3. Getting data out: database queries

Querying

There are two good reasons for using a database to store information. Firstly, as you have seen, databases are very good at storing data in an accurate and consistent way (if we take care with the design, anyway). The second reason is that they provide a very efficient way to get out exactly the data you need.

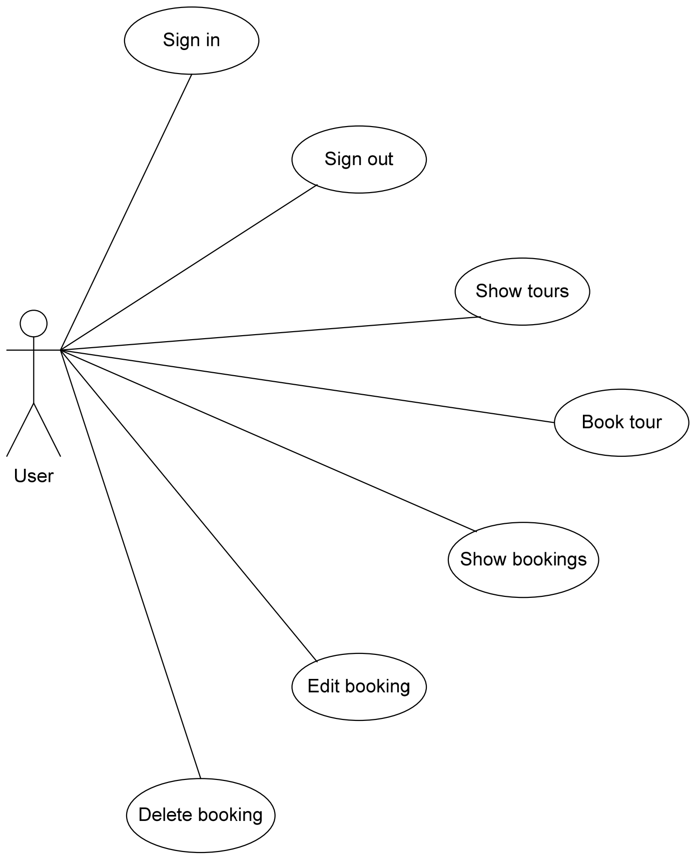
A large database may have thousands or millions of rows in its tables, while you may want to select only one, or a few specific data items – for example, to find the bookings placed by a specific user, or to find only the users whose accounts are unpaid.

We get that specific information out of the database by **querying**. A query is *a question* addressed to a database, usually written in a **query language.** For relational databases, the standard query language is **SQL**.

Some database tools provide graphical interfaces for building queries. These tools actually write the SQL for you behind the scenes, which is handy if you don't know SQL. However, it's well worth learning SQL because you can use it to query *any* RDBMS. Also, if you're accessing a database from an enterprise application you usually need to write some SQL.

Queries and use cases

The questions we need to ask the database are related to the uses cases for the system. A use case diagram for GCUTours is shown in the figure below.



A query selects only the **subset** of the data we actually need to carry out the actions of a use case. Some of these use cases will obviously lead to queries. To carry out the *'Show tours'* use case, we will need to query the database to find all the available tours.

Others are slightly less obvious. For example, *'Sign in'* will need to query the database to find the stored password for the user in order to check it matches the one which the user types in.

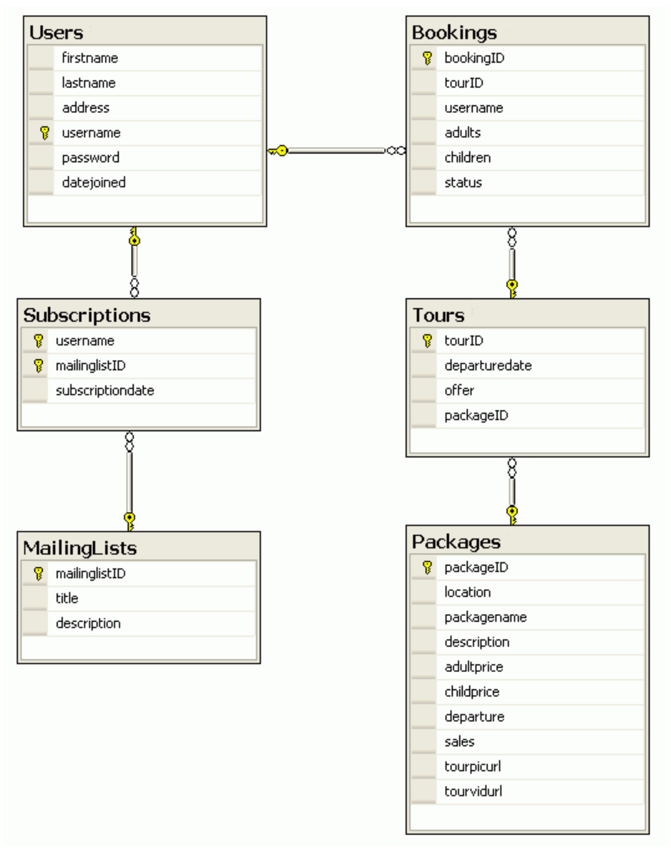
NOTE

Other use cases, such as *'Edit booking'* relate to **updating** and **deleting** data, which we will look at later.

The GCUTours database tables

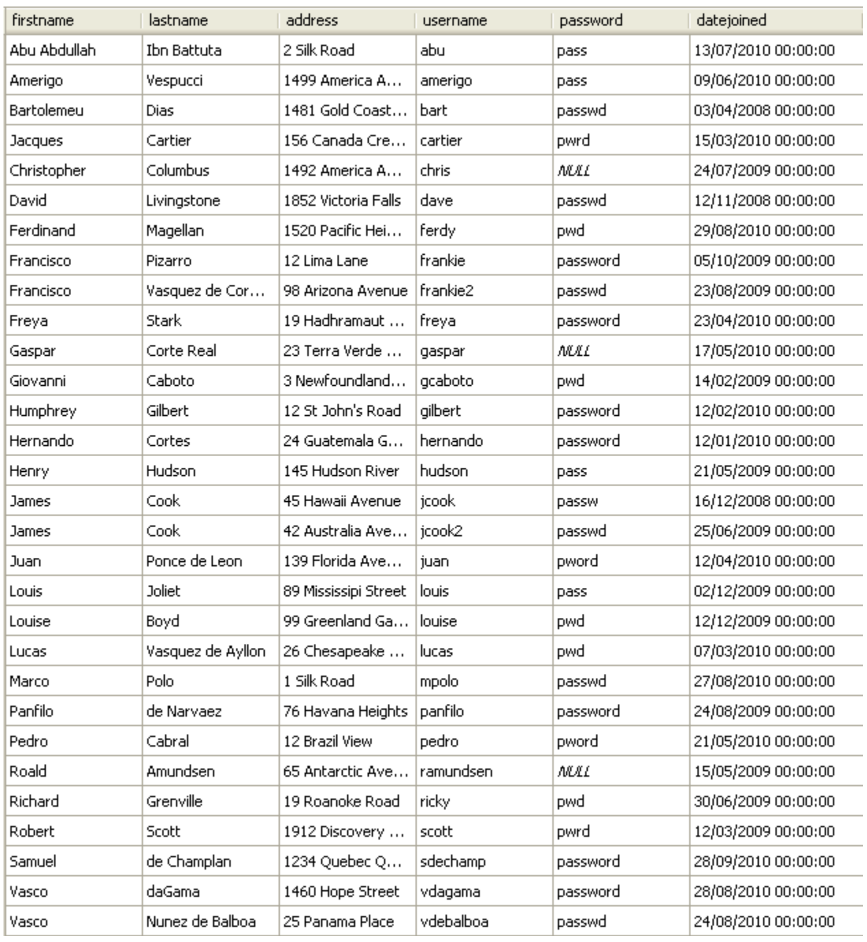
These notes use a version of the GCUTours database which has been populated with a bit more data than you saw before. You can download this database, inside a WebMatrix website, from your module website.

The tables and relationships in this database are summarised in the figure below:



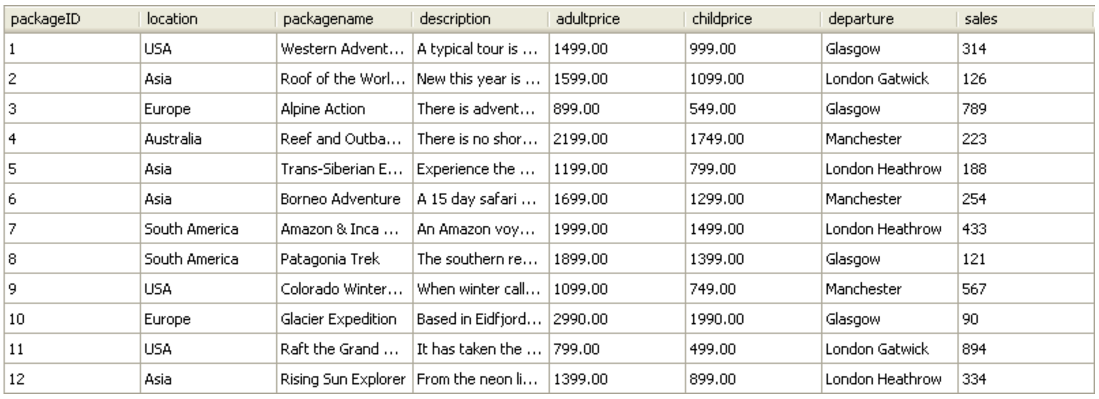
We will only be using the Users, Bookings, Tours and Packages tables here. The data in the tables is shown on the following pages – you can refer back to this as you read the notes.

Users



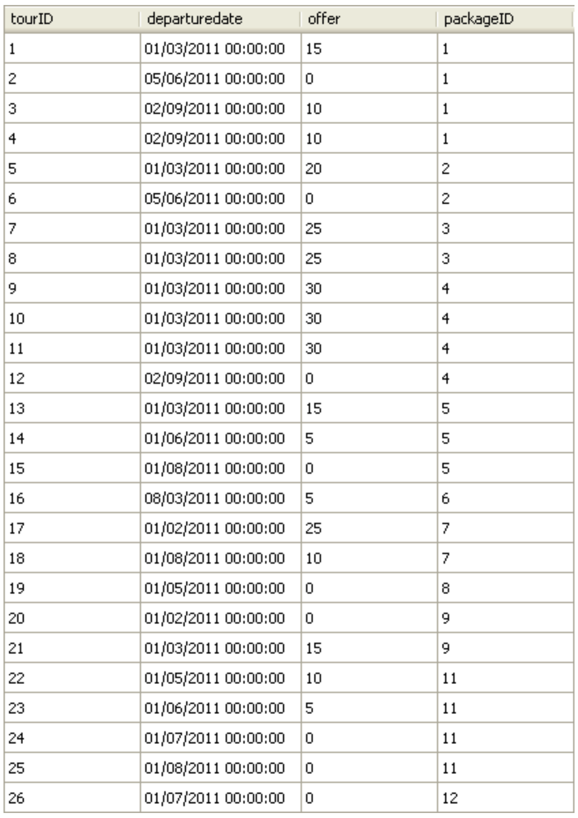
firstname	lastname	address	username	password	datejoined
Abu Abdullah	Ibn Battuta	2 Silk Road	abu	pass	13/07/2010 00:00:00
Amerigo	Vespucci	1499 America A...	amerigo	pass	09/06/2010 00:00:00
Bartolemeu	Dias	1481 Gold Coast...	bart	passwd	03/04/2008 00:00:00
Jacques	Cartier	156 Canada Cre...	cartier	pwrd	15/03/2010 00:00:00
Christopher	Columbus	1492 America A...	chris	*NULL*	24/07/2009 00:00:00
David	Livingstone	1852 Victoria Falls	dave	passwd	12/11/2008 00:00:00
Ferdinand	Magellan	1520 Pacific Hei...	ferdy	pwd	29/08/2010 00:00:00
Francisco	Pizarro	12 Lima Lane	frankie	password	05/10/2009 00:00:00
Francisco	Vasquez de Cor...	98 Arizona Avenue	frankie2	passwd	23/08/2009 00:00:00
Freya	Stark	19 Hadhramaut ...	freya	password	23/04/2010 00:00:00
Gaspar	Corte Real	23 Terra Verde ...	gaspar	*NULL*	17/05/2010 00:00:00
Giovanni	Caboto	3 Newfoundland...	gcaboto	pwd	14/02/2009 00:00:00
Humphrey	Gilbert	12 St John's Road	gilbert	password	12/02/2010 00:00:00
Hernando	Cortes	24 Guatemala G...	hernando	password	12/01/2010 00:00:00
Henry	Hudson	145 Hudson River	hudson	pass	21/05/2009 00:00:00
James	Cook	45 Hawaii Avenue	jcook	passw	16/12/2008 00:00:00
James	Cook	42 Australia Ave...	jcook2	passwd	25/06/2009 00:00:00
Juan	Ponce de Leon	139 Florida Ave...	juan	pword	12/04/2010 00:00:00
Louis	Joliet	89 Mississipi Street	louis	pass	02/12/2009 00:00:00
Louise	Boyd	99 Greenland Ga...	louise	pwd	12/12/2009 00:00:00
Lucas	Vasquez de Ayllon	26 Chesapeake ...	lucas	pwd	07/03/2010 00:00:00
Marco	Polo	1 Silk Road	mpolo	passwd	27/08/2010 00:00:00
Panfilo	de Narvaez	76 Havana Heights	panfilo	password	24/08/2009 00:00:00
Pedro	Cabral	12 Brazil View	pedro	pword	21/05/2010 00:00:00
Roald	Amundsen	65 Antarctic Ave...	ramundsen	*NULL*	15/05/2009 00:00:00
Richard	Grenville	19 Roanoke Road	ricky	pwd	30/06/2009 00:00:00
Robert	Scott	1912 Discovery ...	scott	pwrd	12/03/2009 00:00:00
Samuel	de Champlan	1234 Quebec Q...	sdechamp	password	28/09/2010 00:00:00
Vasco	daGama	1460 Hope Street	vdagama	password	28/08/2010 00:00:00
Vasco	Nunez de Balboa	25 Panama Place	vdebalboa	passwd	24/08/2010 00:00:00

Packages



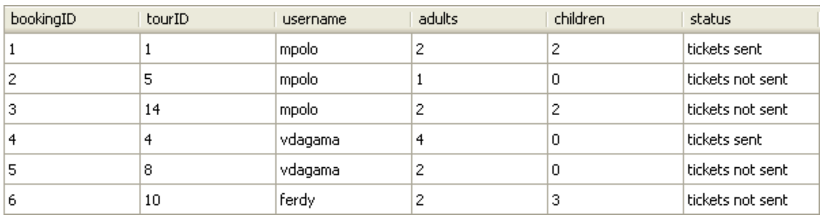
packageID	location	packagename	description	adultprice	childprice	departure	sales
1	USA	Western Advent...	A typical tour is ...	1499.00	999.00	Glasgow	314
2	Asia	Roof of the Worl...	New this year is ...	1599.00	1099.00	London Gatwick	126
3	Europe	Alpine Action	There is advent...	899.00	549.00	Glasgow	789
4	Australia	Reef and Outba...	There is no shor...	2199.00	1749.00	Manchester	223
5	Asia	Trans-Siberian E...	Experience the ...	1199.00	799.00	London Heathrow	188
6	Asia	Borneo Adventure	A 15 day safari ...	1699.00	1299.00	Manchester	254
7	South America	Amazon & Inca ...	An Amazon voy...	1999.00	1499.00	London Heathrow	433
8	South America	Patagonia Trek	The southern re...	1899.00	1399.00	Glasgow	121
9	USA	Colorado Winter...	When winter call...	1099.00	749.00	Manchester	567
10	Europe	Glacier Expedition	Based in Eidfjord...	2990.00	1990.00	Glasgow	90
11	USA	Raft the Grand ...	It has taken the ...	799.00	499.00	London Gatwick	894
12	Asia	Rising Sun Explorer	From the neon li...	1399.00	899.00	London Heathrow	334

Tours



tourID	departuredate	offer	packageID
1	01/03/2011 00:00:00	15	1
2	05/06/2011 00:00:00	0	1
3	02/09/2011 00:00:00	10	1
4	02/09/2011 00:00:00	10	1
5	01/03/2011 00:00:00	20	2
6	05/06/2011 00:00:00	0	2
7	01/03/2011 00:00:00	25	3
8	01/03/2011 00:00:00	25	3
9	01/03/2011 00:00:00	30	4
10	01/03/2011 00:00:00	30	4
11	01/03/2011 00:00:00	30	4
12	02/09/2011 00:00:00	0	4
13	01/03/2011 00:00:00	15	5
14	01/06/2011 00:00:00	5	5
15	01/08/2011 00:00:00	0	5
16	08/03/2011 00:00:00	5	6
17	01/02/2011 00:00:00	25	7
18	01/08/2011 00:00:00	10	7
19	01/05/2011 00:00:00	0	8
20	01/02/2011 00:00:00	0	9
21	01/03/2011 00:00:00	15	9
22	01/05/2011 00:00:00	10	11
23	01/06/2011 00:00:00	5	11
24	01/07/2011 00:00:00	0	11
25	01/08/2011 00:00:00	0	11
26	01/07/2011 00:00:00	0	12

Bookings



bookingID	tourID	username	adults	children	status
1	1	mpolo	2	2	tickets sent
2	5	mpolo	1	0	tickets not sent
3	14	mpolo	2	2	tickets not sent
4	4	vdagama	4	0	tickets sent
5	8	vdagama	2	0	tickets not sent
6	10	ferdy	2	3	tickets not sent

You have previously seen how to create a query in WebMatrix which allowed you to write an SQL statement to create a database table. We will now look at SQL statements which **query the database** to retrieve information.

The simplest situation is when all the data you want to get is in just **one table**. For now, we'll use the GCUTours database tables to illustrate the main types of query that you can do on a single table.

Project operations

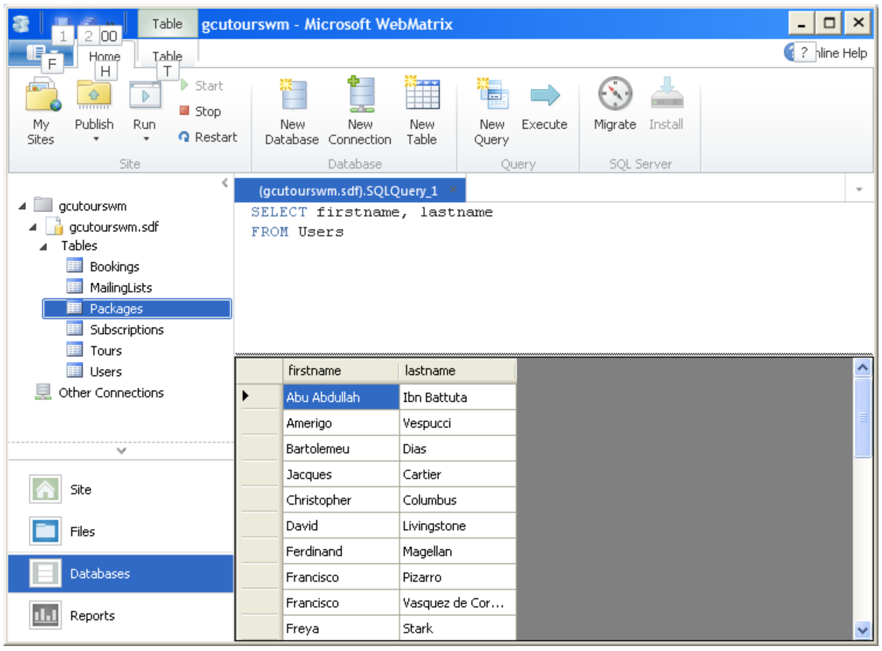
Project operations allow us to retrieve **only the fields we are interested in**. Let's say we only want to see the names (first and last) of all the users. We can write this query in SQL as:

```
SELECT firstname, lastname
FROM Users
```

This query uses the following SQL **key words**:

`SELECT` – introduces the **list of fields** to be included
`FROM` – specifies which **table** to get data from

The result of executing this SQL in a query in WebMatrix is shown below:



```
SELECT firstname, lastname
FROM Users
```

firstname	lastname
Abu Abdullah	Ibn Battuta
Amerigo	Vespucci
Bartolemeu	Dias
Jacques	Cartier
Christopher	Columbus
David	Livingstone
Ferdinand	Magellan
Francisco	Pizarro
Francisco	Vasquez de Cor…
Freya	Stark

The * operator

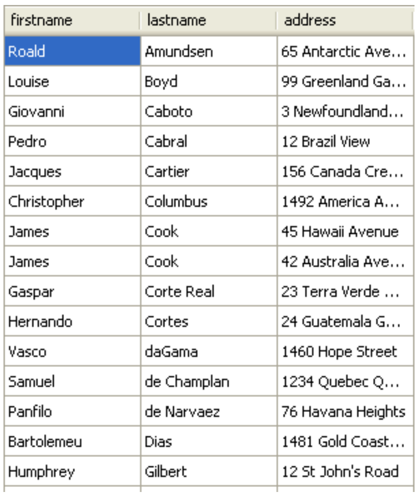
Using * after SELECT instead of a list of field names causes **all available fields** to be included. The result of this query is that the entire contents of the table are shown.

```
SELECT *
FROM Users
```

Ordering

The results of any query can be **sorted in order of the values** in any field (or fields) by using the `ORDER BY` key word at the end of the query. The following example orders by the value of *lastname*, then by *firstname* if there are any identical values of *lastname*. The `ASC` key word means sort in ascending order.

```
SELECT firstname, lastname, address
FROM Users
ORDER BY lastname, firstname ASC
```

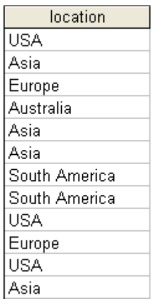


firstname	lastname	address
Roald	Amundsen	65 Antarctic Ave...
Louise	Boyd	99 Greenland Ga...
Giovanni	Caboto	3 Newfoundland...
Pedro	Cabral	12 Brazil View
Jacques	Cartier	156 Canada Cre...
Christopher	Columbus	1492 America A...
James	Cook	45 Hawaii Avenue
James	Cook	42 Australia Ave...
Gaspar	Corte Real	23 Terra Verde ...
Hernando	Cortes	24 Guatemala G...
Vasco	daGama	1460 Hope Street
Samuel	de Champlan	1234 Quebec Q...
Panfilo	de Narvaez	76 Havana Heights
Bartolemeu	Dias	1481 Gold Coast...
Humphrey	Gilbert	12 St John's Road

Duplicate rows

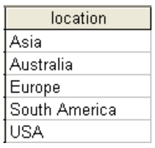
Sometimes there are duplicate values in the fields included in a project operation (even though each complete row is unique). Compare the following two queries, and note the effect of the `DISTINCT` key word:

```
SELECT location
FROM Packages
```



location
USA
Asia
Europe
Australia
Asia
Asia
South America
South America
USA
Europe
USA
Asia

```
SELECT DISTINCT location
FROM Packages
```

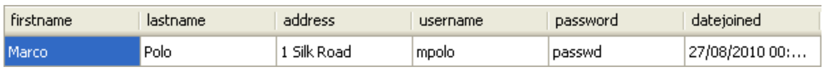


location
Asia
Australia
Europe
South America
USA

Select operations

Select operations allow us to retrieve just **some of the rows**. For example, we might want to find the data stored about a particular user, identified by username. We use the WHERE key word to specify the **condition**, or **filter**, which decides which row or rows to retrieve.

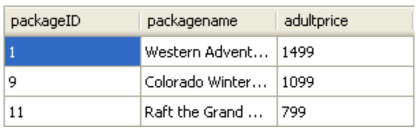
```
SELECT *
FROM Users
WHERE username = 'mpolo'
```



firstname	lastname	address	username	password	datejoined
Marco	Polo	1 Silk Road	mpolo	passwd	27/08/2010 00:...

Most queries in reality are a **combination of project and select** operations, for example:

```
SELECT packageID, packagename, adultprice
FROM Packages
WHERE location = 'USA'
```



packageID	packagename	adultprice
1	Western Advent...	1499
9	Colorado Winter...	1099
11	Raft the Grand ...	799

NOTE

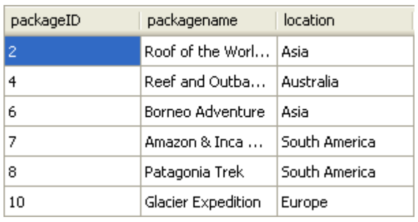
If the value to be matched is text, it needs to be in **single quotes**. Numerical values are **not** written in quotes.

Types of condition in a Select operation

The last two queries retrieved data where a value in the database **exactly** matched the specified condition because we used the `=` operator in the condition. There are other operators we can use to match data in less exact ways. Here are some examples:

Find packages with adultprice greater than £1500 using the `>` operator

```
SELECT packageID, packagename, location
FROM Packages
WHERE adultprice > 1500
```

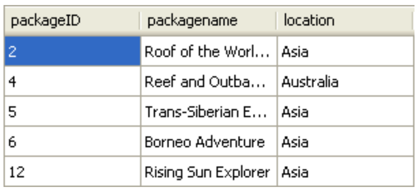


packageID	packagename	location
2	Roof of the Worl...	Asia
4	Reef and Outba...	Australia
6	Borneo Adventure	Asia
7	Amazon & Inca ...	South America
8	Patagonia Trek	South America
10	Glacier Expedition	Europe

Other similar **relative operators** you can use are `<`, `<=`, `>=`, `<>` (not equal)

Find packages where the location starts with the letter A using the `LIKE` operator

```
SELECT packageID, packagename, location
FROM Packages
WHERE location LIKE 'A%'
```



packageID	packagename	location
2	Roof of the Worl...	Asia
4	Reef and Outba...	Australia
5	Trans-Siberian E...	Asia
6	Borneo Adventure	Asia
12	Rising Sun Explorer	Asia

`%` is the wildcard character, which matches any character. `A%` means *A followed by any other characters*.

The underscore character `_` matches a single character. `A_` means *A followed by any single character*.

Some RDBMSs also allow you to have wildcards which match characters within a specified list or range

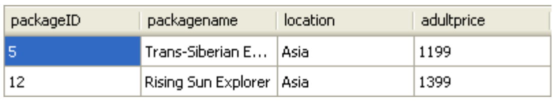
More than one condition

Conditions in a Select operation can be **combined** using the AND and OR operators, as shown in the following examples:

Find holidays located in Asia with adultprice less than £1500

We use the AND operator when **both** conditions must be true:

```
SELECT packageID, packagename, location, adultprice
FROM Packages
WHERE location = 'Asia' AND adultprice < 1500
```

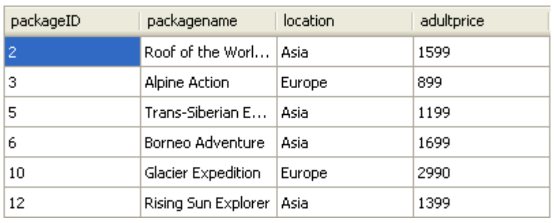


packageID	packagename	location	adultprice
5	Trans-Siberian E...	Asia	1199
12	Rising Sun Explorer	Asia	1399

Find all holidays in Asia or Europe

We use the OR operator when either condition may be true:

```
SELECT packageID, packagename, location, adultprice
FROM Packages
WHERE location = 'Asia' OR location = 'Europe'
```

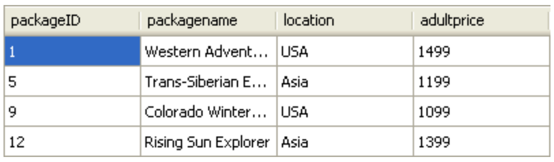


packageID	packagename	location	adultprice
2	Roof of the Worl...	Asia	1599
3	Alpine Action	Europe	899
5	Trans-Siberian E...	Asia	1199
6	Borneo Adventure	Asia	1699
10	Glacier Expedition	Europe	2990
12	Rising Sun Explorer	Asia	1399

Find holidays with adultprice between £1000 and £1500

We can use the AND operator with relative operators to match data within a **range of values**:

```
SELECT packageID, packagename, location, adultprice
FROM Packages
WHERE adultprice > 1000 AND adultprice < 1500
```



packageID	packagename	location	adultprice
1	Western Advent...	USA	1499
5	Trans-Siberian E...	Asia	1199
9	Colorado Winter...	USA	1099
12	Rising Sun Explorer	Asia	1399

You can also use the `BETWEEN` keyword for this situation. This gives exactly the same result as the previous query:

```sql
SELECT packageID, packagename, location, adultprice
FROM Packages
WHERE adultprice BETWEEN 1000 AND 1500
```

Date formats in queries

Dates are more complicated than most of the other data types, and it is important to understand how your particular RDBMS interprets them.

The following query refers to a DateTime field:

```sql
SELECT firstname, lastname
FROM Users
WHERE datejoined > '1/1/2010'
```

This condition will return all rows with *datejoined* later than 1st January 2010. What if the condition was this:

```sql
WHERE datejoined > '14/3/2010'
```

Or what about this:

```sql
WHERE datejoined > '3/14/2010'
```

It is likely that one of these will work and the other will cause an error. This may depend on the way your computer is configured. 14 is not a valid month value, but 3 is, so the first version represents a valid date in **UK form** (*dd/mm/yyyy*), while the second is valid in the **US form** (*mm/dd/yyyy*).

What about this:

```sql
WHERE datejoined > '3/7/2010'
```

This will not cause an error as 3 and 7 are both valid month values. However, it may not give you the answer you expect. You can even find that dates in queries are interpreted as US dates, while the results are shown in UK format.

To be safe, you can specify dates in the **ISO standard date format** (yyyy-mm-dd). The following condition ensures that you mean 7th March 2010:

```sql
WHERE datejoined > '2010-03-07'
```

Aggregates

The queries you have seen so far retrieve data row by row. It can also be useful to be able to ask questions which combine the data in more than one row into a single value. For example, you might want to count the number of rows which match a condition, or to add up all the values in a particular column of a table.

SQL provides **aggregate functions**, including `COUNT`, `SUM`, `AVG`, `MAX` and `MIN`, to allow you to do queries like that.

Counting

The simplest aggregate query **counts all the rows** in a table, for example:

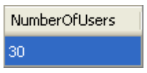
```
SELECT COUNT(*)
FROM Users
```

`COUNT(*)` simply means *"count each record"*.

The result of this query is the single number 30, as there are 30 users in the table.

It can be helpful to give the results of aggregate functions meaningful labels when they are displayed, using the `AS` key word:

```
SELECT COUNT(*) AS NumberOfUsers
FROM Users
```

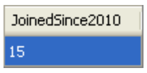


```
NumberOfUsers
30
```

You can use a Select operation to **count only some of the rows**:

Count the number of Users who have joined since the beginning of 2010

```
SELECT Count(*) AS JoinedSince2010
FROM Users
WHERE datejoined > '2010-01-01'
```



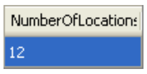
```
JoinedSince2010
15
```

As before, the `DISTINCT` key word can deal with duplicate values. For example, the Packages table has **12 records**, but only contains **5 different locations**.

Compare the following two queries, where we are counting the number of values in a particular field rather than the number of complete records:

```
SELECT COUNT(location)
AS NumberOfLocations
FROM Packages
```

```
SELECT COUNT(DISTINCT location)
AS NumberOfLocations
FROM Packages
```



NumberOfLocation
12

should give **5**

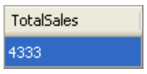
Note that no actual result is shown for the second version – this use of `DISTINCT` **is** standard SQL, but isn't supported in SQL Server Compact!

Summarising

We'll use the *sales* figures in *Packages* to demonstrate SQL summarising functions:

Find the total number of sales for all packages

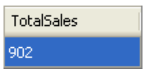
```
SELECT SUM(sales) AS TotalSales
FROM Packages
```



TotalSales
4333

Find the total number of sales for all packages to Asia

```
SELECT SUM(sales) AS TotalSales
FROM Packages
WHERE location = 'Asia'
```



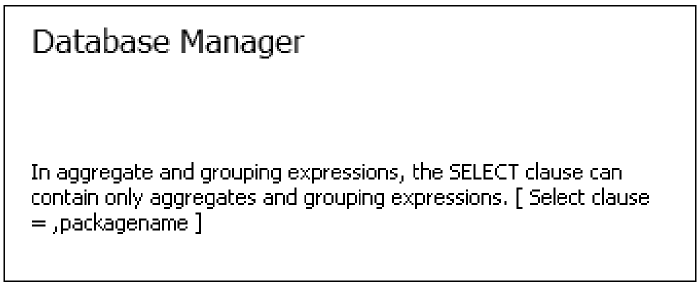
TotalSales
902

Take care with aggregates

There are some common mistakes that people make when using aggregates. For example, to find which package had the highest number of sales, and also list the name of that package alongside its sales value, you might try this query:

```
SELECT packagename, MAX(sales)
FROM Packages
```

If you run this, though, you'll get an error message like this:


Database Manager

In aggregate and grouping expressions, the SELECT clause can contain only aggregates and grouping expressions. [Select clause = ,packagename]

The reason is as follows:

- *packagename* has **one** value for **each row** of the table
- *MAX(sales)* has **one** value for the **entire table**

Think about the results you are **actually** asking this query to give you:

packagename	Max(Sales)
Western Adventure	894
Roof of the World Explorer	
Alpine Action	
Reef and Outback Adventure	
Trans-Siberian Express	
Borneo Adventure	
Amazon & Inca Adventure	
Patagonia Trek	
Colorado Winter Adventure	
Glacier Expedition	
Raft the Grand Canyon	
Rising Sun Explorer	

12 rows

1 row

This *can't* work! The result of a query must have the **same number of rows in each column**, just like a table has. A simple rule to remember is:

» *all the fields or functions after the SELECT key word must have the* ***same number of values***

You will see later on a better way of getting the information we were trying to get here.

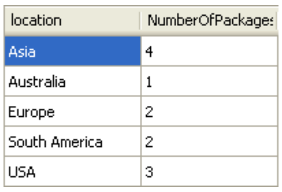
Grouping

So far we have used aggregates to count or summarise an entire table, or a set of rows matching a condition. It's also very useful to **group rows which have something in common together**, and to count or summarise each group. This is done using the `GROUP BY` key word.

In the *Packages* table, there is a group of rows which refer to packages in Asia, another group of packages in the USA, and so on. Let's count the number of rows in each group.

Find the number of packages available in each location

```
SELECT location, COUNT(*) AS NumberOfPackages
FROM Packages
GROUP BY location
```



location	NumberOfPackages
Asia	4
Australia	1
Europe	2
South America	2
USA	3

What about the numbers of values?

- *location* has **one** value for **each group of rows** (as we are grouping rows with the same location together)
- *COUNT(*)* has **one** value for **each group of rows** (as aggregates are applied to **each group separately** when the `GROUP BY` key word appears)

So the numbers of values in this query match up – so it works!

Why would the following NOT work?

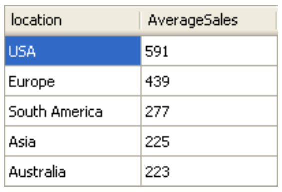
```
SELECT location, packagename, COUNT(*) AS NumberOfPackages
FROM Packages
GROUP BY location
```

The **packagename** field causes the problem here – it has one value for each row of the table, and the grouping by **location** has no effect on the **packagename** values.

Here is another example of grouping. This time we list the groups in order of the result of the AVG function (denoted by the label *AverageSales*) for each group:

Find the average number of sales for each location and show the results in order of popularity, most popular first

```
SELECT location, AVG(sales) AS AverageSales
FROM Packages
GROUP BY location
ORDER BY AverageSales DESC
```



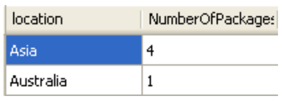
location	AverageSales
USA	591
Europe	439
South America	277
Asia	225
Australia	223

Applying conditions to groups

You saw earlier how to use the `WHERE` keyword to apply **conditions** to decide which rows to retrieve. You can do the same with groups of rows using the `HAVING` key word.

Find the number of packages available in each location, only showing locations beginning with the letter A

```
SELECT location, COUNT(*) AS NumberOfPackages
FROM Packages
GROUP BY location
HAVING location LIKE 'A%'
```



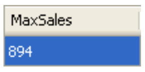
location	NumberOfPackages
Asia	4
Australia	1

Subqueries

You saw the MAX function earlier. This function can be used like this:

Find the maximum number of sales for any one package

```
SELECT MAX(sales) AS MaxSales
FROM Packages
```



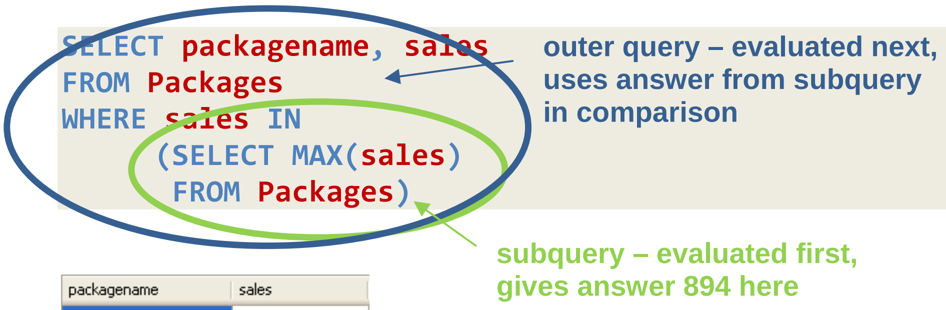
MaxSales
894

This query result isn't very helpful, as it doesn't tell us **which package** is the most popular. We tried earlier to list the name of the most popular package along with its sales earlier, but ran into problems doing so. We could, however, get that information by using the result of the MAX(sales) query, 894, in another query:

```
SELECT packagename, sales
FROM Packages
WHERE sales = 894
```

It's inconvenient to use two queries like this, so fortunately there's a better way. In the example below, the first query, which found the maximum value, is used as a **subquery** for the second (outer) query. The subquery is in written in brackets:

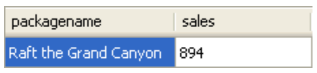
Find the name and number of sales of the most popular package



```
SELECT packagename, sales
FROM Packages
WHERE sales IN
     (SELECT MAX(sales)
      FROM Packages)
```

outer query – evaluated next, uses answer from subquery in comparison

subquery – evaluated first, gives answer 894 here



packagename	sales
Raft the Grand Canyon	894

Here's another example, this time a question which can be answered with grouping and a subquery:

Find the locations where the highest sales for any package to that location is less than the average of sales for all packages

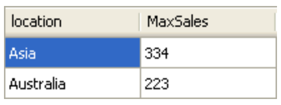
We need to find the **average of all sales** before we can do a comparison:

```
SELECT AVG(Sales)
FROM Packages
```

This is the subquery. The **outer query** needs to find the **maximum sales for each location**, so it needs to group by location. The maximum for each group needs to be compared with the subquery result.

The full query is:

```
SELECT location, MAX(sales) AS [MaxSales] FROM Packages
GROUP BY location
HAVING MAX(sales) < ALL
     (SELECT AVG(Sales)
      FROM Packages)
```



location	MaxSales
Asia	334
Australia	223

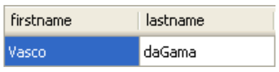
Subqueries can also be useful if we want to get information from **more than one table**. For example:

Find the full name of the user who made the booking with ID 4

The *Bookings* table simply identifies the user by *username*. The full name (*firstname* and *lastname*) is in the *Users* table.

We can use a subquery to find which *username* is related to the booking with ID 4 in *Bookings*, and then use the result of that to query *Users* for the full name.

```
SELECT firstname, lastname
FROM Users
WHERE username IN
      (SELECT username FROM Bookings
       WHERE bookingID = 4)
```



firstname	lastname
Vasco	daGama

There is actually another way of getting information from more than one table, which you will see in a later chapter.