RDBMS Concepts

1. Introduction

The objective of this chapter is to introduce the main concepts of data storage and retrieval in the context of database information systems.

In view of their prominence this booklet concentrates on the general characteristics of Relational Database Management Systems (RDBMS) and the Structured Query Language SQL and does not consider any of the numerous other types of databases. No prior knowledge of SQL is assumed.

It is intended that the SQL presented in this booklet be followed interactively and that you should try all the given examples in the order in which they are presented. At the end of the booklet you should have attained a thorough knowledge of SQL and its capabilities as an interactive statement language.

In the main the SQL covered complies with the standard definition for SQL were proprietary SQL features are referred to this will be made clear. On this basis the skills obtained from this unit should be transferable across a wide range of RDBMS's which support SQL.

To facilitate interactive, study local RDBMS facilities will be made available to you.

To enable you to make effective use of your local facilities a number of RDBMS specific appendices containing access instructions and other supplementary information specific to your environment have been included.

2. Files, Databases and Database Management Systems

Data and its storage may be considered to be the heart of any information system. Data has to be up-to-date, accurate, accessible in the required form and available to one or perhaps many users at the same time.

For data to be of value it must be presented in a form that supports the various operational, financial, managerial, decision-making, administrative and clerical activities within an organisation.

To meet these objectives data needs to be stored efficiently - to avoid lengthy access times - and with minimal duplication - to avoid lengthy update times and the possibility of inconsistency and inaccuracy. For the data stored by a given organisation to have any value at all its integrity (consistency and accuracy) must always be assured.

In this section we are going to consider what is know as the conceptual view (user view) of stored data. As such we do not need to be concerned with how data is physically stored on specific types of storage media; neither do we need to consider the various storage structures and access methods applicable to retrieval of data from such media.

2.1 Data Storage

The data of an organisation takes the form of an abstract representation of objects and events that occur within the organisation's environment. Within the context of an airline company for example, such objects might include Aircraft, Passengers and Airports, and include events such as Flights and the issuing of Tickets.

There are two principle approaches to the storage of data in a computer-based information system. Data may be stored in separate files belonging to isolated information systems operating within individual departments, or data may be stored in a database that may serve as a resource available across all departments and functional areas. The following sections consider these two approaches and their relative merits.

2.2 File Based Systems

Before we consider the issues arising from file based storage let us firstly, establish some general concepts and definitions relating to the storage of data in files.

2.2.1 Data Files

A file is a complete, named collection of information and the basic unit of storage that enables a computer to distinguish one set of information from another. For example a file named "Aircraft" might contain information about the different types of aircraft used by a particular airline company and a file named "Airport" might hold details of all the airports from which the airline operates.

2.2.2 Records

The data held within a file are organised into structured groups of related elements called records. For example, a record describing an individual aircraft might be composed of the data elements: "*identifying number*", "*name of manufacturer*"; "*description*", "*classification*" (turbo-prop, jet, etc) "*seating capacity*" and so on. The aircraft file then contains zero, one, or many such records; where each record describes an individual aircraft.

2.2.3 Fields

The individual elements of a record are referred to as fields. Hence from the example above, "*identifying number*", "*name of manufacturer*", "*description*", "*classification*" and "*seating capacity*", each represent an individual field (element) of the aircraft record.

2.2.4 Data Types

The data to be held within each field of a given record will possess certain characteristics in terms of size (length measured in characters or digits) and type (numeric, alphabetic, dates, etc). Each field of a record is allocated a particular data type that describes the allowed characteristics of the data to be held by the field and further indicates the range of operations which can be carried out on the field. For example, arithmetic operations would be valid on fields containing numeric data but not on fields containing an address or a narrative description.

2.2.5 Keys

A key is a field or combination of fields used to identify a record. When a key uniquely identifies a record it is referred to as the *primary key*.

Continuing with the example of the Aircraft record, if a given value of the field "*identifying number*" identifies an individual aircraft then it could serve as the primary key. Other fields such as "*manufacturer*" for example, could also serve as alternative keys (secondary keys) by which a *set of records* (eg all aircraft from a particular manufacturer) could be identified.

2.3 Issues Arising From File Based Systems

2.3.1 Data Duplication

The use of individual data files each serving separate information system tends to lead to situations in which an organisation maintains many copies of the same basic information.

For example, consider a sales department which calculates bonuses payable to individual sales personnel on the value of their sales; with actual bonus payments being made only after confirmation that the sales invoices on which bonuses have been calculated have been fully paid.

If the sales department was to operate its bonus scheme based on its own sales and receipts files then the sales department would undoubtedly be holding copies of data such as employee name, payroll number, invoice numbers and amounts received. These data would already be recorded elsewhere in the organisation, perhaps by the personnel department and by the accounts department.

A change in marital status resulting in a change of surname would result in the need to update several files and queries against invoices resulting in debit or credit notes being raised would also require the updating of several files.

2.3.2 Data Inconsistency and Integrity

Where several discrete files exist serving the immediate requirements of individual departments there is a strong likelihood that the common or duplicated data held by these files will get out of step with each other resulting in different versions of data being held by the organisation. If the accounts receivable system in the example above issues a credit note against an incorrectly totalled invoice and fails to advise the sales department of this then the integrity of the sales department's files would be lost and as a result incorrect bonus payments would be made.

2.3.3 File Design

Not withstanding the above consequences of file based information systems, taken from the view point of an individual information system, it is a relatively easy matter to ensure that the required files are designed to "perfectly" suit user/application needs. Taken in isolation, such information systems are capable of presenting information in exactly the form required by their users and also of providing highly efficient usage of storage and rapid retrieval times.

2.4 Databases and Database Management Systems

For a generalised consideration of databases we may continue to apply the data storage concepts (files, records, keys, etc) previously introduced for file based systems. However, databases are not just a collection of files; through specific access controls provided by the Database Management System (DBMS) databases are able to provide a central resource of data that can be shared between users on an organisation wide basis.

2.4.1 Database Design

To satisfy the information needs of users across an organisation the database has to be designed (in terms of "files" and "record layouts") in a generalised manner.

Consider the personnel and sales department's views of personnel records. The sales department is only interested in a specific type of personnel record; those records for sales men and women. The sales department might also only be interested in fields such as payroll number, name and bonus. The personnel department would be interested in all personnel records irrespective of *job description* (a field in the personnel record) and would want record fields in addition to those of interest to the sales department. These might include home-address, marital status, date-of-appointment, department, salary-scale and scale-point, national insurance number and so on.

2.4.2 User Views

It is the role of the DBMS to provide facilities that enable data (from a generalised definition) to be presented in the form required by specific users. So the DBMS should provide the sales department with just those record fields which they require about sales personnel whilst at the same time providing the personnel department with their requirements.

2.4.3 Database Security and Integrity

There are two major consequences arising from the shared usage of data, namely, security and integrity.

Firstly, the DBMS must ensure that users are only allowed access to data that they are authorised to access. In addition, access authorisation must also restrict the type of access; limiting some users to read-only access for certain instances of data for example. So, if only the accounts receivable users are permitted to adjust the value of invoices, by the issuing of credit notes, then the sales department, whilst interested to see when full payment of an invoice has been received, should not be permitted to change invoice records in any way.

Secondly, the DBMS must also ensure that conflicting updates do not occur. In a stock control system, a user updating the database for a customer order, must be given exclusive access to the records of the ordered items so their "*quantity on hand*" fields may be updated. Such exclusive access should prevent anyone from looking at out of date quantity on hand figures and should also prevent two or more users from trying to update the same quantity on hand values at the same time. This *serialisation* of record updating is controlled through the DBMS, that in the case of the example above would issue locks on the records required by each customer order transaction.

2.4.4 Database Performance

Because databases maintain data in a generalised form, converting this generalised form into a series of user views as required, they are generally less efficient in terms of data storage usage and in terms of access times compared with their individual counterpart file based systems.

3. Relational Databases

3.1 The Relational Model

The relational model was developed by E F Codd at the IBM San Jose Research Laboratory in the late 1960s. This work was published in 1970 under the title:

"A Relational Model of Data For Large Shared Data Banks".

In this paper Codd defines the relational model and its capabilities mathematically.

Following this publication a number of research projects were undertaken in the early 1970s with the aim of implementing a relational database management system. The earliest of these projects included, **System R** at IBM, San Jose and **INGRES** at the University of California, Berkeley.

3.1.1 Relational Query Languages

The relational database model as defined by Codd included a number of alternative relational query languages.

The INGRES project developed a query language called Quel that broadly complies with Codd's definition of a *tuple relational calculus* query language. Quel is still a part of the INGRES DBMS available today; although in view of current trends SQL is generally chosen.

The System R project developed a series of query languages; the first of these called SQUARE, was later developed into a more convenient form called SEQUEL. SEQUEL was itself further developed into the form of today's SQL.

In 1986 the American National Standards Institute ANSI published an SQL standard the:

"Systems Application Architecture Database Interface (SAA SQL)".

Query languages have two main components:

Data Manipulation Language (DML), and
Data Definition Language (DDL);

The DML part of the language is used to retrieve, delete and amend instances of data in the database and where the DDL part of the language is used to describe the type of data to be held by the database.

3.2 Relational Database Terminology

In section 2 you were introduced to some general concepts and definitions relating to the storage and retrieval of data. In a Relational Database all data may be viewed in the form of simple *two-dimensional tables* and to distinguish this representation of data from that of other representations we use a separate terminology to describe the data held in a Relational Database.

There are in fact alternative terms used to describe the data in a relational database. The first is taken from the formal definition of the relational model and the second is based on the ability to view data in the form of simple tables. We will adopt the latter terminology for the remainder of this booklet.

Equivalent Terms		
Relational Databases		**Non Database**
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

Where "tuple" is pronounced tup^el (as in couple).

3.3 Database Tables and Relationships

The table definitions for an "Airline Database" are given in detail in Appendix A6 and summarised here.

Table Name	Role
Airport	Records all airports from which the Airline operates.
Route	Records all routes flown by the Airline.
Fares	Contains details of different classes of Fare which are available, eg Standby Single, Business Return, APEX, etc.
Tariff	Details the prices of fares as applicable to each of the operated routes.
Aircraft	Records each type of aircraft used by the Airline.
Flight	Holds details of all timetabled flights, times of departure, destination, and service.
Passenger	Records all passengers who have flown with the airline.
Ticket	Records tickets currently allocated to passengers.
Itinerary	Details for a given ticket, all flights and flight-dates in order.

The data held by these tables do not exist independently. Hence, there are a number of inter-relationships that must be considered. The relationships between the tables in the Airline Database are as follows:

- A Passenger may have more than one Ticket.
- A Ticket comprises an Itinerary of one or more Flights.
- A Flight occurs between two Airports (departs from - arrives at).
- Each Flight has a specific type of Aircraft allocated to it; a given type of Aircraft may be allocated to a number of different flights.
- Each Flight serves a particular Route; a Route being served by several Flights occurring at different times of the day.
- Routes may be domestic or continental, may appeal to different sorts of passengers (business, excursions), etc and therefore each Route has a different range of Fares and Tariffs.

In a relational database, relationships such as these are implemented via so called ***foreign keys.***

Any column of a given table constitutes a *foreign key* if it can contain value that refers to a single row of another table; ie if the given column contains the value of the primary key of another table. In the following example, ***Itinerary. TicketNo*** is a foreign key that supports the one-to-many relationship between "ticket" and "itinerary"; ie a *ticket* comprises an itinerary of one or more flight legs.

TICKET

<u>**TICKETNO**</u>	**TICKETDATE**	**PID**
100001	01-07-95	26
100002	25-08-95	28
100010	09-08-95	29
100011	11-08-95	24
100012	01-09-95	21

ITINERARY(*example data of flight legs on ticket 100012*)

<u>**TICKETNO**</u>	<u>**FLIGHTNO**</u>	<u>**LEGNO**</u>	**FLIGHTDATE**	**FARETYPE**
100012	BD776	1	15-09-92	SDR
100012	BD655	2	15-09-92	APR
100012	BD652	3	20-09-92	APR
100012	BD775	4	20-09-92	SDR

In Appendix A6 the primary key column(s) of each table appear underlined. If you look closely you will notice that these columns also appear in other tables not underlined; where they serve as a foreign key enabling one table to be referenced from another.

The ability to reference one table from another enables us to implement relationships between one table and another.

For example, to find the seating capacity of the aircraft allocated to a particular flight we would take the value of AircraftType for that flight, eg DC9, and look for the single row in the Aircraft table with a key value of DC9. Conversely if we wanted to find all flights which have been allocated with a DC9 aircraft we would list from the Flight table all rows with a value of AircraftType equal to DC9.

The following exercise is designed to help you familiarise yourself with the tables of the Airline Database and the relationships between them.

Desk Exercise

Turn to the database schema and table contents given in Appendix A6 and attempt the following questions:

1) Each airport is identified by a short code. What are the identification codes for Heathrow, Leeds/Bradford and Brussels?

2) The airline provides flights on a number of different Routes; what is the description for RouteNo 9.

3) On any given route the airline offers a number of different types of fare, "Standby Single", "Eurobudget", etc. Find and list the complete range of "Return" fares offered by the Airline.

4) What is the Tariff (Price) on Route 9 of an "Advanced Purchase Return"?

5) What is the seating capacity (NoSeats) of the aircraft allocated to FlightNo BD412?

6) What are the names (AName) of the airports (FromAirport-ToAirport) used on FlightNo BD80?

7) List the Itinerary (FlightNo, LegNo, FlightDate) for Ticket No 100001. What is the Fare for this ticket and how much would the ticket cost?

8) List the names (AName) of the airports that will be visited by passenger R H Miller.

4. SQL Data Manipulation Language (DML)

SQL is a non-procedural language. This means it allows the user to concentrate on specifying what data is required rather than concentrating on the how to get it.

The non-procedural nature of SQL is one of the principle characteristics of all 4GLs - Fourth Generation Languages - and contrasts with 3GLs (eg, C, Pascal, Modula-2, COBOL, etc) in which the user has to give particular attention to how data is to be accessed in terms of storage method, primary/secondary indices, end-of-file conditions, error conditions (e.g. Record NOT Found), and so on.

The DML component of SQL comprises four basic statements:

`SELECT`	to retrieve rows from tables
`UPDATE`	to modify the rows of tables
`DELETE`	to remove rows from tables
`INSERT`	to add new rows to tables.

Sections 4.1 through 4.5 illustrate with examples the data retrieval capabilities of the SELECT statement as well as introducing more general features of SQL such as arithmetic and logical expressions which are equally as applicable to the UPDATE, DELETE and INSERT statements which are dealt with separately in section 4.6.

Each section contains examples that illustrate the various options available with each statement. You are advised to try out these examples for yourself. In addition, at the end of each section there are a number of exercise queries designed to test your understanding of the section.

Before proceeding further you should now refer to the appendices which explain how to access to your local database facilities.

If you have any difficulties at this point ask your tutor for help.

4.1 Selecting Columns

The simplest form of the SELECT statement consists of just two clauses.

```
SELECT column-list
FROM table-list ;
```

4.1.1 Specifying Required Column List

Example 4.1.1 - List full details of all Aircraft

In order to list all the details (columns) recorded for each aircraft type we will list the name of each column in the Aircraft table as follows.

```
SELECT AircraftType, ADescription, NoSeats
FROM Aircraft ;
```

Result:

aircra	Adescription	noseat
DC9	Advanced Turbo Prop	48
ATP	McDonnel Douglas Jet	120
737	Boeing 737-300 Jet	300

For those queries in which all columns of a particular table are to be retrieved the column-list may be replaced by an asterisk. Hence:

```
SELECT *
FROM Aircraft ;
```

gives the same result as before.

Example 4.1.2 - List the Names and Addresses of all Passengers

This query requires the selection of individual columns from the Passenger table and so we simply list the columns required as follows:

```
SELECT Name, Address
FROM Passenger ;
```

Result:

name	address
J Millar	Englewood Cliffs
J D Ullman	1 Microsoft Way
A Smithson	16 Bedford St
D Etheridge	4 Maylands Avenue
E Simon	8 Cherry Street
D N Hamer	1 St Paul's Churchyard
D E Avison	5 Chancery Lane
G B Davis	25 Allenby Road
C Evans	63 Kew Green
A N Smith	81 Digby Crescent
T Pittman	The Little House
J Peters	31 Lucas Road
K E Kendall	11 Rosedale Avenue
R H Miller	155 Kingston Road

4.1.2 Removing Duplicate Rows

Example 4.1.3 - List all Flight No's with tickets issued against them

Flights with issued tickets are recorded in the Itinerary table. The following simple query lists Flight No's from the Itinerary table.

```
SELECT FlightNo
FROM Itinerary ;
```

Result:

flight
BD80
BD95
BD80
BD95
BD82
BD54
BD776
BD655
BD652
BD775
BD772
BD655
BD652
BD412
BD419
BD412
BD419
BD224
BD255
BD256
BD275
BD412
BD582
BD589
BD332
BD51
BD774
BD659
BD658
BD771
BD54

31 rows retrieved

However, a given FlightNo will appear as many times as the number of tickets issued for that flight, therefore the result of this query contains duplicate rows. To remove duplicates from the result of this sort of query SQL provides the DISTINCT function that is used as follows.

```
SELECT DISTINCT FlightNo
FROM Itinerary ;
```

Result:

flight
BD224
BD225
BD256
BD275
BD332
BD412
BD419
BD51
BD54
BD582
BD589
BD652
BD655
BD658
BD659
BD771
BD772
BD774
BD775
BD776
BD80
BD82
BD95

23 rows retrieved

4.1.3 Arithmetic Expressions

SQL allows arithmetic expressions to be included in the SELECT clause. An arithmetic expression consists of a number of column names and values connected by any of the following operators:

+	Add
-	Subtract
*	Multiply
/	Divide

When included in the SELECT clause the results of an expression are displayed as a calculated table column.

Example 4.1.4 - Assuming Tariffs are recorded excluding VAT, calculate and list Tariffs inclusive of VAT

The are a number of different ways of calculating Fare prices inclusive of VAT (17.5%). The following are all equivalent and valid arithmetic expressions for calculating VAT inclusive values within the SELECT clause.

i) Price + Price * .175
ii) Price + Price * 17.5 / 100
iii) Price * 1.175

```
SELECT RouteNo, FareType, Price, Price*1.175
FROM Tariff ;
```

Result:

See next page.

Routen	farety	price	col4
3	BUR	$117.00	$137.48
3	SDR	$158.00	$185.65
3	SDS	$79.00	$92.83
4	SDR	$162.00	$190.35
4	SBS	$49.00	$57.58
6	BUR	$117.00	$137.48
6	SBS	$42.00	$49.35
6	KFS	$53.00	$62.28
7	SDR	$128.00	$150.40
8	SDS	$74.00	$86.95
9	PXR	$153.00	$179.78
9	EUR	$181.00	$212.68
9	APR	$95.00	$111.63
11	KFS	$59.00	$69.33
13	EXR	$121.00	$142.18
14	SDR	$110.00	$129.25
14	SBS	$33.00	$38.78
15	SBS	$33.00	$38.78

To obtain a *column name* for the result column enter the following statement:

```
SELECT RouteNo, FareType, Price, Price*1.175 as vat_price
FROM Tariff ;
```

The result-column name must not have embedded spaces and it does not have quotes around it.

INGRES applies standard operator precedence ie:

Multiply (*) and divide (/) are evaluated before plus (+) and minus (-).

Multiply and divide are equal in precedence and plus and minus are equal in precedence.

Operators of equal precedence are evaluated in order of appearance from left to right. Hence:

6 + 4 / 2 ... evaluates to 8 (6+2) and not 5 (10/2).

The order of evaluation may be controlled by the use of brackets. If in the above example, we had wanted the + operator to be evaluated first then the following use of brackets would make the expression

(6 + 4) / 2 ... evaluates to 5.

4.1.4 Aggregate (Group) Functions

SQL provides a number of special functions called aggregate functions that may be included in the SELECT clause. Each of these functions operates over a named column and returns (calculated from all selected rows) a *single value*.

AVG(*column-name*)	returns the average value
COUNT(*column-name*)	returns the number of non-null values
COUNT(DISTINCT *column-name*)	returns the number of distinct values
COUNT(*)	returns the number of rows
MAX(*column-name*)	returns highest value
MIN(*column-name)*	returns the lowest value
SUM(*column-name*)	calculates the total of values

Example 4.1.5 - List the average aircraft seating capacity

```
SELECT AVG(NoSeats)
FROM Aircraft ;
```

Result:

col1
156.00

Example 4.1.6 - Count the number of Routes flown by the airline

```
SELECT COUNT(*)
FROM Route ;
```

Result:

col1
10

Aggregate functions return a single valued result, ie a calculated column with only one row. As such these functions cannot be included in the SELECT clause with columns that are multi-valued, ie resulting in two or more rows. On this basis **the following query is illegal** and would not be processed by the DBMS.

```
SELECT AircraftType, AVG(NoSeats)
FROM Aircraft ;
```

Once we have extended our consideration of SELECT to accommodate slightly more complicated queries we will see examples of how to use aggregate functions with "group columns" in the SELECT clause.

Exercise 1

Give the SQL required to

1. List the full details of all Airports.

2. List the distinct AircraftTypes assigned to flights.

3. List all Tickets with TicketDate appearing in the left most column.

4. From amongst all of the different aircraft types find the largest seating capacity. *You are not asked to find which particular aircraft type this seating capacity belongs to*.

5. Calculate how many different AircraftTypes are allocated to timetabled flights by querying the flights table.

6. Format the query given in Example 4.1.4 so that the expression **price*1.175** is given a *result-column* name of **vat_price**

4.2 Selecting Rows

In the previous section we were concerned only with listing one or more columns from a table. In this type of query all rows of the queried table are returned. More usually however, we will only be interested in one or a small number of rows from a given table, those that satisfy a particular condition. For example, if we wanted to find the number of seats on a McDonnell Douglas DC9 Jet then only the one row in the Aircraft table that records this type of aircraft would be of interest.

In order to select specific rows from a table we use the WHERE clause that is placed after the FROM as follows.

```
SELECT column-list
FROM table-list
WHERE conditional-expression ;
```

4.2.1 Specifying Selection Conditions

When you use a WHERE clause, you are specifying a condition for SELECT to test when processing each row of the table. Only those rows that test True against the condition are listed in the result.

Example 4.2.1 - Find the Seating Capacity on a DC9

```
SELECT NoSeats
FROM Aircraft
WHERE AircaftType = 'DC9' ;
```

Result:

noseat
120

This illustrates the most commonly used form of the WHERE clause in which the expressions consists of three elements:

i.	a column name	**AircraftType**
ii.	a comparison operator	**=** {equal to}
iii.	a column name, constant value, or list of values	**'DC9'**

Comparison Operators

Any of the following comparison operators may be used:

=	equal to
<> (!=)	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

A number of special comparison operators are also provide:

BETWEEN	Compares a range of values.
IN	Tests against values in a list.
(ANY	Used in conjunction with comparison
(ALL	operators to test against values in a subquery.
LIKE	Compares a character pattern.

Constants

Where a constant contains one or more alphabetic characters it is distinguished from a column-name by placing the constant between apostrophes.

List of Values

See section 4.2.3 Matching a Value in a List.

Example 4.2.2 - List Descriptions of Aircraft with more than 50 seats

```
SELECT ADescription, NoSeats
FROM Aircraft
WHERE NoSeats > 50 ;
```

Result:

adescription	noseat
McDonnel Douglas Jet	120
Boeing 737-300 Jet	300

Expressions formed from these comparison operators are known as logical expressions because they evaluate to one of two possible logic states True or False.

In the above example, if a given row has a value in the Noseats column which is greater than 50 then the expression will evaluate to true, if the value is less than or equal to 50 the expression will then evaluate to false. Only rows that cause the WHERE clause expression to evaluate to true are listed in the result.

4.2.2 Selecting Rows From a Range

The BETWEEN operator provides a convenient way of selecting rows with a column value that is within a specified range.

Example 4.2.3 - List the names and addresses of passengers with Pid's in the range 25 to 35 inclusive

```
SELECT Pid, Name, Address
FROM Passenger
WHERE Pid BETWEEN 25 AND 35 ;
```

Result:

pid	name	address
26	J Millar	Englewood Cliffs
28	J D Ullman	1 Microsoft Way
29	A Smithson	16 Bedford St
30	D Etheridge	4 Maylands Avenue
34	E Simon	8 Cherry Street

If required the NOT operator may be used to negate the result of the BETWEEN comparison. Use of the NOT operator in this case would specify passengers whose Pid's are less than 25 or greater than 35, ie

```
SELECT Pid, Name, Address
FROM Passenger
WHERE Pid NOT BETWEEN 25 AND 35 ;
```

Result:

pid	name	address
10	D N Hamer	1 St Paul's Churchyard
20	D E Avison	5 Chancery Lane
21	G B Davis	25 Allenby Road
24	C Evans	63 Kew Green
90	A N Smith	81 Digby Crescent
91	T Pittman	The Little House
92	J Peters	31 Lucas Road
93	K E Kendall	11 Rosedale Avenue
94	R H Miller	155 Kingston Road

4.2.3 Matching a Value in a List

The IN operator permits the selection of rows with a column value that matches any value from a set of values.

Example 4.2.4 - Find the Tickets (No's) issued on Flight Nos BD54, BD80, BD412, BD582, or BD332

```
SELECT TicketNo, FlightNo
FROM Itinerary
WHERE FlightNo IN ('BD54','BD80','BD412', 'BD582', 'BD332') ;
```

Result:

ticketno	flight
100001	BD80
100002	BD80
100011	BD54
100021	BD412
100022	BD412
100041	BD412
100051	BD582
100052	BD332
100100	BD54

(*See the equivalent boolean expression in Example 4.2.8*)

The NOT operator may be used to negate this result, ie

```
SELECT TicketNo, FlightNo
FROM Itinerary
WHERE FlightNo NOT IN ('BD54','BD80','BD412',
'BD582','BD332' );
```

Result:

ticketno	flight
100001	BD95
100002	BD95
100010	BD82
100012	BD776
100012	BD655
100012	BD652
100012	BD775
100020	BD772
100020	BD655
100020	BD652
100021	BD419
100022	BD419
100030	BD224
100030	BD255
100030	BD256
100030	BD275
100051	BD589
100100	BD51
100100	BD774
100100	BD659
100100	BD658
100100	BD771

4.2.4 Matching a Character Pattern

The comparison operators = and IN match exact values, however there are occasions when an exact value for comparison is not known or where or a partial match only is all that is required.

In these situations the LIKE operator allows rows to be selected that partially match a given pattern of characters.

The LIKE operator recognises two special character symbols

%	that represents any sequence of zero or more characters
_	(underscore) which represents any single character

Example 4.2.5 List the description of all Routes using Birmingham

For this query we are interested in any Route which includes the character constant 'Birmingham' anywhere within its description.

```
SELECT *
FROM Route
WHERE RDescription LIKE '%Birmingham%' ;
```

Result:

routeno	rdescription
9	Birmingham-Brussels
14	Heathrow-Birmingham

Example 4.2.6 - Find the telephone numbers of all passengers with a surname of either 'Miller' or 'Millar'

```
SELECT Name, TelNo
FROM Passenger
WHERE Name LIKE '%Mill_r' ;
```

Result:

name	telno
J Millar	061 343 881
R H Miller	0638 4672

4.2.5 Compound Expressions with Boolean Operators

So far we have only considered the WHERE clause composed of single logical expressions. There are many occasions however when selections need to be based on several conditions. There may be a number of alternative conditions by which rows can be selected or there may be a number of conditions that must be satisfied by all selected rows.

Individual logical expressions may be combined within the WHERE clause through the use of the two Boolean operators:

```
AND
OR
```

Expressions are then combined as follows:

```
WHERE logical-exp AND logical-exp AND ...
                                  logical-exp OR logical-exp ...
```

Example 4.2.7 - List the names and addresses of passengers with Pid's in the range 25 to 35 inclusive

```
SELECT Pid, Name, Address
FROM Passenger
WHERE Pid >= 25 AND Pid <= 35 ;
```

Result:

pid	name	address
26	J Millar	Englewood Cliffs
28	J D Ullman	1 Microsoft Way
29	A Smithson	16 Bedford St
30	D Etheridge	4 Maylands Avenue
34	E Simon	8 Cherry Street

Logical Operator Precedence

Comparison operators (=, <, IN, LIKE, etc) are evaluated first, then the AND operators then the OR operators. As with arithmetic operators brackets may be used to change the order of evaluation. Contents of brackets being evaluated first.

If we take the query from Example 4.2.7 and consider three rows from the Passenger table with values in the Pid column of 36, 24 and 30 respectively, then given the above precedence rules, they would be evaluated as summarised in the table below.

Pid	Pid >=25	Pid <= 35	AND	-- Result --
36	true	false	false	not selected
24	false	true	false	not selected
30	true	true	true	selected

As can be observed, for the AND operator to evaluate to true both of the tested conditions must also be true.

The action of the AND and OR operators are summarised in the following tables in which 1 represents *true* and 0 represents *false*.

AND-Operator	OR-Operator
0 AND 0 = 0	0 OR 0 = 0
1 AND 0 = 0	1 OR 0 = 1
0 AND 1 = 0	0 OR 1 = 1
1 AND 1 = 1	1 OR 1 = 1

Example 4.2.8 - Find the Tickets (No's) issued on Flights BD54, BD80, BD412, BD582, or BD332

In Example 4.2.4 we used the IN operator for this query; this is the equivalent SQL using the OR operator.

```
SELECT TicketNo, FlightNo
FROM Itinerary
WHERE FlightNo  = 'BD54'
          OR FlightNo = 'BD80'
          OR FlightNo = 'BD412'
          OR FlightNo = 'BD582'
          OR FlightNo = 'BD332' ;
```

Result:

ticketno	flight
100001	BD80
100002	BD80
100011	BD54
100021	BD412
100022	BD412
100041	BD412
100051	BD582
100052	BD332
100100	BD54

The WHERE clause may be contain a combination of different types of expressions as illustrated in the following example.

Example 4.2.9 - List the names and addresses of passengers who either have Pid's in the range 25 to 35 inclusive or have a surname of 'Smith'

```
SELECT Pid, Name, Address
FROM Passenger
WHERE Pid >= 25 AND Pid <= 35
OR Name LIKE '%Smith' ;
```

Result

pid	name	address
26	J Millar	Englewood Cliffs
28	J D Ullman	1 Microsoft Way
29	A Smithson	16 Bedford St
30	D Etheridge	4 Maylands Avenue
34	E Simon	8 Cherry Street
90	A N Smith	81 Digby Crescent

4.2.6 Arithmetic Expressions

SQL allows arithmetic expressions to be included as part of the WHERE clause.

Example 4.2.10 - List the Tariffs available on Route No 9 (Birmingham-Brussels) which when VAT (@17.5%) is added cost more than $170.00

```
SELECT FareType, Price, Price*1.175
FROM Tariff
WHERE RouteNo = 9
      AND (Price*1.175) > 170 ;
```

Result:

faretype	price	col3
PXR	$153.00	$179.78
EUR	$181.00	$212.68

Exercise 2

Give the SQL required to:

1. List the FlightNo's on the itinerary for Ticket 100030.

2. Calculate the total ticket price for Ticket 100030 given the Tariff for each flight is $50.00. Assign a suitable title to your *result-column*.

3. List full details of flights where the allocated aircraft is either a DC9 or a 737.

4. List full details of all flights from HROW (Heathrow), EMID (East Midlands) or BIRM (Birmingham) where the allocated aircraft is a DC9.

5. List details of aircraft with seat capacities between 50 and 120.

6. List the FlightNo's and service details of flights from HROW (Heathrow) to BIRM (Birmingham) with departure times after 07:00 and before 15:00.

7. List the names and addresses of all passengers whose names start with an initial of 'A' and have surnames of either 'Smith' or 'Smithson'.

4.3 Ordering the Rows of the Result

The relational database model places no significance on the order in which rows appear in a table. As such the order in which rows will appear in a given query result should be considered to be indeterminate. Where a particular order is a required the ORDER BY clause must be used.

```
SELECT column-list
FROM table-list
WHERE conditional-expression
ORDER BY order-columns ;
```

When used the ORDER BY clause must appear as the last clause in the SELECT statement.

The column or columns on which the result is to be ordered must appear in the column-list of the SELECT clause.

Columns in the ORDER BY clause may be of type numeric, character or date.

4.3.1 Ordering on a Single Column

Example 4.3.1 - List in descending order of FlightNo the Tickets (No's) and FlightNo's for Flights BD54, BD80, BD412, BD582, or BD332

```
SELECT TicketNo, FlightNo, FlightDate
FROM Itinerary
WHERE FlightNo IN ('BD54','BD80','BD412', 'BD582', 'BD332')
ORDER BY FlightDate DESC ;
```

Result:

ticketno	flight	Flightdate
100100	BD54	02/10/94
100052	BD332	09/09/94
100051	BD582	07/09/94
100011	BD54	12/08/94
100022	BD412	07/08/94
100001	BD80	05/08/94
100002	BD80	05/08/94
100021	BD412	02/08/94
100041	BD412	02/08/94

4.3.2 Ordering on Multiple Columns

As well as being able to order the results of a query by a single column we can specify a list of columns on which to order the result. The first column on the list provides the primary order, the second column is the ordered within the first and so on.

Example 4.3.2 - List, in ascending order of FlightDate within TicketNo - TicketNos, FlightNo's and FlightDates for TicketNos in the range 100010 to 100020

```
SELECT TicketNo, FlightNo, FlightDate
FROM Itinerary
WHERE TicketNo BETWEEN 100010 AND 100020
ORDER BY TicketNo, FlightDate ;
```

Result:

ticketno	flight	flightdate
100010	BD82	10/08/94
100011	BD54	12/08/94
100012	BD655	15/08/94
100012	BD776	15/08/94
100012	BD652	20/08/94
100012	BD775	20/08/94
100020	BD655	20/08/94
100020	BD772	20/08/94
100020	BD652	23/08/94

Example 4.3.3 - List in descending order of FlightDate within ascending TicketNo, TicketNo's, FLightNo's and their FlightDates for Ticket Nos in the range 100010 to 100020

```
SELECT TicketNo, FlightNo, FlightDate
FROM Itinerary
WHERE TicketNo BETWEEN 100010 AND 100020
ORDER BY TicketNo, FlightDate DESC ;
```

Result:

ticketno	flight	flightdate
100010	BD82	10/08/94
100011	BD54	12/08/94
100012	BD652	20/08/94
100012	BD775	20/08/94
100012	BD655	15/08/94
100012	BD776	15/08/94
100020	BD652	23/08/94
100020	BD655	20/08/94
100020	BD772	20/08/94

4.3.3 Ordering on Calculated (Result) Columns

In addition to being able to order the result of a query by a given table column it is also possible to order by the values of an expression that appears within the SELECT clause, ie by a calculated column.

Example 4.3.4 - List the Tariffs available on Route No 9 (Birmingham-Brussels) that when VAT (17.5%) is added cost more than $170.00

If we wished to Order the result of this query in descending order of the calculated VAT inclusive column the SQL would appear as follows.

```
SELECT FareType, Price*1.175 As VatPrice
FROM Tariff
WHERE RouteNo = 9
         AND (Price*1.175) > 170
ORDER BY VatPrice DESC;
```

Result:

faretype	Vatprice
EUR	$212.68
PXR	$179.78

Exercise 3

Give the SQL required to:

1. Modify the query given for Example 4.3.1 so that the result is listed in ascending order by FlightNo.

2. List full details of flights from BIRM (Birmingham) to BRUS (Brussels) in descending order by DepTime.

3. List full details of all flights in ascending order of Service within descending order of Deptime.

4.4 Grouping the Rows of the Result

We noted in section 4.1.4 that an aggregate (group) column could not be mixed in the SELECT clause with multi-valued columns. There are however, occasions when it is useful to be able to list several groups within a single query. To enable this the SELECT statement has an optional GROUP BY clause which is included immediately after the WHERE clause (or after the FROM clause if the WHERE clause is not present).

```
SELECT column-list
FROM table-list
WHERE conditional-expression
GROUP BY group-columns ;
```

4.4.1 Using the GROUP BY Clause

Example 4.4.1 - Count the number of flights on the Itineraries of TicketNo's 100010 to100020 inclusive

```
SELECT TicketNo, COUNT(FlightNo)
FROM Itinerary
WHERE TicketNo BETWEEN 100010 AND 100020
GROUP BY TicketNo ;
```

Result:

Ticketno	**col2**
100010	1
100011	1
100012	4
100020	3

4.4.2 Using the HAVING Clause

In the same way that you can select specific rows through the use of the WHERE clause, you can also select specific groups with the HAVING clause.

The HAVING clause compares some property of the group with a constant value. If a group fulfils the condition of the logical expression in the HAVING clause it is included in the result.

```
SELECT column-list
FROM table-list
WHERE conditional-expression
GROUP BY group-columns
HAVING group-conditional-expression ;
```

Example 4.4.2 - List TicketNo's with itineraries of 4 or more flights

```
SELECT TicketNo, COUNT(FlightNo)
FROM Itinerary
GROUP BY TicketNo
HAVING COUNT(FlightNo) >= 4 ;
```

Result:

ticketno	col2
100012	4
100030	4
100100	6

Exercise 4

Give the SQL required to:

1. The SQL given in Example 4.4.2 counts the FlightNo column. However, as there is one row for each flight in a given Ticket's itinerary it is not necessary for the SQL to specify any particular column. Modify the SQL given to reflect this fact.

2. List Routes (RouteNo) with less than 4 timetabled flights.

3. List the most popular FareType based on currently recorded Itineraries. Your query should list each FareType and its number of occurrences.

4.5 Joining Tables

All of the queries considered up to this point have been based on single tables, ie resulting in one or more columns selected from a single table named within the FROM clause.

Where queries are to be based on the columns of two or more tables the required results are obtained by joining these tables together. The joined tables are specified in the FROM clause, and are "linked" to each other by one or more common columns; ie columns containing a range of values that each table has in common. The WHERE clause is then be used to specify the conditions between these common columns on which the required rows will be selected.

4.5.1 Equi-Joins

Example 4.5.1 - Find the seating capacity of the aircraft allocated to flight BD80

The type of aircraft (AircraftType) allocated to a particular flight (FlightNo) is recorded in the Flight table whilst the seating capacity for each type of aircraft is recorded in the Aircraft table.

Inspection of the Aircraft and Flight tables reveals that they are related to each other via the common column, AircraftType. Hence we can join the Aircraft table and the Flight table, selecting rows, where FlightNo in the Flight table is BD80 and AircraftType in the Flight table is equal to AircraftType in Aircraft table.

The following SQL illustrates the required join; known as an equi-join because the comparison operator in the join condition is = (equals).

```
SELECT NoSeats
FROM Aircraft, Flight
WHERE FlightNo = 'BD80'
    AND Aircraft.AircraftType = Flight.AircraftType ;
```

Result:

noseat
300

We are not restricted to joins involving only two tables here is an example of a query that needs to join three tables to obtain the required result.

Example 4.5.2 - List the description and seating capacity of the aircraft allocated to the flights on TicketNo 100010

```
SELECT ADescription, NoSeats
FROM Aircraft, Flight, Itinerary
WHERE TicketNo = 100010
         AND Itinerary.FlightNo = Flight.FlightNo
         AND Flight.AircraftType = Aircraft.AircraftType;
```

Result:

adescription	**noseat**
Boeing 737-300 Jet	300

4.5.2 Table References and Table Aliases (Correlation Names)

When joined tables have a common column-name you must make it clear which table is being referenced by prefixing the column with its table-name. The column NoSeats in the previous examples is not prefixed because this column-name appears only in the Aircraft table and is therefore unambiguous.

The need to use a table-name prefix leads to fairly long column-name references. To avoid this, tables may be allocated an abbreviated name (an alias) within the FROM clause by which the table will be referenced elsewhere within the SELECT statement. The FROM clause has the following general form

```
... FROM table-name alias, ... , table-name alias, ...
```

where the alias appears immediately after the table-name (separated by a space).

Using table aliases the previous example could be rewritten as:-

```
SELECT ADescription, NoSeats
FROM Aircraft A, Flight F, Itinerary I
WHERE TicketNo = 100010
         AND I.FlightNo = F.FlightNo
         AND F.AircraftType = A.AircraftType;
```

Result:

adescription	**noseat**
Boeing 737-300 Jet	300

4.5.3 Non-Equi-Joins

Example 4.5.3 - A passenger requests details of alternative flights departing earlier than his/her currently booked Flight (BD659 18:25 Birmingham->Brussels).

The objective of this query is to find alternative flights that serve the same route as that served by BD659 with the same departure airport but with earlier departure times.

The conditional expression for the join used in this query uses the inequality operator > {greater than} and therefore represents a non-equi-join.

```
SELECT E.FromAirport, E.FlightNo, E.DepTime
FROM Flight F, Flight E
WHERE F.FlightNo = 'BD659'
AND F.FromAirport = E.FromAirport
AND F.RouteNo = E.RouteNo
AND F.DepTime > E.DepTime ;
```

Result:

fromai	flight	deptime
BIRM	BD651	07:30
BIRM	BD655	15:00
BIRM	BD657	17:30

4.5.4 Joining A Table to Itself

Table aliases used with example 4.5.2 to abbreviate table-names also allow you to give a table a double alias thereby allowing you to join a table to itself as though it were two separate tables. This is a very useful facility where a query needs to compare rows within the same table. Example 4.5.3, as well as illustrating a non-equi-join also makes use of a self-join with two aliases of the Flight table. This allows alternative flights to be found which serve the same route as the given (currently booked) flight.

Here is another example of the use of table aliases to enable a self-join.

Example 4.5.4 - To fly from Heathrow (HROW) to Brussels (BRUS) passengers must undertake their journey in two stages. List the FlightNo's, Airport's, Departure and Arrival times for inter-connecting flights from Heathrow to Brussels.

```
SELECT DISTINCT B.FlightNo, B.FromAirport,B.DepTime,  B.ArrTime
FROM Flight A, Flight B
WHERE A.FromAirport = 'HROW'
AND       A.ToAirport   = B.FromAirport
AND       B.ToAirport   = 'BRUS';
```

Result:

flight	fromai	deptime	Arrtime
BD651	BIRM	07:30	09:35
BD655	BIRM	15:00	17:05
BD657	BIRM	17:30	19:35
BD659	BIRM	18:25	20:30

Exercise 5

Give the SQL required to:

1. List the full name of the airport from which flight BD275 departs.

2. List the full description and seating capacity of the aircraft allocated to flight BD582.

3. List the description and seating capacity of all the aircraft allocated to flights on the Itineraries of Tickets issue to 'R H Miller'.

4. List the FareTypes and Descriptions of the Fares available on RouteNo 4 (Heathrow - Edinburgh).

5. Calculate the total ticket price for Ticket number 100010 using Price from the Tariff table.

6. List the full name of the airport from which flight BD257 departs and the full name of the airport at which flight BD257 arrives. *Hint: each flight record refers to two Airport records therefore the Airport table should appear twice in the FROM clause. Use aliases to differentiate the first Airport reference from the second.*

7. List for the flights on all East-Midlands Routes the FlightNo, service provided, aircraft used, and seating capacity. *Hint: use LIKE to obtain RouteNo's and join Route and Flight tables on RouteNo. How are you going to find the seating capacity for each flight*?

4.6 Modifying the Contents of Database Tables

SQL provides three statements with which you can modify the contents of database tables.

UPDATE	modifies data in existing rows of a table.
INSERT	inserts new rows into a table.
DELETE	deletes existing rows from a table.

4.6.1 Updating Rows

The UPDATE statement consists of three clauses

UPDATE	*tablename*
SET	*column-assignment-list*
WHERE	*conditional-expression ;*

where the *column-assignment-list* lists the columns to be updated and the values they are to be set to and takes the general form:

```
column-name = value, column-name = value, ...
```

where value may either be a constant or a column-expression which returns a value of the same type as column-name.

The WHERE clause is optional. When used, the WHERE clause specifies a condition for UPDATE to test when processing each row of the table. Only those rows that test True against the condition are updated with the new values.

Example 4.6.1 - Increase the price of the Standby Single (STS) Tariff on Route 13 (East Midlands-Paris) by $5.00

```
UPDATE Tariff
SET Price = Price + 5
WHERE RouteNo = 13
AND   FareType = 'STS' ;
```

Without the WHERE clause the UPDATE statement updates all rows of the table.

Example 4.6.2 - Increase the price of Tariffs on all Routes by 10 percent

```
UPDATE Tariff
SET Price = Price * 1.1 ;
```

4.6.2 Inserting Rows

The INSERT statement has two distinct variations the first, and simplest, inserts a single row into the named table.

```
INSERT INTO tablename [( column-list )]
         VALUES ( constant-list ) ;
```

Example 4.6.3 - Assume that two new types of aircraft are to be brought into service by the airline-company; the Shorts-360 and Fokker-Friendship F24

The type code (AircraftType) for these aircraft will be 'S60' and 'F24' respectively. The 'S60' has a seating capacity of 36 and the 'F24' has a seating capacity of 48.

```
INSERT INTO Aircraft
VALUES ('S60', 'Shorts-360', 36);

INSERT INTO Aircraft
VALUES ('F24', 'Fokker-Friendship', 48);
```

Note: if the input values match the order and number of columns in the table then *column-list* can be omitted.

4.6.3 Inserting Rows Copied from Another Table

The INSERT statement may also be used in conjunction with a SELECT statement query to copy the rows of one table to another.

The general form of this variation of the INSERT statement is as follows:

```
INSERT INTO tablename [( column-list )]
          SELECT column-list
          FROM table-list
          WHERE conditional-expression ;
```

where the SELECT statement replaces the VALUES clause.

Only the specified columns of those rows selected by the query are inserted into the named table.

The columns of the table being copied-from and those of the table being copied-to must be type compatible.

If the columns of both tables match in type and order then the column-list may be omitted from the INSERT clause.

Example 4.6.4 - Copy Passenger records with Pids in the range 91 - 94 to an archive table APassenger

```
INSERT INTO APassenger
          SELECT *
          FROM Passenger
          WHERE Pid BETWEEN 91 AND 94 ;
```

Note: the APassenger table must exist at the time this statement is executed.

4.6.4 Deleting Rows

The general form of the DELETE statement is

```
DELETE FROM tablename
WHERE conditional-expression
```

The DELETE statement removes those rows from a given table that satisfy the condition specified in the WHERE clause.

Example 4.6.5 - Delete rows from the Passenger table for Pid's 91, 92, 93, and 94

```
DELETE
FROM Passenger
WHERE Pid IN (91, 92, 93, 94) ;
```

Example 4.6.6 - Remove all Archived passengers records from the 'live' Passenger table

This example of DELETE employs a sub-query as a part of the WHERE clause to compile a list of Pid's from the "archived passenger table", APassenger.

Where rows in the Passenger table are found with a Pid value which matches a value in the list they are removed by the DELETE statement. See section 6.1 on sub-queries later on.

```
DELETE
FROM Passenger
WHERE Pid IN
          ( SELECT Pid
            FROM APassenger ) ;
```

The WHERE clause is optional, when omitted all rows of the named table are removed. For example

```
DELETE
FROM Aircraft ;
```

will delete all rows from the aircraft table.

4.6.5 Using Rollback, Savepoint and Commit

Table updates, deletions and insertions are not made permanent until you end your **sql** (**or isql**) session and exit from the DBMS. Whilst your database tables are in this uncommitted state you can see the changes you have made as if they were permanent but other users who may have privileges to access your tables cannot.

At any time during the period that your tables are in an uncommitted state you may reinstate the changed tables to their state prior to those changes by using the **rollback** statement that is simply entered as:

```
rollback ;
```

In addition it is possible to limit how much of your work INGRES will rollback by issuing *savepoints* between the consecutive SQL statements of a given transaction. Savepoints are issued with a *name* (which may begin with a number) that is then used in conjunction with rollback to identify the point in the transaction to rollback to.

A transaction commences on the execution of your first SQL statement and includes all subsequent SQL statements issued up to the point where you issue a **commit** or at the point where you end your INGRES session. So, for example:

```
INSERT INTO Aircraft
VALUES ('S60', 'Shorts-360', 36);
savepoint 1;
INSERT INTO Aircraft
VALUES ('F24', 'Fokker-Friendship', 48);

rollback 1;           /*undoes the last insert only*/
rollback;       /*undoes the whole transaction*/
```

At any time during an SQL session your uncommitted database changes may be committed to the database by issuing the **commit** statement as follows:

```
commit ;
```

Having committed your *pending* database changes all users will then be able to see those changes.

Committed changes cannot be undone.

5. SQL Data Definition Language

5.1 Database Administration

All of the statements considered in previous sections belong to the data manipulation part of the SQL language and allow users to interrogate and change the data of selected database tables.

In this section we will consider two statements which form the data definition part of the INGRES/SQL language, namely:

CREATE	used to create table definitions
DROP	used to remove unwanted tables from the database.

In a live database, ie a database supporting some aspect of an organisation's operation, these statements (and others beside) would be used by the database administrator (DBA) to establish and maintain the definition of the database tables which support the organisations information needs.

The following examples of the CREATE statement are shown with INGRES data types. See Appendix A4 for details of the data types and storage structures supported by INGRES.

Appendix A6 contains specifications for the tables of the Airline Database.

5.1.1 Creating Tables

In it simplest form the SQL (DDL) statement used to create a database table has the following syntax:

```
CREATE TABLE table-name
             ( column-definition-list ) ;
```

Example 5.1.1 - Consider the CREATE statement used to create the Airport table definition for the Airline Database

```
CREATE TABLE Airport
(airport        char(4)     not null,
 aname          varchar(20),
 checkin        varchar(50),
 resvtns        varchar(12),
 flightinfo     varchar(12) );
```

`Table Name` `(Airport)`

The name chosen for a table must be a valid INGRES name as detailed in Appendix A2.3.

`Column Names` `(Airport, AName, ..., FlightInfo)`

The names chosen for the columns of a table must also be valid INGRES names as detailed in Appendix A2.3.

Data Types

Each column must be allocated an appropriate data type - INGRES Data Types Appendix A4.

In addition, key columns, ie columns used to uniquely identify individual rows of a given table, may be specified to be NOT NULL. The DBMS will then ensure that columns specified as NOT NULL always contain a value. Note: zero and space are values.

5.1.2 Copying Tables

It is also possible with the following variation of the CREATE statement to create a new table definition and copy rows into the new table with a single statement.

```
CREATE TABLE table-name ( column-definition-list )
AS  SELECT column-list
          FROM table-list
          WHERE conditional-expression ;
```

Example 5.1.2 - Let us suppose that want to create an exact copy of the Ticket table that we will call Ticket2

```
CREATE TABLE Ticket2 (TicketNo, TicketDate, Pid)
AS  SELECT TicketNo, TicketDate, Pid
          FROM Ticket ;
```

As Ticket2 is inheriting the same number of columns, the same order of columns, and the same column names as Ticket, the column-lists used in this example could have been omitted.

If we had wanted to create Ticket2 containing only a sub-set of the rows from the Ticket table then we would have used the WHERE clause to specify the required selection conditions.

5.1.3 Full INGRES Create Table Syntax

Create a new base table **owned** by the user who issues the statement:

```
create table tablename
    ( columnname format {, columnname format} )
    [with_clause]
```

To create a table and load from another table**:**

```
create table tablename
    [( columnname {, columnname} )] as subselect {union [all]
subselect}
    [with_clause]
```

Where the ***with_clause*** consists of the word **with** followed by a comma-separated list of one or more of the following:

```
structure = hash | heap | isam | btree
/*default is heap */
location = (locationname {, locationname})
/*default is ii_database */
```

```
[no ]journaling
/* default without journaling */
[no ]duplicates
/* default duplicates allowed */
key = (columnlist)
fillfactor = n
/* defaults -   50%hash, 80%isam,80%btree */
minpages = n
/* No of hash primary pages -     default 16 */
maxpages = n
/* No of hash primary pages - default no limit */
leaffill = n
/* %fill - btree leaf index pages -     default 70% */
nonleaffill = n
/* %fill - tree nonleaf indexpages - default 80%*/
compression[ = ([[no]key] [,[no]data])] | nocompression
```

A table can have a maximum of 300 columns and a table row may be a maximum of 2000 bytes wide. Note: nullable columns require an additional byte of storage for the null indicator.

The name and data type of each column in the new table are specified by *columnname* and *format* arguments respectively.

Columnname can be any valid INGRES name - must be unique within the table.

Forma*t* specifies the datatype and length of the column and uses the following syntax:

```
data type [not null [with | not default] | with null]
[not | with system_maintained]
```

(See INGRES data types Appendix A4.)

with null (assumed if [**not null**] omitted)

The column accepts nulls. INGRES inserts a null if no value supplied.

not null with default

The column does not accept nulls and INGRES supplies a default value if no value given.

not null (= not null not default)

The user must supply a value.

with system_maintained

Used in conjunction with a ***logical key*** data types (ie table_key or object_key). INGRES automatically assigns a unique value to the column when a row is appended to the table.

Only compatible with **not null with default.**

System_maintained columns cannot be changed by users or applications.

5.1.4 Removing Unwanted Tables

When a database table becomes redundant it may be dropped from the database as follows:

```
DROP TABLE Table2 ;
```

Exercise 6 (*This exercise must be followed in sequence.*)

Give the appropriate SQL to following:

1) Create the archive table **APassenger** with the same column names and data types as the Passenger table. Reference Appendix A6.2 or use the **help** statement to see the Passenger table definition.

2) Copy rows to the Apassenger table from the Passenger table according to the SQL specification given in Example 4.6.4. Use appropriate SQL to confirm that you have copied the required rows into the APassenger table.

3) Create using a single statement a new table called **People** that is an exact copy of (ie contains the same rows of data) the Passenger table.

4) Issue a COMMIT after creating the People table.

5) Delete those rows (passenger records) in the People table that appear in the Apassenger table. Use appropriate SQL to confirm that you have deleted the required records.

6) Issue a ROLLBACK. Use appropriate SQL to confirm the results.

7) Create using a single statement a new table called **Craft** which is an exact copy of (ie contains the same rows of data) as the Aircraft table.

8) Add the two new aircraft types given in Example 4.6.3 to the **Craft** table. Use appropriate SQL to confirm that the two new aircraft have been inserted correctly.

9) Reduce by 4 the seating capacity of all aircraft recorded in the Craft table. Use appropriate SQL to confirm that you have updated the table correctly.

10) **KEEP** the **Craft** table you will need it later! However, make a note of how you could have removed this table at this point <u>**without**</u> using the **Drop** statement.

Drop the APassenger and People tables created in this exercise.

5.2 VIEWS ***(Virtual Tables and Data Security)***

A View, as we will see, is a definition for a "virtual table" (virtual because there is no permanent allocation of storage space) that is assembled at reference time from selected rows and/or columns of one or more real tables.

A view may be queried in exactly the same way as a real table.

Views are useful for two main reasons:

1) They enable users to see data, from a generalised database design, in the form most convenient for their needs.

2) They may be employed as a security mechanism for restricting user access to specific tables columns and/or rows.

The statement used to create a view has the following general form:

```
CREATE VIEW view-name [( column-list )]
AS         SELECT column-list
           FROM table-list
           WHERE conditional-expression ;
```

You can display the specification of the views you have created using the help statement whose general syntax is as follows:

`help view` *`view_name {,view_name}`*

When a view is no longer required it may be dropped from the database with the DROP statement:

```
DROP VIEW table-name ;
```

5.2.1 Views Designed to Simplify Queries

As we noted with Example 4.5.4, there are no direct flights from Heathrow (HROW) to Brussels (BRUS). To simplify the query required to list the departure times of interconnecting flights we will specify a view called Brussels-Link.

```
CREATE VIEW Brussels_Link
AS         SELECT DISTINCT B.FlightNo,B.FromAirport,
           B.DepTime, B.ArrTime
           FROM Flight A, Flight B
           WHERE A.FromAirport = 'HROW'
           AND      A.ToAirport   = B.FromAirport
           AND      B.ToAirport   = 'BRUS' ;
```

Example 5.2.1 - List the FlightNo's, Airport's, Departure and Arrival times for flights from 1500 that link Heathrow with Brussels

```
SELECT FromAirport, FlightNo, DepTime, ArrTime
FROM Brussels_Link
WHERE DepTime >= '15:00';
```

Result:

fromai	flightn	deptime	arrtime
BIRM	BD655	15:00	17:05
BIRM	BD657	17:30	19:35
BIRM	BD659	18:25	20:30

5.2.2 Views Providing Database Security

There are many instances where access to private data within a database needs to be restricted to specific users.

It is possible to create such access restrictions using views. In this section we will consider just one example of how access may be controlled by creating a view which consists of different rows depending on the user who is querying the view.

Example 5.2.2 - Let us suppose that we want to allow all passengers to view their itineraries from a visual display at the airport by logging on to the Airline's DBMS under their Passenger ID as held in the Passenger table (the Pid column)

The DBMS provides a pseudo-column **user** that holds the name (login) of the person currently querying the database. The pseudo-column **user** may then be used in the WHERE clause like any other table column. The following view will automatically return the itinerary belonging to the enquiring passenger.

```
CREATE VIEW PassengerItinerary
AS SELECT I.LegNo, I.TicketNo, I.FlightNo, I.FlightDate
    FROM        Passenger P, Itinerary I
    WHERE       P.Pid       = user
    AND         P.TicketNo  = I.TicketNo
    ORDER BY    TicketNo, LegNo ;
```

This view would need to be accessible to all, so we would need to assign read-only privileges to the view. We would do this with the **grant** statement as follows:

```
grant select on PassengerItinerary to public;
```

The following SQL would then list only the itinerary belonging to the enquiring passenger.

```
SELECT * FROM PassengerItinerary ;
```

Exercise 7

Give the SQL required to:

1. Passengers travelling from Heathrow to Paris must pick up a link flight from East Midlands. Create a View of the interconnecting flights between HROW (Heathrow) and PARI (Paris).

2. List, based on an appropriate join with the view created in (1), the possible arrival times at Paris based upon a departure from Heathrow on flight BD224.

3. Remove the View created in (1) from the database.

6. Further Data Manipulation Language Techniques

This section returns to the SELECT statement and considers some slightly more complex queries that demonstrate the flexibility and strength of SQL as a data manipulation language.

6.1 Sub-Queries

Example 6.1.1 - List the description of the aircraft with the largest seating capacity

This appears to be a very straightforward and simple query. But it requires a query that will need to compare every aircraft row with all other rows of the aircraft table to find the one row for the aircraft with the largest seating capacity.

The SQL needed to find the largest seating capacity (the largest value of NoSeats) is simply:

```
SELECT MAX(NoSeats)
FROM Aircraft;
```

Result:

col1
300

But to which AircraftType does this seating capacity belong? We need to list the AircraftType that corresponds to this value of NoSeats, ie

```
SELECT AircraftType
FROM Aircraft
WHERE NoSeats = "largest seating capacity"
```

SQL allows us to specify a sub-query within the WHERE clause which is executed **before** the main outer query to return one or more rows which may then be compared with the rows returned by the outer query. Putting the above queries together uses the following SQL.

```
SELECT AircraftType
FROM Aircraft
WHERE NoSeats = (     SELECT MAX(NoSeats)
                      FROM Aircraft );
```

Result:

aircra
737

Note: the sub-query appears is in brackets indicating that it will return its result before the execution of the outer query.

This query could have been alternatively expressed as:

```
SELECT AircraftType
FROM Aircraft
WHERE NoSeats >= ALL    ( SELECT NoSeats
                            FROM Aircraft );
```

Example 6.1.2 - List all aircraft that are not allocated to any timetabled Flights

For this example we will consider the SQL components required for each part of the query. Firstly, we need to obtain a list of aircraft types from the Flights tables; those aircraft which are allocated to one or more timetabled flights, ie:

```
SELECT DISTINCT AircraftType
FROM Flights;
```

Result:

aircra
ATP
DC9
737

We require the AircraftTypes recorded in the Aircraft table that do not appear in this list, ie:

Note: craft table created earlier will be used for this example.

```
SELECT AircraftType
FROM Craft
WHERE AircraftType NOT IN ("the list of allocated aircraft")
```

Putting these two queries together as before gives:

```
SELECT AircraftType
FROM Craft
WHERE AircraftType NOT IN
          ( SELECT DISTINCT AircraftType
            FROM Flights );
```

Result:

```
    aircra
    ------
    S60
    F24
```

Note equivalencies between the **IN** and **ANY**/**ALL** operators when used to compare against a subquery generated list:

= ANY (subquery)	≡	**IN (subquery)**
!= ALL (subquery)	≡	**NOT IN (subquery)**

Exercise 8

1. By use of the IN operator and a sub-query list the names and addresses of passengers with tickets for flight BD80.

2. List the names and addresses of passengers with Standby Single (SBS) or Standard Return (SDR) tickets for flight BD54.

3. Provide an alternative to the SQL used in Example 6.1.2.

4. Find the AircraftType with the <u>smallest</u> seating capacity:
   a) using an appropriate aggregate function;
   b) without using an aggregate function.

6.2 Correlated Subqueries

In the previous examples of subqueries each subquery was seen to be executed once; returning its result for use by the main (outer) query.

In correlated sub-queries the sub-query executes once for each row returned by the outer query.

Example 6.2.1 - List the names and addresses of passengers with tickets made up of itineraries with only one flight

```
SELECT Name, Address
FROM Passenger P
WHERE 1 =  ( SELECT COUNT(*)
               FROM Ticket T, Itinerary I
               WHERE P.Pid = T.Pid AND
               T.TicketNo = I.TicketNo );
```

Result:

name	**address**
A Smithson	16 Bedford St
C Evans	63 Kew Green
T Pittman	The Little House
K E Kendall	11 Rosedale Avenue

For each row in the passenger table the value of Pid for that row is passed into the subquery. The subquery executes once for each value of Pid passed from the outer query and returns the value of COUNT (*the number of rows in the ticket-itinerary join, ie flight legs, which satisfy the subquery WHERE condition, ie flights for this Pid*) to the WHERE clause in the outer query. Where COUNT equals one for the current Pid its corresponding row in the passenger table is listed in the result.

6.2.1 Testing for Existence

The EXISTS operator enables us to test whether or not a subquery has returned any rows.

EXISTS evaluates to true if one or more rows have been selected and evaluates to false if zero rows have been selected.

NOT EXISTS evaluates to true if zero rows have been selected and evaluates to false if one or more rows have been selected.

Example 6.2.2 - List the names and addresses of passengers who purchased their Tickets on 01-AUG-92

```
SELECT Name, Address
FROM Passenger P
WHERE           EXISTS (   SELECT *
                     FROM Ticket T
                     WHERE P.Pid = T.Pid AND
                     T.TicketDate      = '01/08/94' );
```

Result:

name	address
J Millar	Englewood Cliffs
T Pittman	The Little House

6.3 Set Operators

We have seen how a query may be composed of subqueries whose results are returned for evaluation within a WHERE clause.

We can also combine the results of pairs of queries and/or subqueries by using the following Set Operators.

`MINUS`	returns query difference; ie all rows in the result of the first query that do not appear in the second query.
`INTERSECTION`	returns common rows; ie only those rows that appear in the results of both the first query and the second query.
`UNION`	combines query results; ie returns the distinct rows returned by either the first query or the second query.

It is important to note that queries to be operated on by these operators must be union compatible, ie the SELECT clauses of both queries must contain the same number and type of columns.

Note: MINUS and INTERSECTION are included here for completeness they are NOT available in INGRES.

6.3.1 Query Difference

The MINUS operator takes as its operands, the results of two separate queries and returns (as its result) the rows from the first query which do not appear in the results of second query.

Example 6.3.1 - Find names of the airports from which direct flights may be taken to <u>all</u> of the following destinations:

BELF (Belfast)
BIRM (Birmingham)
EDIN (Edinburgh)
EMID (East Midlands)
LBDR (Leeds/Bradford)
LVPL (Liverpool)
TEES (Teeside)

```
SELECT AName
FROM Airport A
WHERE NOT EXISTS
                ( ( SELECT DISTINCT F.ToAirport
                    FROM Flight F
                    WHERE F.ToAirport IN ('BELF','BIRM','EDIN',
                      'EMID','LBDR','LVPL','TEES') )
               MINUS
                      ( SELECT DISTINCT T.ToAirport
                  FROM Flight T
                  WHERE A.Airport = T.FromAirport ) );
```

This query would return **Heathrow** as the only airport.

The first subquery returns all flight destinations (airports) that exist in the Flights table corresponding with the given list.

The second subquery (correlated) takes each airport in turn from the Airport table (outer query) as a departure point (FromAirport) and generates a list of all possible direct destinations (ToAirport).

The MINUS operator compares the set of available destination airports with the set of direct destinations possible from each departure point (Airport). If there are no airports in the first query that are not in the second, then we have found a departure point from which we can reach all of the available destinations directly.

If there are zero rows returned, ie the list of available destinations not reached is zero, then the NOT EXISTS evaluates to true and the current ANAME from the Airport table is listed in the result.

6.3.2 Common Rows

The INTERSECT operator allows us to find the common rows from the results of two separate queries (or subqueries). The following example shows the intersection of two queries.

Example 6.3.2 - Find the Names and Addresses of passengers flying from HROW (Heathrow) to BIRM (Birmingham) and from BIRM to HROW. *Not necessarily on the same ticket.*

```
SELECT Name, Address
FROM Passenger P, Ticket T, Itinerary I, Flight F
WHERE P.Pid           = T.Pid     AND
         T.TicketNo  = I.TicketNo      AND
         I.FlightNo  = F.FlightNo      AND
         F.FromAirport     = 'HROW'    AND
         F.ToAirport   = 'BIRM'
INTERSECT
SELECT Name, Address
FROM Passenger P, Ticket T, Itinerary I, Flight F
WHERE P.Pid           = T.Pid     AND
         T.TicketNo  = I.TicketNo      AND
         I.FlightNo  = F.FlightNo      AND
         F.FromAirport     = 'BIRM'    AND
         F.ToAirport = 'HROW' ;
```

Result would be:

```
NAME            ADDRESS
-----------     ---------------------------
G B Davis       25 Allenby Road
R H Miller      155 Kingston Road
```

The first query returns the names and addresses of passengers flying from Heathrow to Birmingham.

The second query returns the names and addresses of passengers flying from Birmingham to Heathrow.

The INTERSECT operator returns the rows from the first query which also appear in the rows from the second query, ie those passengers flying from Heathrow to Birmingham and from Birmingham to Heathrow.

6.3.3 Combining Results

The **UNION** operator allows us to combine distinct rows from the results of two separate queries (or subqueries). The following example shows the union of two queries.

Example 6.3.3 - Find the Names and Addresses of passengers flying from HROW (Heathrow) to BIRM (Birmingham) and from BIRM to HROW or both

```
SELECT Name, Address
FROM Passenger P, Ticket T, Itinerary I, Flight F
WHERE P.Pid         = T.Pid      AND
         T.TicketNo    = I.TicketNo AND
         I.FlightNo    = F.FlightNo AND
         F.FromAirport = 'HROW'     AND
         F.ToAirport   = 'BIRM'
UNION
SELECT Name, Address
FROM Passenger P, Ticket T, Itinerary I, Flight F
WHERE P.Pid           = T.Pid     AND
         T.TicketNo  = I.TicketNo      AND
         I.FlightNo  = F.FlightNo      AND
         F.FromAirport     = 'BIRM'    AND
         F.ToAirport = 'HROW' ;
```

Result:

Name	address
D Etheridge	4 Maylands Avenue
G B Davis	25 Allenby Road
R H Miller	155 Kingston Road

The first query returns the names and addresses of passengers flying from Heathrow to Birmingham.

The second query returns the names and addresses of passengers flying from Birmingham to Heathrow.

The UNION operator returns the rows from the first query plus the rows from the second query and removes any resulting duplicates. We see listed those passengers whom are either flying from Heathrow to Birmingham, or whom are flying from Birmingham to Heathrow, or both.

Exercise 9

Give the SQL required to:

1. List the names and addresses of any passenger on a single ticket with more than 5 flights.

2. Find the names of those passengers who are taking <u>all</u> of the flights that J Millar is taking.

3. Rewrite the SQL statement given for Example 6.3.2 so that it compares flights for the same ticket and find those passenger's with *return* tickets HROW to BIRM and back. Note: you will need to use correlated sub-queries and an alternative query strategy to find the intersection.

4. Using correlated sub-queries find the Names of those passengers with a ticket to fly from HROW (Heathrow) to PARI (Paris) without a return flight to HROW; ie those passengers with *non-return* tickets.

7. About CHL...

CHL provides courses, products and services in e-commerce, client/server applications, databases, networks and programming.

Founded in 1994 CHL is an association of experienced Information Systems professionals and educators.

Courses

Our short public and bespoke courses are designed for management, users and practitioners and include:

- E-commerce
- Internet basics
- Business searching on the Internet
- Microsoft Office 2000 applications
- HTML
- Web site design
- Project Management
- Systems Analysis and Design
- Programming

Custom Solutions

We provide custom programming services in various languages and environments including:

- Visual Basic
- Microsoft Office integration using VBA
- ASP
- JavaScript