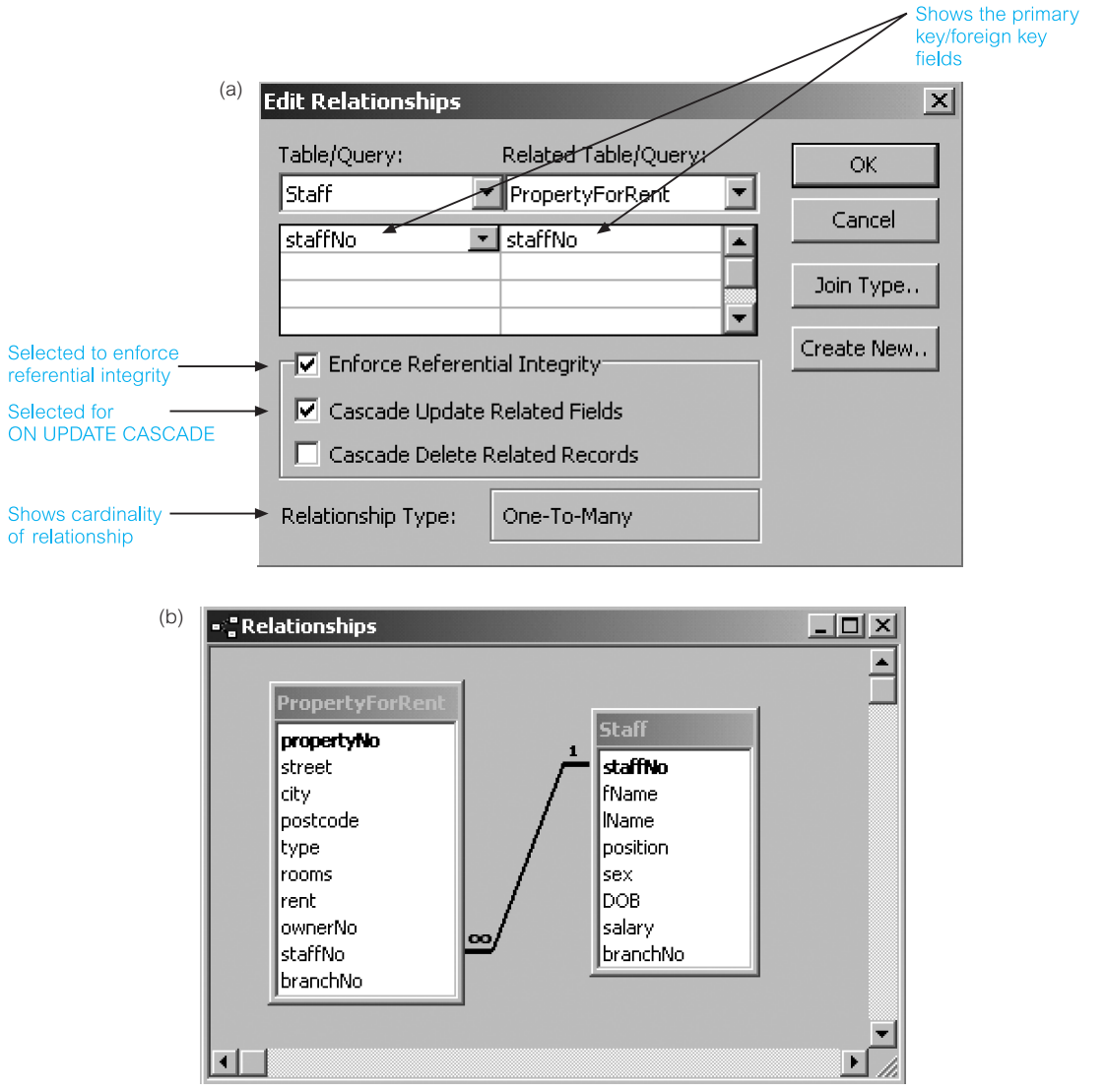


Figure 8.3 (a) Setting the referential integrity constraints for the one-to-many Staff *Manages* PropertyForRent relationship; (b) relationship window with the one-to-many Staff *Manages* PropertyForRent relationship displayed.

8.1.5 General Constraint Definition

There are several ways to create general constraints in Microsoft Office Access using, for example:

- validation rules for fields;
- validation rules for records;
- validation for forms using Visual Basic for Applications (VBA).

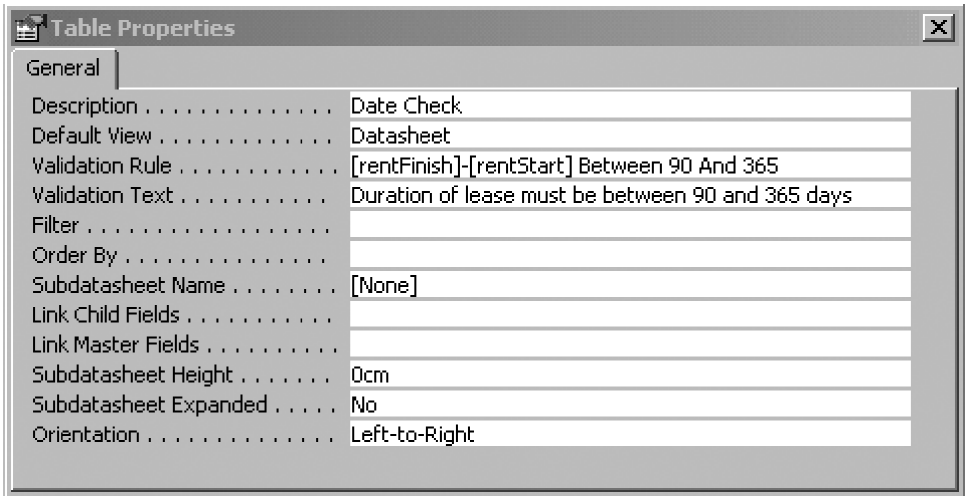


Figure 8.4
Example of record validation in Microsoft Office Access.

We have already seen an example of field validation in Section 8.1.3. In this section, we illustrate the other two methods with some simple examples.

Validation rules for records

A record validation rule controls when an entire record can be saved. Unlike field validation rules, record validation rules can refer to more than one field. This can be useful when values from different fields in a table have to be compared. For example, *DreamHome* has a constraint that the lease period for properties must be between 90 days and 1 year. We can implement this constraint at the record level in the Lease table using the validation rule:

[dateFinish] – [dateStart] Between 90 and 365

Figure 8.4 shows the Table Properties box for the Lease table with this rule set.

Validation for forms using VBA

DreamHome also has a constraint that prevents a member of staff from managing more than 100 properties at any one time. This is a more complex constraint that requires a check on how many properties the member of staff currently manages. One way to implement this constraint in Office Access is to use an **event procedure**. An **event** is a specific action that occurs on or with a certain object. Microsoft Office Access can respond to a variety of events such as mouse clicks, changes in data, and forms opening or closing. Events are usually the result of user action. By using either an event procedure or a macro (see Section 8.1.8), we can customize a user response to an event that occurs on a form, report, or control. Figure 8.5 shows an example of a BeforeUpdate event procedure, which is triggered before a record is updated to implement this constraint.

In some systems, there will be no support for some or all of the general constraints and it will be necessary to design the constraints into the application, as we have shown in Figure 8.5 that has built the constraint into the application's VBA code. Implementing a

Figure 8.5
VBA code to check that a member of staff does not have more than 100 properties to manage at any one time.

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
Dim MyDB As Database
Dim MySet As Recordset
Dim MyQuery As String

'Set up query to select all records for specified member'
MyQuery = "SELECT staffNo FROM PropertyForRent WHERE staffNo = '" + staffNoField + "'"
'Open the database and run the query'
Set MyDB = DBEngine.Workspaces(0).Databases(0)
Set MySet = MyDB.OpenRecordset(MyQuery)

'Check if any records have been returned, then move to the end of the file to allow RecordCount'
'property to be correctly set'
If (NOT MySet.EOF) Then
        MySet.MoveLast
        If (MySet.RecordCount = 100) Then      'If currently 100 – cannot manage any more'
                MsgBox "Member currently managing 100 properties"
                Me.Undo
        End If
End If
MySet.Close
MyDB.Close
End Sub
```

Name of field on form

general constraint in application code is potentially dangerous and can lead to duplication of effort and, worse still, to inconsistencies if the constraint is not implemented everywhere that it should be.

8.1.6 Forms

Microsoft Office Access Forms allow a user to view and edit the data stored in the underlying base tables, presenting the data in an organized and customized manner. Forms are constructed as a collection of individual design elements called *controls* or *control objects*. There are many types of control, such as *text boxes* to enter and edit data, *labels* to hold field names, and *command buttons* to initiate some user action. Controls can be easily added and removed from a form. In addition, Office Access provides a Control Wizard to help the user add controls to a form.

A form is divided into a number of sections, of which the three main ones are:

- *Form Header* This determines what will be displayed at the top of each form, such as a title.
- *Detail* This section usually displays a number of fields in a record.
- *Form Footer* This determines what will be displayed at the bottom of each form, such as a total.

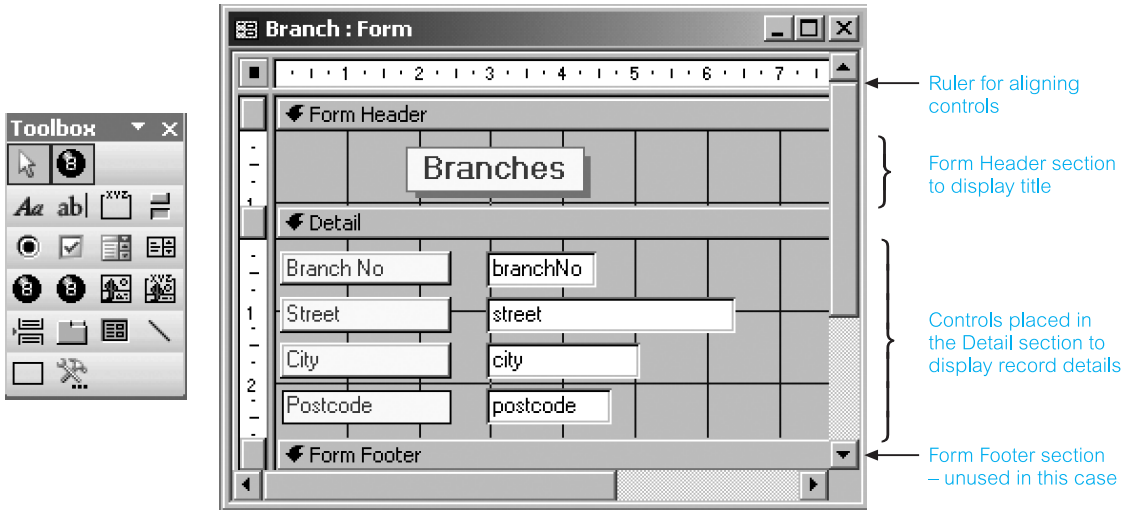


Figure 8.6
Example of a form in Design View with the adjacent toolbox.

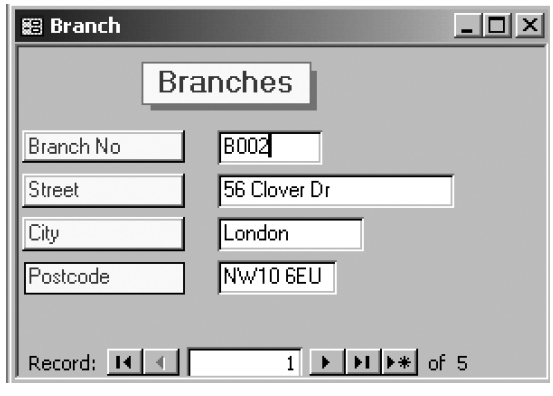
It is also possible for forms to contain other forms, called *subforms*. For example, we may want to display details relating to a branch (the master form) and the details of all staff at that branch (the subform). Normally, subforms are used when there is a relationship between two tables (in this example, we have a one-to-many relationship Branch *Has* Staff).

Forms have three views: Design View, Form View, and Datasheet View. Figure 8.6 shows the construction of a form in Design View to display branch details; the adjacent toolbox gives access to the controls that can be added to the form. In Datasheet View, multiple records can be viewed in the conventional row and column layout and, in Form View, records are typically viewed one at a time. Figure 8.7 shows an example of the branch form in both Datasheet View and Form View.

Figure 8.7
Example of the branch form:
(a) Datasheet View;
(b) Form View.

Office Access allows forms to be created from scratch by the experienced user. However, Office Access also provides a Form Wizard that takes the user through a series of interactive pages to determine:

(a)



(b)

- the table or query that the form is to be based on;
- the fields to be displayed on the form;
- the layout for the form (Columnar, Tabular, Datasheet, or Justified);
- the style for the form based on a predefined set of options;
- the title for the form.

8.1.7 Reports

Microsoft Office Access Reports are a special type of continuous form designed specifically for printing, rather than for displaying in a Window. As such, a Report has only read-access to the underlying base table(s). Among other things, an Office Access Report allows the user to:

- sort records;
- group records;
- calculate summary information;
- control the overall layout and appearance of the report.

As with Forms, a Report's Design View is divided into a number of sections with the main ones being:

- *Report Header* Similar to the Form Header section, this determines what will be displayed at the top of the report, such as a title.
- *Page Header* Determines what will be displayed at the top of each page of the report, such as column headings.
- *Detail* Constitutes the main body of the report, such as details of each record.
- *Page Footer* Determines what will be displayed at the bottom of each page, such as a page number.
- *Report Footer* Determines what will be displayed at the bottom of the report, such as sums or averages that summarize the information in the body of the report.

It is also possible to split the body of the report into groupings based on records that share a common value, and to calculate subtotals for the group. In this case, there are two additional sections in the report:

- *Group Header* Determines what will be displayed at the top of each group, such as the name of the field used for grouping the data.
- *Group Footer* Determines what will be displayed at the bottom of each group, such as a subtotal for the group.

A Report does not have a Datasheet View, only a Design View, a Print Preview, and a Layout Preview. Figure 8.8 shows the construction of a report in Design View to display property for rent details. Figure 8.9 shows an example of the report in Print Preview. Layout Preview is similar to Print Preview but is used to obtain a quick view of the layout of the report and not all records may be displayed.

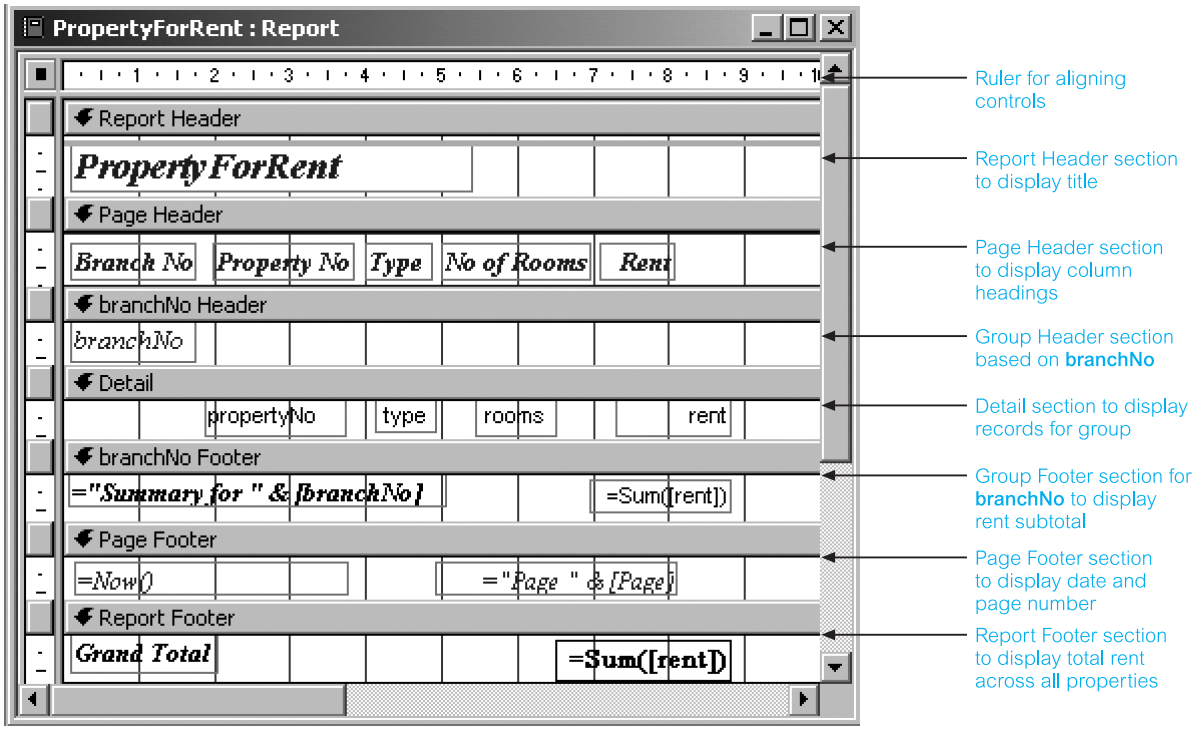


Figure 8.8
Example of a report in Design View.

Office Access allows reports to be created from scratch by the experienced user. However, Office Access also provides a Report Wizard that takes the user through a series of interactive pages to determine:

- the table or query the report is to be based on;
- the fields to be displayed in the report;
- any fields to be used for grouping data in the report along with any subtotals required for the group(s);
- any fields to be used for sorting the data in the report;
- the layout for the report;
- the style for the report based on a predefined set of options;
- the title for the report.

Macros 8.1.8

As discussed earlier, Microsoft Office Access uses an event-driven programming paradigm. Office Access can recognize certain events, such as:

- mouse events, which occur when a mouse action, such as pressing down or clicking a mouse button, occurs;

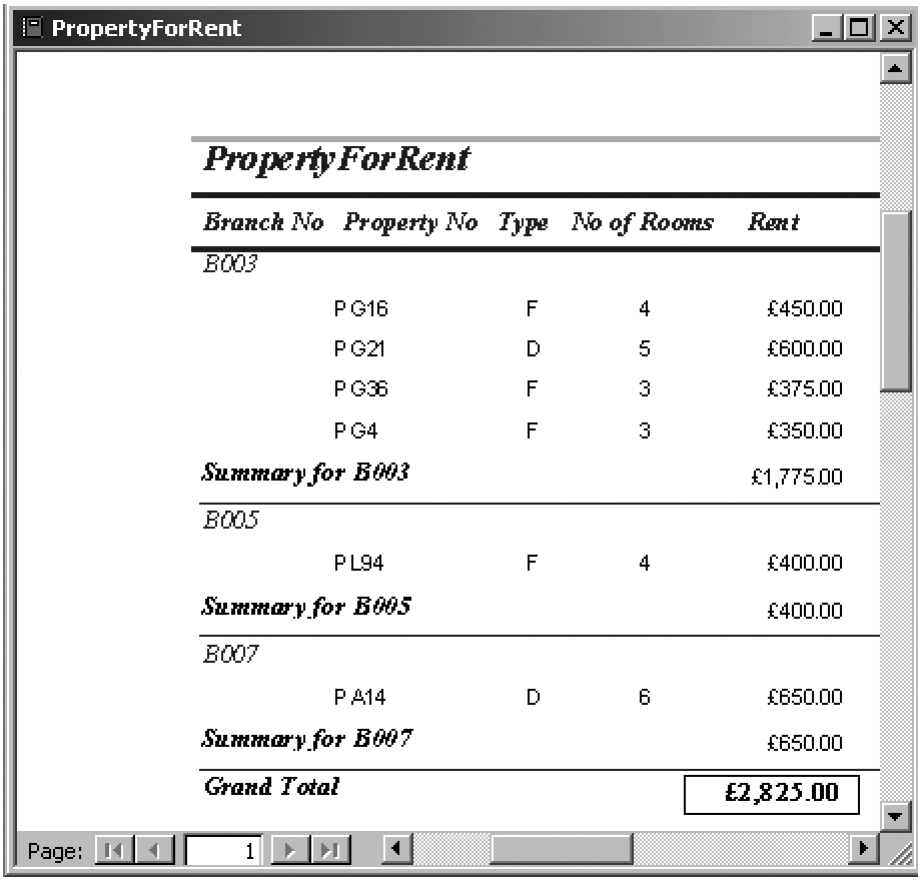


Figure 8.9
Example of a report for the PropertyForRent table with a grouping based on the branchNo field in Print Preview.

- keyboard events, which occur, for example, when the user types on the keyboard;
- focus events, which occur when a form or form control gains or loses focus or when a form or report becomes active or inactive;
- data events, which occur when data is entered, deleted, or changed in a form or control, or when the focus moves from one record to another.

Office Access allows the user to write **macros** and **event procedures** that are triggered by an event. We saw an example of an event procedure in Section 8.1.5. In this section, we briefly describe macros.

Macros are very useful for automating repetitive tasks and ensuring that these tasks are performed consistently and completely each time. A macro consists of a list of *actions* that Office Access is to perform. Some actions duplicate menu commands such as Print, Close, and ApplyFilter. Some actions substitute for mouse actions such as the SelectObject action, which selects a database object in the same way that a database object is selected by clicking the object's name. Most actions require additional information as *action arguments* to determine how the action is to function. For example, to use the SetValue action,

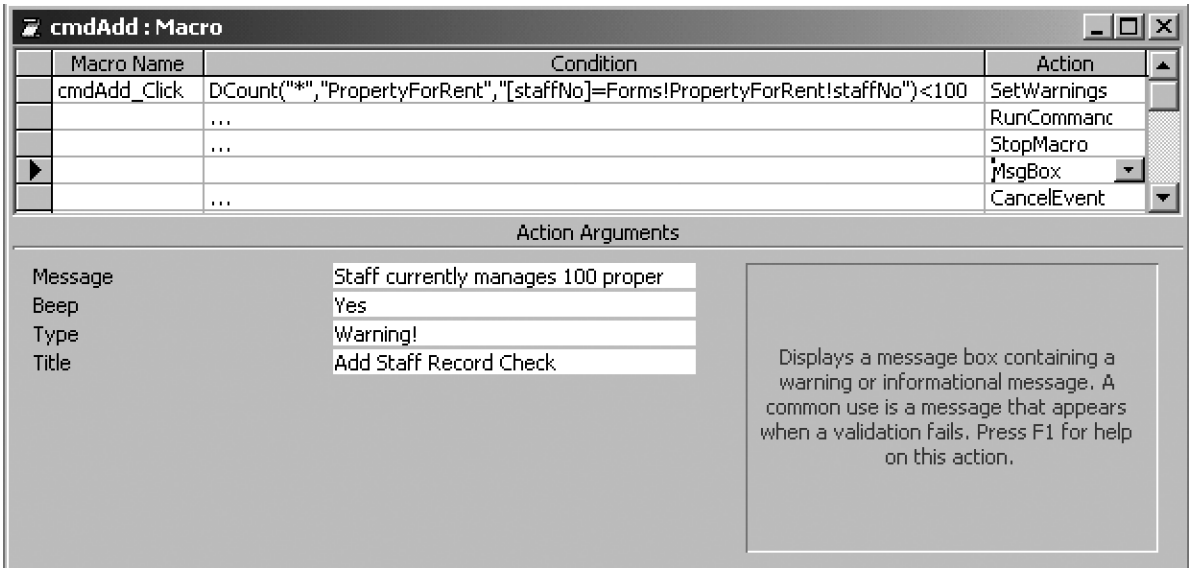


Figure 8.10 Macro to check that a member of staff currently has fewer than 100 properties to manage.

which sets the value of a field, control, or property on a form or report, we need to specify the item to be set and an expression representing the value for the specified item. Similarly, to use the MsgBox action, which displays a pop-up message box, we need to specify the text to go into the message box.

Figure 8.10 shows an example of a macro that is called when a user tries to add a new property for rent record into the database. The macro enforces the enterprise constraint that a member of staff cannot manage more than 100 properties at any one time, which we showed previously how to implement using an event procedure written in VBA (see Figure 8.5). In this example, the macro checks whether the member of staff specified on the PropertyForRent form (Forms!PropertyForRent!staffNo) is currently managing less than 100 properties. If so, the macro uses the RunCommand action with the argument Save (to save the new record) and then uses the StopMacro action to stop. Otherwise, the macro uses the MsgBox action to display an error message and uses the CancelEvent macro to cancel the addition of the new record. This example also demonstrates:

- use of the DCOUNT function to check the constraint instead of a SELECT COUNT(*) statement;
- use of an ellipsis (. . .) in the Condition column to run a series of actions associated with a condition.

In this case, the SetWarnings, RunCommand, and StopMacro actions are called if the condition

DCOUNT(“*”, “PropertyForRent”, “[staffNo] = Forms!PropertyForRent!staffNo”) < 100

evaluates to true, otherwise the MsgBox and CancelEvent actions are called.

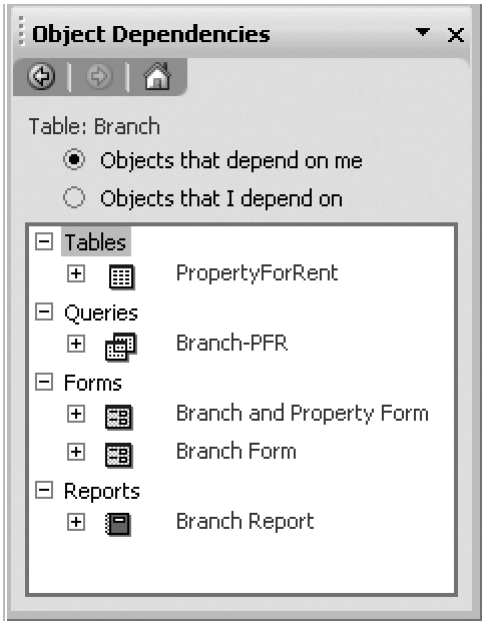


Figure 8.11 Object Dependencies task pane showing the dependencies for the Branch table.

8.1.9 Object Dependencies

Microsoft Office Access now allows dependencies between database objects (tables, queries, forms, and reports) to be viewed. This can be particularly useful for identifying objects that are no longer required or for maintaining consistency after an object has been modified. For example, if we add a new field to the Branch table, we can use the Object Dependencies task pane shown in Figure 8.11 to identify which queries, forms, and reports may need to be modified to include the additional field. It is also possible to list the objects that are being used by a selected object.

8.2 Oracle9*i*

The Oracle Corporation is the world's leading supplier of software for information management, and the world's second largest independent software company. With annual revenues of about US$10 billion, the company offers its database, tools, and application products, along with related services, in more than 145 countries around the world. Oracle is the top-selling multi-user RDBMS with 98% of Fortune 100 companies using Oracle Solutions (Oracle Corporation, 2003).

Oracle's integrated suite of business applications, Oracle E-Business Suite, covers business intelligence, financials (such as accounts receivable, accounts payable, and general ledger), human resources, procurement, manufacturing, marketing, projects, sales, services, asset enterprise management, order fulfilment, product development, and treasury.

Oracle has undergone many revisions since its first release in the late 1970s, but in 1997 Oracle8 was released with extended object-relational capabilities, and improved performance and scalability features. In 1999, Oracle8*i* was released with added functionality

Table 8.2 Oracle9*i* family of products.

Product	Description
Oracle9*i* Standard Edition	Oracle for low to medium volume OLTP (Online Transaction Processing) environments.
Oracle9*i* Enterprise Edition	Oracle for a large number of users or large database size, with advanced management, extensibility, and performance features for mission-critical OLTP environments, query intensive data warehousing applications, and demanding Internet applications.
Oracle9*i* Personal Edition	Single-user version of Oracle, typically for development of applications deployed on Oracle9*i* Standard/Enterprise Edition.

supporting Internet deployment and in 2001 Oracle9*i* was released with additional functionality aimed at the e-Business environments. There are three main products in the Oracle9*i* family, as shown in Table 8.2.

Within this family, Oracle offers a number of advanced products and options such as:

- *Oracle Real Application Clusters* As performance demands increase and data volumes continue to grow, the use of database servers with multiple CPUs, called symmetric multiprocessing (SMP) machines, are becoming more common. The use of multiple processors and disks reduces the time to complete a given task and at the same time provides greater availability and scalability. The Oracle Real Application Clusters supports parallelism within a single SMP server as well as parallelism across multiple nodes.
- *Oracle9i Application Server* (Oracle9*i*AS) Provides a means of implementing the middle tier of a three-tier architecture for Web-based applications. The first tier is a Web browser and the third tier is the database server. We discuss the Oracle9*i* Application Server in more detail in Chapter 29.
- *Oracle9iAS Portal* An HTML-based tool for developing Web-enabled applications and content-enabled Web sites.
- *i*FS Bundled now with Oracle9*i*AS, Oracle Internet File System (*i*FS) makes it possible to treat an Oracle9*i* database like a shared network drive, allowing users to store and retrieve files managed by the database as if they were files managed by a file server.
- *Java support* Oracle has integrated a secure Java Virtual Machine with the Oracle9*i* database server. Oracle JVM supports Java stored procedures and triggers, Java methods, CORBA objects, Enterprise JavaBeans (EJB), Java Servlets, and JavaServer Pages (JSPs). It also supports the Internet Inter-Object Protocol (IIOP) and the HyperText Transfer Protocol (HTTP). Oracle provides JDeveloper to help develop basic Java applications. We discuss Java support in more detail in Chapter 29.
- *XML support* Oracle includes a number of features to support XML. The XML Development Kit (XDK) allows developers to send, receive, and interpret XML data from applications written in Java, C, C++, and PL/SQL. The XML Class Generator creates Java/C++ classes from XML Schema definitions. The XML SQL utility

supports reading and writing XML data to and from the database using SQL (through the DBMS–XMLGEN package). Oracle9*i* also includes the new XMLType data type, which allows an XML document to be stored in a character LOB column (see Table 8.3 on page 253), with built-in functions to extract individual nodes from the document and to build indexes on any node in the document. We discuss XML in Chapter 30.

- *inter*MEDIA Enables Oracle9*i* to manage text, documents, image, audio, video, and locator data. It supports a variety of Web client interfaces, Web development tools, Web servers, and streaming media servers.
- *Visual Information Retrieval* Supports content-based queries based on visual attributes of an image, such as color, structure, and texture.
- *Time Series* Allows timestamped data to be stored in the database. Includes calendar functions and time-based analysis functions such as calculating moving averages.
- *Spatial* Optimizes the retrieval and display of data linked to spatial information.
- *Distributed database features* Allow data to be distributed across a number of database servers. Users can query and update this data as if it existed in a single database. We discuss distributed DBMSs and examine the Oracle distribution facilities in Chapters 22 and 23.
- *Advanced Security* Used in a distributed environment to provide secure access and transmission of data. Includes network data encryption using RSA Data Security's RC4 or DES algorithm, network data integrity checking, enhanced authentication, and digital certificates (see Chapter 19).
- *Data Warehousing* Provides tools that support the extraction, transformation, and loading of organizational data sources into a single database, and tools that can then be used to analyze this data for strategic decision-making. We discuss data warehouses and examine the Oracle data warehouse facilities in Chapters 31 and 32.
- *Oracle Internet Developer Suite* A set of tools to help developers build sophisticated database applications. We discuss this suite in Section 8.2.8.

8.2.1 Objects

The user interacts with Oracle and develops a database using a number of objects, the main objects being:

- *Tables* The base tables that make up the database. Using the Oracle terminology, a table is organized into *columns* and *rows*. One or more tables are stored within a tablespace (see Section 8.2.2). Oracle also supports temporary tables that exist only for the duration of a transaction or session.
- *Objects* Object types provide a way to extend Oracle's relational data type system. As we saw in Section 6.1, SQL supports three regular data types: characters, numbers, and dates. Object types allow the user to define new data types and use them as regular relational data types would be used. We defer discussion of Oracle's object-relational features until Chapter 28.

- *Clusters* A cluster is a set of tables physically stored together as one table that shares common columns. If data in two or more tables are frequently retrieved together based on data in the common column, using a cluster can be quite efficient. Tables can be accessed separately even though they are part of a cluster. Because of the structure of the cluster, related data requires much less input/output (I/O) overhead if accessed simultaneously. Clusters are discussed in Appendix C and we give guidelines for their use.
- *Indexes* An index is a structure that provides accelerated access to the rows of a table based on the values in one or more columns. Oracle supports index-only tables, where the data and index are stored together. Indexes are discussed in Appendix C and guidelines for when to create indexes are provided in Step 5.3 in Chapter 17.
- *Views* A view is a *virtual table* that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request (see Section 6.4).
- *Synonyms* These are alternative names for objects in the database.
- *Sequences* The Oracle sequence generator is used to automatically generate a unique sequence of numbers in cache. The sequence generator avoids the user having to create the sequence, for example by locking the row that has the last value of the sequence, generating a new value, and then unlocking the row.
- *Stored functions* These are a set of SQL or PL/SQL statements used together to execute a particular function and stored in the database. PL/SQL is Oracle's procedural extension to SQL.
- *Stored procedures* Procedures and functions are identical except that functions always return a value (procedures do not). By processing the SQL code on the database server, the number of instructions sent across the network and returned from the SQL statements are reduced.
- *Packages* These are a collection of procedures, functions, variables, and SQL statements that are grouped together and stored as a single program unit in the database.
- *Triggers* Triggers are code stored in the database and invoked (*triggered*) by events that occur in the database.

Before we discuss some of these objects in more detail, we first examine the architecture of Oracle.

Oracle Architecture 8.2.2

Oracle is based on the client–server architecture examined in Section 2.6.3. The Oracle server consists of the *database* (the raw data, including log and control files) and the *instance* (the processes and system memory on the server that provide access to the database). An instance can connect to only one database. The database consists of a *logical structure*, such as the database schema, and a *physical structure*, containing the files that make up an Oracle database. We now discuss the logical and physical structure of the database and the system processes in more detail.

Oracle's logical database structure

At the logical level, Oracle maintains *tablespaces*, *schemas*, and *data blocks* and *extents*/*segments*.

Tablespaces

An Oracle database is divided into logical storage units called **tablespaces**. A tablespace is used to group related logical structures together. For example, tablespaces commonly group all the application's objects to simplify some administrative operations.

Every Oracle database contains a tablespace named SYSTEM, which is created automatically when the database is created. The SYSTEM tablespace always contains the system catalog tables (called the *data dictionary* in Oracle) for the entire database. A small database might need only the SYSTEM tablespace; however, it is recommended that at least one additional tablespace is created to store user data separate from the data dictionary, thereby reducing contention among dictionary objects and schema objects for the same datafiles (see Figure 16.2 in Chapter 16). Figure 8.12 illustrates an Oracle database consisting of the SYSTEM tablespace and a USER_DATA tablespace.

A new tablespace can be created using the CREATE TABLESPACE command, for example:

```
CREATE TABLESPACE user_data
DATAFILE 'DATA3.ORA' SIZE 100K
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
```

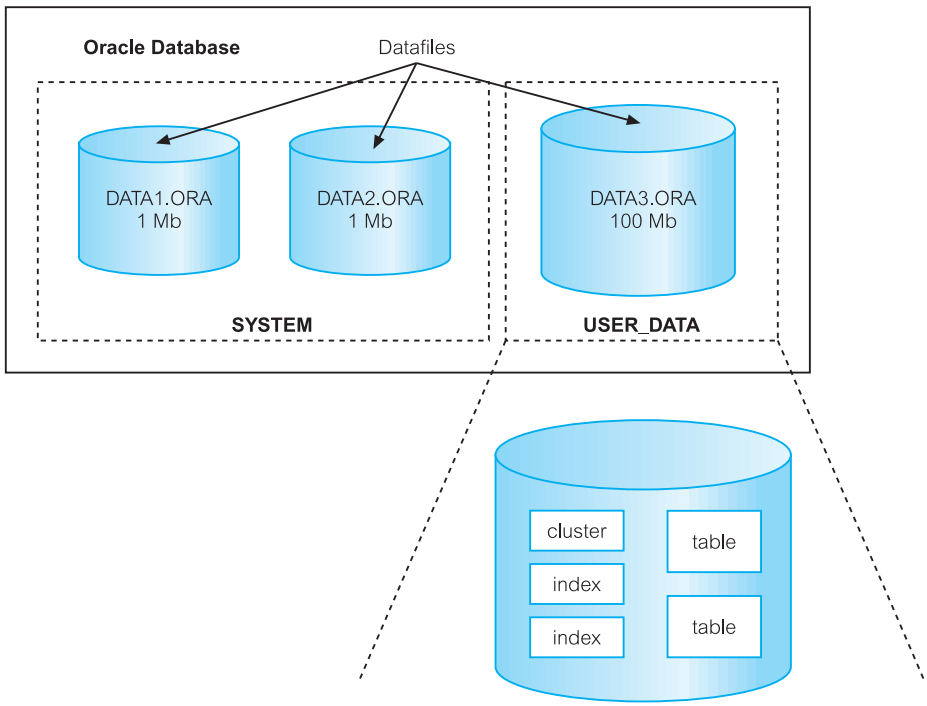


Figure 8.12 Relationship between an Oracle database, tablespaces, and datafiles.

A table can then be associated with a specific tablespace using the CREATE TABLE or ALTER TABLE statement, for example:

```
CREATE TABLE PropertyForRent (propertyNo VARCHAR2(5) NOT NULL, . . . )
TABLESPACE user_data;
```

If no tablespace is specified when creating a new table, the default tablespace associated with the user when the user account was set up is used. We see how this default tablespace can be specified in Section 18.4.

Users, schemas, and schema objects

A **user** (sometimes called a **username**) is a name defined in the database that can connect to, and access, objects. A **schema** is a named collection of schema objects, such as tables, views, indexes, clusters, and procedures, associated with a particular user. Schemas and users help DBAs manage database security.

To access a database, a user must run a database application (such as Oracle Forms or SQL*Plus) and connect using a username defined in the database. When a database user is created, a corresponding schema of the same name is created for the user. By default, once a user connects to a database, the user has access to all objects contained in the corresponding schema. As a user is associated only with the schema of the same name, the terms 'user' and 'schema' are often used interchangeably. (Note there is no relationship between a tablespace and a schema: objects in the same schema can be in different tablespaces, and a tablespace can hold objects from different schemas.)

Data blocks, extents, and segments

The **data block** is the smallest unit of storage that Oracle can use or allocate. One data block corresponds to a specific number of bytes of physical disk space. The data block size can be set for each Oracle database when it is created. This data block size should be a multiple of the operating system's block size (within the system's maximum operating limit) to avoid unnecessary I/O. A data block has the following structure:

- *Header* Contains general information such as block address and type of segment.
- *Table directory* Contains information about the tables that have data in the data block.
- *Row directory* Contains information about the rows in the data block.
- *Row data* Contains the actual rows of table data. A row can span blocks.
- *Free space* Allocated for the insertion of new rows and updates to rows that require additional space. Since Oracle8*i*, Oracle can manage free space automatically, although there is an option to manage it manually.

We show how to estimate the size of an Oracle table using these components in Appendix G. The next level of logical database space is called an **extent**. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information. The level above an extent is called a **segment**. A segment is a set of extents allocated for a certain logical structure. For example, each table's data is stored in its own data segment, while each index's data is stored in its own index segment. Figure 8.13 shows the relationship between data blocks, extents, and segments. Oracle dynamically allocates space when the existing extents of a segment become full. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on disk.

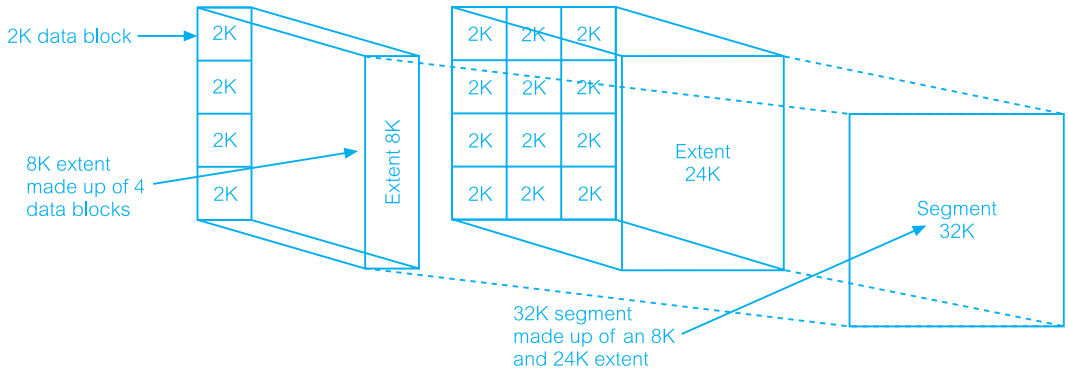


Figure 8.13 Relationship between Oracle data blocks, extents, and segments.

Oracle's physical database structure

The main physical database structures in Oracle are datafiles, redo log files, and control files.

Datafiles

Every Oracle database has one or more physical datafiles. The data of logical database structures (such as tables and indexes) is physically stored in these datafiles. As shown in Figure 8.12, one or more datafiles form a tablespace. The simplest Oracle database would have one tablespace and one datafile. A more complex database might have four tablespaces, each consisting of two datafiles, giving a total of eight datafiles.

Redo log files

Every Oracle database has a set of two or more redo log files that record all changes made to data for recovery purposes. Should a failure prevent modified data from being permanently written to the datafiles, the changes can be obtained from the redo log, thus preventing work from being lost. We discuss recovery in detail in Section 20.3.

Control files

Every Oracle database has a control file that contains a list of all the other files that make up the database, such as the datafiles and redo log files. For added protection, it is recommended that the control file should be multiplexed (multiple copies may be written to multiple devices). Similarly, it may be advisable to multiplex the redo log files as well.

The Oracle instance

The Oracle instance consists of the Oracle processes and shared memory required to access information in the database. The instance is made up of the Oracle background processes, the user processes, and the shared memory used by these processes, as illustrated in Figure 8.14. Among other things, Oracle uses shared memory for caching data and indexes as well as storing shared program code. Shared memory is broken into various

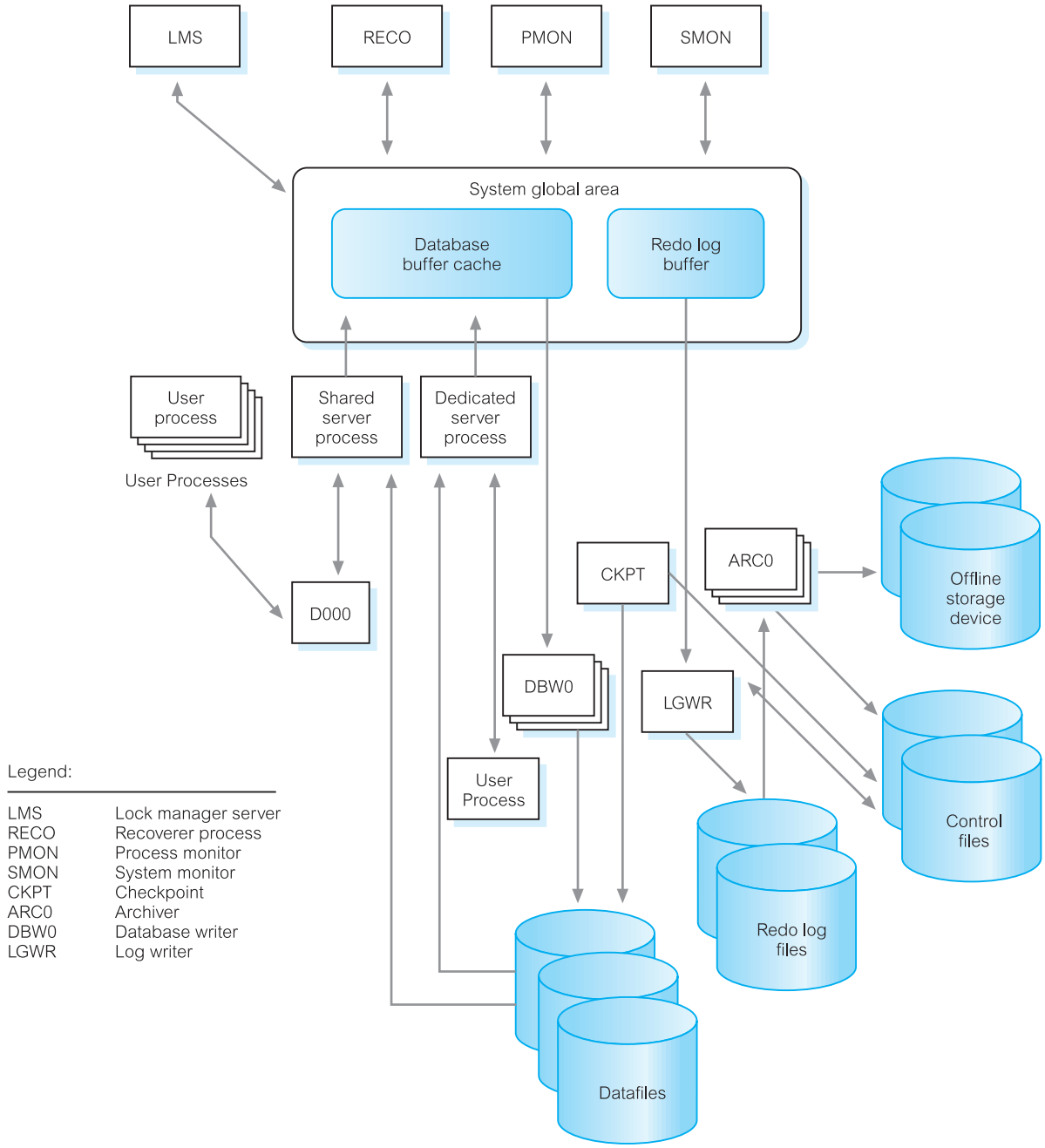


Figure 8.14 The Oracle architecture (from the Oracle documentation set).

memory structures, of which the basic ones are the System Global Area (SGA) and the Program Global Area (PGA).

- *System global area* The SGA is an area of shared memory that is used to store data and control information for one Oracle instance. The SGA is allocated when the Oracle instance starts and deallocated when the Oracle instance shuts down. The information in the SGA consists of the following memory structures, each of which has a fixed size and is created at instance startup:
  - *Database buffer cache* This contains the most recently used data blocks from the database. These blocks can contain modified data that has not yet been written to disk (*dirty blocks*), blocks that have not been modified, or blocks that have been written to disk since modification (*clean blocks*). By storing the most recently used blocks, the most active buffers stay in memory to reduce I/O and improve performance. We discuss buffer management policies in Section 20.3.2.
  - *Redo log buffer* This contains the redo log file entries, which are used for recovery purposes (see Section 20.3). The background process LGWR writes the redo log buffer to the active online redo log file on disk.
  - *Shared pool* This contains the shared memory structures such as shared SQL areas in the library cache and internal information in the data dictionary. The shared SQL areas contain parse trees and execution plans for the SQL queries. If multiple applications issue the same SQL statement, each can access the shared SQL area to reduce the amount of memory needed and to reduce the processing time used for parsing and execution. We discuss query processing in Chapter 21.
- *Program global area* The PGA is an area of shared memory that is used to store data and control information for the Oracle server processes. The size and content of the PGA depends on the Oracle server options installed.
- *User processes* Each user process represents the user's connection to the Oracle server (for example, through SQL*Plus or an Oracle Forms application). The user process manipulates the user's input, communicates with the Oracle server process, displays the information requested by the user and, if required, processes this information into a more useful form.
- *Oracle processes* Oracle (server) processes perform functions for users. Oracle processes can be split into two groups: *server processes* (which handle requests from connected user processes) and *background processes* (which perform asynchronous I/O and provide increased parallelism for improved performance and reliability). From Figure 8.14, we have the following background processes:
  - *Database Writer (DBWR)* The DBWR process is responsible for writing the modified (dirty) blocks from the buffer cache in the SGA to datafiles on disk. An Oracle instance can have up to ten DBWR processes, named DBW0 to DBW9, to handle I/O to multiple datafiles. Oracle employs a technique known as write-ahead logging (see Section 20.3.4), which means that the DBWR process performs *batched writes* whenever the buffers need to be freed, not necessarily at the point the transaction commits.
  - *Log Writer (LGWR)* The LGWR process is responsible for writing data from the log buffer to the redo log.

- *Checkpoint (CKPT)* A checkpoint is an event in which all modified database buffers are written to the datafiles by the DBWR (see Section 20.3.3). The CKPT process is responsible for telling the DBWR process to perform a checkpoint and to update all the datafiles and control files for the database to indicate the most recent checkpoint. The CKPT process is optional and, if omitted, these responsibilities are assumed by the LGWR process.
- *System Monitor (SMON)* The SMON process is responsible for crash recovery when the instance is started following a failure. This includes recovering transactions that have died because of a system crash. SMON also defragments the database by merging free extents within the datafiles.
- *Process Monitor (PMON)* The PMON process is responsible for tracking user processes that access the database and recovering them following a crash. This includes cleaning up any resources left behind (such as memory) and releasing any locks held by the failed process.
- *Archiver (ARCH)* The ARCH process is responsible for copying the online redo log files to archival storage when they become full. The system can be configured to run up to ten ARCH processes, named ARC0 to ARC9. The additional archive processes are started by the LWGR when the load dictates.
- *Recoverer (RECO)* The RECO process is responsible for cleaning up failed or suspended distributed transactions (see Section 23.4).
- *Dispatchers (Dnnn)* The D*nnn* processes are responsible for routing requests from the user processes to available shared server processes and back again. Dispatchers are present only when the Shared Server (previously known as the Multi-Threaded Server, MTS) option is used, in which case there is at least one D*nnn* process for every communications protocol in use.
- *Lock Manager Server (LMS)* The LMS process is responsible for inter-instance locking when the Oracle Real Application Clusters option is used.

In the foregoing descriptions we have used the term 'process' generically. Nowadays, some systems will implement processes as *threads*.

Example of how these processes interact

The following example illustrates an Oracle configuration with the server process running on one machine and a user process connecting to the server from a separate machine. Oracle uses a communication mechanism called Oracle Net Services to allow processes on different physical machines to communicate with each other. Oracle Net Services supports a variety of network protocols such as TCP/IP. The services can also perform network protocol interchanges, allowing clients that use one protocol to interact with a database server using another protocol.

(1) The client workstation runs an application in a user process. The client application attempts to establish a connection to the server using the Oracle Net Services driver.

(2) The server detects the connection request from the application and creates a (dedicated) server process on behalf of the user process.

(3) The user executes an SQL statement to change a row of a table and commits the transaction.

(4) The server process receives the statement and checks the shared pool for any shared SQL area that contains an identical SQL statement. If a shared SQL area is found, the server process checks the user's access privileges to the requested data and the previously existing shared SQL area is used to process the statement; if not, a new shared SQL area is allocated for the statement so that it can be parsed and processed.

(5) The server process retrieves any necessary data values from the actual datafile (table) or those stored in the SGA.

(6) The server process modifies data in the SGA. The DBWR process writes modified blocks permanently to disk when doing so is efficient. Since the transaction committed, the LGWR process immediately records the transaction in the online redo log file.

(7) The server process sends a success/failure message across the network to the application.

(8) During this time, the other background processes run, watching for conditions that require intervention. In addition, the Oracle server manages other users' transactions and prevents contention between transactions that request the same data.

8.2.3 Table Definition

In Section 6.3.2, we examined the SQL CREATE TABLE statement. Oracle9*i* supports many of the SQL CREATE TABLE clauses, so we can define:

- primary keys, using the PRIMARY KEY clause;
- alternate keys, using the UNIQUE keyword;
- default values, using the DEFAULT clause;
- not null attributes, using the NOT NULL keyword;
- foreign keys, using the FOREIGN KEY clause;
- other attribute or table constraints using the CHECK and CONSTRAINT clauses.

However, there is no facility to create domains, although Oracle9*i* does allow user-defined types to be created, as we discuss in Section 28.6. In addition, the data types are slightly different from the SQL standard, as shown in Table 8.3.

Sequences

In the previous section we mentioned that Microsoft Office Access has an Autonumber data type that creates a new sequential number for a column value whenever a row is inserted. Oracle does not have such a data type but it does have a similar facility through the SQL CREATE SEQUENCE statement. For example, the statement:

```
CREATE SEQUENCE appNoSeq
START WITH 1 INCREMENT BY 1 CACHE 30;
```

creates a sequence, called appNoSeq, that starts with the initial value 1 and increases by 1 each time. The CACHE 30 clause specifies that Oracle should pre-allocate 30 sequence

Table 8.3 Partial list of Oracle data types

Data type	Use	Size
char(size)	Stores fixed-length character data (default size is 1).	Up to 2000 bytes
nchar(size)	Unicode data types that store Unicode character data. Same as char data type, except the maximum length is determined by the character set of the database (for example, American English, eastern European, or Korean).	Up to 4000 bytes
varchar2(size)	Stores variable length character data.	
nvarchar2(size)	Same as varchar2 with the same caveat as for nchar data type.	Up to 2000 bytes
varchar	Currently the same as char. However, use of varchar2 is recommended as varchar might become a separate data type with different comparison semantics in a later release.	
number(l, d)	Stores fixed-point or floating-point numbers, where **l** stands for length and **d** stands for the number of decimal digits. For example, number(5, 2) could contain nothing larger than 999.99 without an error.	±1.0E−130 . . . ±9.99E125 (up to 38 significant digits)
decimal(l, d), dec(l, d), or numeric(l, d)	Same as number. Provided for compatibility with SQL standard.	
integer, int, or smallint	Provided for compatibility with SQL standard. Converted to number(38).	
date	Stores dates from 1 Jan 4712 BC to 31 Dec 4712 AD	Up to 4 gigabytes
blob	A binary large object.	Up to 4 gigabytes
clob	A character large object.	Up to 2000 bytes
raw(size)	Raw binary data, such as a sequence of graphics characters or a digitized picture.	

numbers and keep them in memory for faster access. Once a sequence has been created, its values can be accessed in SQL statements using the following pseudocolumns:

- CURRVAL Returns the current value of the sequence.
- NEXTVAL Increments the sequence and returns the new value.

For example, the SQL statement:

```
INSERT INTO Appointment(appNo, aDate, aTime, clientNo)
VALUES (appNoSeq.nextval, SYSDATE, '12.00', 'CR76');
```

inserts a new row into the Appointment table with the value for column appNo (the appointment number) set to the next available number in the sequence. We now illustrate how to create the PropertyForRent table in Oracle with the constraints specified in Example 6.1.

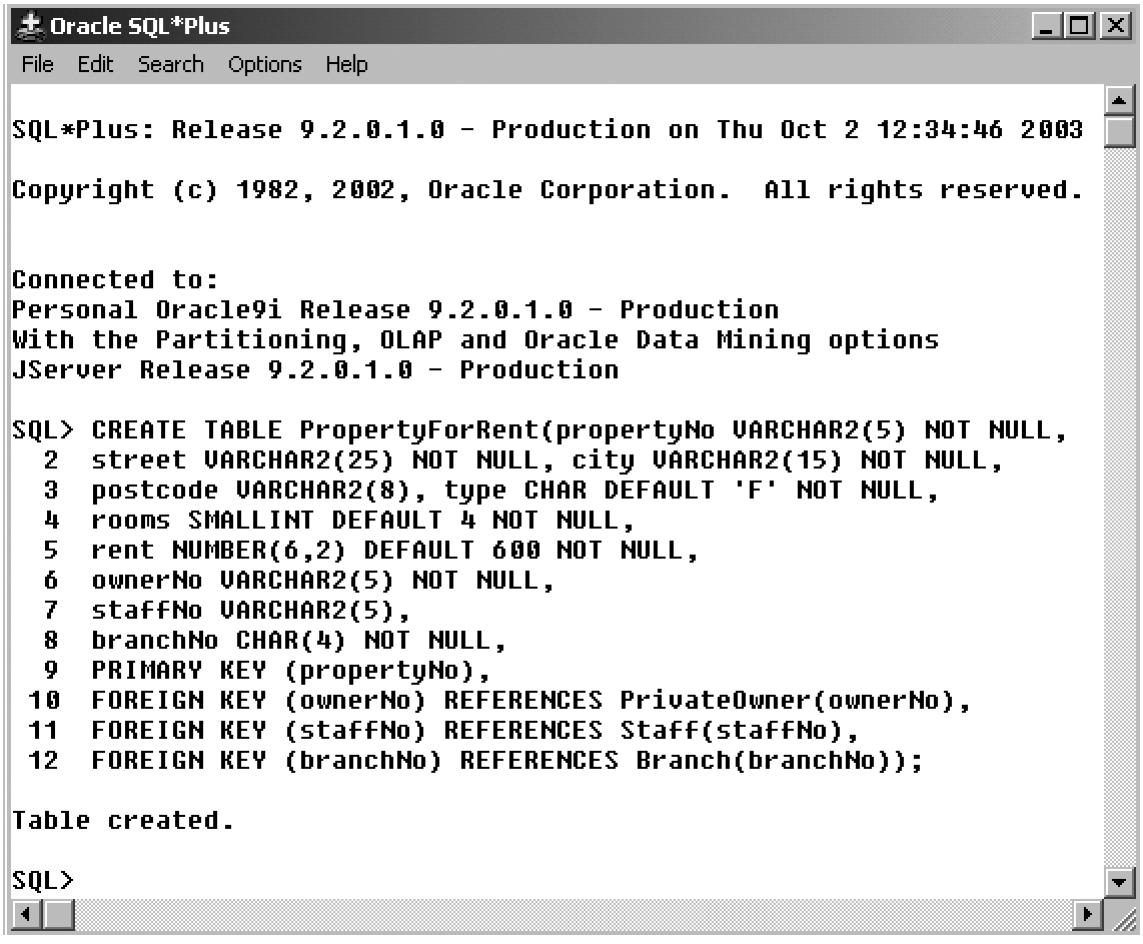


Figure 8.15 Creating the PropertyForRent table using the Oracle SQL CREATE TABLE statement in SQL*Plus.

Creating a blank table in Oracle using SQL*Plus

To illustrate the process of creating a blank table in Oracle, we first use **SQL*Plus**, which is an interactive, command-line driven, SQL interface to the Oracle database. Figure 8.15 shows the creation of the PropertyForRent table using the Oracle SQL CREATE TABLE statement.

By default, Oracle enforces the referential actions ON DELETE NO ACTION and ON UPDATE NO ACTION on the named foreign keys. It also allows the additional clause ON DELETE CASCADE to be specified to allow deletions from the parent table to cascade to the child table. However, it does not support the ON UPDATE CASCADE action or the SET DEFAULT and SET NULL actions. If any of these actions are required, they have to be implemented as triggers or stored procedures, or within the application code. We see an example of a trigger to enforce this type of constraint in Section 8.2.7.

Creating a table using the Create Table Wizard

An alternative approach in Oracle9*i* is to use the **Create Table Wizard** that is part of the **Schema Manager**. Using a series of interactive forms, the Create Table Wizard takes the user through the process of defining each of the columns with its associated data type, defining any constraints on the columns and/or constraints on the table that may be required, and defining the key fields. Figure 8.16 shows the final form of the Create Table Wizard used to create the PropertyForRent table.

General Constraint Definition 8.2.4

There are several ways to create general constraints in Oracle using, for example:

- SQL, and the CHECK and CONSTRAINT clauses of the CREATE and ALTER TABLE statements;
- stored procedures and functions;
- triggers;
- methods.

The first approach was dealt with in Section 6.1. We defer treatment of methods until Chapter 28 on Object-Relational DBMSs. Before we illustrate the remaining two approaches, we first discuss Oracle's procedural programming language, PL/SQL.

PL/SQL 8.2.5

PL/SQL is Oracle's procedural extension to SQL. There are two versions of PL/SQL: one is part of the Oracle server, the other is a separate engine embedded in a number of Oracle tools. They are very similar to each other and have the same programming constructs, syntax, and logic mechanisms, although PL/SQL for Oracle tools has some extensions to suit the requirements of the particular tool (for example, PL/SQL has extensions for Oracle Forms).

PL/SQL has concepts similar to modern programming languages, such as variable and constant declarations, control structures, exception handling, and modularization. PL/SQL is a block-structured language: blocks can be entirely separate or nested within one another. The basic units that comprise a PL/SQL program are procedures, functions, and anonymous (*unnamed*) blocks. As illustrated in Figure 8.17, a PL/SQL block has up to three parts:

- an optional declaration part in which variables, constants, cursors, and exceptions are defined and possibly initialized;
- a mandatory executable part, in which the variables are manipulated;
- an optional exception part, to handle any exceptions raised during execution.

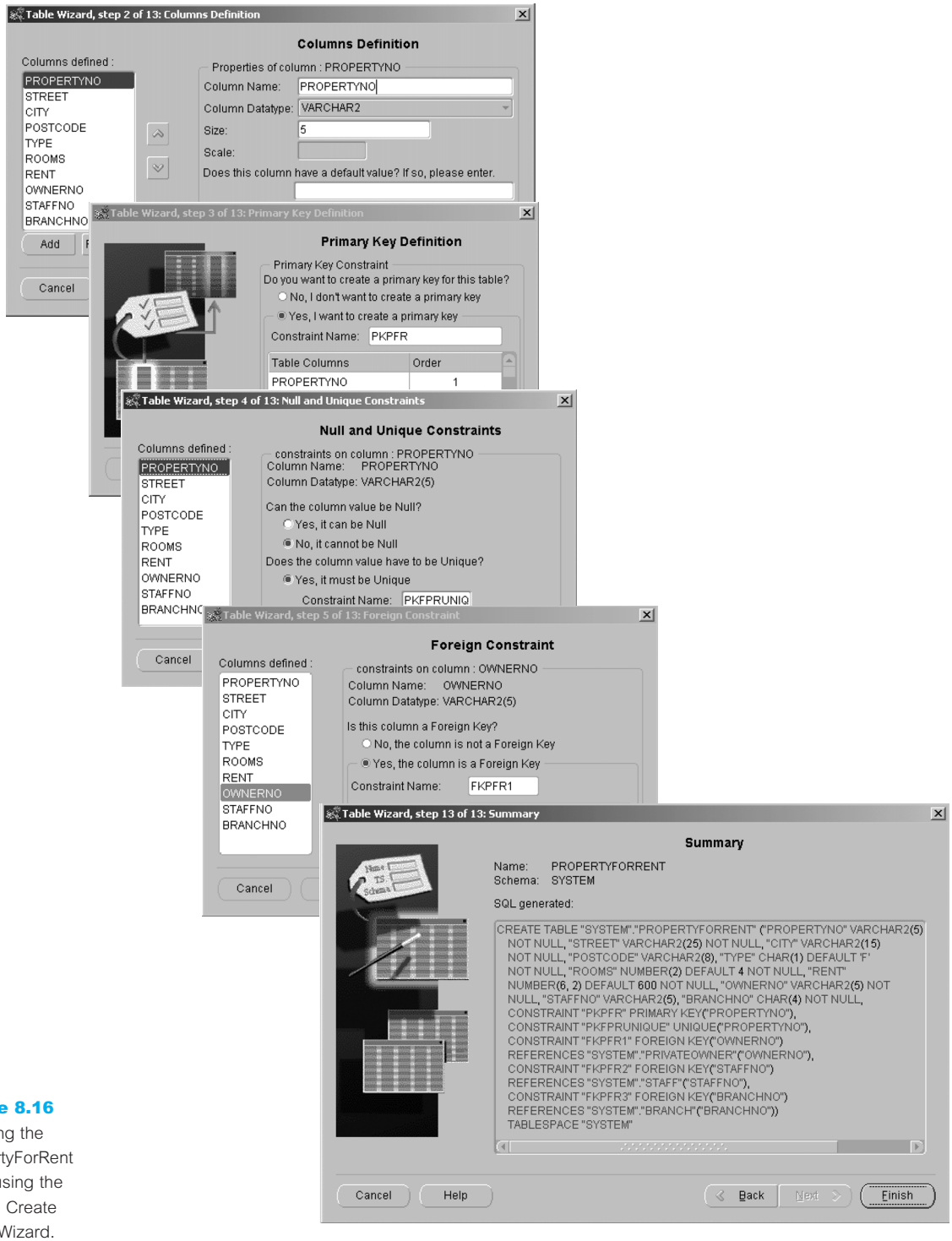


Figure 8.16
Creating the PropertyForRent table using the Oracle Create Table Wizard.

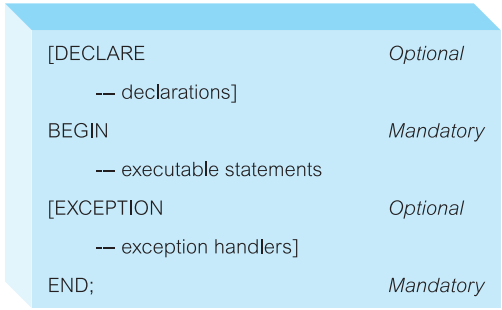


Figure 8.17
General structure of a PL/SQL block.

Declarations

Variables and constant variables must be declared before they can be referenced in other statements, including other declarative statements. The types of variables are as shown in Table 8.3. Examples of declarations are:

vStaffNo **VARCHAR2**(5);
vRent **NUMBER**(6, 2) **NOT NULL** := 600;
MAX_PROPERTIES **CONSTANT NUMBER** := 100;

Note that it is possible to declare a variable as NOT NULL, although in this case an initial value must be assigned to the variable. It is also possible to declare a variable to be of the same type as a column in a specified table or another variable using the %TYPE attribute. For example, to declare that the vStaffNo variable is the same type as the staffNo column of the Staff table we could write:

vStaffNo Staff.staffNo%**TYPE**;
vStaffNo1 vStaffNo%**TYPE**;

Similarly, we can declare a variable to be of the same type as an entire row of a table or view using the %ROWTYPE attribute. In this case, the fields in the record take their names and data types from the columns in the table or view. For example, to declare a vStaffRec variable to be a row from the Staff table we could write:

vStaffRec Staff%**ROWTYPE**;

Assignments

In the executable part of a PL/SQL block, variables can be assigned in two ways: using the normal assignment statement (:=) or as the result of an SQL SELECT or FETCH statement. For example:

vStaffNo := 'SG14';
vRent := 500;
SELECT COUNT (*) **INTO** x **FROM** PropertyForRent **WHERE** staffNo = vStaffNo;

In the latter case, the variable x is set to the result of the SELECT statement (in this case, equal to the number of properties managed by staff member SG14).

Control statements

PL/SQL supports the usual conditional, iterative, and sequential flow-of-control mechanisms:

- IF–THEN–ELSE–END IF;
- LOOP–EXIT WHEN–END LOOP; FOR–END LOOP; and WHILE–END LOOP;
- GOTO.

We present examples using some of these structures shortly.

Exceptions

An **exception** is an identifier in PL/SQL raised during the execution of a block, which terminates its main body of actions. A block always terminates when an exception is raised although the exception handler can perform some final actions. An exception can be raised automatically by Oracle – for example, the exception NO_DATA_FOUND is raised whenever no rows are retrieved from the database in a SELECT statement. It is also possible for an exception to be raised explicitly using the RAISE statement. To handle raised exceptions, separate routines called **exception handlers** are specified.

As mentioned earlier, a user-defined exception is defined in the declarative part of a PL/SQL block. In the executable part a check is made for the exception condition and, if found, the exception is raised. The exception handler itself is defined at the end of the PL/SQL block. An example of exception handling is given in Figure 8.18. This example also illustrates the use of the Oracle-supplied package DBMS_OUTPUT, which allows

Figure 8.18
Example of exception handling in PL/SQL.

```
DECLARE
   vpCount       NUMBER;
   vStaffNo PropertyForRent.staffNo%TYPE := 'SG14';
-- define an exception for the enterprise constraint that prevents a member of staff
-- managing more than 100 properties
   e_too_many_properties EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_too_many_properties, -20000);
BEGIN
         SELECT COUNT(*) INTO vpCount
         FROM PropertyForRent
         WHERE staffNo = vStaffNo;
         IF vpCount  = 100
-- raise an exception for the enterprise constraint
              RAISE e_too_many_properties;
         END IF;
         UPDATE PropertyForRent SET staffNo = vStaffNo WHERE propertyNo = 'PG4';
EXCEPTION
   -- handle the exception for the enterprise constraint
   WHEN e_too_many_properties THEN
              dbms_output.put_line('Member of staff ' || staffNo || 'already managing 100 properties');
END;
```

output from PL/SQL blocks and subprograms. The procedure put_line outputs information to a buffer in the SGA, which can be displayed by calling the procedure get_line or by setting SERVEROUTPUT ON in SQL*Plus.

Cursors

A SELECT statement can be used if the query returns *one and only one* row. To handle a query that can return an arbitrary number of rows (that is, zero, one, or more rows) PL/SQL uses **cursors** to allow the rows of a query result to be accessed one at a time. In effect, the cursor acts as a pointer to a particular row of the query result. The cursor can be advanced by 1 to access the next row. A cursor must be *declared* and *opened* before it can be used, and it must be *closed* to deactivate it after it is no longer required. Once the cursor has been opened, the rows of the query result can be retrieved one at a time using a FETCH statement, as opposed to a SELECT statement. (In Appendix E we see that SQL can also be embedded in high-level programming languages and cursors are also used for handling queries that can return an arbitrary number of rows.)

Figure 8.19 illustrates the use of a cursor to determine the properties managed by staff member SG14. In this case, the query can return an arbitrary number of rows and so a cursor must be used. The important points to note in this example are:

- In the DECLARE section, the cursor propertyCursor is defined.
- In the statements section, the cursor is first opened. Among others, this has the effect of parsing the SELECT statement specified in the CURSOR declaration, identifying the rows that satisfy the search criteria (called the *active set*), and positioning the pointer just before the first row in the active set. Note, if the query returns no rows, PL/SQL does not raise an exception when the cursor is open.
- The code then loops over each row in the active set and retrieves the current row values into output variables using the FETCH INTO statement. Each FETCH statement also advances the pointer to the next row of the active set.
- The code checks if the cursor did not contain a row (propertyCursor%NOTFOUND) and exits the loop if no row was found (EXIT WHEN). Otherwise, it displays the property details using the DBMS_OUTPUT package and goes round the loop again.
- The cursor is closed on completion of the fetches.
- Finally, the exception block displays any error conditions encountered.

As well as %NOTFOUND, which evaluates to true if the most recent fetch does not return a row, there are some other cursor attributes that are useful:

- %FOUND Evaluates to true if the most recent fetch returns a row (complement of %NOTFOUND).
- %ISOPEN Evaluates to true if the cursor is open.
- %ROWCOUNT Evaluates to the total number of rows returned so far.

Passing parameters to cursors

PL/SQL allows cursors to be parameterized, so that the same cursor definition can be reused with different criteria. For example, we could change the cursor defined in the above example to:

Figure 8.19
Using cursors in PL/SQL to process a multi-row query.

```
DECLARE
        vPropertyNo     PropertyForRent.propertyNo%TYPE;
        vStreet         PropertyForRent.street%TYPE;
        vCity           PropertyForRent.city%TYPE;
        vPostcode       PropertyForRent.postcode%TYPE;
        CURSOR propertyCursor IS
                SELECT propertyNo, street, city, postcode
                FROM PropertyForRent
                WHERE staffNo = 'SG14'
                ORDER by propertyNo;
BEGIN
-- Open the cursor to start of selection, then loop to fetch each row of the result table
      OPEN propertyCursor;
      LOOP
-- Fetch next row of the result table
        FETCH propertyCursor
                INTO vPropertyNo, vStreet, vCity, vPostcode;
        EXIT WHEN  propertyCursor%NOTFOUND;

-- Display data
        dbms_output.put_line('Property number: ' || vPropertyNo);
        dbms_output.put_line('Street:          ' || vStreet);
        dbms_output.put_line('City:            ' || vCity);
        IF postcode IS NOT NULL THEN
                dbms_output.put_line('Post Code:        ' || vPostcode);
        ELSE
                dbms_output.put_line('Post Code:        NULL');
        END IF;
    END LOOP;
    IF propertyCursor%ISOPEN THEN CLOSE propertyCursor END IF;

-- Error condition - print out error
EXCEPTION
  WHEN OTHERS THEN
        dbms_output.put_line('Error detected');
        IF propertyCursor%ISOPEN THEN CLOSE propertyCursor; END IF;
END;
```

```
CURSOR propertyCursor (vStaffNo VARCHAR2) IS
        SELECT propertyNo, street, city, postcode
        FROM PropertyForRent
        WHERE staffNo = vStaffNo
        ORDER BY propertyNo;
```

and we could open the cursor using the following example statements:

```
vStaffNo1 PropertyForRent.staffNo%TYPE := 'SG14';
OPEN propertyCursor('SG14');
OPEN propertyCursor('SA9');
OPEN propertyCursor(vStaffNo1);
```

Updating rows through a cursor

It is possible to update and delete a row after it has been fetched through a cursor. In this case, to ensure that rows are not changed between declaring the cursor, opening it, and fetching the rows in the active set, the FOR UPDATE clause is added to the cursor declaration. This has the effect of locking the rows of the active set to prevent any update conflict when the cursor is opened (locking and update conflicts are discussed in Chapter 20).

For example, we may want to reassign the properties that SG14 manages to SG37. The cursor would now be declared as:

```
CURSOR propertyCursor IS
SELECT propertyNo, street, city, postcode
FROM PropertyForRent
WHERE staffNo = 'SG14'
ORDER BY propertyNo
FOR UPDATE NOWAIT;
```

By default, if the Oracle server cannot acquire the locks on the rows in the active set in a SELECT FOR UPDATE cursor, it waits indefinitely. To prevent this, the optional NOWAIT keyword can be specified and a test can be made to see if the locking has been successful. When looping over the rows in the active set, the WHERE CURRENT OF clause is added to the SQL UPDATE or DELETE statement to indicate that the update is to be applied to the current row of the active set. For example:

```
UPDATE PropertyForRent
SET staffNo = 'SG37'
WHERE CURRENT OF propertyCursor;
. . .
COMMIT;
```

8.2.6 Subprograms, Stored Procedures, Functions, and Packages

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprogram called (**stored**) **procedures** and **functions**. Procedures and functions can take a set of parameters given to them by the calling program and perform a set of actions. Both can modify and return data passed to them as a parameter. The difference between a procedure and a function is that a function will always return a single value to the caller, whereas a procedure does not. Usually, procedures are used unless only one return value is needed.

Procedures and functions are very similar to those found in most high-level programming languages, and have the same advantages: they provide modularity and extensibility,

they promote reusability and maintainability, and they aid abstraction. A parameter has a specified name and data type but can also be designated as:

- IN parameter is used as an input value only.
- OUT parameter is used as an output value only.
- IN OUT parameter is used as both an input and an output value.

For example, we could change the anonymous PL/SQL block given in Figure 8.19 into a procedure by adding the following lines at the start:

```
CREATE OR REPLACE PROCEDURE PropertiesForStaff
  (IN vStaffNo VARCHAR2)
AS . . .
```

The procedure could then be executed in SQL*Plus as:

```
SQL> SET SERVEROUTPUT ON;
SQL> EXECUTE PropertiesForStaff('SG14');
```

Packages

A **package** is a collection of procedures, functions, variables, and SQL statements that are grouped together and stored as a single program unit. A package has two parts: a specification and a body. A package's *specification* declares all public constructs of the package, and the *body* defines all constructs (public and private) of the package, and so implements the specification. In this way, packages provide a form of encapsulation. Oracle performs the following steps when a procedure or package is created:

- It compiles the procedure or package.
- It stores the compiled code in memory.
- It stores the procedure or package in the database.

For the previous example, we could create a package specification as follows:

```
CREATE OR REPLACE PACKAGE StaffPropertiesPackage AS
  procedure PropertiesForStaff(vStaffNo VARCHAR2);
END StaffPropertiesPackage;
```

and we could create the package body (that is, the implementation of the package) as:

```
CREATE OR REPLACE PACKAGE BODY StaffPropertiesPackage
AS
. . .
END StaffPropertiesPackage;
```

To reference the items declared within a package specification, we use the dot notation. For example, we could call the PropertiesForStaff procedure as follows:

```
StaffPropertiesPackage.PropertiesForStaff('SG14');
```

Triggers 8.2.7

A **trigger** defines an action that the database should take when some event occurs in the application. A trigger may be used to enforce some referential integrity constraints, to enforce complex enterprise constraints, or to audit changes to data. The code within a trigger, called the *trigger body*, is made up of a PL/SQL block, Java program, or 'C' callout. Triggers are based on the Event–Condition–Action (ECA) model:

- The *event* (or *events*) that trigger the rule. In Oracle, this is:
  - an INSERT, UPDATE, or DELETE statement on a specified table (or possibly view);
  - a CREATE, ALTER, or DROP statement on any schema object;
  - a database startup or instance shutdown, or a user logon or logoff;
  - a specific error message or any error message.

  It is also possible to specify whether the trigger should fire *before* the event or *after* the event.
- The *condition* that determines whether the action should be executed. The condition is optional but, if specified, the action will be executed only if the condition is true.
- The *action* to be taken. This block contains the SQL statements and code to be executed when a triggering statement is issued and the trigger condition evaluates to true.

There are two types of trigger: *row-level* triggers that execute for each row of the table that is affected by the triggering event, and *statement-level* triggers that execute only once even if multiple rows are affected by the triggering event. Oracle also supports INSTEAD-OF triggers, which provide a transparent way of modifying views that cannot be modified directly through SQL DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD-OF triggers because, unlike other types of trigger, Oracle fires the trigger *instead of* executing the original SQL statement. Triggers can also activate themselves one after the other. This can happen when the trigger action makes a change to the database that has the effect of causing another event that has a trigger associated with it.

For example, *DreamHome* has a rule that prevents a member of staff from managing more than 100 properties at the same time. We could create the trigger shown in Figure 8.20 to enforce this enterprise constraint. This trigger is invoked before a row is inserted into the PropertyForRent table or an existing row is updated. If the member of staff currently manages 100 properties, the system displays a message and aborts the transaction. The following points should be noted:

- The **BEFORE** keyword indicates that the trigger should be executed before an insert or update is applied to the PropertyForRent table.
- The **FOR EACH ROW** keyword indicates that this is a row-level trigger, which executes for each row of the PropertyForRent table that is updated in the statement.
- The **new** keyword is used to refer to the new value of the column. (Although not used in this example, the **old** keyword can be used to refer to the old value of a column.)

Figure 8.20
Trigger to enforce the constraint that a member of staff cannot manage more than 100 properties at any one time.

```
CREATE TRIGGER StaffNotHandlingTooMuch
BEFORE INSERT OR UPDATE ON PropertyForRent
FOR EACH ROW
DECLARE
  vpCount          NUMBER;
BEGIN
        SELECT COUNT(*) INTO vpCount
        FROM PropertyForRent
        WHERE staffNo = :new.staffNo;
        IF vpCount = 100
            raise_application_error(-20000, ('Member' || :new.staffNo || 'already managing 100 properties');
        END IF;
END;
```

Using triggers to enforce referential integrity

We mentioned in Section 8.2.3 that, by default, Oracle enforces the referential actions ON DELETE NO ACTION and ON UPDATE NO ACTION on the named foreign keys. It also allows the additional clause ON DELETE CASCADE to be specified to allow deletions from the parent table to cascade to the child table. However, it does not support the ON UPDATE CASCADE action, or the SET DEFAULT and SET NULL actions. If any of these actions are required, they will have to be implemented as triggers or stored procedures, or within the application code. For example, from Example 6.1 in Chapter 6 the foreign key staffNo in the PropertyForRent table should have the action ON UPDATE CASCADE. This action can be implemented using the triggers shown in Figure 8.21.

Trigger 1 (PropertyForRent_Check_Before)

The trigger in Figure 8.21(a) is *fired* whenever the staffNo column in the PropertyForRent table is updated. The trigger checks *before* the update takes place that the new value specified exists in the Staff table. If an Invalid_Staff exception is raised, the trigger issues an error message and prevents the change from occurring.

Changes to support triggers on the Staff table

The three triggers shown in Figure 8.21(b) are fired whenever the staffNo column in the Staff table is updated. Before the definition of the triggers, a sequence number updateSequence is created along with a public variable updateSeq (which is accessible to the three triggers through the seqPackage package). In addition, the PropertyForRent table is modified to add a column called updateId, which is used to flag whether a row has been updated, to prevent it being updated more than once during the cascade operation.

Trigger 2 (Cascade_StaffNo_Update1)

This (statement-level) trigger fires before the update to the staffNo column in the Staff table to set a new sequence number for the update.

```
-- Before the staffNo column is updated in the PropertyForRent table, fire this trigger
-- to verify that the new foreign key value is present in the Staff table.
CREATE TRIGGER PropertyForRent_Check_Before
        BEFORE UPDATE OF staffNo ON PropertyForRent
        FOR EACH ROW WHEN (new.staffNo IS NOT NULL)
DECLARE
        dummy CHAR(5);
        invalid_staff EXCEPTION;
        valid_staff EXCEPTION;
        mutating_table EXCEPTION;
        PRAGMA EXCEPTION_INIT (mutating_table, -4091);
-- Use cursor to verify parent key value exists.
-- Use FOR UPDATE OF to lock parent key's row so that it cannot be deleted
-- by another transaction until this transaction completes.
CURSOR update_cursor (sn CHAR(5)) IS
        SELECT staffNo FROM Staff
        WHERE staffNo = sn
        FOR UPDATE OF staffNo;
        BEGIN
                OPEN update_cursor (:new.staffNo);
                FETCH update_cursor INTO dummy;
-- Verify parent key. Raise exceptions as appropriate.
                IF update_cursor%NOTFOUND THEN
                        RAISE invalid_staff;
                ELSE
                        RAISE valid_staff;
                END IF;
                CLOSE update_cursor;
        EXCEPTION
         WHEN invalid_staff THEN
                CLOSE update_cursor;
                raise_application_error(-20000, 'Invalid Staff Number' || :new.staffNo);
         WHEN valid_staff THEN
                CLOSE update_cursor;
-- A mutating table is a table that is currently being modified by an INSERT, UPDATE,
-- or DELETE statement, or one that might need to be updated by the effects of a declarative
-- DELETE CASCADE referential integrity constraint.
-- This error would raise an exception, but in this case the exception is OK, so trap it,
-- but don't do anything.
        WHEN mutating_table THEN
                NULL;
        END;
```

before trigger

condition for trigger to fire

row-level trigger

Invalid **staffNo** specified

(a)

Figure 8.21
Oracle triggers to enforce ON UPDATE CASCADE on the foreign key staffNo in the PropertyForRent table when the primary key staffNo is updated in the Staff table: (a) trigger for the PropertyForRent table.

Figure 8.21
(b) Triggers for the Staff table.

```
-- Create a sequence number and a public variable UPDATESEQ.
CREATE SEQUENCE updatesequence INCREMENT BY 1 MAXVALUE 500 CYCLE;
CREATE PACKAGE seqpackage AS
        updateseq NUMBER;
END seqpackage;
CREATE or REPLACE PACKAGE BODY seqpackage AS END seqpackage;

-- Add a new attribute to the PropertyForRent table to flag changed rows.
ALTER TABLE PropertyForRent ADD updateid NUMBER;

-- Before updating the Staff table using this statement trigger, generate a new
-- sequence number and assign it to the public variable UPDATESEQ.
CREATE TRIGGER Cascade_StaffNo_Update1
  BEFORE UPDATE OF staffNo ON Staff
  DECLARE
        dummy NUMBER;
        BEGIN
                SELECT updatesequence.NEXTVAL
                INTO dummy FROM dual;
                seqpackage.updateseq := dummy;
        END;

-- Create a row after-trigger that cascades the update to the PropertyForRent table.
-- Only cascade the update if the child row has not already been updated by the trigger.
CREATE TRIGGER Cascade_StaffNo_Update2
  AFTER UPDATE OF staffNo ON Staff
  FOR EACH ROW
  BEGIN
        UPDATE PropertyForRent SET staffNo = :new.staffNo,
                                   updateid = seqpackage.updateseq
        WHERE staffNo = :old.staffNo AND updateid  IS NULL;
  END;

-- Create a final statement after-trigger to reset the updateid flags
CREATE TRIGGER Cascade_StaffNo_Update3
  AFTER UPDATE OF staffNo ON Staff
  BEGIN
        UPDATE PropertyForRent SET updateid = NULL
        WHERE updateid = seqpackage.updateseq;
  END;
```

Package to hold sequence number

add extra column to **PropertyForRent** table

statement-level before trigger

set new sequence number for update

row-level after trigger

update **PropertyForRent** table and set updated flag for these rows

statement-level after trigger; resets flags for updated rows

(b)

Trigger 3 (Cascade_StaffNo_Update2)

This (row-level) trigger fires to update all rows in the PropertyForRent table that have the old staffNo value (:old.staffNo) to the new value (:new.staffNo), and to flag the row as having been updated.

Trigger 4 (Cascade_StaffNo_Update3)

The final (statement-level) trigger fires after the update to reset the flagged rows back to unflagged.

Oracle Internet Developer Suite 8.2.8

The Oracle Internet Developer Suite is a set of tools to help developers build sophisticated database applications. The suite includes:

- Oracle Forms Developer, a set of tools to develop form-based applications for deployment as traditional two-tier client–server applications or as three-tier browser-based applications.
- Oracle Reports Developer, a set of tools for the rapid development and deployment of sophisticated paper and Web reports.
- Oracle Designer, a graphical tool for Rapid Application Development (RAD) covering the database system development lifecycle from conceptual design, to logical design (schema generation), application code generation, and deployment. Oracle Designer can also reverse engineer existing logical designs into conceptual schemas.
- Oracle JDeveloper, to help develop Java applications. JDeveloper includes a Data Form wizard, a Beans-Express wizard for creating JavaBeans and BeanInfo classes, and a Deployment wizard.
- Oracle9*i*AS Portal, an HTML-based tool for developing Web-enabled applications and content-driven websites.

In this section we consider the first two components of the Oracle Developer Suite. We consider Web-based development in Chapter 29.

Oracle9*i* Forms Developer

Oracle9*i* Forms Developer is a set of tools that help developers create customized database applications. In conjunction with Oracle9*i*AS Forms Services (a component of the Oracle9*i* Application Server), developers can create and deploy Oracle Forms on the Web using Oracle Containers for J2EE (OC4J). The Oracle9*i*AS Forms Services component renders the application presentation as a Java applet, which can be extended using Java components, such as JavaBeans and Pluggable Java Components (PJCs), so that developers can quickly and easily deliver sophisticated interfaces.

Forms are constructed as a collection of individual design elements called *items*. There are many types of items, such as *text boxes* to enter and edit data, *check boxes*, and *buttons* to initiate some user action. A form is divided into a number of sections, of which the main ones are:

- *Canvas* This is the area on which items are placed (akin to the canvas that an artist would use). Properties such as layout and color can be changed using the Layout Editor. There are four types of canvas: a *content canvas* is the visual part of the application and

must exist; a *stacked canvas*, which can be overlayed with other canvases to hide or show parts of some information when other data is being accessed; a *tab canvas*, which has a series of pages, each with a named tab at the top to indicate the nature of the page; a *toolbar*, which appears in all forms and can be customized.

- *Frames* A group of items which can be manipulated and changed as a single item.
- *Data blocks* The control source for the form, such as a table, view, or stored procedure.
- *Windows* A container for all visual objects that make up a Form. Each window must have a least one canvas and each canvas must be assigned to a window.

Like Microsoft Office Access, Oracle Forms applications are event driven. An event may be an *interface event*, such as a user pressing a button, moving between fields, or opening/closing a form, or an *internal processing event* (a system action), such as checking the validity of an item against validation rules. The code that responds to an event is a *trigger*; for example, when the user presses the close button on a form the WHEN-WINDOW-CLOSED trigger is fired. The code written to handle this event may, for example, close down the application or remind the user to save his/her work.

Forms can be created from scratch by the experienced user. However, Oracle also provides a Data Block Wizard and a Layout Wizard that takes the user through a series of interactive pages to determine:

- the table/view or stored procedure that the form is to be based on;
- the columns to be displayed on the form;
- whether to create/delete a master–detail relationship to other data blocks on the form;
- the name for the new data block;
- the canvas the data block is to be placed on;
- the label, width, and height of each item;
- the layout style (Form or Tabular);
- the title for the frame, along with the number of records to be displayed and the distance between records.

Figure 8.22 shows some screens from these wizards and the final form displayed through Forms Services.

Oracle9*i* Reports Developer

Oracle9*i* Reports Developer is a set of tools that enables the rapid development and deployment of sophisticated paper and Web reports against a variety of data sources, including the Oracle9*i* database itself, JDBC, XML, text files, and Oracle9*i* OLAP. Using J2EE technologies such as JSP and XML, reports can be published in a variety of formats, such as HTML, XML, PDF, delimited text, Postscript, PCL, and RTF, to a variety of destinations, such as e-mail, Web browser, Oracle9*i*AS Portal, and the file system. In conjunction with Oracle9*i*AS Reports Services (a component of the Oracle9*i* Application Server), developers can create and deploy Oracle Reports on the Web.

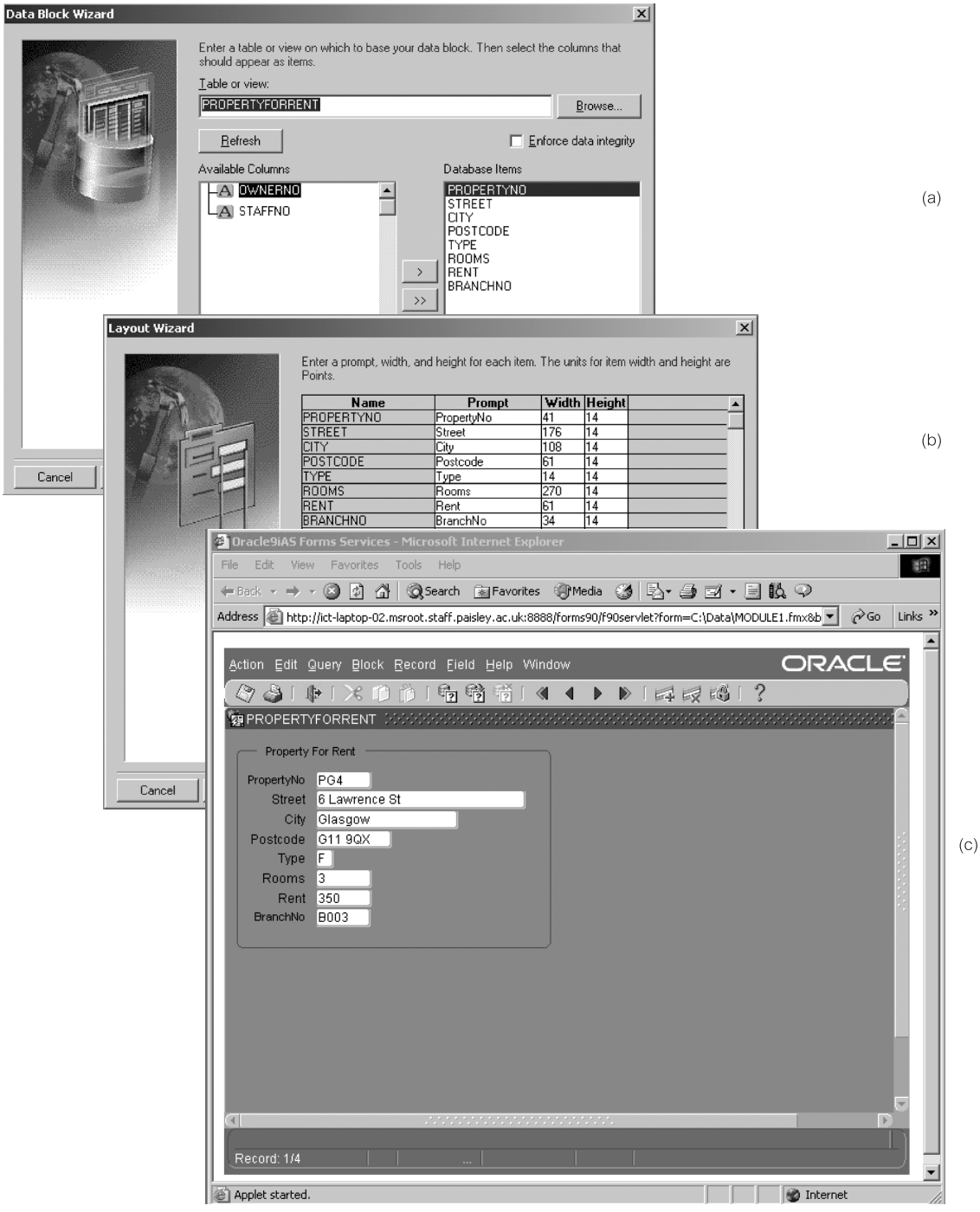


Figure 8.22 Example of a form being created in Oracle Forms Builder: (a) a page from the Data Block Wizard; (b) a page from the Layout Wizard; (c) the final form displayed through Forms Services.

The Oracle9*i* Reports Developer includes:

- wizards that guide the user through the report design process;
- pluggable data sources (PDSs), such as JDBC and XML, that provide access to data from any source for reports;
- a query builder with a graphical representation of the SQL statement to obtain report data;
- default report templates and layout styles that can be customized;
- an editor that allows paper report layouts to be modified in WYSIWYG mode (Paper Design view);
- an integrated graph builder to graphically represent report data;
- the ability to execute dynamic SQL statements within PL/SQL procedures;
- event-based reporting (report execution based on database events);

Reports are constructed as a collection of objects, such as:

- data model objects (queries, groups, database columns, links, user parameters);
- layout objects (frames, repeating frames, fields, boilerplate, anchors);
- parameter form objects (parameters, fields, boilerplate);
- PL/SQL objects (program units, triggers).

Queries provide the data for the report. Queries can select data from any data source, such as an Oracle9*i* database, JDBC, XML, or PDSs. *Groups* are created to organize the columns in the report. Groups can separate a query's data into sets and can also filter a query's data. A *database column* represents a column that is selected by the query containing the data values for a report. For each column selected in the query, the Reports Builder automatically creates a column in the report's data model. Summaries and computations on database column values can be created manually in the Data Model view or by using the Report Wizard (for summary columns). A *data link* (or parent–child relationship) relates the results of multiple queries. A data link causes the child query to be executed once for each instance of its parent group. The child query is executed with the value of the parent's primary key.

Frames surround objects and protect them from being overwritten by other objects. For example, a frame might be used to surround all objects owned by a group, to surround column headings, or to surround summaries. *Repeating frames* surround all the fields that are created for a group's columns. The repeating frame prints once for each record in the group. Repeating frames can enclose any layout object, including other repeating frames. Nested repeating frames are typically used to produce master/detail and break reports. *Fields* are placeholders for parameters, columns, and other data such as the page number or current date. A *boilerplate* object is any text, lines, or graphics that appear in a report every time it is run. A *parameter* is a variable whose value can be set at runtime.

Like Oracle Forms, Oracle Reports Developer allows reports to be created from scratch by the experienced user and it also provides a Data Block Wizard and a Layout Wizard that take the user through a series of interactive pages to determine:

- the report style (for example, tabular, group left, group above, matrix, matrix with group);
- the data source (Express Server Query for OLAP queries, JDBC Query, SQL Query, Text Query, XML Query);
- the data source definition (for example, an SQL query);
- the fields to group on (for a grouped report);
- the fields to be displayed in the report;
- the fields for any aggregated calculations;
- the label, width, and height of each item;
- the template to be used for the report, if any.

Figure 8.23 shows some screens from this wizard and the final form displayed through Reports Services Note that it is also possible to build a report using SQL*Plus. Figure 8.24 illustrates some of the commands that can be used to build a report using SQL*Plus:

- The **COLUMN** command provides a title and format for a column in the report.
- **BREAK**s can be set to group the data, skip lines between attributes, or separate the report into pages. Breaks can be defined on an attribute, expression, alias, or the report itself.
- **COMPUTE** performs a computation on columns or expressions selected from a table. The **BREAK** command must accompany the compute command.

Other Oracle Functionality 8.2.9

We will examine Oracle in more depth in later parts of this book, including:

- Oracle file organizations and indexing in Chapter 17 and Appendix C;
- basic Oracle security features in Chapter 19;
- how Oracle handles concurrency and recovery in Chapter 20;
- how Oracle handles query optimization in Chapter 21;
- Oracle's data distribution mechanism in Chapter 23;
- Oracle's data replication mechanism in Chapter 24;
- Oracle's object-relational features in Chapter 28;
- the Oracle9*i* Application Server in Chapter 29;
- Oracle's support for XML in Chapter 30;
- Oracle's data warehousing functionality in Chapter 32.

Oracle10*g* 8.2.10

At the time of writing, Oracle had just announced the next version of its product, Oracle10*g*. While the '*i*' in Oracle9*i* stands for 'Internet', the '*g*' in the next release stands for 'grid'. The product line targets **grid computing**, which aims to pool together low-cost

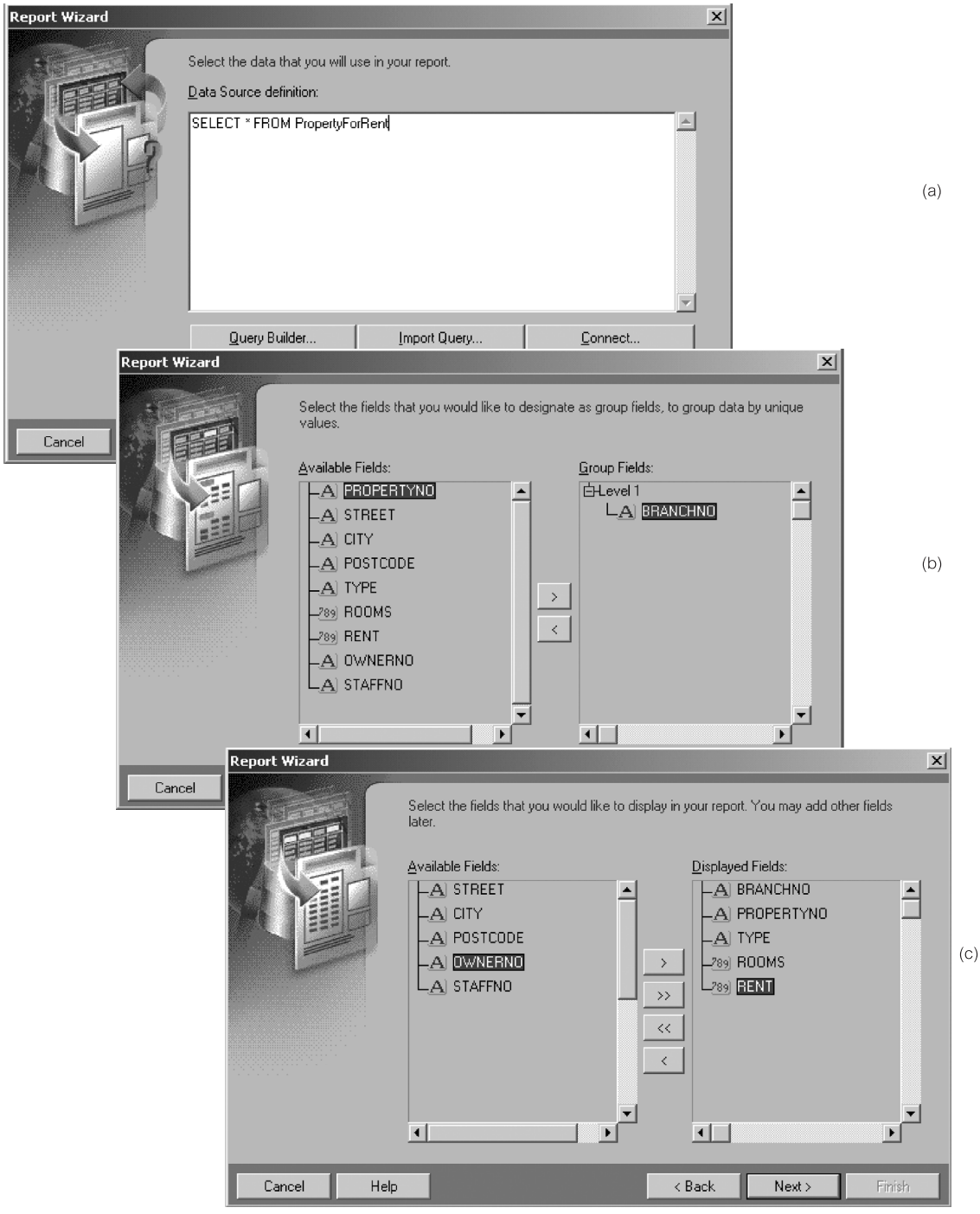
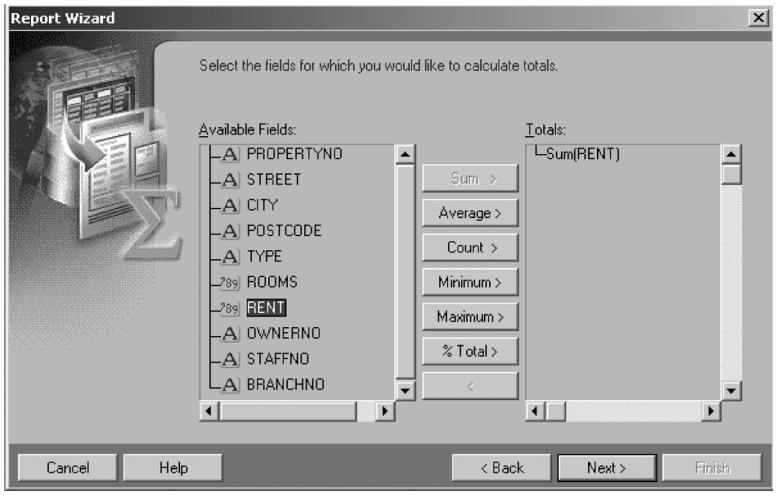
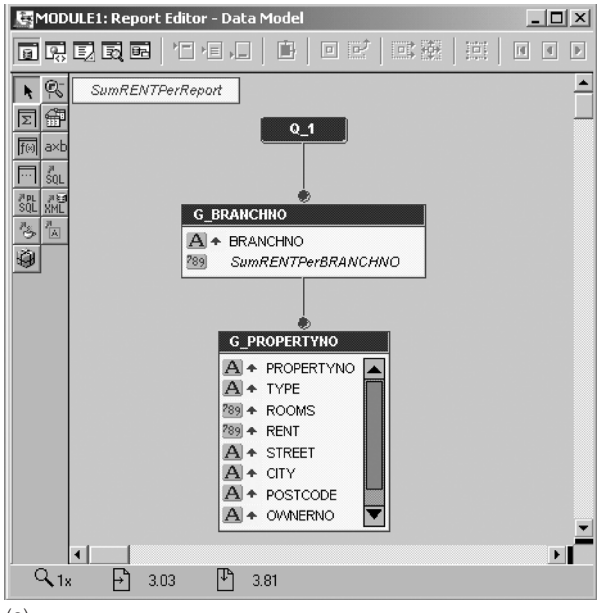


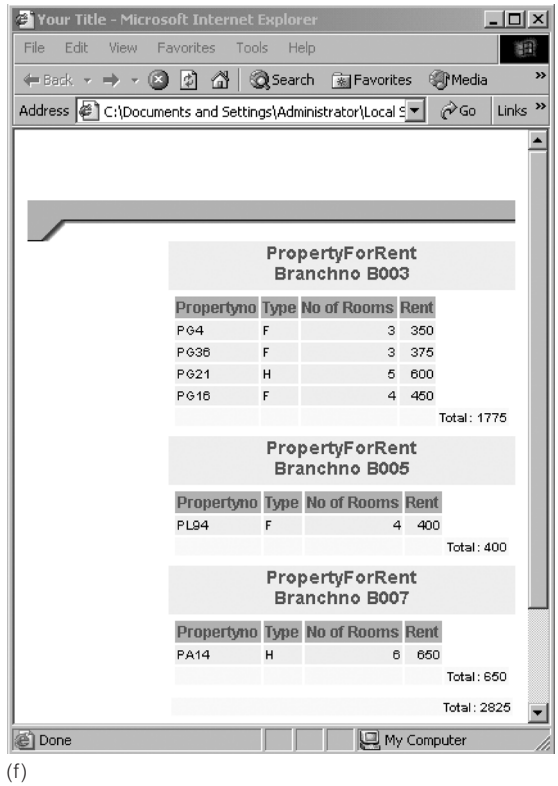
Figure 8.23 Example of a report being created in Oracle Reports Builder: (a)–(d) pages from the Data Block Wizard and Layout Wizard; (e) the data model for the report; (f) the final form displayed through Reports Services.



(d)



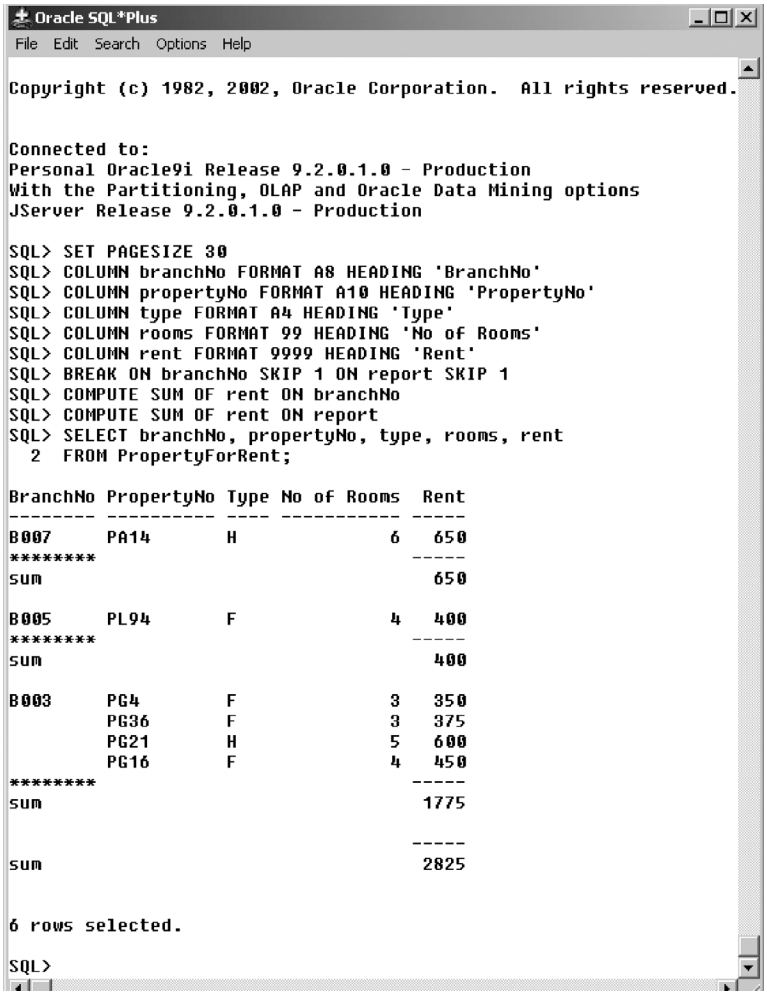
(e)



(f)

Figure 8.23 (*cont'd*)

Figure 8.24
Example of a report being created through SQL*Plus.



modular storage and servers to create a virtual computing resource that the organization has at its disposal. The system transparently distributes workload to use capacity efficiently, at low cost, and with high availability, thus providing computing capacity 'on demand'. In this way, computing is considered to be analogous to a utility, like an electric power grid or telephone network: a client does not care where data is stored within the grid or where the computation is performed; the client is only concerned about getting the necessary data as and when required.

Oracle has announced three grid-enhanced products:

- Oracle Database 10*g*;
- Oracle Application Server 10*g*;
- Oracle Enterprise Manager 10*g* Grid Control.

Oracle Database 10*g*

The database component of the grid architecture is based on the Real Application Clusters feature, which was introduced in Oracle9*i*. Oracle Real Application Clusters enables a single database to run across multiple clustered nodes. New integrated clusterware has been added to simplify the clustering process, allowing the dynamic addition and removal of an Oracle cluster. Automatic storage management (ASM) allows a DBA to define a *disk group* (a set of disk devices) that Oracle manages as a single, logical unit. For example, if a disk group has been defined as the default disk group for a database, Oracle will automatically allocate the necessary storage and create/delete the associated files. Using RAID, ASM can balance I/O from multiple databases across all the devices in the disk group and improve performance and reliability with striping and mirroring (see Section 19.2.6). In addition, ASM can reassign disks from node to node and cluster to cluster.

As well as dynamically allocating work across multiple nodes and data across multiple disks, Oracle can also dynamically move data or share data across multiple databases, potentially on different operating systems, using Oracle Streams. Self-managing features of the database include automatically diagnosing problems such as poor lock contention and slow SQL queries, resolving some problems and alerting the DBA to others with suggested solutions.

Oracle Application Server 10*g* and Oracle Enterprise Manager 10*g* Grid Control

Oracle9*i*AS, an integrated suite of application infrastructure software, and the Enterprise Manager have been enhanced to run enterprise applications on computing grids. Enhancements include:

- streamlined installation and configuration of software across multiple nodes in the grid;
- cloning facilities, to clone servers, their configurations, and the applications deployed on them;
- facilities to automate frequent tasks across multiple servers;
- advanced security including Java2 security support, SSL support for all protocols, and a PKI-based security infrastructure (see Chapter 19);
- a Security Management Console, to create users, roles and to define user identity and access control privileges across the grid (this information is stored in the Oracle Internet Directory, an LDAP-compliant Directory Service that can be integrated with other security environments);
- Oracle Enterprise Single Sign-On Service, to allow users to authenticate to a number of applications and services on the grid;
- a set of tools to monitor and tune the performance of the system; for example, the Dynamic Monitoring Service (DMS) collects resource consumption statistics such as CPU, memory, and I/O usage; Application Performance Monitoring (APM) allows DBAs to track the resource usage of a transaction through the various infrastructure components, such as network, Web servers, application servers, and database servers.

Chapter Summary

- The Relational Database Management System (RDBMS) has become the dominant data-processing software in use today, with estimated new licence sales of between US$6 billion and US$10 billion per year (US$25 billion with tools sales included).
- Microsoft Office Access is the mostly widely used relational DBMS for the Microsoft Windows environment. It is a typical PC-based DBMS capable of storing, sorting, and retrieving data for a variety of applications. Office Access provides a GUI to create tables, queries, forms, and reports, and tools to develop customized database applications using the Microsoft Office Access macro language or the Microsoft Visual Basic for Applications (VBA) language.
- The user interacts with Microsoft Office Access and develops a database and application using tables, queries, forms, reports, data access pages, macros, and modules. A **table** is organized into columns (called *fields*) and rows (called *records*). **Queries** allow the user to view, change, and analyze data in different ways. Queries can also be stored and used as the source of records for forms, reports, and data access pages. **Forms** can be used for a variety of purposes such as to create a data entry form to enter data into a table. **Reports** allow data in the database to be presented in an effective way in a customized printed format. A **data access page** is a special type of Web page designed for viewing and working with data (stored in a Microsoft Office Access database or a Microsoft SQL Server database) from the Internet or an intranet. **Macros** are a set of one or more actions that each performs a particular operation, such as opening a form or printing a report. **Modules** are a collection of VBA declarations and procedures that are stored together as a unit.
- Microsoft Office Access can be used as a standalone system on a single PC or as a multi-user system on a PC network. Since the release of Office Access 2000, there is a choice of two data engines in the product: the original Jet engine and the new Microsoft SQL Server Desktop Engine (MSDE), which is compatible with Microsoft's backoffice SQL Server.
- The Oracle Corporation is the world's leading supplier of software for information management, and the world's second largest independent software company. With annual revenues of about US$10 billion, the company offers its database, tools, and application products, along with related services in more than 145 countries around the world. Oracle is the top-selling multi-user RDBMS with 98% of Fortune 100 companies using Oracle solutions.
- The user interacts with Oracle and develops a database using a number of objects. The main objects in Oracle are **tables** (a table is organized into columns and rows); **objects** (a way to extend Oracle's relational data type system); **clusters** (a set of tables physically stored together as one table that shares a common column); **indexes** (a structure used to help retrieve data more quickly and efficiently); **views** (*virtual tables*); **synonyms** (an alternative name for an object in the database); **sequences** (generates a unique sequence of numbers in cache); **stored functions/procedures** (a set of SQL or PL/SQL statements used together to execute a particular function); **packages** (a collection of procedures, functions, variables, and SQL statements that are grouped together and stored as a single program unit); **triggers** (code stored in the database and invoked – *triggered* – by events that occur in the application).
- Oracle is based on the client–server architecture. The Oracle server consists of the *database* (the raw data, including log and control files) and the *instance* (the processes and system memory on the server that provide access to the database). An instance can connect to only one database. The database consists of a *logical structure*, such as the database schema, and a *physical structure*, containing the files that make up an Oracle database.

Review Questions

8.1 Describe the objects that can be created within Microsoft Office Access.
8.2 Discuss how Office Access can be used in a multi-user environment.
8.3 Describe the main data types in Office Access and when each type would be used.
8.4 Describe two ways to create tables and relationships in Office Access.
8.5 Describe three ways to create enterprise constraints in Office Access.
8.6 Describe the objects that can be created within Oracle.
8.7 Describe Oracle's logical database structure.
8.8 Describe Oracle's physical database structure.
8.9 Describe the main data types in Oracle and when each type would be used.
8.10 Describe two ways to create tables and relationships in Oracle.
8.11 Describe three ways to create enterprise constraints in Oracle.
8.12 Describe the structure of a PL/SQL block.