SQL Course

The Simple Select Statement

- Demonstrate the use of the simple **SELECT** statement.
- Highlight the capabilities of the **SELECT** and **FROM** clauses.
- Illustrate the simple **SELECT** statement with a few examples.

In many ways, queries are at the heart of the SQL language. The **SELECT** statement, which is used to express SQL queries, is the most powerful and complex of the SQL statements. Despite the many options afforded by the **SELECT** statement, it's possible to start simply and work up to more complex queries.
This module discusses the syntax of the **SELECT** statement and its capabilities before entering into more detail in later modules.
The simple select statement consists solely of using the two main clauses:
SELECT and **FROM**.
In the simple case covered in this module we will only ever be querying from one table. It is possible to select from more than one table and this will be covered in more detail in later modules.

The **SELECT** statement in its simplest form is as follows :

> **SELECT** [**ALL**|**DISTINCT**] *{select_list | *}* **FROM** *table_name*;

The **SELECT** clause specifies the data items to be retrieved by the query. These items are usually specified by a select list. This is a list of select items separated by a comma. Each item in the list generates a single column of query results, in left to right order. Each item can be one of the following :
Column name
This must identify a column from the table in the **FROM** clause. In such situations SQL simply takes the value of that column from each row of the database table and places it in the corresponding row of query results.
Constant
This means the same constant value will appear in every row of the query results.
SQL expression
Indicating that SQL must calculate the value to be placed into the query results, in the style specified by the expression.

When selecting a column name we can use its full (qualified) name which has the form :

table_name.column_name

Columns can be renamed for display purposes, for example :

> **SELECT** *customer_name "Customer Name"* .....

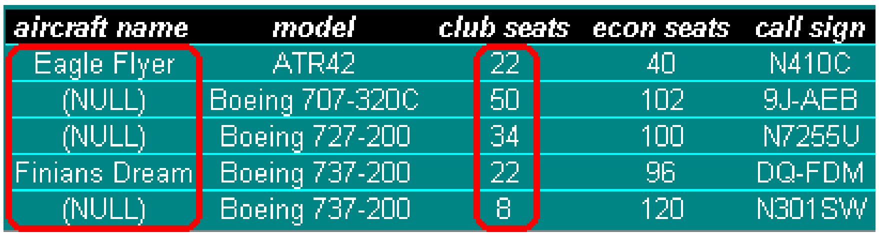
This will print the customer names as before but this time the column heading will be *Customer Name* instead of *customer_name*. As in this case, double quotes are only needed when the new column name contains a space or other special characters.

The **FROM** clause consists of the keyword **FROM**, followed by a list of table specifications separated by commas. Each table specification identifies a table containing data to be

retrieved by the query. This module will only cover examples where only one table is ever specified in the **FROM** clause. Multi table queries are covered in later modules.
Sometimes it's convenient to display all the contents of all the columns of a table. This can be particularly useful when you encounter a new database and you want to get familiar with its structure and data. As a convenience, SQL lets you use an asterix (*) as an abbreviation for "all columns". The ANSI/ISO standard specifies that a **SELECT** statement can either have an all column selection or a column list, but not both. However many implementation treat the * as just another element of the select list.
If a query result contains the primary key of a table in its column list then every row of the query will be unique. If the primary key is not included in the query results, duplicate rows can occur. You can eliminate duplicate rows of query results by inserting the keyword **DISTINCT** in the **SELECT** statement just before the select list. You may also specify the keyword **ALL** to indicate that duplicate rows are to be retained.

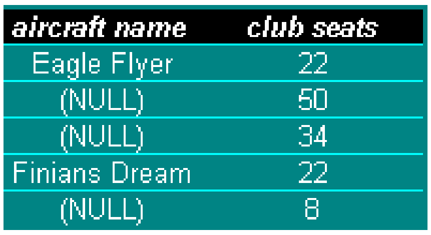
The simplest SQL queries request columns of data from a single table in the database.
The first example shows how the SQL **SELECT** is used to implement the relational algebra **PROJECT** operator, and will retrieve the name of all aircrafts and the number of club seats on each plane :

SELECT *aircraft_name, no_club_seats* **FROM** *aircraft*;



aircraft name	model	club seats	econ seats	call sign
Eagle Flyer	ATR42	22	40	N410C
(NULL)	Boeing 707-320C	50	102	9J-AEB
(NULL)	Boeing 727-200	34	100	N7255U
Finians Dream	Boeing 737-200	22	96	DQ-FDM
(NULL)	Boeing 737-200	8	120	N301SW

Table returned from the Projection.



aircraft name	club seats
Eagle Flyer	22
(NULL)	50
(NULL)	34
Finians Dream	22
(NULL)	8

The following is an example of a **PROJECT**ion of columns from the customer table :

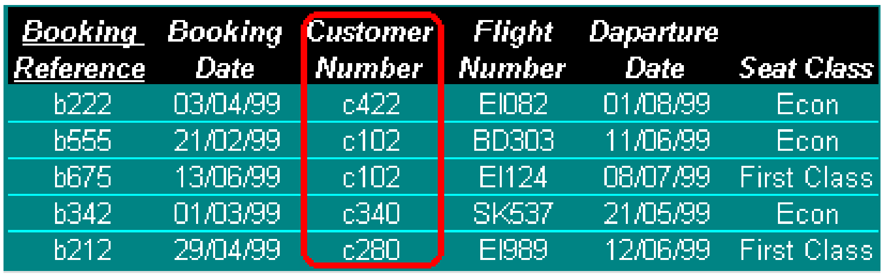
SELECT *customer_name, telephone* **FROM** *customer*;

The following **SELECT** just displays all the rows and columns in one table :

SELECT * **FROM** *airport*;

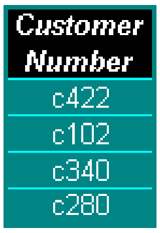
Each row in a table is unique, but by **SELECT**ing only some of the columns, duplicate rows can be produced. The keyword **DISTINCT** is used to qualify the column name so that no duplicate values will be displayed.
The following example returns the list of customers (by customer number) who have made bookings. Duplicate customer numbers are eliminated.

SELECT DISTINCT *customer_number* **FROM** *booking*;



Booking Reference	Booking Date	Customer Number	Flight Number	Daparture Date	Seat Class
b222	03/04/99	c422	EI082	01/08/99	Econ
b555	21/02/99	c102	BD303	11/06/99	Econ
b675	13/06/99	c102	EI124	08/07/99	First Class
b342	01/03/99	c340	SK537	21/05/99	Econ
b212	29/04/99	c280	EI989	12/06/99	First Class

Table returned from Select statement using Distinct.



Customer Number
c422
c102
c340
c280

SQL Course

The Select Statement

- Introduce the **SELECT** statement and describe its use.
- Describe fully the syntax of the **SELECT** statement.

Here it is shown how the SQL **SELECT** can be used to implement the relational algebra **PROJECT** operator.
The **SELECT** statement lets you find and view your data in a variety of ways.You can use it to answer questions based on your data i.e. how many, where, what kind of, even what if. Once you become comfortable with its sometimes dauntingly complex syntax, you'll be amazed at what the **SELECT** statement can do.

- All SQL retrievals are made using the **SELECT** statement.
- The **SELECT** statement requires a **FROM** clause to specify the table(s) to be searched.
- Data retrieval is relationally complete. This means you can :
  - Retrieve any atomic piece of data.
  - Retrieve all data.
  - Retrieve any subset of data
    (any column or row subset combination).
  - Retrieve any set of subsets of data.
  - Data can be retrieved in a sorted order.

The **SELECT** statement retrieves data From a database and returns it to you in the form of query results.
Later modules will look at depth at the variations and possibilities capable when using the **SELECT** statement.

The full **SELECT** statement is specified as follows :

> **SELECT** [**ALL**|**DISTINCT**] *{select_list* | **}* **FROM** *{table_name* | *view_name}* [, *{table_name* | *view_name}*] [**WHERE** *search_conditions*] [**GROUP BY** *column_name* [,*column_name*] ...] [**HAVING** *search_conditions*] [**ORDER BY** {*column_name* | *select_list_number*} [**ASC** | **DESC**] [,{*column_name* | *select_list_number*} [**ASC** | **DESC**]] ...**]**;

Although SQL is a free form language, you do have to keep the clauses in the **SELECT** statement in syntactical order (i.e. a **GROUP BY** clause must come before an **ORDER BY** clause). Otherwise you'll get syntax errors when you try to execute the query.

You may also need to qualify the names of database objects if there is any ambiguity about which object you mean. The most you would ever have to qualify a table or column name is by :

- database.
- owner.
- table_name.
- column_name.

This would result in: **database.owner.table_name.column_name**

Qualifiers are usually omitted in most books, articles, and reference manuals on SQL because the short forms make the **SELECT** statements more readable. However it's never wrong to include them.

The **SELECT** and **FROM** clauses are required. The remaining four clauses are optional. You can use them when you want to use the functionality they provide. The function of each clause is summarised below :

Select

> This clause lists the data items to be retrieved by the **SELECT** statement. The items may be columns from the database, or columns to be calculated by SQL as it performs the query.

From

> This clause lists the tables that contain the data to be retrieved by the query.

Where

> The **WHERE** clause tells SQL to include only certain rows of data in the query results. A search condition is used to specify the desired rows.

Group By

> This specifies a summary query. A summary query groups together similar rows and then produces one summary row of query results for each group.

Having

> This tells SQL to include only certain groups produced by the **GROUP BY** clause. It also uses a search condition to specify the desired groups.

Order By

> This clause sorts the query results based on the data in one or more columns.

The query results will always have the same row/column format as the actual tables in the database that are being queried. Some queries may return no rows. Even in this situation the returned result is still regarded as a table with zero rows. Query results may also contain **NULL** values if the column or columns they were retrieved from also contained **NULL** values. The advantage of the query results being in tabular form is that these results may in turn be queried or used in conjunction with other SQL statements.

A **SELECT** statement can include :

- Expressions of column values and constants :
  e.g. *c_name || c_address, 1.5 * p_economy.*
- Functions of column values and constants :
  e.g. *round (salary), upper (c_name).*
- Group functions of sets of rows :
  e.g. *avg(salary).*

SQL Course

Expression Operators

- Demonstrate the use of expression operators in SQL **SELECT** statements.
- Illustrate the use of expression operators with a few examples.

In addition to columns whose values come directly from the database, a SQL query can include calculated columns whose values are calculated from the stored data values. To request a calculated column, you specify a SQL expression operator in the select list. The ANSI/ISO standard specifies three categories of expression operators that may be used in a **SELECT** statement :

- Arithmetic Operators.
- Time/Date Operators.
- String Operators.

  The ANSI/ISO standard specifies four arithmetic operations that can be used in expressions :

- Addition.
- Subtraction.
- Multiplication.
- Division.

The arithmetic operators **-**, ***** and **/** can be used in expressions involving constants, column values and functions of column values.
Parentheses can be used to form more complicated expressions. The ANSI/ISO standard specifies that multiplication and division have a higher precedence than addition or subtraction and as such parentheses are not strictly required. However, parentheses should be used to make your SQL statements unambiguous and also because different SQL dialects may use different rules. The standard also specifies automatic data type conversion from integers to decimal numbers and from decimal point numbers to floating point numbers, as required.
Of course the columns referenced in an arithmetic expression must have a numeric type. The calculated column is calculated row by row using the data values of the current row. SQL constants can also be used by themselves in the select list. This can be useful for producing query results that are easier to read and interpret.

Date Operators

The following expressions are allowed with DATE types :

date + number	adds a number of days 'number' to a date.
date - number	subtracts a number of days 'number' from a date.
date - date	subtracts two dates to give a number of days.

The SQL2 standard supports addition and subtraction of DATE, TIME, and TIMESTAMP data, for occasions when these operations make sense.

String Operators

Column values can be concatenated using the '**||**' operator as implemented by DB2.

The following displays the total number of seats on each aircraft :

SELECT *no_club_seats* **+** no_economy_seats **FROM** *aircraft***;**

The following example will calculate the time elapsed between the booking of the flight and it's departure :

SELECT *departure_date* **-** *booking_date* **FROM** *booking***v;**

The following example retrieves the customer name and address as one column :

SELECT *customer_name* **||** *address* **FROM** *customer***;**

SQL Course

The Where Clause

- Demonstrate how to select only certain rows from a table using the **SELECT** statement.
- Illustrate the use of the **WHERE** clause with a simple example.

In a real world environment queries which retrieve all rows of a table are rarely any use. Invariably these queries lead to redundant information being retrieved. Most often a user will only want to retrieve some of the rows from a table and include those rows in the query results. The user may do this by using the **WHERE** clause.
The **WHERE** clause consists of the keyword **WHERE** followed by a search condition that specifies the rows to be retrieved. An in depth coverage of search conditions is available in the *Search Conditions* module.
Conceptually SQL goes through each row of the table one by one and applies the search condition to each row. If the search condition evaluates to be **TRUE** then the row is included in the query results, if the search condition is **FALSE** then the row is excluded and if the search condition has a **NULL** unknown value then the row is excluded from the query results.
This section shows how the SQL **SELECT** can be used to implement the relational algebra select operator. Note that the relational algebra select operator is sometimes called **RESTRICT**.

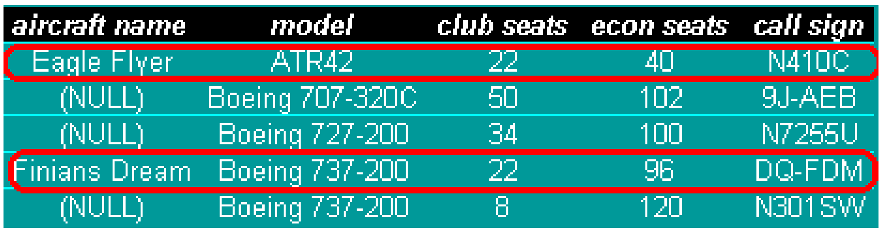
A select statement can have the form :

> **SELECT** *select_list* **FROM** *table_name* **WHERE** *search_condition* **AND | OR** *search_condition***;**

- The **WHERE** clause restricts which rows are displayed by the **SELECT** statement.
- There is a wide range of search conditions to restrict the rows which are chosen.

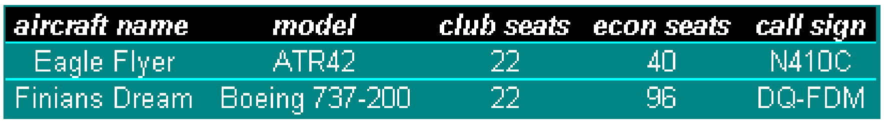
The following example displays all the aircraft details that have club class accommodation of 22 :

> **SELECT * FROM** *aircraft* **WHERE** *no_club_seats* **=** *22***;**



aircraft name	model	club seats	econ seats	call sign
Eagle Flyer	ATR42	22	40	N410C
(NULL)	Boeing 707-320C	50	102	9J-AEB
(NULL)	Boeing 727-200	34	100	N7255U
Finians Dream	Boeing 737-200	22	96	DQ-FDM
(NULL)	Boeing 737-200	8	120	N301SW

Table returned from this selection -



aircraft name	model	club seats	econ seats	call sign
Eagle Flyer	ATR42	22	40	N410C
Finians Dream	Boeing 737-200	22	96	DQ-FDM

SQL Course

Search Conditions

- Show the various search conditions that may be used in the **WHERE** clause.
- Illustrate how different search conditions can be combined.
- Illustrate the use of search conditions with a few examples.

SQL offers a rich set of search conditions that allow you to specify many different kinds of queries efficiently and naturally. Five different search conditions, called predicates in the ANSI/ISO standard, are described in this module.

- **Comparison Test** : Compares the value of one expression to the value of another expression.
- **Range Test** : Tests whether the value of an expression falls between a specified range of values.
- **Set Membership Test** : Checks whether the value of an expression matches one of a set of values.
- **Pattern Matching Test** : Checks whether the value of a column containing string data matches a specified pattern.
- **Null Value Test** : Checks whether or not a column has a **NULL** value.

The **WHERE** clause may be followed by one of the search conditions listed above, as in the following syntax. Search conditions may be combined using the **AND**, **OR** and **NOT** keywords.

> **WHERE** *search_condition* ([**AND|OR|NOT** *search_condition*] [**AND|OR|NOT** *search_condition*] ...)**;**

Comparison Tests

> In a comparison test, SQL computes and compares the values of two SQL expressions for each row of data. The expressions can be as simple as a column name or a constant or even more complex arithmetic expressions.

SQL offers six different ways of comparing the two expressions, shown in the following list :

=	Equality
<>	Inequality (also **!=** and **^=**)
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to

For the inequality comparison the ANSI/ISO standard specifies **<>** to be used. Several other specifications use alternate notations such as **!=** (SQLServer), **~=**(DB2 and SQL/DS).
For the inequality comparison the ANSI/ISO standard specifies **<>** to be used. Several other specifications use alternate notations such as **!=** (SQLServer), **~=**(DB2 and SQL/DS).
Strings can be compared using any of the following operators :

= != > >= < <=

In the test if either of the two expressions produce a **NULL** value then the comparison also yields a **NULL** result. Only rows which yield a **TRUE** result from the search condition are included in the query results. This is due to SQLs three valued logic (**TRUE**, **FALSE**, **NULL**).

The range test checks whether a data value lies between two specified values. **BETWEEN** *expression1* **AND** *expression2* is used for the range test. It involves three SQL expressions.

- The first expression defines the value to be tested.
- The second expression defines the low end of the range to be tested.
- The third expression defines the high end of the range to be tested.
- The data types of the three expressions must be comparable.
- The negated version of the range test (**NOT BETWEEN**) checks for values that fall outside the range specified by the second and third expressions.

The first expression is usually just a column name. The ANSI/ISO standard specifies the following rules for the range test to handle null values :

- If the test expression produces a **NULL** value or if both expressions defining the range produce **NULL** values then the **BETWEEN** test returns a **NULL** result.
- If the expression defining the lower end of the range produces a **NULL** value, then the **BETWEEN** test returns **FALSE** if the the test value is greater than the upper bound, and **NULL** otherwise.
- If the expression defining the upper end of the range produces a **NULL** value, then the **BETWEEN** test returns **FALSE** if the the test value is less than the lower bound, and **NULL** otherwise.

The **BETWEEN** test can be easily expressed as two comparison tests joined by the keyword **AND**.

Set Membership Tests

The set membership test, specified by the keyword **IN**, tests whether a data value matches one of a list of target values. The list of target values are surrounded by brackets and separated by commas. The following rules apply for the set membership test :

- This test can also be negated by using the **NOT IN** statement.
- The test expression is usually just a column name but can also be a more complicated expression.
- If the test expression produces a **NULL** result then the test will produce a **NULL** result.
- All of the items in the list of target values must have the same data type, and that data type must be comparable to the data type of the test expression.
- The **IN** test may also be replaced by comparison tests joined together by the keyword **OR**.

The ANSI/ISO standard doesn't specify a maximum limit to the number of items that can appear in the value list as most implementations don't either. For portability reasons it's generally a good idea to avoid lists with only a single item.

Pattern Matching Test

SQLs pattern matching test can be used to retrieve the data based on a partial match of a text string such as a customers name. The pattern matching test (**LIKE**) checks to see whether the data value in a column matches a specified pattern. The **LIKE** test must be applied to a column with a string data type. The pattern is a string that may include one or more wild-card characters. These wild-card characters and their use is given below :

%

This character can match any sequence of zero or more characters.

_

The underscore character can match any single character.

The following rules apply to wild-card characters :

- Wild-card characters can appear anywhere in a string.
- There can be several wild-card characters in a single string.
- The test can be negated to retrieve strings that do not match a pattern using the **NOT LIKE** statement.
- If the data value in a column is **NULL**, then the test will return a **NULL** result.

One of the problems with pattern matching is how to include the wild-card characters themselves as characters in the pattern. You can't just include them in the pattern, as the DBMS will interpret it as a wild-card.
The ANSI/ISO standard does specify a way to match these special characters by using a special escape character. When the escape character appears in a pattern, the character immediately following it is treated as an ordinary character. The escape character can itself be escaped. The escape character is specified as a one character constant string in the **ESCAPE** clause of the search condition.
For Example :

WHERE *customer_no* **LIKE**
'AHG£%HGF£%'
ESCAPE '£';

As the **ESCAPE** clause has not been widely implemented in commercial products, it is best avoided so as to enhance portability.

Null Value Test

For any given row, the result of a search condition may be **TRUE**, **FALSE** or **NULL** depending on the contents of the column being evaluated. Sometimes it's useful to check explicitly for **NULL** values in a search condition and handle them directly. The following rules apply for **NULL** values :

- SQL provides a **NULL** value test, **IS NULL**, to handle this. The negated form of the null value test, **IS NOT NULL**, finds rows that do not contain a **NULL** value.
- The **NULL** value test cannot produce a **NULL** result, it can only produce either a **TRUE** or **FALSE** result.
- It may seem strange that the **NULL** keyword can't be used in a comparison test but it makes sense as the **NULL** value has no meaning, it is just a flag, and as such would cause unpredictable results if allowed in these tests.

WHERE clause conditions can be constructed from multiple search criteria nested together using the standard Boolean logic listed below :

- **OR**.
- **AND**.

- **NOT**.

OR is used to combine search conditions when at least one of the search conditions must be **TRUE**.
AND is used when all the search conditions must be **TRUE**.
Finally you can use the keyword **NOT** to select rows where a search condition is **FALSE**.
You can build very complex queries using these three keywords. When two or more search conditions are combined **AND**, **OR**, and **NOT**, the ANSI/ISO standard specifies that **NOT** has the highest precedence, followed by **AND** and then **OR**. To ensure portability it's usually a good idea to use parentheses and remove any ambiguity. The use of parentheses forces precedence.

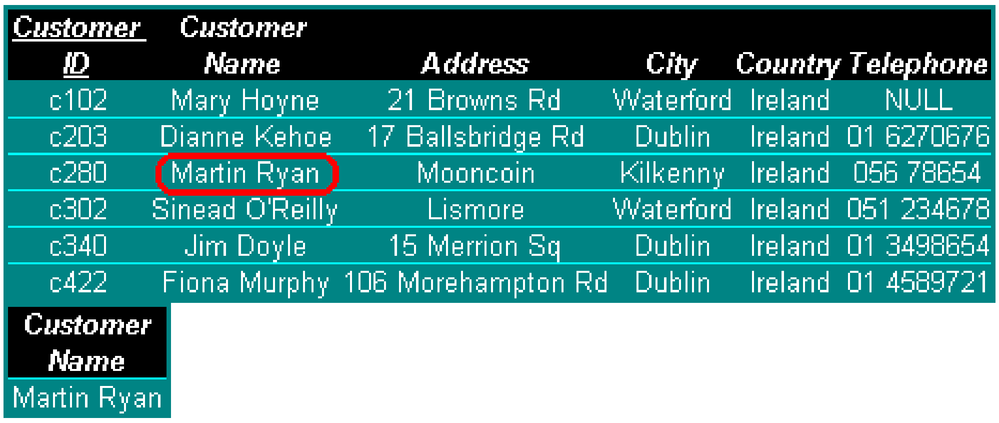
The SQL2 standard adds another logical search condition, the **IS** test. The **IS** test, checks to see whether the logical value of an expression or comparison test is **TRUE**, **FALSE** or **UNKNOWN** (**NULL**).

Note :

- **=Any** is equivalent to **In**.
- **!=All** is equivalent to **Not In**.
- **< >=All** is equivalent to **Not In**.

To display all the names of customers who are not from Waterford nor Dublin :

> **SELECT** *customer_name* **FROM** *customer* **WHERE** *city* **NOT IN** ("Waterford", "Dublin");



Customer ID	Customer Name	Address	City	Country	Telephone
c102	Mary Hoyne	21 Browns Rd	Waterford	Ireland	NULL
c203	Dianne Kehoe	17 Ballsbridge Rd	Dublin	Ireland	01 6270676
c280	Martin Ryan	Mooncoin	Kilkenny	Ireland	056 78654
c302	Sinead O'Reilly	Lismore	Waterford	Ireland	051 234678
c340	Jim Doyle	15 Merrion Sq	Dublin	Ireland	01 3498654
c422	Fiona Murphy	106 Morehampton Rd	Dublin	Ireland	01 4589721

Customer Name
Martin Ryan

To display all the aircraft that have club class accommodation :

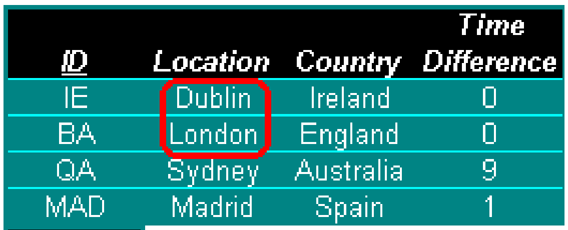
> **SELECT** * **FROM** *aircraft* **WHERE** *no_club_seats* **<>** 0;

To display all booking details for economy seats :

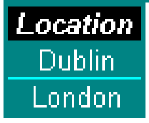
> **SELECT** * **FROM** *booking* **WHERE** *no_seat_class* **=** "Econ";

To display the location of all airports from Ireland and England :

> **SELECT DISTINCT** *location* **FROM** *airport* **WHERE** *country* **=** "Ireland" **AND** *country* **=** "England";



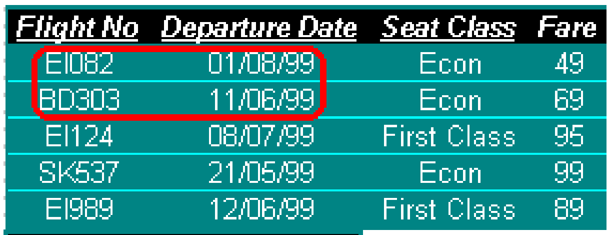
ID	Location	Country	Time Difference
IE	Dublin	Ireland	0
BA	London	England	0
QA	Sydney	Australia	9
MAD	Madrid	Spain	1



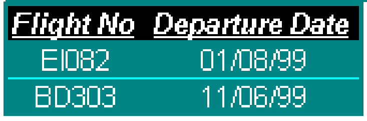
Location
Dublin
London

To display the flight number and departure date of flights which cost less than £70 :

SELECT *flight_no, departure_date* **FROM** *fare* **WHERE** *fare* **<** 70**;**



Flight No	Departure Date	Seat Class	Fare
EI082	01/08/99	Econ	49
BD303	11/06/99	Econ	69
EI124	08/07/99	First Class	95
SK537	21/05/99	Econ	99
EI989	12/06/99	First Class	89



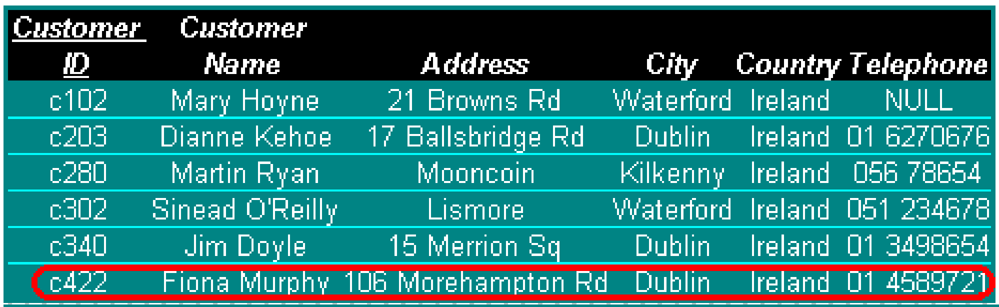
Flight No	Departure Date
EI082	01/08/99
BD303	11/06/99

To display details on flights which cost in the range £40 - £90 :

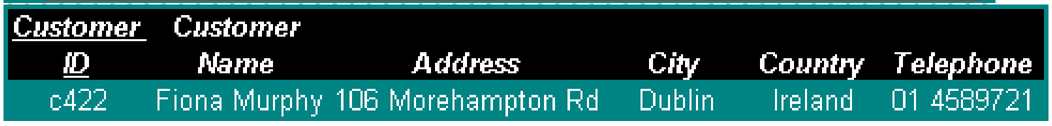
SELECT * **FROM** *fare* **WHERE** *fare* **BETWEEN** 40 **AND** 90**;**

To display details on customers whose names include the characters **on** :

SELECT * **FROM** *customer* **WHERE** *customer_name* **LIKE** "**%**on**%**"**;**



Customer ID	Customer Name	Address	City	Country	Telephone
c102	Mary Hoyne	21 Browns Rd	Waterford	Ireland	NULL
c203	Dianne Kehoe	17 Ballsbridge Rd	Dublin	Ireland	01 6270676
c280	Martin Ryan	Mooncoin	Kilkenny	Ireland	056 78654
c302	Sinead O'Reilly	Lismore	Waterford	Ireland	051 234678
c340	Jim Doyle	15 Merrion Sq	Dublin	Ireland	01 3498654
c422	Fiona Murphy	106 Morehampton Rd	Dublin	Ireland	01 4589721



Customer ID	Customer Name	Address	City	Country	Telephone
c422	Fiona Murphy	106 Morehampton Rd	Dublin	Ireland	01 4589721

SQL Course

Aggregate Functions

- Introduce the functions available for use in a SQL query.
- Illustrate the use of the many functions using a few examples.

- SQL lets you summarise data from the database through a set of column functions.
- A SQL column function takes an entire column of data as its argument and produces a single data item that summarises the column.

SQL offers six different column functions, as shown below :
Sum()
Computes the total of a column. The result of the **SUM()** function has the same basic data type as the data in the column, but the result may have a higher precision. The syntax is as follows :

SUM ([**DISTINCT|ALL**] *column*)**;**

Avg()
Computes the average value in a column. The result of the **AVG()** function may be of a different data type then the values in the column. The syntax is as follows:

AVG([**DISTINCT | ALL**] *column*)**;**

Min()
Finds the smallest value in a column. The syntax is as follows :

MIN (expr)**;**

Max()
Finds the largest value in a column. The syntax is as follows :

MAX (expr)**;**

Count()
Counts the number of values in a column. The syntax is as follows :

COUNT([**DISTINCT|ALL**] expr)**;**

Count(*)
Counts rows of query results. The syntax is as follows :

COUNT (*)**;**

Both **MIN()** and **MAX()** can operate on numeric, string or date/time information. For numeric numbers, SQL compares the numbers in algebraic order. Dates are compared sequentially. Durations are compared based on their length. The comparison of two strings depend on the character set being used.

There are also a few other SQL functions :
Stddev()

Finds the Standard Deviation in a column. The syntax is as follows :

STDDEV ([**DISTINCT**|**ALL**] *column*)**;**

Variance()
Finds the Variance in a column. The syntax is as follows :

VARIENCE ([**DISTINCT**|**ALL**] *column*)**;**

You can ask SQL to eliminate duplicate values from a column before applying a column function to it. To eliminate duplicate values, the keyword **DISTINCT** is included before the column function argument, immediately after the opening parenthesis.

One of the best ways to think about column functions is to imagine the query processing broken down into two steps :

- First, imagine how the query would work without the column functions, producing many rows of query results.
- Then imagine the DBMS applying the column functions to the detailed query results, producing a single summary row.

It's illegal to nest column functions.It's also illegal to mix column functions and ordinary column names in a select list, because the resulting query doesn't make sense.
As stated, an aggregate function returns one single data item that summarises the chosen column, but if you want to categorise that column by a value in another column you must use the **GROUP BY** clause.

The ANSI/ISO standard specifies that **NULL** values are ignored by column functions. This can lead to inconsistencies when counting the number of rows in a query result. The ANSI/ISO standard specifies these precise rules for handling **NULL** values in column functions :

- **NULL** values are ignored by column functions.
- If every data item in a column is **NULL** , then **SUM()**, **AVG()**, **MIN()** and **MAX()** return a **NULL** result. **COUNT()** returns a value of zero.
- If there are no data items in the column, then **SUM()**, **AVG()**, **MIN()** and **MAX()** return a **NULL** value and **COUNT()** returns a value of zero.
- **COUNT(*)** does not depend on the presence or absence of **NULL** so will work as usual.

These rules may vary from specific implementation to implementation.

The SQL1 standard specifies that when **DISTINCT** is used, the argument to the column function must be a simple column name, it cannot be an expression. SQL2 relaxes these restrictions and allows the **DISTINCT** keyword to be applied for any of the column functions, and permitting expressions as arguments for any of the functions as well.
The **DISTINCT** keyword can only be specified once in a query. If it appears in one column function, then it cannot appear in any other. If it is specified before the column list then it cannot appear in any column functions. The only exception is that it may appear within a sub-query.

SQL1 did not have support for built in functions, even though most implementations incorporated their own. In general, a built in function can be specified in a SQL expression anywhere that a constant of the same data type can be specified. The SQL2 standard incorporated the most useful built in functions from commercial implementations, in many cases with slightly different syntax. The functions are summarised in the table below :

Built-In functions in SQL2

Function	Returns
Bit_Length(string)	numbers of bits in a string
Cast(value AS data_type)	the value, converted to the specified data type
Char_Length(string)	Numbers of bits in a string
Conver(string **Using** conv)	string converted as specified by a named conversion function
Current_Date	current date
Current_Time(precision)	current time with the specified precision
Current_Timestamp(precision)	current date and time with the specified precision
Extract(part **From** source)	specified part (**Day**, **Hour**, etc.) from a **Datetime** value
Lower(string)	string converted to all lower case letters
Octet_Length(string)	number of 8 bit bytes in a character string
Position(target **In** source)	position where the target string appears within the source string
Substring(source **From** n **For** len)	a portion of the source string, beginning at the nth character, for a length of len
Translate(string **Using** trans)	string translated as specified by a named translation function
Trim(**Both** char **From** string)	string with both leading and trailing occurrences of char trimmed off
Trim(**Leading** char **From** string)	string with any leading occurrences of char trimmed off
Trim(**Trailing** char **From** string)	string with any trailing occurrences of char trimmed off
Upper(string)	string converted to all upper case letters

There are many types of function which include :
Arithmetic Functions

Arithmetic Functions are used in expressions involving column values:

Abs (n)	absolute value of n
Ceil (n)	smallest integer greater than or equal to n
Floor (n)	largest integer greater than or equal to n
Mod (m,n)	remainder of m divided by n
Power (m,n)	m raised to nth power. If n is not an integer it will be truncated.

Other arithmetic functions include **Round**, **Sign**, **Sqrt** and **Trunc**.

Functions on Character Strings
There are many Character string functions including:

Ascii (c)	**Ascii** value of first character of c
Chr (n)	The character with **Ascii** value n (integer)
Initcap (c)	Capitalises the first letter of each word contained in c

Other functions include **Upper**, **Substr**, **Lower**, **Length** and **Soundex** .

Date functions
Several functions which operate on **Date** types (columns holding values defined by type date) include:

Add_Months (d,n)	Adds a number of months n to a date d
Last_Day (d)	Get last day of month containing date d
Months_Between (d,e)	Number of months between dates d and e.

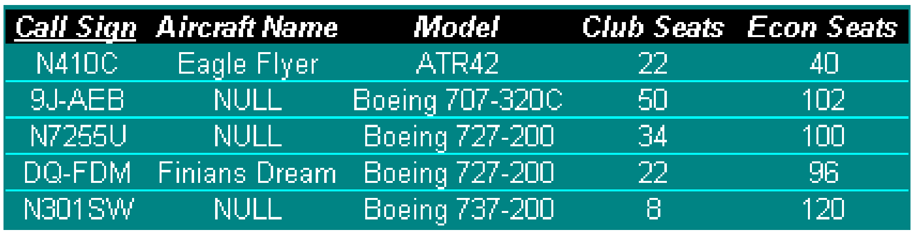
Other date functions include **New_Time**, **Next_Day** and **Trunc**.

Miscellaneous and conversion functions
There are several MISCELLANEOUS functions:
Decode, **Dump**, **Greatest**, **Least** , **Nvl** and **Vsize**.

Another set of functions will perform CONVERSIONS of columns or values into different datatypes e.g. **To_Number**, **To_Char** and **To_Date**.

The following example will count the number of flights in the database from the table aircraft_flight :

SELECT COUNT(*) FROM *aircraft_flight*;

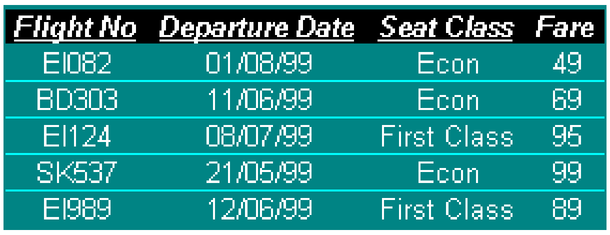


Call Sign	Aircraft Name	Model	Club Seats	Econ Seats
N410C	Eagle Flyer	ATR42	22	40
9J-AEB	NULL	Boeing 707-320C	50	102
N7255U	NULL	Boeing 727-200	34	100
DQ-FDM	Finians Dream	Boeing 727-200	22	96
N301SW	NULL	Boeing 737-200	8	120

The answer returned is 5

The following example calculates the minimum fare paid on any flight :

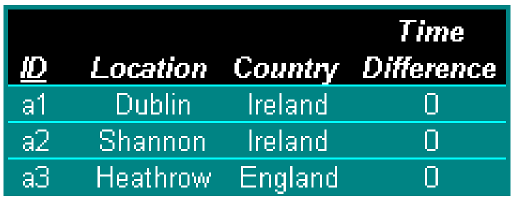
SELECT MIN(*fare***) FROM** *fare***;**



Flight No	Departure Date	Seat Class	Fare
EI082	01/08/99	Econ	49
BD303	11/06/99	Econ	69
EI124	08/07/99	First Class	95
SK537	21/05/99	Econ	99
EI989	12/06/99	First Class	89

The answer returned is 4

SELECT*country*, **COUNT(***id***)FROM** *airport* **GROUP BY** *country*;



ID	Location	Country	Time Difference
a1	Dublin	Ireland	0
a2	Shannon	Ireland	0
a3	Heathrow	England	0

The answer returned is:

Ireland 2 England 1