5: Doing more with queries

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	• **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	• **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
Abu Abdullah	Ibn Battuta	2 Silk Road	abu	pass
Amerigo	Vespucci	1499 America A...	amerigo	pass
Bartolemeu	Dias	1481 Gold Coast...	bart	passwd
Jacques	Cartier	156 Canada Cre...	cartier	pwrd
Christopher	Columbus	1492 America A...	chris	*NULL*
David	Livingstone	1852 Victoria Falls	dave	passwd
Ferdinand	Magellan	1520 Pacific Hei...	ferdy	pwd
Francisco	Pizarro	12 Lima Lane	frankie	password
Francisco	Vasquez de Cor...	98 Arizona Avenue	frankie2	passwd
Freya	Stark	19 Hadhramaut ...	freya	password
Gaspar	Corte Real	23 Terra Verde ...	gaspar	*NULL*

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**.

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?

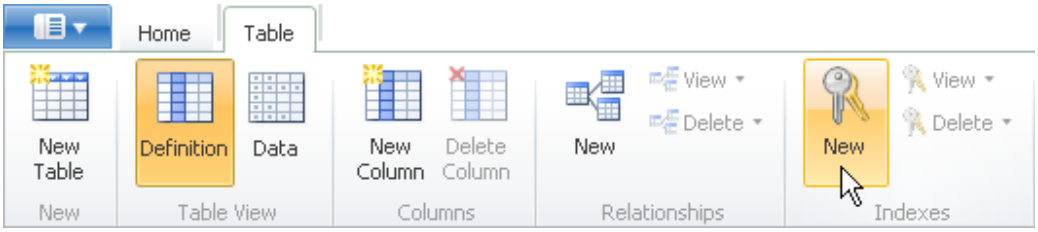
We've said that indexes speed up queries – so, if all the fields are indexed then all queries should run faster, shouldn't they? Well, they might, but the problem then is the amount of additional storage space which would be needed to hold all the indexes. This could be a major problem for a large database.

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

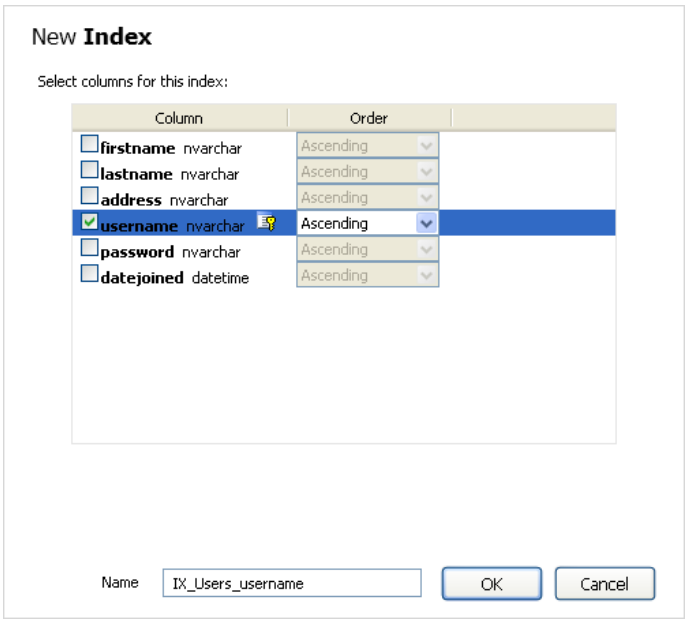
One of the decisions which we must take when designing a database is how to choose indexes carefully to speed up common queries without having too many indexes.

Creating indexes

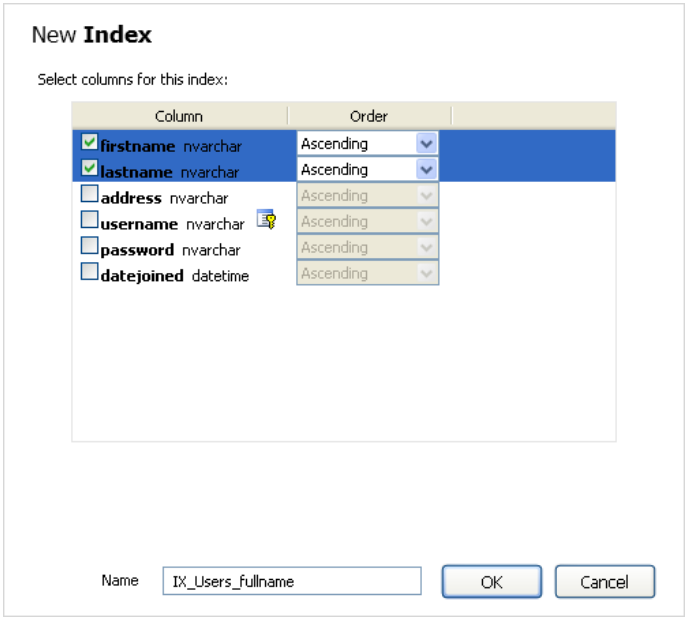
You can create an index for a table in Web Matrix when the table is open in Table Definition view. Select the **New Indexes** icon on the Table ribbon.



You can then select the index field in the **New Index** window.



To create a **compound index** you can select more than one field for the index. In the figure below, the index called *IX_Users_fullname* includes *lastname* and *firstname* fields.



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

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

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

```
DROP INDEX Users.IX_Users_fullname
```

Joining tables

What query will the *Show tours* use case need? We might want to show the **packagename**, **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	packagename	description	adultprice	childprice	departure	sales
1	USA	Western Advent...	A typical tour is ...	1499	999	Glasgow	314
2	Asia	Roof of the Worl...	New this year is ...	1599	1099	London Gatwick	126
3	Europe	Alpine Action	There is advent...	899	549	Glasgow	789
4	Australia	Reef and Outba...	There is no sho...	2199	1749	Manchester	223
5	Asia	Trans-Siberian E...	Experience the ...	1199	799	London Heathrow	188
6	Asia	Borneo Adventure	A 15 day safari ...	1699	1299	Manchester	254
7	South America	Amazon & Inca ...	An Amazon voy...	1999	1499	London Heathrow	433
8	South America	Patagonia Trek	The southern re...	1899	1399	Glasgow	121

foreign key : packageID

Tours

tourID	departuredate	offer	packageID
1	01/03/2011 00:...	15	1
2	05/06/2011 00:...	0	1
3	02/09/2011 00:...	10	1
4	02/09/2011 00:...	10	1
5	01/03/2011 00:...	20	2
6	05/06/2011 00:...	0	2
7	01/03/2011 00:...	25	3

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 **'01/03/2011'** 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 **1599** for package 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, packagename, adultprice, departuredate
FROM Packages, Tours
WHERE Packages.packageID = Tours.packageID
```

packageID	packagename	adultprice	departuredate
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	02/09/2011 00:...
5	Trans-Siberian E...	1199	01/03/2011 00:...
5	Trans-Siberian E...	1199	01/06/2011 00:...
5	Trans-Siberian E...	1199	01/08/2011 00:...
6	Borneo Adventure	1699	08/03/2011 00:...
7	Amazon & Inca ...	1999	01/02/2011 00:...
7	Amazon & Inca ...	1999	01/08/2011 00:...
8	Patagonia Trek	1899	01/05/2011 00:...
9	Colorado Winter...	1099	01/02/2011 00:...
9	Colorado Winter...	1099	01/03/2011 00:...
11	Raft the Grand ...	799	01/05/2011 00:...
11	Raft the Grand ...	799	01/06/2011 00:...
11	Raft the Grand ...	799	01/07/2011 00:...
11	Raft the Grand ...	799	01/08/2011 00:...
12	Rising Sun Explorer	1399	01/07/2011 00:...

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.packagename*, *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, packagename, adultprice, departuredate
FROM Packages, Tours
WHERE Packages.packageID = Tours.packageID
AND location = 'Asia'
```

packageID	packagename	adultprice	departuredate
2	Roof of the Worl...	1599	01/03/2011 00:...
2	Roof of the Worl...	1599	05/06/2011 00:...
5	Trans-Siberian E...	1199	01/03/2011 00:...
5	Trans-Siberian E...	1199	01/06/2011 00:...
5	Trans-Siberian E...	1199	01/08/2011 00:...
6	Borneo Adventure	1699	08/03/2011 00:...
12	Rising Sun Explorer	1399	01/07/2011 00:...

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, packagename, 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 26, so the query gives (12x26) = **312 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	packagename
Marco	Polo	2	2	01/03/2011 00:...	Western Advent...
Marco	Polo	1	0	01/03/2011 00:...	Roof of the Worl...
Marco	Polo	2	2	01/06/2011 00:...	Trans-Siberian E...
Vasco	daGama	4	0	02/09/2011 00:...	Western Advent...
Vasco	daGama	2	0	01/03/2011 00:...	Alpine Action
Ferdinand	Magellan	2	3	01/03/2011 00:...	Reef and Outba...

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

packageID	packagename	adultprice	departuredate
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	02/09/2011 00:...
5	Trans-Siberian E...	1199	01/03/2011 00:...
5	Trans-Siberian E...	1199	01/06/2011 00:...
5	Trans-Siberian E...	1199	01/08/2011 00:...
6	Borneo Adventure	1699	08/03/2011 00:...
7	Amazon & Inca ...	1999	01/02/2011 00:...
7	Amazon & Inca ...	1999	01/08/2011 00:...
8	Patagonia Trek	1899	01/05/2011 00:...
9	Colorado Winter...	1099	01/02/2011 00:...
9	Colorado Winter...	1099	01/03/2011 00:...
11	Raft the Grand ...	799	01/05/2011 00:...
11	Raft the Grand ...	799	01/06/2011 00:...
11	Raft the Grand ...	799	01/07/2011 00:...
11	Raft the Grand ...	799	01/08/2011 00:...
12	Rising Sun Explorer	1399	01/07/2011 00:...

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

packageID	packagename	adultprice	departuredate
2	Roof of the Worl...	1599	01/03/2011 00:...
2	Roof of the Worl...	1599	05/06/2011 00:...
5	Trans-Siberian E...	1199	01/03/2011 00:...
5	Trans-Siberian E...	1199	01/06/2011 00:...
5	Trans-Siberian E...	1199	01/08/2011 00:...
6	Borneo Adventure	1699	08/03/2011 00:...
12	Rising Sun Explorer	1399	01/07/2011 00:...

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

packageID	packagename	adultprice	departuredate
1	Western Advent...	1499	02/09/2011 00:...
2	Roof of the Worl...	1599	01/03/2011 00:...
2	Roof of the Worl...	1599	05/06/2011 00:...
3	Alpine Action	899	01/03/2011 00:...
3	Alpine Action	899	01/03/2011 00:...
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	01/03/2011 00:...
4	Reef and Outba...	2199	02/09/2011 00:...
5	Trans-Siberian E...	1199	01/03/2011 00:...
5	Trans-Siberian E...	1199	01/06/2011 00:...
5	Trans-Siberian E...	1199	01/08/2011 00:...
6	Borneo Adventure	1699	08/03/2011 00:...
7	Amazon & Inca ...	1999	01/02/2011 00:...
7	Amazon & Inca ...	1999	01/08/2011 00:...
8	Patagonia Trek	1899	01/05/2011 00:...
9	Colorado Winter...	1099	01/02/2011 00:...
9	Colorado Winter...	1099	01/03/2011 00:...
10	Glacier Expedition	2990	
11	Raft the Grand ...	799	01/05/2011 00:...
11	Raft the Grand ...	799	01/06/2011 00:...
11	Raft the Grand ...	799	01/07/2011 00:...
11	Raft the Grand ...	799	01/08/2011 00:...
12	Rising Sun Explorer	1399	01/07/2011 00:...

NULL

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
```

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. Here it is again to remind you.

```
INSERT INTO Users
(firstname,lastname,address,username,password,datejoined)
VALUES
('Ferdinand', 'Magellan', '1520 Pacific Heights', 'ferdy', 'pwd',
'2007-08-29')
```

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	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

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 **two 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 safer 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 using SQL when you define the foreign key. The following could be part of a **CREATE TABLE** or **ALTER TABLE** statement.

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

This option should be used with care

Parameterized 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.

You can make the value of username a **parameter** for the query. Every time the query is executed, a different value can be supplied for the parameter. The query needs to be modified slightly:

```
SELECT *
FROM Users
WHERE username = @username
```

The `@` symbol indicates that the value is to be treated as a parameter, not an actual value.

You can't actually execute this query directly in a WebMatrix query window as there is no way to supply a parameter value. You will see in the next chapter how parameterized queries help when building a data-driven web application with WebMatrix.