

3. Getting data out: database queries

Querying	1
Queries and use cases.....	2
The GCUTours database tables	3
Queries in Access.....	5
Project operations	6
Select operations	8
Date formats in queries.....	11
Aggregates	12
Grouping.....	14
Subqueries.....	16

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.

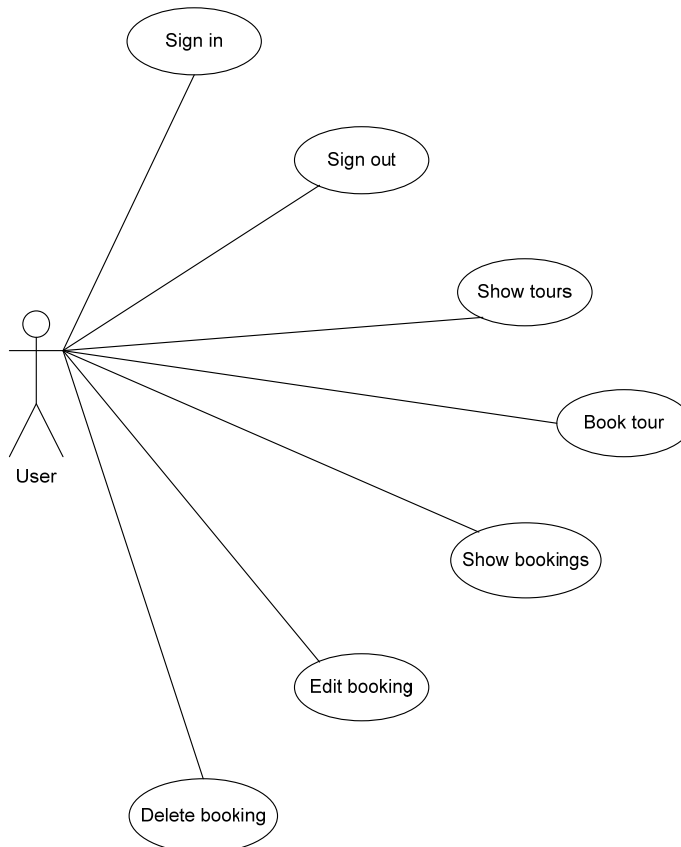
A large database may have thousands or millions of rows in its tables, while you may want to select only one, or a few specific data items – for example, to find the bookings placed by a specific user, or to find only the users whose accounts are unpaid.

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, including Access, 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.

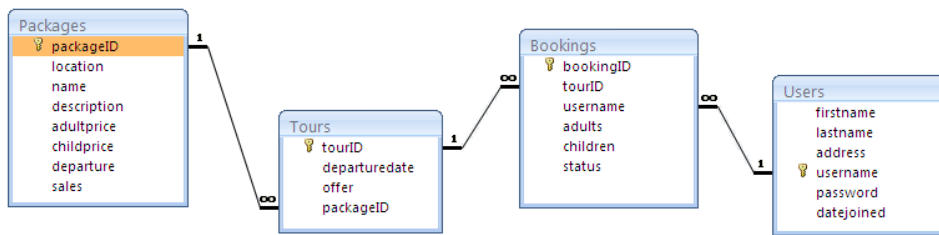
NOTE

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 from your module website.

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



The data in the tables is shown below – you can refer back to this as you read the notes:

Users

	firstname	lastname	address	username	password	datejoined
+	Abu Abdullah	Ibri Battuta	2 Silk Road	abu	pass	13/07/2007
+	Amerigo	Vespucci	1499 America Avenue	amerigo	pass	09/06/2007
+	Bartolemeu	Dias	1481 Gold Coast Road	bart	passwd	03/04/2005
+	Christopher	Columbus	1492 America Avenue	chris	passwd	24/07/2006
+	David	Livingstone	1852 Victoria Falls	dave	passwd	12/11/2005
+	Ferdinand	Magellan	1520 Pacific Heights	ferdy	pwd	29/08/2007
+	Francisco	Pizarro	12 Lima Lane	frankie	password	05/10/2006
+	Francisco	Vasquez de Co	98 Arizona Avenue	frankie2	passwd	23/08/2006
+	Freya	Stark	19 Hadhramaut Street	freya	password	23/04/2007
+	Gaspar	Corte Real	23 Terra Verde Way	gaspar	passwd	17/05/2007
+	Giovanni	Caboto	3 Newfoundland Road	gcaboto	pwd	14/02/2006
+	Henry	Hudson	145 Hudson River	hudson	pass	21/05/2006
+	Hernando	Cortez	24 Guatemala Gardens	hernando	password	12/01/2007
+	Humphrey	Gilbert	12 St. John's Road	gilbert	password	12/02/2007
+	Jacques	Cartier	156 Canada Crescent	cartier	pwd	15/03/2007
+	James	Cook	42 Australia Avenue	jcook2	passwd	25/06/2006
+	James	Cook	45 Hawaii Avenue	jcook	passw	16/12/2005
+	Juan	Ponce de Leon	139 Florida Avenue	juan	pwd	12/04/2007
+	Louis	Joliet	89 Mississippi Street	louis	pass	02/12/2006
+	Louise	Boyd	99 Greenland Gardens	louise	pwd	12/12/2006
+	Lucas	Vasquez de Ay	26 Chesapeake Avenue	lucas	pwd	07/03/2007
+	Marco	Polo	1 Silk Road	mpolo	passwd	27/08/2007
+	Panfilo	de Narvaez	76 Havana Heights	panfilo	password	24/08/2006
+	Pedro	Cabral	12 Brazil View	pedro	pwd	21/05/2007
+	Richard	Grenville	19 Roanoke Road	ricky	pwd	30/06/2006
+	Roald	Amundsen	65 Antarctic Avenue	ramundsen	pwd	15/05/2006
+	Robert	Scott	1912 Discovery Drive	scott	pwd	12/03/2006
+	Samuel	de Champlan	1234 Quebec Quadrant	sdechamp	password	28/09/2007
+	Vasco	daGama	1460 Hope Street	vdagama	password	28/08/2007
+	Vasco	Nunez de Balbo	25 Panama Place	vdebalboa	passwd	24/08/2006

Packages

	packageID	location	name	description	adultprice	childprice	departure	sales
+	1	USA	Western Adventure	A typical tour is our Western Adventure f	£1,499.00	£999.00	Glasgow	314
+	2	Asia	Roof of the World Explorer	New this year is our Roof of the World to	£1,599.00	£1,099.00	London Gatwick	126
+	3	Europe	Alpine Action	There is adventure to be found closer to l	£899.00	£549.00	Glasgow	789
+	4	Australia	Reef and Outback Adventure	There is no shortage of adventure on the	£2,199.00	£1,749.00	Manchester	223
+	5	Asia	Trans-Siberian Express	Experience the world's greatest railway a	£1,199.00	£799.00	London Heathrow	188
+	6	Asia	Borneo Adventure	A 15 day safari exploring the forests and	£1,699.00	£1,299.00	Manchester	254
+	7	South America	Amazon & Inca Adventure	An Amazon voyage like no other on boar	£1,999.00	£1,499.00	London Heathrow	433
+	8	South America	Patagonia Trek	The southern region of Argentina and Ch	£1,899.00	£1,399.00	Glasgow	121
+	9	USA	Colorado Winter Adventure	When winter calls, Colorado answers wit	£1,099.00	£749.00	Manchester	567
+	10	Europe	Glacier Expedition	Based in Eidfjord, Norway, this expeditio	£299.00	£199.00	Glasgow	90
+	11	USA	Raft the Grand Canyon	It has taken the Colorado River millions c	£799.00	£499.00	London Gatwick	894
+	12	Asia	Rising Sun Explorer	From the neon lights and modern glitz of	£1,399.00	£899.00	London Heathrow	334

Tours

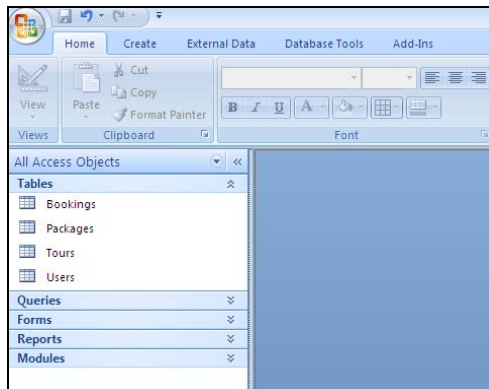
	tourID	departuredate	offer	packageID
+	1	01/03/2008	15	1
+	2	05/06/2008	0	1
+	3	02/09/2008	10	1
+	4	01/03/2008	20	2
+	5	05/06/2008	0	2
+	6	01/03/2008	25	3
+	7	01/03/2008	30	4
+	8	02/09/2008	0	4
+	9	01/03/2008	15	5
+	10	01/06/2008	5	5
+	11	01/08/2008	0	5
+	12	08/03/2008	5	6
+	13	01/02/2008	25	7
+	14	01/08/2008	10	7
+	15	01/05/2008	0	8
+	16	01/02/2008	0	9
+	17	01/03/2008	15	9
+	18	01/05/2008	10	11
+	19	01/06/2008	5	11
+	20	01/07/2008	0	11
+	21	01/08/2008	0	11
+	22	01/07/2008	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
7	13	ferdy	2	3	tickets sent

Queries in Access

In Access, queries are database objects in the same way as tables (and others such as forms and reports which we'll look at later). Usually the word query means the question which is addressed to the database. In Access, a query also means the **view** of the data produced in response to the question.



Looking at the objects in an Access database

As it does for tables, Access gives *design* and *datasheet* views of queries. It also lets you look at an **SQL view**, which lets you write your own SQL queries (it also lets you see the SQL written automatically when a query is built in design view).

We'll look at a few examples in design and SQL view, but we'll mainly concentrate on using SQL.

NOTE

The SQL that Access writes for you is often correct but a bit more complicated than it absolutely needs to be, so don't worry if you try the query builder and see some unfamiliar looking SQL in SQL view.

The simplest situation is when all the data you want to get is in just **one table**. 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
```

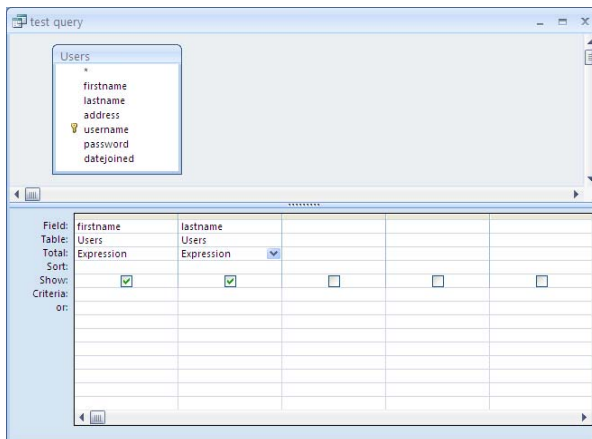
This query uses the following SQL **key words**:

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

Part of the data set retrieved by this query is shown below:

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

You could also create this query in the Access query builder:



Using ***** after SELECT instead of a list of field names causes **all available fields** to be included:

```
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 *
FROM Users
ORDER BY lastname, firstname ASC
```

firstname	lastname	address
Ronald	Amundsen	65 Antarctic Avenue
Louise	Boyd	99 Greenland Gardens
Giovanni	Caboto	3 Newfoundland Road
Pedro	Cabral	12 Brazil View
Jacques	Cartier	156 Canada Crescent
Christopher	Columbus	1492 America Avenue
James	Cook	45 Hawaii Avenue
James	Cook	42 Australia Avenue
Gaspar	Corte Real	23 Terra Verde Way
Hernando	Cortes	24 Guatemala Gardens
Vasco	daGama	1460 Hope Street
Samuel	de Champlan	1234 Quebec Quadrant
Panfilo	de Narvaez	76 Havana Heights
Bartolemeu	Dias	1481 Gold Coast Road
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	27/08/2007

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

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

	packageID	name	adultprice
	1	Western Adventure	£1,499.00
	9	Colorado Winter Adventure	£1,099.00
	11	Raft the Grand Canyon	£799.00

NOTE

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

Look at the Access query builder screenshot on page 6 – how do you think you would specify a SELECT condition?

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, name, location
FROM Packages
WHERE adultprice > 1500
```

packageID	name	location
2	Roof of the World Explorer	Asia
4	Reef and Outback Adventure	Australia
6	Borneo Adventure	Asia
7	Amazon & Inca Adventure	South America
8	Patagonia Trek	South America

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, name, location
FROM Packages
WHERE location LIKE 'A*'
```

packageID	name	location
2	Roof of the World Explorer	Asia
4	Reef and Outback Adventure	Australia
5	Trans-Siberian Express	Asia
6	Borneo Adventure	Asia
12	Rising Sun Explorer	Asia

***** is the wildcard character, which matches any letters. **A*** means *A followed by any other letters*. There are other wildcard characters, including:

- ?** – matches a single character
- #** - matches a single digit in a numeric value

NOTE

Most RDBMSs use different wildcard characters to those Access uses, for example % instead of *. Access databases can be switched to a SQL Server compatible mode in which it too uses the same wildcards as most other databases.

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, name, location, adultprice
FROM Packages
WHERE location = 'Asia' AND adultprice < 1500
```

packageID	name	location	adultprice
5	Trans-Siberian Express	Asia	£1,199.00
12	Rising Sun Explorer	Asia	£1,399.00

Find all holidays in Asia or Europe

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

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

packageID	name	location	adultprice
2	Roof of the World Explorer	Asia	£1,599.00
3	Alpine Action	Europe	£899.00
5	Trans-Siberian Express	Asia	£1,199.00
6	Borneo Adventure	Asia	£1,699.00
10	Glacier Expedition	Europe	£299.00
12	Rising Sun Explorer	Asia	£1,399.00

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, name, location, adultprice
FROM Packages
WHERE adultprice > 1000 AND adultprice < 1500
```

packageID	name	location	adultprice
1	Western Adventure	USA	£1,499.00
5	Trans-Siberian Express	Asia	£1,199.00
9	Colorado Winter Adventure	USA	£1,099.00
12	Rising Sun Explorer	Asia	£1,399.00

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

```
SELECT packageID, name, 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 Date/Time field:

```
SELECT firstname, lastname  
FROM Users  
WHERE datejoined > #1/1/2007#
```

Note that date values in queries should be enclosed in hash (#) characters.

This condition will return all rows with *datejoined* later than 1st January 2007. What if the condition was this:

```
WHERE datejoined > #14/3/2007#
```

As you might expect, this gives all rows with *datejoined* later than 14th March 2007. What about this?

```
WHERE datejoined > #3/14/2007#
```

Access sees that 14 is not a valid month value, but 3 is, and assumes you mean a date in the **US form** (mm/dd/yyyy). Therefore this **also** gives all rows with *datejoined* later than 14th March 2007. So what does it do with the following condition?

```
WHERE datejoined > #3/7/2007#
```

3 and 7 are both valid month values, so this date would be valid in both **UK** (dd/mm/yyyy) and **US forms**. By default, Access chooses to interpret this as a US date, and gives all rows with *datejoined* later than 7th March 2007, which might not be what you were expecting.

To be safe, you can specify dates in the **ISO standard date format** (yyyy-mm-dd). The following condition ensures that you mean 3rd July 2007:

```
WHERE datejoined > #2007-07-03#
```

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 2007

```
SELECT Count(*) AS JoinedSince2007  
FROM Users  
WHERE datejoined > #2007-01-01#
```

	JoinedSince2007
▶	14

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
```

NumberOfLocations
12

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

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 Access!

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

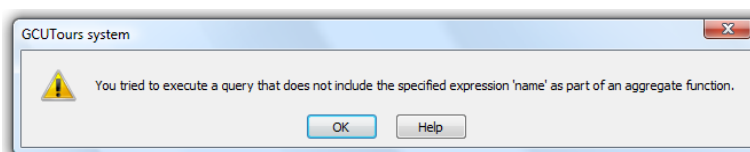
Take care with aggregates

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 name, MAX(sales)
FROM Packages
```

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



The reason is as follows:

- *name* 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:

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

The **name** 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 **name** values.

Here is another example of grouping. This time we list the groups in order of the result of the AVG function 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 AVG(sales) DESC
```

	location	AverageSales
	USA	591.666666666667
	Europe	439.5
	South America	277
	Asia	225.5
▶	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 name, 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 written in brackets.

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

```
SELECT name, sales
FROM Packages
WHERE sales =
  (SELECT MAX(sales)
   FROM Packages)
```

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

	name	sales
▶	Raft the Grand Canyon	894

subquery – evaluated first,
gives answer 894 here

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) <
    (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 =
    (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.