Part 2

The Relational Model and Languages

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 3</td>
<td>The Relational Model</td>
<td>69</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Relational Algebra and Relational Calculus</td>
<td>88</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>SQL: Data Manipulation</td>
<td>112</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>SQL: Data Definition</td>
<td>157</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Query-By-Example</td>
<td>198</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Commercial RDBMSs: Office Access and Oracle</td>
<td>225</td>
</tr>
</tbody>
</table>
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.
Chapter 3  The Relational Model

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.

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.

**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
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>
<thead>
<tr>
<th><strong>Table 3.1</strong> Alternative terminology for relational model terms.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formal terms</strong></td>
</tr>
<tr>
<td>Relation</td>
</tr>
<tr>
<td>Tuple</td>
</tr>
<tr>
<td>Attribute</td>
</tr>
</tbody>
</table>
Mathematical Relations

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:

\[
\prod_{i=1}^{n} D_i
\]

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:

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

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

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.

<table>
<thead>
<tr>
<th>Superkey</th>
<th>An attribute, or set of attributes, that uniquely identifies a tuple within a relation.</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>Candidate key</th>
<th>A superkey such that no proper subset is a superkey within the relation.</th>
</tr>
</thead>
</table>

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 \textit{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, \textit{branchNo}, we can determine at most one tuple, so that \textit{branchNo} is a candidate key. Similarly, \textit{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 (\textit{clientNo}), a property number (\textit{propertyNo}), a date of viewing (\textit{viewDate}) and, optionally, a comment (\textit{comment}). Given a client number, \textit{clientNo}, there may be several corresponding viewings for different properties. Similarly, given a property number, \textit{propertyNo}, there may be several clients who viewed this property. Therefore, \textit{clientNo} by itself or \textit{propertyNo} by itself cannot be selected as a candidate key. However, the combination of \textit{clientNo} and \textit{propertyNo} identifies at most one tuple, so, for the Viewing relation, \textit{clientNo} and \textit{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.

<table>
<thead>
<tr>
<th><strong>Primary key</strong></th>
<th>The candidate key that is selected to identify tuples uniquely within the relation.</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th><strong>Foreign key</strong></th>
<th>An attribute, or set of attributes, within one relation that matches the candidate key of some (possibly the same) relation.</th>
</tr>
</thead>
</table>

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**

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

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

#### Staff

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

#### PropertyForRent

<table>
<thead>
<tr>
<th>propertyNo</th>
<th>street</th>
<th>city</th>
<th>postcode</th>
<th>type</th>
<th>rooms</th>
<th>rent</th>
<th>ownerNo</th>
<th>staffNo</th>
<th>branchNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA14</td>
<td>16 Holhead</td>
<td>Aberdeen</td>
<td>AB7 5SU</td>
<td>Flat</td>
<td>6</td>
<td>650</td>
<td>CO46</td>
<td>SA9</td>
<td>B007</td>
</tr>
<tr>
<td>PL94</td>
<td>6 Argyll St</td>
<td>London</td>
<td>NW2</td>
<td>Flat</td>
<td>4</td>
<td>400</td>
<td>CO87</td>
<td>SL41</td>
<td>B005</td>
</tr>
<tr>
<td>PG4</td>
<td>6 Lawrence St</td>
<td>Glasgow</td>
<td>G11 9QX</td>
<td>Flat</td>
<td>3</td>
<td>350</td>
<td>CO40</td>
<td>B003</td>
<td></td>
</tr>
<tr>
<td>PG36</td>
<td>2 Manor Rd</td>
<td>Glasgow</td>
<td>G32 4QX</td>
<td>Flat</td>
<td>3</td>
<td>375</td>
<td>CO93</td>
<td>SG37</td>
<td>B003</td>
</tr>
<tr>
<td>PG21</td>
<td>18 Dale Rd</td>
<td>Glasgow</td>
<td>G12</td>
<td>House</td>
<td>5</td>
<td>600</td>
<td>CO87</td>
<td>SG37</td>
<td>B003</td>
</tr>
<tr>
<td>PG16</td>
<td>5 Novar Dr</td>
<td>Glasgow</td>
<td>G129AX</td>
<td>Flat</td>
<td>4</td>
<td>450</td>
<td>CO93</td>
<td>SG14</td>
<td>B003</td>
</tr>
</tbody>
</table>

#### Client

<table>
<thead>
<tr>
<th>clientNo</th>
<th>fName</th>
<th>lName</th>
<th>telNo</th>
<th>prefType</th>
<th>maxRent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>0207-774-5612</td>
<td>Flat</td>
<td>425</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Steve</td>
<td>0141-848-1825</td>
<td>Flat</td>
<td>350</td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ron</td>
<td>01475-392178</td>
<td>House</td>
<td>750</td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregar</td>
<td>01224-196720</td>
<td>Flat</td>
<td>600</td>
</tr>
</tbody>
</table>

#### PrivateOwner

<table>
<thead>
<tr>
<th>ownerNo</th>
<th>fName</th>
<th>lName</th>
<th>address</th>
<th>telNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO46</td>
<td>Joe</td>
<td>Keogh</td>
<td>2 Fergus Dr, Aberdeen AB2 7SX</td>
<td>01224-861212</td>
</tr>
<tr>
<td>CO87</td>
<td>Carol</td>
<td>Farrell</td>
<td>6 Achray St, Glasgow G32 9DX</td>
<td>0141-357-7419</td>
</tr>
<tr>
<td>CO40</td>
<td>Tina</td>
<td>Murphy</td>
<td>63 Well St, Glasgow G42</td>
<td>0141-943-1728</td>
</tr>
<tr>
<td>CO93</td>
<td>Tony</td>
<td>Shaw</td>
<td>12 Park Pl, Glasgow G4 0QR</td>
<td>0141-225-7025</td>
</tr>
</tbody>
</table>

#### Viewing

<table>
<thead>
<tr>
<th>clientNo</th>
<th>propertyNo</th>
<th>viewDate</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR56</td>
<td>PA14</td>
<td>24-May-04</td>
<td>too small</td>
</tr>
<tr>
<td>CR76</td>
<td>PG4</td>
<td>20-Apr-04</td>
<td>too remote</td>
</tr>
<tr>
<td>CR56</td>
<td>PG4</td>
<td>26-May-04</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR62</td>
<td>PA14</td>
<td>14-May-04</td>
<td></td>
</tr>
<tr>
<td>CR56</td>
<td>PG36</td>
<td>28-Apr-04</td>
<td></td>
</tr>
</tbody>
</table>

#### Registration

<table>
<thead>
<tr>
<th>clientNo</th>
<th>branchNo</th>
<th>staffNo</th>
<th>dateJoined</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR76</td>
<td>B005</td>
<td>SL41</td>
<td>2-Jan-04</td>
</tr>
<tr>
<td>CR56</td>
<td>B003</td>
<td>SG37</td>
<td>11-Apr-03</td>
</tr>
<tr>
<td>CR74</td>
<td>B003</td>
<td>SG37</td>
<td>16-Nov-02</td>
</tr>
<tr>
<td>CR62</td>
<td>B007</td>
<td>SA9</td>
<td>7-Mar-03</td>
</tr>
</tbody>
</table>
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

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

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

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

| 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

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**

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.
4.1 The Relational Algebra

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.

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.

Unary Operations

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

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.

\[ \sigma_{\text{salary} > 10000}(\text{Staff}) \]

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 \( \land \) (AND), \( \lor \) (OR) and \( \neg \) (NOT).

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

Figure 4.2 Selecting salary > 10000 from the Staff relation.

Projection

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.

\[ \Pi_{\text{staffNo}, \text{fName}, \text{lName}, \text{salary}}(\text{Staff}) \]

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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>30000</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>12000</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>18000</td>
</tr>
<tr>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>9000</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>24000</td>
</tr>
<tr>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>9000</td>
</tr>
</tbody>
</table>

Figure 4.3 Projecting the Staff relation over the staffNo, fName, lName, and salary attributes.
Chapter 4  Relational Algebra and Relational Calculus

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

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.

\[ \Pi_{\text{city}}(\text{Branch}) \cup \Pi_{\text{city}}(\text{PropertyForRent}) \]

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.

Set difference

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.*

\[ \Pi_{\text{city}}(\text{Branch}) - \Pi_{\text{city}}(\text{PropertyForRent}) \]

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.

Intersection

\[ R \cap S \]

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.*

\[ \Pi_{\text{city}}(\text{Branch}) \cap \Pi_{\text{city}}(\text{PropertyForRent}) \]

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.

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

\[ R \cap S = R - (R - S) \]

**Cartesian product**

\[ R \times S \]

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 \( (I \times J) \) 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:

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

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:

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

The result of this operation is shown in Figure 4.8.

<table>
<thead>
<tr>
<th>client.clientNo</th>
<th>fName</th>
<th>lName</th>
<th>Viewing.clientNo</th>
<th>propertyNo</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>CR56</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>CR56</td>
<td>PG4</td>
<td>too remote</td>
</tr>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>CR56</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>CR62</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>CR56</td>
<td>PG36</td>
<td></td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>CR56</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>CR56</td>
<td>PG4</td>
<td>too remote</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>CR56</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>CR62</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>CR56</td>
<td>PG36</td>
<td></td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ritchie</td>
<td>CR56</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ritchie</td>
<td>CR76</td>
<td>PG4</td>
<td>too remote</td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ritchie</td>
<td>CR36</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ritchie</td>
<td>CR62</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR74</td>
<td>Mike</td>
<td>Ritchie</td>
<td>CR36</td>
<td>PG36</td>
<td></td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>CR56</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>CR76</td>
<td>PG4</td>
<td>too remote</td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>CR56</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>CR62</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>CR56</td>
<td>PG36</td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
\text{TempViewing}(\text{clientNo}, \text{propertyNo}, \text{comment}) \leftarrow \Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing})
\]
\[
\text{TempClient}(\text{clientNo}, \text{fName}, \text{Name}) \leftarrow \Pi_{\text{clientNo}, \text{fName}, \text{Name}}(\text{Client})
\]
\[
\text{Comment}(\text{clientNo}, \text{fName}, \text{Name}, \text{vclientNo}, \text{propertyNo}, \text{comment}) \leftarrow \text{TempClient} \times \text{TempViewing}
\]
\[
\text{Result} \leftarrow O_{\text{clientNo} = \text{vclientNo}}(\text{Comment})
\]

Alternatively, we can use the Rename operation \( \rho \) (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.

\[
\rho_s(E) \text{ or } \rho_s(a_1, a_2, \ldots, a_n)(E)
\]

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 \( \theta \) may be one of the comparison operators (\(<\), \(\leq\), \(>\), \(\geq\), \(=\), \(!=\)).

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:

\[ (\Pi_{clientNo, fName, lName}(Client)) \bowtie_{clientNo} (\Pi_{clientNo, propertyNo, comment}(Viewing)) \]

or

\[ \text{Result } \leftarrow \text{TempClient} \bowtie_{\text{TempClient.clientNo} = \text{TempViewing.clientNo}} \text{TempViewing} \]

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:

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

or

Result ← TempClient \bowtie TempViewing

The result of this operation is shown in Figure 4.9.

<table>
<thead>
<tr>
<th>clientNo</th>
<th>fName</th>
<th>lName</th>
<th>propertyNo</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>PG4</td>
<td>too remote</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PG36</td>
<td></td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
</tbody>
</table>

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.

\[ R \bowtie S \]

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:

\[(\Pi_{\text{propertyNo, street, city}}(\text{PropertyForRent})) \bowtie \text{Viewing}\]

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.

<table>
<thead>
<tr>
<th>propertyNo</th>
<th>street</th>
<th>city</th>
<th>clientNo</th>
<th>viewDate</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA14</td>
<td>16 Holhead</td>
<td>Aberdeen</td>
<td>CR56</td>
<td>26-May-04</td>
<td>too small</td>
</tr>
<tr>
<td>PA14</td>
<td>16 Holhead</td>
<td>Aberdeen</td>
<td>CR62</td>
<td>14-May-04</td>
<td>no dining room</td>
</tr>
<tr>
<td>PL94</td>
<td>6 Argyll St</td>
<td>London</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>PG4</td>
<td>6 Lawrence St</td>
<td>Glasgow</td>
<td>CR50</td>
<td>20-Apr-04</td>
<td>too remote</td>
</tr>
<tr>
<td>PG4</td>
<td>6 Lawrence St</td>
<td>Glasgow</td>
<td>CR56</td>
<td>26-May-04</td>
<td></td>
</tr>
<tr>
<td>PG36</td>
<td>2 Manor Rd</td>
<td>Glasgow</td>
<td>CR56</td>
<td>28-Apr-04</td>
<td></td>
</tr>
<tr>
<td>PG21</td>
<td>18 Dale Rd</td>
<td>Glasgow</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>PG16</td>
<td>5 Novar Dr</td>
<td>Glasgow</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

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

\[R \bowtie_F S\]  
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:

\[R \bowtie_F S = \Pi_A(R \bowtie_F S)\]  
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.

\[
\text{Staff} \bowtie \text{Staff.branchNo = Branch.branchNo.} (\sigma_{\text{city} = \text{"Glasgow"}} (\text{Branch}))
\]

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

Figure 4.11 Semijoin of Staff and Branch relations.

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.

\[
R \div S
\]

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.

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

\[ \exists_{AL}(R) \] Applies the aggregate function list, AL, to the relation R to define a relation over the aggregate list. AL contains one or more \(<\text{aggregate_function},\text{attribute}>\) 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:

\[ \rho_{R}(myCount) \exists_{\text{propertyNo}}(\text{COUNT}_{\text{propertyNo}}(\sigma_{\text{rent} > 350}(\text{PropertyForRent}))) \]

(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:
4.1 The Relational Algebra

The relational algebra includes the grouping operation, denoted \( \rho \), which 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:

\[
a_1, a_2, \ldots, a_n \ \Join \ \{(a_{p_1}, a_{q_1}), \ldots, (a_{p_z}, a_{q_z})\} \ (R)
\]

where \( R \) is any relation, \( a_1, a_2, \ldots, a_n \) are attributes of \( R \), \( a_{p_1}, a_{q_1}, \ldots, a_{p_z}, a_{q_z} \) are other attributes of \( R \), and \( a_{p_1}, a_{q_1}, \ldots, a_{p_z}, a_{q_z} \) 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, \( \text{branchNo} \), and then use the aggregate functions \( \text{COUNT} \) and \( \text{SUM} \) to produce the required relation. The relational algebra expression is as follows:

\[
\rho_{\text{branchNo}}(\text{myCount, mySum}) \ \Join \ \{(\text{COUNT staffNo, SUM salary})\} \ (\text{Staff})
\]

The resulting relation is shown in Figure 4.14.
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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$\sigma_{\text{predicate}}(R)$</td>
<td>Produces a relation that contains only those tuples of $R$ that satisfy the specified $\text{predicate}$.</td>
</tr>
<tr>
<td>Projection</td>
<td>$\Pi_{a_{1}, \ldots, a_{n}}(R)$</td>
<td>Produces a relation that contains a vertical subset of $R$, extracting the values of specified attributes and eliminating duplicates.</td>
</tr>
<tr>
<td>Union</td>
<td>$R \cup S$</td>
<td>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.</td>
</tr>
<tr>
<td>Set difference</td>
<td>$R - S$</td>
<td>Produces a relation that contains all the tuples in $R$ that are not in $S$. $R$ and $S$ must be union-compatible.</td>
</tr>
<tr>
<td>Intersection</td>
<td>$R \cap S$</td>
<td>Produces a relation that contains all the tuples in both $R$ and $S$. $R$ and $S$ must be union-compatible.</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>$R \times S$</td>
<td>Produces a relation that is the concatenation of every tuple of relation $R$ with every tuple of relation $S$.</td>
</tr>
<tr>
<td>Theta join</td>
<td>$R \bowtie_{F} S$</td>
<td>Produces a relation that contains tuples satisfying the predicate $F$ from the Cartesian product of $R$ and $S$.</td>
</tr>
<tr>
<td>Equijoin</td>
<td>$R \bowtie_{F} S$</td>
<td>Produces a relation that contains tuples satisfying the predicate $F$ (which only contains equality comparisons) from the Cartesian product of $R$ and $S$.</td>
</tr>
<tr>
<td>Natural join</td>
<td>$R \bowtie S$</td>
<td>An Equijoin of the two relations $R$ and $S$ over all common attributes $x$. One occurrence of each common attribute is eliminated.</td>
</tr>
<tr>
<td>(Left) Outer join</td>
<td>$R \bowtie_{F} S$</td>
<td>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.</td>
</tr>
<tr>
<td>Semijoin</td>
<td>$R \triangleright_{F} S$</td>
<td>Produces a relation that contains the tuples of $R$ that participate in the join of $R$ with $S$ satisfying the predicate $F$.</td>
</tr>
<tr>
<td>Division</td>
<td>$R + S$</td>
<td>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 $S$.</td>
</tr>
<tr>
<td>Aggregate</td>
<td>$\mathcal{A}_{\text{AL}}(R)$</td>
<td>Applies the aggregate function list, $\text{AL}$, to the relation $R$ to define a relation over the aggregate list. $\text{AL}$ contains one or more $(&lt;\text{aggregate_function}&gt;,$ $&lt;\text{attribute}&gt;)$ pairs.</td>
</tr>
<tr>
<td>Grouping</td>
<td>$<em>{GA}\mathcal{A}</em>{\text{AL}}(R)$</td>
<td>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 $(&lt;\text{aggregate_function}&gt;,$ $&lt;\text{attribute}&gt;)$ pairs. The resulting relation contains the grouping attributes, $GA$, along with the results of each of the aggregate functions.</td>
</tr>
</tbody>
</table>
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:

\( \{ x | P(x) \} \)

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

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:

\( \text{Staff}(S) \)

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

\( \{ S | F(S) \} \)
$F$ 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:

$$\{ S | \text{Staff}(S) \land S\text{.salary} > 10000 \}$$

$S\text{.salary}$ means the value of the salary attribute for the tuple variable $S$. To retrieve a particular attribute, such as salary, we would write:

$$\{ S\text{.salary} | \text{Staff}(S) \land S\text{.salary} > 10000 \}$$

**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:

$$\text{Staff}(S) \land (\exists B) \ (\text{Branch}(B) \land (B\text{.branchNo} = S\text{.branchNo}) \land B\text{.city} = \text{‘London’})$$

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:

$$(\forall B) \ (B\text{.city} \neq \text{‘Paris’})$$

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:

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

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

$$(\exists X)(F_1(X) \land F_2(X)) \equiv \sim(\forall X)(\sim(F_1(X)) \lor \sim(F_2(X)))$$

$$(\forall X)(F_1(X) \land F_2(X)) \equiv \sim(\exists X)(\sim(F_1(X)) \lor \sim(F_2(X)))$$

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

$$\sim(\exists B) \ (B\text{.city} = \text{‘Paris’})$$

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:

$$\{ S\text{.fName, S.lName} | \text{Staff}(S) \land (\exists B) \ (\text{Branch}(B) \land (B\text{.branchNo} = S\text{.branchNo}) \land B\text{.city} = \text{‘London’}) \}$$

$S$ 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:

\[
\{ S_1.a_1, S_2.a_2, \ldots, S_n.a_n \mid F(S_1, S_2, \ldots, S_n) \}
\]

where \( S_1, S_2, \ldots, S_n \) 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 \textit{atoms}, where an atom has one of the following forms:

- \( R(S) \), where \( S \) is a tuple variable and \( R \) is a relation.
- \( S, a, \theta S_a \), where \( S \) and \( S_a \) are tuple variables, \( a \) is an attribute of the relation over which \( S \) ranges, and \( \theta \) is one of the comparison operators (\(<\), \(\leq\), \(>\), \(\geq\), \(=\), \(\neq\)); the attributes \( a \) and \( a_2 \) must have domains whose members can be compared by \( \theta \).
- \( S, a, \theta c \), where \( S \) is a tuple variable, \( a \) is an attribute of the relation over which \( S \) ranges, \( c \) is a constant from the domain of attribute \( a \), and \( \theta \) 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 \( F_1 \land F_2 \), their disjunction \( F_1 \lor F_2 \), and the negation \( \neg F_1 \).
- 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.

\[
\{ S, S.fName, S.lName \mid Staff(S) \land S.position = 'Manager' \land S.salary > 25000 \}
\]

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

\[
\{ S \mid Staff(S) \land (\exists P) (PropertyForRent(P) \land (P.staffNo = S.staffNo) \land P.city = 'Glasgow') \}
\]

The \textit{staffNo} attribute in the \textit{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 \textit{PropertyForRent} for that member of staff with the value of the attribute \textit{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 \textit{PropertyForRent} such that the city is Glasgow and perform their join with the \textit{Staff} relation’, which has an implied order of execution.

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

\[
\{ \text{S.fName, S.lName} \mid \text{Staff(S) \land (\neg (\exists P) (\text{PropertyForRent(P) \land (S.staffNo = P.staffNo))))} \}
\]

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

\[
\{ \text{S.fName, S.lName} \mid \text{Staff(S) \land ((\forall P) (\neg \text{PropertyForRent(P) \lor (\neg (S.staffNo = P.staffNo)))))} \}
\]

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

\[
\{ \text{C.fName, C.lName} \mid \text{Client(C) \land ((\exists V) (\exists P) (\text{Viewing(V) \land \text{PropertyForRent(P) \land (C.clientNo = V.clientNo) \land (V.propertyNo = P.propertyNo) \land P.city = ‘Glasgow’})))} \}
\]

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) \textit{List all cities where there is either a branch office or a property for rent.}

\[
\{ \text{T.city} \mid (\exists B) (\text{Branch(B) \land B.city = ‘T.city’) \lor (\exists P) (\text{PropertyForRent(P) \land P.city = T.city})) \}
\]

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

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

\[
\{ \text{B.city} \mid \text{Branch(B) \land (\neg (\exists P) (\text{PropertyForRent(P) \land B.city = P.city}))} \}
\]

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

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

\[
\{ \text{B.city} \mid \text{Branch(B) \land (\exists P) (\text{PropertyForRent(P) \land B.city = P.city})} \}
\]

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:
4.2 The Relational Calculus

\{ S \mid \sim \text{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 \( \text{dom}(E) \). 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**

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:

\[
\{ d_1, d_2, \ldots, d_m \mid F(d_1, d_2, \ldots, d_m) \}
\]

where \( d_1, d_2, \ldots, d_m \) 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 \( \theta \) is one of the comparison operators \( (\lt, \leq, >, \geq, =, \neq) \); the domains \( d_i \) and \( d_j \) must have members that can be compared by \( \theta \).
- \( d_i \theta c \), where \( d_i \) is a domain variable, \( c \) is a constant from the domain of \( d_i \), and \( \theta \) 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 \( F_1 \land F_2 \), their disjunction \( F_1 \lor F_2 \), 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.

\[
\{ fN, IN | (\exists sN, posn, sex, DOB, sal, bN) (Staff(sN, fN, IN, posn, sex, DOB, sal, bN) \land 
\text{posn} = \text{Manager'} \land \text{sal} > 25000) \}
\]

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 \(Staff(sN, fN, \ldots, bN)\) ensures that the domain variables are restricted to be attributes of the same tuple. Thus, we can use the formula \(posn = \text{Manager'}\), rather than \(Staff\).position = \text{Manager'}\). Also note the difference in the use of the existential quantifier. In the tuple relational calculus, when we write \(\exists posn\) 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 \(Staff(sN, fN, IN, posn, sex, DOB, sal, bN)\) 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.

\[
\{ sN, fN, IN, posn, sex, DOB, sal, bN | (\exists sN1, cty) (Staff(sN, fN, IN, posn, sex, DOB, sal, bN) \land 
\text{Staff}\.position = \text{Manager'} \land (sN = sN1) \land 
\text{cty} = \text{Glasgow'})\}
\]

This query can also be written as:

\[
\{ sN, fN, IN, posn, sex, DOB, sal, bN | (Staff(sN, fN, IN, posn, sex, DOB, sal, bN) \land 
PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1))\}
\]

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.

\[
\{ fN, IN | (\exists sN) (Staff(sN, fN, IN, posn, sex, DOB, sal, bN) \land 
(\neg (\exists sN1) (PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN1, bN1) \land (sN = sN1))))\}
\]

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

\[
\{ fN, IN | (\exists cN, cN1, pN, pN1, cty) (Client(cN, fN, IN, tel, pT, mR) \land 
Viewing(cN1, pN1, dt, cmt) \land PropertyForRent(pN, st, cty, pc, typ, rms, rnt, oN, sN, bN) \land 
(cN = cN1) \land (pN = pN1) \land cty = \text{Glasgow'})\}
\]
(e) List all cities where there is either a branch office or a property for rent.

\[ \{ \text{cty} | (\text{Branch}(bN, \text{st}, \text{cty}, \text{pc}) \lor \text{PropertyForRent}(pN, \text{st1}, \text{cty}, \text{pc1}, \text{typ}, \text{rms}, \text{rnt}, \text{oN}, \text{sN}, \text{bN1})) \} \]

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

\[ \{ \text{cty} | (\text{Branch}(bN, \text{st}, \text{cty}, \text{pc}) \land \neg(\exists \text{cty1}) (\text{PropertyForRent}(pN, \text{st1}, \text{cty1}, \text{pc1}, \text{typ}, \text{rms}, \text{rnt}, \text{oN}, \text{sN}, \text{bN1}) \land (\text{cty} = \text{cty1}))) \} \]

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

\[ \{ \text{cty} | (\text{Branch}(bN, \text{st}, \text{cty}, \text{pc}) \land (\exists \text{cty1}) (\text{PropertyForRent}(pN, \text{st1}, \text{cty1}, \text{pc1}, \text{typ}, \text{rms}, \text{rnt}, \text{oN}, \text{sN}, \text{bN1}) \land (\text{cty} = \text{cty1}))) \} \]

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.

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

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) \{ \text{H.hotelName} | \text{Hotel}(H) \land \text{H.city} = 'London' \}
(b) \{ \text{H.hotelName} | \text{Hotel}(H) \land (\exists R) (\text{Room}(R) \land \text{H.hotelNo} = R.hotelNo \land R.price > 50) \}
(c) \{ \text{H.hotelName} | \text{Hotel}(H) \land (\exists B) (\exists G) (\text{Booking}(B) \land \text{Guest}(G) \land \text{H.hotelNo} = B.hotelNo \land B.guestNo = G.guestNo \land G.guestName = 'John Smith') \}
(d) \{ \text{H.hotelName}, \text{G.guestName}, \text{B1.dateFrom}, \text{B2.dateFrom} | \text{Hotel}(H) \land \text{Guest}(G) \land \text{Booking}(B1) \land \text{Booking}(B2) \land \text{H.hotelNo} = B1.hotelNo \land G.guestNo = B1.guestNo \land B2.hotelNo = B1.hotelNo \land B2.guestNo = B1.guestNo \land 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 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.
5.1 Introduction to SQL

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

Objectives of SQL

Ideally, a database language should allow a user to:

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

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

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

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

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:

```sql
INSERT INTO PropertyForRent
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.

Retrieval 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.

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

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>30000.00</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>12000.00</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>18000.00</td>
</tr>
<tr>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>9000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>24000.00</td>
</tr>
<tr>
<td>SL42</td>
<td>Julie</td>
<td>Lee</td>
<td>9000.00</td>
</tr>
</tbody>
</table>

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.
5.3 Data Manipulation

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.
(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:

```sql
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.*

```sql
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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>30000.00</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>Assistant</td>
<td>12000.00</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>18000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>24000.00</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>street</th>
<th>city</th>
<th>postcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>B005</td>
<td>22 Deer Rd</td>
<td>London</td>
<td>SW1 4EH</td>
</tr>
<tr>
<td>B003</td>
<td>163 Main St</td>
<td>Glasgow</td>
<td>G11 9QX</td>
</tr>
<tr>
<td>B002</td>
<td>56 Clover Dr</td>
<td>London</td>
<td>NW10 6EU</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>30000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>24000.00</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
</tr>
</tbody>
</table>

```sql
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:
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:

\[
\text{(propertyNo} = \text{’PG4’ AND comment = ’ ’)}
\]

or

\[
\text{(propertyNo} = \text{’PG4’ AND comment < > ’too remote’)}
\]

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:

\[
\text{SELECT clientNo, viewDate}
\text{FROM Viewing}
\text{WHERE propertyNo = ’PG4’ AND comment IS NULL;}
\]

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.
Sorting Results (ORDER BY Clause)

In general, the rows of an SQL query result table are not arranged in any particular order (although some DBMSs may use a default ordering based, for example, on a primary key). However, we can ensure the results of a query are sorted using the ORDER BY clause in the SELECT statement. The ORDER BY clause consists of a list of column identifiers that the result is to be sorted on, separated by commas. A column identifier may be either a column name or a column number\(^1\) 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>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>30000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>24000.00</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>18000.00</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>12000.00</td>
</tr>
<tr>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>9000.00</td>
</tr>
<tr>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>9000.00</td>
</tr>
</tbody>
</table>

\(^1\) 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.*

```sql
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>
<thead>
<tr>
<th>propertyNo</th>
<th>type</th>
<th>rooms</th>
<th>rent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL94</td>
<td>Flat</td>
<td>4</td>
<td>400</td>
</tr>
<tr>
<td>PG4</td>
<td>Flat</td>
<td>3</td>
<td>350</td>
</tr>
<tr>
<td>PG36</td>
<td>Flat</td>
<td>3</td>
<td>375</td>
</tr>
<tr>
<td>PG16</td>
<td>Flat</td>
<td>4</td>
<td>450</td>
</tr>
<tr>
<td>PA14</td>
<td>House</td>
<td>6</td>
<td>650</td>
</tr>
<tr>
<td>PG21</td>
<td>House</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>

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:

```sql
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.
Using the SQL Aggregate Functions

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

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

These functions operate on a single column of a table and return a single value. COUNT, MIN, and MAX apply to both numeric and 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:

---

Table 5.12(b)  Result table for Example 5.12 with two sort keys.

<table>
<thead>
<tr>
<th>propertyNo</th>
<th>type</th>
<th>rooms</th>
<th>rent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG16</td>
<td>Flat</td>
<td>4</td>
<td>450</td>
</tr>
<tr>
<td>PL94</td>
<td>Flat</td>
<td>4</td>
<td>400</td>
</tr>
<tr>
<td>PG36</td>
<td>Flat</td>
<td>3</td>
<td>375</td>
</tr>
<tr>
<td>PG4</td>
<td>Flat</td>
<td>3</td>
<td>350</td>
</tr>
<tr>
<td>PA14</td>
<td>House</td>
<td>6</td>
<td>650</td>
</tr>
<tr>
<td>PG21</td>
<td>House</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>
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?*

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

Restricting the query to properties that cost more than £350 per month is achieved using the WHERE clause. The total number of properties satisfying this condition can then be found by applying the aggregate function COUNT. The result table is shown in Table 5.13.

**Table 5.13**

<table>
<thead>
<tr>
<th>myCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

**Example 5.14** Use of COUNT(DISTINCT)

*How many different properties were viewed in May 2004?*

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

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.

**Table 5.14**

<table>
<thead>
<tr>
<th>myCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

**Example 5.15** Use of COUNT and SUM

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

```sql
SELECT COUNT(staffNo) AS myCount, SUM(salary) AS mySum
FROM Staff
WHERE position = ‘Manager’;
```
Table 5.15  Result table for Example 5.15.

<table>
<thead>
<tr>
<th>myCount</th>
<th>mySum</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>54000.00</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>myMin</th>
<th>myMax</th>
<th>myAvg</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000.00</td>
<td>30000.00</td>
<td>17000.00</td>
</tr>
</tbody>
</table>

**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.

```sql
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>
<thead>
<tr>
<th>branchNo</th>
<th>myCount</th>
<th>mySum</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>3</td>
<td>54000.00</td>
</tr>
<tr>
<td>B005</td>
<td>2</td>
<td>39000.00</td>
</tr>
<tr>
<td>B007</td>
<td>1</td>
<td>9000.00</td>
</tr>
</tbody>
</table>

Conceptually, SQL performs the query as follows:

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

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:

```sql
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>
<thead>
<tr>
<th>branchNo</th>
<th>myCount</th>
<th>mySum</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>3</td>
<td>54000.00</td>
</tr>
<tr>
<td>B005</td>
<td>2</td>
<td>39000.00</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>Assistant</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
</tr>
</tbody>
</table>

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.

```sql
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:

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

The result table is shown in Table 5.20.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>salDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>13000.00</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>1000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>7000.00</td>
</tr>
</tbody>
</table>

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:

```sql
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'.

```sql
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>
<thead>
<tr>
<th>propertyNo</th>
<th>street</th>
<th>city</th>
<th>postcode</th>
<th>type</th>
<th>rooms</th>
<th>rent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG16</td>
<td>5 Novar Dr</td>
<td>Glasgow</td>
<td>G12 9AX</td>
<td>Flat</td>
<td>4</td>
<td>450</td>
</tr>
<tr>
<td>PG36</td>
<td>2 Manor Rd</td>
<td>Glasgow</td>
<td>G32 4QX</td>
<td>Flat</td>
<td>3</td>
<td>375</td>
</tr>
<tr>
<td>PG21</td>
<td>18 Dale Rd</td>
<td>Glasgow</td>
<td>G12</td>
<td>House</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>30000.00</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>18000.00</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>24000.00</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>30000.00</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>clientNo</th>
<th>fName</th>
<th>lName</th>
<th>propertyNo</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PG36</td>
<td></td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PA14</td>
<td>too small</td>
</tr>
<tr>
<td>CR56</td>
<td>Aline</td>
<td>Stewart</td>
<td>PG4</td>
<td></td>
</tr>
<tr>
<td>CR62</td>
<td>Mary</td>
<td>Tregear</td>
<td>PA14</td>
<td>no dining room</td>
</tr>
<tr>
<td>CR76</td>
<td>John</td>
<td>Kay</td>
<td>PG4</td>
<td>too remote</td>
</tr>
</tbody>
</table>

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:

```sql
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.*

```sql
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>
<thead>
<tr>
<th>branchNo</th>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>propertyNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>PG16</td>
</tr>
<tr>
<td>B003</td>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>PG21</td>
</tr>
<tr>
<td>B003</td>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>PG36</td>
</tr>
<tr>
<td>B005</td>
<td>ST41</td>
<td>Julie</td>
<td>Lee</td>
<td>PL94</td>
</tr>
<tr>
<td>B007</td>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>PA14</td>
</tr>
</tbody>
</table>

**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.*

```sql
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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>city</th>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>propertyNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>PG16</td>
</tr>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>PG21</td>
</tr>
<tr>
<td>B005</td>
<td>London</td>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>PL94</td>
</tr>
<tr>
<td>B007</td>
<td>Aberdeen</td>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>PA14</td>
</tr>
</tbody>
</table>

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(a).

<table>
<thead>
<tr>
<th>branchNo</th>
<th>staffNo</th>
<th>myCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>SG14</td>
<td>1</td>
</tr>
<tr>
<td>B003</td>
<td>SG37</td>
<td>2</td>
</tr>
<tr>
<td>B005</td>
<td>SL41</td>
<td>1</td>
</tr>
<tr>
<td>B007</td>
<td>SA9</td>
<td>1</td>
</tr>
</tbody>
</table>
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:

```sql
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:

<table>
<thead>
<tr>
<th>Branch1</th>
<th>PropertyForRent1</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchNo</td>
<td>propertyNo</td>
</tr>
<tr>
<td></td>
<td>pCity</td>
</tr>
<tr>
<td>B003</td>
<td>PA14</td>
</tr>
<tr>
<td>B004</td>
<td>PL94</td>
</tr>
<tr>
<td>B002</td>
<td>PG4</td>
</tr>
<tr>
<td></td>
<td>Aberdeen</td>
</tr>
<tr>
<td></td>
<td>London</td>
</tr>
<tr>
<td></td>
<td>Glasgow</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>bCity</th>
<th>propertyNo</th>
<th>pCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>PG4</td>
<td>Glasgow</td>
</tr>
<tr>
<td>B002</td>
<td>London</td>
<td>PL94</td>
<td>London</td>
</tr>
</tbody>
</table>

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 all 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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>bCity</th>
<th>propertyNo</th>
<th>pCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>PG4</td>
<td>Glasgow</td>
</tr>
<tr>
<td>B004</td>
<td>Bristol</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>B002</td>
<td>London</td>
<td>PL94</td>
<td>London</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>bCity</th>
<th>propertyNo</th>
<th>pCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>PA14</td>
<td>Aberdeen</td>
</tr>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>PG4</td>
<td>Glasgow</td>
</tr>
<tr>
<td>B002</td>
<td>London</td>
<td>PL94</td>
<td>London</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>bCity</th>
<th>propertyNo</th>
<th>pCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>PA14</td>
<td>Aberdeen</td>
</tr>
<tr>
<td>B003</td>
<td>Glasgow</td>
<td>PG4</td>
<td>Glasgow</td>
</tr>
<tr>
<td>B004</td>
<td>Bristol</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>B002</td>
<td>London</td>
<td>PL94</td>
<td>London</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
</tr>
<tr>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>Assistant</td>
</tr>
</tbody>
</table>

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:

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

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

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

**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
 FROM Branch
 WHERE city IS NOT NULL)
 UNION
(SELECT city
 FROM PropertyForRent
 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.

**Example 5.33 Use of INTERSECT**

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

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

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

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

or

```sql
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.

```sql
(SELECT city
FROM Branch)
EXCEPT
(SELECT *
FROM Branch)
```

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.

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

```sql
SELECT DISTINCT city
FROM Branch
WHERE city NOT IN (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>
<thead>
<tr>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bristol</td>
</tr>
</tbody>
</table>

**Database Updates**

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

- **INSERT** – adds new rows of data to a table;
- **UPDATE** – modifies existing data in a table;
- **DELETE** – removes rows of data from a table.
Adding data to the database (INSERT)

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

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

*TableName* may be either a base table or an updatable view (see Section 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
```

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)
```
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.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>propCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>1</td>
</tr>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>0</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>2</td>
</tr>
<tr>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>1</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>0</td>
</tr>
<tr>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>1</td>
</tr>
</tbody>
</table>

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.