

5: Doing more with queries

Improving query performance with indexes.....	1
Joining tables.....	4
Different kinds of join.....	7
Joining tables with the Access query builder.....	9
Data definition queries.....	10
Insert queries.....	10
Update queries.....	11
Delete queries.....	11
Parameter queries.....	13

Improving query performance with indexes

When you run queries on the GCUTours database you see the results virtually instantaneously. With a small database, the **performance** of a query, or the time it takes to complete, is not an issue. However, when there are many thousands of rows in the database tables, it can take a lot more time to extract the rows you want. If query performance is poor, the system can seem sluggish and unresponsive to the users.

Indexes can enable particular rows of a database table to be found more quickly. Database indexes are much like the index you would find at the back of a book:

Book index	<ul style="list-style-type: none">• Contains an ordered list of topics in a separate section of the book• Each topic has a page number which points to the page in the main section of the book with the full details of the topic• You can find the topic you want very quickly as the index is in order• Finding the topic using the index is much faster than searching through the whole book
Database index	<ul style="list-style-type: none">• Contains an ordered list of the values in particular field(s), stored in a separate part of the database• Each value has a reference which points to the full record which contains that value• The database can find the record you want very quickly as the index is in order• Finding the record using the index is much faster than searching through all the records

Which fields should be indexed?

You should index the fields which will be most often used as conditions in queries, which will depend on the **use cases** for the system.

Let's look at the Users table, part of which is shown below:

	firstname	lastname	address	username	password	datejoined
+	Abu Abdullah	Ibn 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
+	Jacques	Cartier	156 Canada Crescent	cartier	pwrld	15/03/2007
+	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 Coronado	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

It's likely that we'll want to search for users by *username*, for example in the *Log in* use case. Therefore **the *username* field should be indexed**.

In fact, the *username* field is the **primary key** for this table (each user must have a unique *username*). Primary keys are **automatically indexed**, so we don't need to do anything further.

What about the other fields? There might be a need to search for users by name. Therefore we might want to create an index on *lastname*. We could go further and create an index on *lastname* AND *firstname*. Then, a search for David Livingstone will use the first part of the index to find all the Livingstones in the table, then the second part to find the Davids among them. This is an example of a **compound index**.

Why not just index everything?

We've said that indexes speed up queries – so, if all the fields are indexed then all queries should run faster, shouldn't they? Well, they might, but the problem then is the amount of additional storage space which would be needed to hold all the indexes. This could be a major problem for a large database.

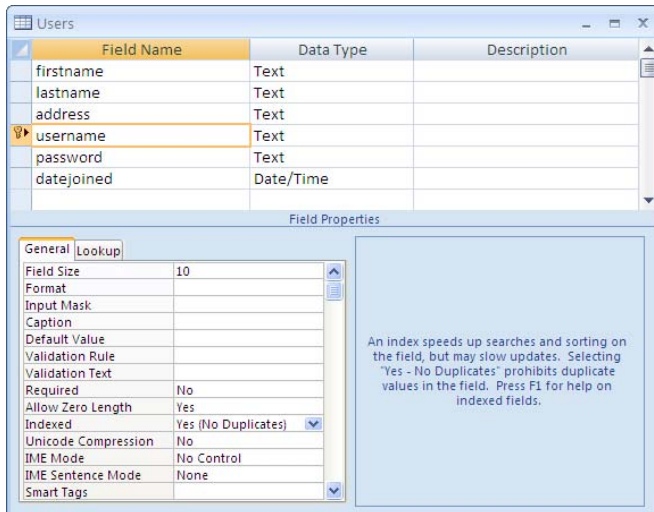
Having too many indexes can also cause performance problems when adding new data and deleting data as all the indexes need to be updated.

One of the decisions which we must take when designing a database is how to choose indexes carefully to speed up common queries without having too many indexes.

Creating indexes

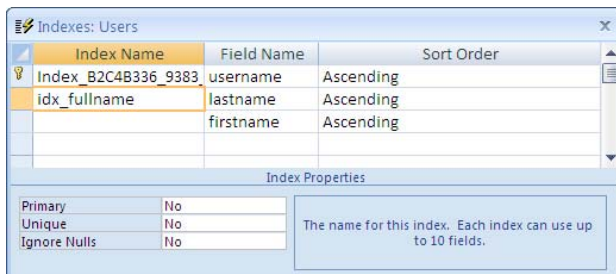
You saw in chapter 2 how to create a primary key in Access in table design view and using SQL. The figure below shows the properties of the *username* primary key field of

the *Users* table in design view. Note that the *Indexed* property is shown as *Yes (No Duplicates)*.



You can create further indexes by setting the *Indexed* property of any other fields to *Yes (Duplicates OK)*, or to *Yes (No Duplicates)* if you want to force the values to be unique.

To create a **compound index** you can open the **Indexes** window when in table design view. In the figure below, the index called *idx_fullname* includes the *lastname* and *firstname* fields.



As always, you can also use SQL, like this:

```
CREATE INDEX idx_fullname ON Users (lastname, firstname)
```

You can also remove an index using SQL. The opposite of **CREATE** in SQL is **DROP**.

```
DROP INDEX idx_fullname ON Users
```

Joining tables

What query will the *Show tours* use case need? We might want to show the **name**, **adult price** and **date of departure** of each tour. The problem is that the name and prices are in the *Packages* table, while the departure date is in the *Tours* table.

The data modelling process encouraged us to treat packages and tours as separate entities and so put them in **separate tables** in the database. If it hadn't, the rules of normalisation would have made us do so anyway to avoid duplication of data and possible inaccuracies in the data.

Database design generally results in data being spread across multiple tables. It's actually fairly uncommon to find all the data need for a particular action stored in a single table.

Packages

packageID	location	name	description	adultprice	childprice	departure	sales
1	USA	Western Adventure	A typical tour is our Western Adventure	£1,499.00	£999.00	Glasgow	314
2	Asia	Roof of the World Explorer	New this year is our Roof of the World	£1,599.00	£1,099.00	London Gatwick	126
3	Europe	Alpine Action	There is adventure to be found closer to	£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	£1,199.00	£799.00	London Heathrow	188
6	Asia	Romeo 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 boat	£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

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

foreign key : packageid

Part of the data in the *Packages* and *Tours* tables

The query now needs to **join** the tables to gather all the information that we need. If you were to get the information just by looking at the tables you would look at each tour in turn, and then look up the corresponding package.

For example, for the row of *Tours* highlighted in the figure above you would pick out the value of '01/03/2008' for departure date, and then look for the matching row of *Packages*. These tables are related by the **foreign key field**, *packageID*, so you would look for the row with the value **2**, and pick out the values 'Roof of the World Explorer' and **£1099** for name and adult price.

NOTE

The next row of *Tours* would match the same row of *Packages* as it also has the value of 2 for *packageID*.

We can do the same thing with an SQL query. Looking up matching rows is done using a join condition:

WHERE Packages.packageID = Tours.packageID

This means

“join each row in Tours to the row in Packages where the value of packageID in Packages matches the value of packageID in Tours”

The **dot notation**, for example *Packages.packageID*, is used to specify a field belonging to a particular table - then we know exactly which one we mean if different tables have fields with identical names.

The full query also needs to say what fields we want to get, and needs to specify that the data will come from both tables.

SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages, Tours
WHERE Packages.packageID = Tours.packageID

packageID	name	adultprice	departuredate
1	Western Adventure	£1,499.00	01/03/2008
1	Western Adventure	£1,499.00	05/06/2008
1	Western Adventure	£1,499.00	02/09/2008
2	Roof of the World Explorer	£1,599.00	01/03/2008
2	Roof of the World Explorer	£1,599.00	05/06/2008
3	Alpine Action	£899.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	02/09/2008
5	Trans-Siberian Express	£1,199.00	01/03/2008
5	Trans-Siberian Express	£1,199.00	01/06/2008
5	Trans-Siberian Express	£1,199.00	01/08/2008
6	Borneo Explorer	£1,699.00	08/03/2008
7	Amazon & Inca Adventure	£1,999.00	01/02/2008
7	Amazon & Inca Adventure	£1,999.00	01/08/2008
8	Patagonia Trek	£1,899.00	01/05/2008
9	Colorado Winter Adventure	£1,099.00	01/02/2008
9	Colorado Winter Adventure	£1,099.00	01/03/2008
11	Raft the Grand Canyon	£799.00	01/05/2008
11	Raft the Grand Canyon	£799.00	01/06/2008
11	Raft the Grand Canyon	£799.00	01/07/2008
11	Raft the Grand Canyon	£799.00	01/08/2008
12	Rising Sun Explorer	£1,399.00	01/07/2008

Note that the *packageID* has been included in this query. Since this field is in both tables, dot notation has been used to say which one to display. We **should** really give the **full name** for every field, for example *Packages.name*, *Tours.departuredate*, but we get away without doing so here because these field names are unique in the database.

NOTE

There is a lot of repeated data in the query result. This is OK because this is just a **view** of the data – there is still no repetition in the **stored data**.

You can combine a join condition with any other condition using **AND** if you want to filter the query results, for example:

```
SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages, Tours
WHERE Packages.packageID = Tours.packageID
AND location = 'Asia'
```

packageID	name	adultprice	departuredate
2	Roof of the World Explorer	£1,599.00	01/03/2008
2	Roof of the World Explorer	£1,599.00	05/06/2008
5	Trans-Siberian Express	£1,199.00	01/03/2008
5	Trans-Siberian Express	£1,199.00	01/06/2008
5	Trans-Siberian Express	£1,199.00	01/08/2008
6	Borneo Explorer	£1,699.00	08/03/2008
12	Rising Sun Explorer	£1,399.00	01/07/2008

Don't forget the join condition

A common mistake people make when writing join queries is to miss out the join condition, like this:



```
SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages, Tours
```

The database will then **join every row of the first table to every row of the second**. In this database Packages has 12 rows and Tours has 22, so the query gives (12x22) = **264 rows**, and the result is probably not very useful!

Joining more than two tables

The example query above joins two tables. However, related data can be split between many tables in a database, and may need to be gathered together in with a query. You can join as many tables as you like as long as you include join conditions for each pair of related tables combined with **AND**, for example:

```
FROM Packages, Tours, Bookings
WHERE Packages.packageID = Tours.packageID
AND Tours.tourID = Bookings.tourID
```

The *Show bookings* use case might need to show the following information:

- *firstname* and *lastname* from **Users**
- *adults* and *children* from **Bookings**
- *departuredate* from **Tours**
- *name* from **Packages**

Think about how you would write a query to get the following result

	firstname	lastname	adults	children	departuredate	name
	Marco	Polo	2	2	01/03/2008	Western Adventure
	Marco	Polo	1	0	05/06/2008	Roof of the World Explorer
	Marco	Polo	2	2	01/08/2008	Amazon & Inca Adventure
	Vasco	daGama	4	0	01/03/2008	Roof of the World Explorer
	Vasco	daGama	2	0	02/09/2008	Reef and Outback Adventure
	Ferdinand	Magellan	2	3	01/06/2008	Trans-Siberian Express
▶	Ferdinand	Magellan	2	3	01/02/2008	Amazon & Inca Adventure

Different kinds of join

Inner join

The examples you have seen so far are called simple joins, or **inner joins**. An inner join returns data **only from rows where there are matching values in both tables**.

SQL gives you another way of writing joins, for example:

```
SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages
INNER JOIN Tours ON Packages.packageID = Tours.packageID
```

packageID	name	adultprice	departuredate
1	Western Adventure	£1,499.00	01/03/2008
1	Western Adventure	£1,499.00	05/06/2008
1	Western Adventure	£1,499.00	02/09/2008
2	Roof of the World Explorer	£1,599.00	01/03/2008
2	Roof of the World Explorer	£1,599.00	05/06/2008
3	Alpine Action	£899.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	02/09/2008
5	Trans-Siberian Express	£1,199.00	01/03/2008
5	Trans-Siberian Express	£1,199.00	01/06/2008
5	Trans-Siberian Express	£1,199.00	01/08/2008
6	Borneo Explorer	£1,699.00	08/03/2008
7	Amazon & Inca Adventure	£1,999.00	01/02/2008
7	Amazon & Inca Adventure	£1,999.00	01/08/2008
8	Patagonia Trek	£1,899.00	01/05/2008
9	Colorado Winter Adventure	£1,099.00	01/02/2008
9	Colorado Winter Adventure	£1,099.00	01/03/2008
11	Raft the Grand Canyon	£799.00	01/05/2008
11	Raft the Grand Canyon	£799.00	01/06/2008
11	Raft the Grand Canyon	£799.00	01/07/2008
11	Raft the Grand Canyon	£799.00	01/08/2008
▶	12 Rising Sun Explorer	£1,399.00	01/07/2008

The result is exactly the same as the first join query we looked at.

You can still apply conditions in addition to the join:

```
SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages
INNER JOIN Tours ON Packages.packageID = Tours.packageID
WHERE location = 'Asia'
```

packageID	name	adultprice	departuredate
2	Roof of the World Explorer	£1,599.00	01/03/2008
2	Roof of the World Explorer	£1,599.00	05/06/2008
5	Trans-Siberian Express	£1,199.00	01/03/2008
5	Trans-Siberian Express	£1,199.00	01/06/2008
5	Trans-Siberian Express	£1,199.00	01/08/2008
6	Borneo Explorer	£1,699.00	08/03/2008
12	Rising Sun Explorer	£1,399.00	01/07/2008

Outer join

If you look at the results of the inner join query carefully you will see that there is no row with **packageID = 10**. However, there *is* a package in the *Packages* table with that ID. Look at the results of the following query, which uses an **outer join**:

```
SELECT Packages.packageID, name, adultprice, departuredate
FROM Packages
LEFT OUTER JOIN Tours ON Packages.packageID = Tours.packageID
```

packageID	name	adultprice	departuredate
1	Western Adventure	£1,499.00	01/03/2008
1	Western Adventure	£1,499.00	05/06/2008
1	Western Adventure	£1,499.00	02/09/2008
2	Roof of the World Explorer	£1,599.00	01/03/2008
2	Roof of the World Explorer	£1,599.00	05/06/2008
3	Alpine Action	£899.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	01/03/2008
4	Reef and Outback Adventure	£2,199.00	02/09/2008
5	Trans-Siberian Express	£1,199.00	01/03/2008
5	Trans-Siberian Express	£1,199.00	01/06/2008
5	Trans-Siberian Express	£1,199.00	01/08/2008
6	Borneo Explorer	£1,699.00	08/03/2008
7	Amazon & Inca Adventure	£1,999.00	01/02/2008
7	Amazon & Inca Adventure	£1,999.00	01/08/2008
8	Patagonia Trek	£1,899.00	01/05/2008
9	Colorado Winter Adventure	£1,099.00	01/02/2008
9	Colorado Winter Adventure	£1,099.00	01/03/2008
10	Glacier Expedition	£299.00	NULL
11	Raft the Grand Canyon	£799.00	01/05/2008
11	Raft the Grand Canyon	£799.00	01/06/2008
11	Raft the Grand Canyon	£799.00	01/07/2008
11	Raft the Grand Canyon	£799.00	01/08/2008
12	Rising Sun Explorer	£1,399.00	01/07/2008

A **left outer join** returns data from **all the rows** of the first table and only the rows of the second table which have matching values.

So in this example, the highlighted row has data from *Packages* and a **NULL** for *departuredate* as there is **no matching row** in *Tours* for package 10.

Is this useful? Well, it shows clearly that there is a package for which there are no tours scheduled, which the inner join version did not do. The choice of which query to use would depend on the exact requirements in the use case.

What effect do you think the following would have?

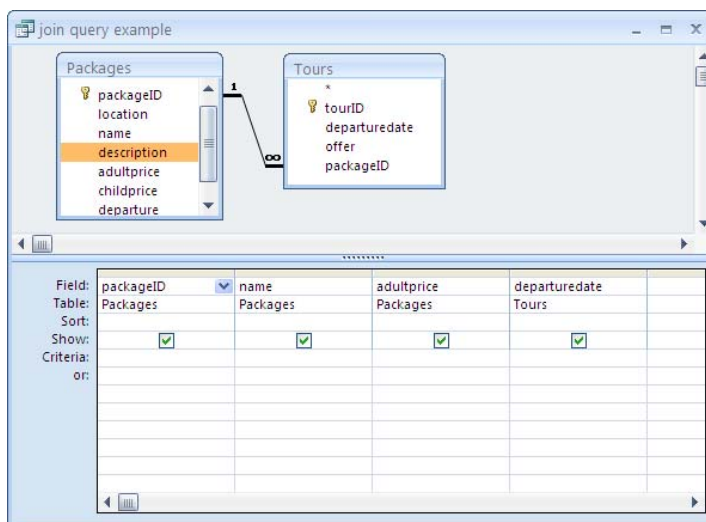
RIGHT OUTER JOIN Tours ON Packages.packageID = Tours.packageID

FULL OUTER JOIN Tours ON Packages.packageID = Tours.packageID

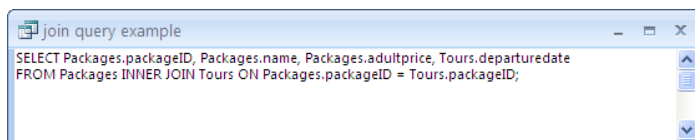
(Access doesn't support the second one, but some RDBMSs do)

Joining tables with the Access query builder

You can create a join query in Access in query design view by choosing more than one table. If the tables are properly related Access will realise this and create join conditions automatically.



Switching to SQL view shows that Access has generated an inner join by default, using the **INNER JOIN** key word.



You can change the join type using the **Join Properties** window. What kind of join has been chosen in the figure below?



Data definition queries

You have already seen several examples of data definition queries. Data definition queries include

- **CREATE TABLE**
- **ALTER TABLE**
- **DROP TABLE**
- **CREATE INDEX**

This part of SQL is sometimes known as **DDL (Data Definition Language)**.

Insert queries

You have also seen an example of an INSERT query, or Append query as it is called by Access. Here it is again to remind you.

```
INSERT INTO Users  
(firstname,lastname,address,username,password,datejoined)  
VALUES  
( 'Ferdinand', 'Magellan', '1520 Pacific Heights', 'ferdy', 'pwd',  
'29/8/2007' )
```

NOTE

A single INSERT query can only apply to **one table**. The same is true for updating and deleting

Update queries

An UPDATE query allows you to change data that is in a table. For example, the *Edit booking* use case may need a query that lets you change the number of *adults* and *children* in the booking. Such a query might look like this:

```
UPDATE Bookings
SET adults = 3, children = 5
WHERE bookingID = 1
```

This example updates the single row with *bookingID* = 1. You can update **many rows**, or even all the rows in a table, at the same time.

Increase the adult price of all packages in Asia by £100

```
UPDATE Packages
SET adultprice = adultprice + 100.0
WHERE location = 'Asia'
```

Delete queries

A DELETE query allows you remove a row or rows from a table.

Delete user with username vdagama from the Users table

```
DELETE FROM Users
WHERE username = 'vdagama'
```

This seems straightforward, but it won't work! The reason is that there are bookings for this user in the *Bookings* table, highlighted in the figure below.

bookingID	tourID	username	adults	children	status
1	1	mpolo	2	2	2 tickets sent
2	5	mpolo	1	0	0 tickets not sent
3	14	mpolo	2	2	2 tickets not sent
4	4	vdagama	4	0	0 tickets sent
5	8	vdagama	2	0	0 tickets not sent
6	10	ferdy	2	3	3 tickets not sent
7	13	ferdy	2	3	3 tickets sent

If we deleted the user, then those bookings would have the value 'vdagama' in the username field, while there would **no longer be a matching value** in the *Users* table. The database will not allow this since there is a foreign key which ensures that each booking must be for a valid user.

There are **three ways** of making it possible to delete this user:

1. Delete all bookings for the user first

Run this query first:

```
DELETE FROM Bookings  
WHERE username = 'vdagama'
```

and then run the query which deletes the user.

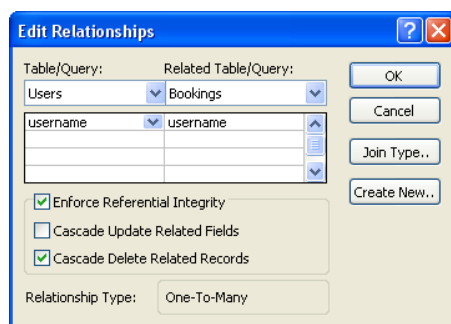
This is the safest option

2. Make deleting a user automatically delete all bookings for the user

This is called a **cascading delete**, and is an option you choose when you create the foreign key relationship between the tables. You should only choose this option if you are sure that it will not result in the loss of important data. You can set it up with SQL¹

```
FOREIGN KEY(username) REFERENCES Users(username) ON DELETE  
CASCADE
```

Or, you can select the **Cascade Delete Related Records** option in the Edit Relationships window. What do you think the Cascade Update Related Fields option will do?

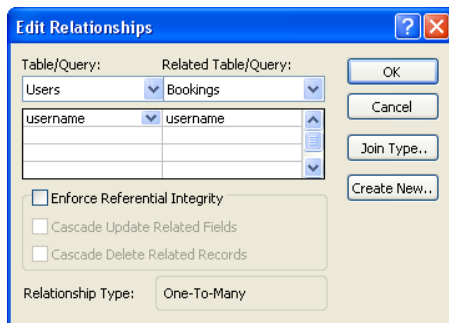


This option should be used with care

¹ Warning if you're trying this: in Access 2007 this statement will only work if you have selected the **SQL Server Compatible Syntax (ANSI 92)** option for **This Database** in the **Object Designers** tab of the **Access Options** dialogue (you can open this from the main Access menu)

3. Choose not to enforce referential integrity

You can do this simply by not creating a foreign key or by deselecting the **Enforce Referential Integrity** option. This means that there will be no check that a booking is for a valid user.



This option is not recommended

Parameter queries

One of the first queries we looked at was this:

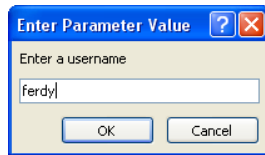
```
SELECT *  
FROM Users  
WHERE username = 'mpolo'
```

This query finds the user with username *mpolo*, and retrieves all his or her details. That sounds like something we might want to do again and again, but not always for the same user.

Instead of retyping the same query each time, you can make the value of username a **parameter** for the query. Every time you run the query you can enter a different value. The query needs to be modified slightly:

```
SELECT *  
FROM Users  
WHERE username = [Enter a username]
```

When you run the query, Access shows a prompt, and you choose what username you want to find:



	firstname	lastname	address	username	password	datejoined
	Ferdinand	Magellan	1520 Pacific Heights	ferdy	pwd	29/08/2007

NOTE

Access treats any word it doesn't recognise, as either a key word or as a valid table or field name, as a parameter. If you see a prompt when you don't expect to, it usually means you have misspelled a table or field name in your query.

The **[brackets]** are there to enclose a phrase which contains spaces. They are **nothing to do with** the fact that we are specifying a parameter.

Parameters are particularly useful if you **save queries** so that they can be opened and run again at any time.

Here's another example of a query (an update query this time) rewritten to use parameters:

```
UPDATE Bookings
SET adults = [Number of adults],
    children = [Number of children]
WHERE bookingID = [Booking ID]
```

In this example, you type in the new values to be stored in the updated row as well as the ID value of the row to be updated.

If you run this query, you'll be prompted for the parameters in the **order they appear in the query**, with *Booking ID* last. It seems sensible to specify the booking first, then enter the new values. You can set the order of parameters in an additional **PARAMETERS** statement with a list of parameter names and their data types. Note this query is actually two SQL statements, separated by a semicolon.

```
PARAMETERS [Booking ID] Long, [Number of adults] Long,
[Number of children] Long;
UPDATE Bookings
SET adults = [Number of adults], children = [Number of children]
WHERE bookingID = [Booking ID];
```

NOTE

Access can't normally run multiple SQL statements in one query. The PARAMETERS statement is an exception to this.

Most other RDBMSs can run scripts consisting of multiple SQL statements separated by semicolons, which can be very useful. You can set up and populate a whole database in one go with a script containing CREATE TABLE and INSERT INTO statements.