CHAPTER 14

Indexes

We've already met indexes during our database travels to date. We were creating clustered indexes when we created primary keys, and we also created unique indexes when we created unique constraints.

Indexes are important because they can greatly affect how well your database performs. The benefits of an index become apparent once your database contains a certain number of rows; it's possible to reduce queries that can take 20 minutes (in some cases) to just two seconds. That's how important they are.

We'll take a tour of the various types of index you can create, look at some examples, and see how to create an indexed view. This is a big subject, so hold on to your hat!

What Is an Index?

A database index is not particularly different from an index in a book. Just as you might use a book index to lookup a particular word or phrase (e.g., `CREATE VIEW`), SQL Server uses an index to quickly find records based on search criteria you supply to it.

Think what happens if you ask SQL Server to bring back all contacts who are developers. If there is an index on the `RoleTitle` column, SQL Server can use the index to find the rows that match the search criteria. If no index exists, SQL Server has to inspect every single row to find a match.

Why Are Indexes Useful?

Whenever it executes a query, SQL Server uses something called the *Query Optimizer*. This is a built-in component of SQL Server that takes the T-SQL code you provide and figures out the fastest way of executing it. Indexes can help the Query Optimizer decide on the best path to take. This often helps to avoid disk input/output operations (disk I/O)—operations in which data has to be read from disk instead of memory.

Disk I/O operations are expensive from a time-taken perspective, as disks are slower to access than memory. Indexes help to reduce disk I/O, as SQL Server can access the data it needs with fewer steps, especially as the data in the index is sorted according to the indexed columns, resulting in faster lookups.

What Do Indexes Affect?

Indexes affect `SELECT`, `UPDATE`, and `DELETE` statements. They also affect the `MERGE` statement, which I won't cover in this book. Anything that uses joins or `WHERE` conditions might benefit from an index.

Identifying Which Columns to Index

Quite often you'll come to a database and find a lot of indexes have been created that make no sense. While I could probably write a thesis on all of the ways you can identify what you should index, and how to develop a good indexing strategy, there are actually a couple of basic rules of thumb that you can apply to determine whether a column should be included in an index or not.

We'll use this statement as a basis for our discussion:

```
SELECT C.ContactId, C.FirstName, C.LastName, C.DateOfBirth, PNT.PhoneNumberType, CPN.PhoneNumber FR
```

The basic steps to index identification are:

- Write or obtain the SQL statements that will be used to return data from the database. This includes any `UPDATE` or `DELETE` statements that use joins or `WHERE` conditions.
- Once you have the statements, make a note of the columns used in joins. Contrary to popular belief, a foreign key column is *not* automatically included in an index. (I've been told this by many non-SQL

developers down the years—it isn't true!)

- Look for any `WHERE` conditions used by the queries and identify the columns used by such queries.
- Finally, make a note of the columns returned by the `SELECT` statements.

The statement returns six columns. We join on:

- `Contacts.ContactId` (this is the primary key and is clustered),
- `ContactPhoneNumbers.ContactId` (a non-indexed foreign key),
- `ContactPhoneNumbers.PhoneNumberTypeId` (a non-indexed foreign key), and
- `PhoneNumberTypes.PhoneNumberTypeId` (a clustered primary key).

Finally, the `WHERE` clause uses two conditions, each of which uses a different column:

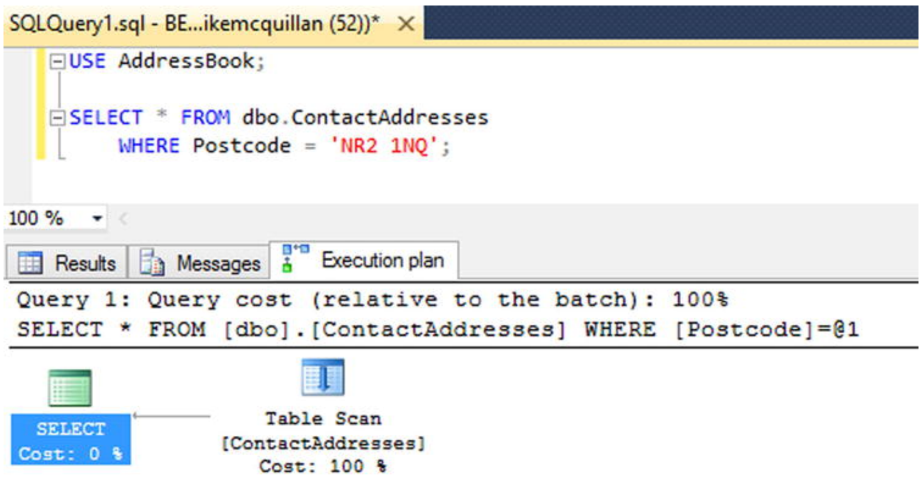
- `Contacts.DateOfBirth` and
- `PhoneNumberTypes.PhoneNumberType`

An index can only apply to one table. You cannot spread an index over two tables (Indexed Views offer a kind of workaround to this problem, as we'll see later). This means you cannot create a single index to optimize the preceding query—you'll have to create appropriate indexes on all of the tables involved in the query.

How Indexes Work

I've already mentioned why indexes are useful. Before we create any indexes, we'll take a glance at how they work. This will help you understand why indexes help queries run faster.

Imagine we have 30 rows in a table, and we write a query that will cause row 17 to be returned. Without an index, SQL Server finds the row we are interested in by executing something called a *Table Scan* (Figure 14-1).



```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses
    WHERE Postcode = 'NR2 1NQ';
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[ContactAddresses] WHERE [Postcode]=@1
```

Figure 14-1. *Executing a query with a Table Scan*

Figure 14-1 shows something called an *Execution Plan*. You can turn these on to help you determine if SQL Server is executing your queries in the most efficient manner. *Table Scans are bad, and most definitely* not *efficient*. To turn execution plans on, click the **Include Actual Execution Plan** option in the **Query** menu of SSMS (or press Ctrl+M). The **Execution Plan** tab will appear after your query completes.

Table Scans

Why is a table scan bad? Because they mean *every single row in your table is being interrogated*. Our query looks for a particular postcode, which can be found in row 17 of 30. To find that row, SQL Server inspected the first 16 rows before finding the match. It marked row 17 as a match, then carried on inspecting rows 18 to 30 to check if any of those matched. This isn't too bad when only 30 rows exist, but imagine if a million rows existe

B-Trees

Indexes offer a much more efficient method of finding data by using something called a *B-Tree*, more formally known as a *Balanced Tree*. This is a structure that causes data to be split into different levels, allowing for very fast data querying. The B-Tree for a clustered index consists of three levels. Figure 14-2 shows how the levels are structured.

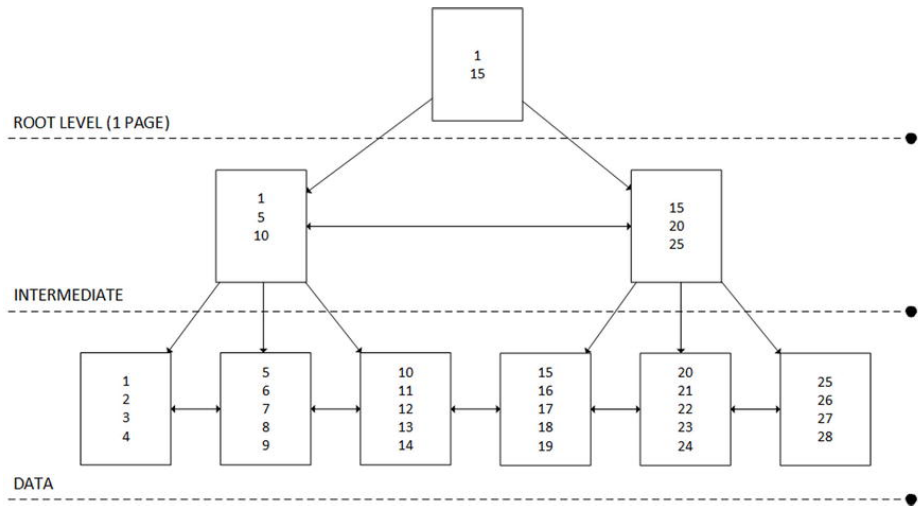
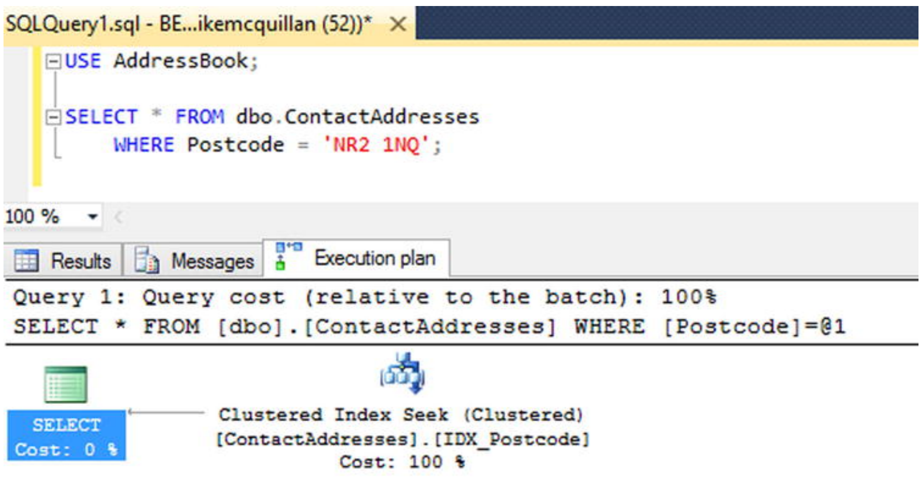


Figure 14-2. *A clustered index B-Tree*

There is only ever one root level, and it only ever contains one page. There can be multiple intermediate levels, depending upon the amount of data the index holds. Finally, there is only ever one data level, and the pages in this level hold the actual data, sorted as per the clustered index.

You are probably wondering what these pages I've mentioned are. I'm not going to delve deeply into how SQL Server structures its data in this book, but here's a quick overview. Do you remember how a SQL Server database can consist of one or more files? Tables may be contained within one of those files, or spread across multiple files. Each file is split into *extents*. You could think of an extent as a folder. Each extent contains eight *pages*. Pages hold rows. If an extent is a folder, a page is a sheet of paper, and a row is a line on that sheet of paper. These pages that hold the data rows are what we are talking about with regards to indexes.

Now, let's say we've added a clustered index on the `Postcode` column, and we're again looking for postcode NR2 1NQ. Figure 14-3 has the execution plan.



```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses
    WHERE Postcode = 'NR2 1NQ';
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[ContactAddresses] WHERE [Postcode]=@1
```

```
SELECT          Clustered Index Seek (Clustered)
Cost: 0 %       [ContactAddresses].[IDX_Postcode]
                Cost: 100 %
```

Figure 14-3. *Running the same query after adding a clustered index*

Now we have a Clustered Index Seek instead of a Table Scan (you may see a Clustered Index Scan—don't worry about it if you do). Is this any better? Let's see. Data is now ordered by the postcode. For simplicity, we'll assume the row we are interested in is still at position 17. We'll also assume the data is ordered using numbe so we can use our earlier diagram (in reality, it would be sorted by postcode). Here's what SQL Server will do

- Start at the root page and look at the value of the record at the start of the page. We have the value 1. The next value is 15. Seventeen is not between 1 and 15, so the index navigates to the page in the intermediate level that begins with value 15.
- The same process occurs. The start value is 15 and the next value is 20. Seventeen is between these two values, so the search now drops down to the data page level.
- The data page contains the index identifier for row 17. As this is a clustered index, it also holds the data. We're done!

We only had to navigate three times to find the value using the clustered index, rather than checking all 30 rows to see if they matched. Much more efficient, especially when you consider what could happen if the table contained many more rows.

This is a much-simplified explanation of how indexes work, but it should demonstrate that indexes can be very effective when used correctly.

Non-Clustered Indexes And B-Trees

Non-clustered indexes work in a very similar manner to clustered indexes, except the data is not stored in a sorted order, and indeed is not stored on the data pages. The data pages in a non-clustered index store an identifier that points at the row containing the data. So once the B-Tree has done its work and found the correct data page, there is an additional step as the index hops over to the actual row to retrieve the columns you have requested. You can see this extra step in Figure 14-4.

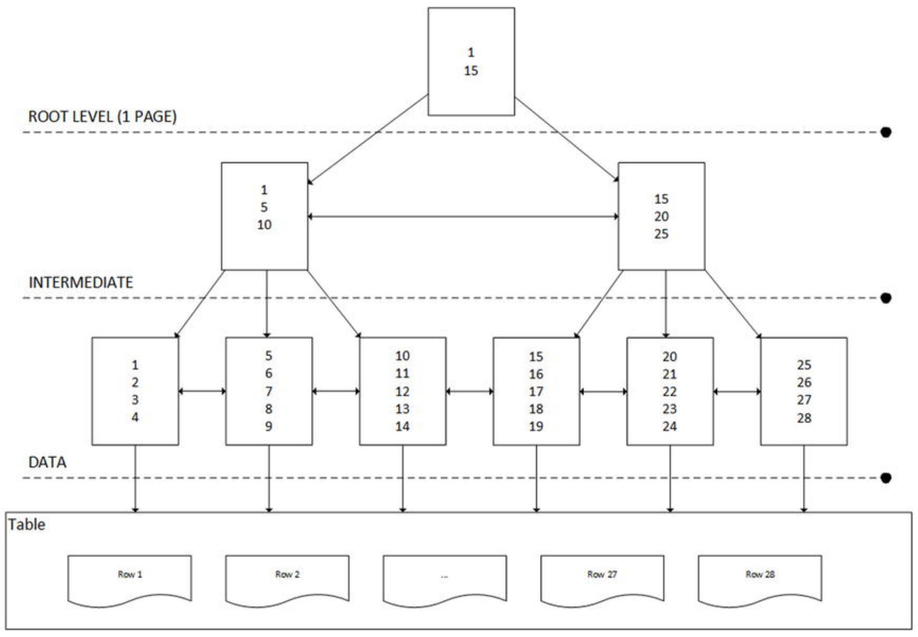


Figure 14-4. *A non-clustered index B-Tree*

Included columns can work around this; you can include columns in an index and the values for those columns will be stored right inside the index. This can avoid the additional step required for non-clustered indexes if all of the relevant columns are available. We'll see how this works in a few pages' time, in the section "Indexed Columns vs. Included Columns".

Basics Of The CREATE INDEX Statement

Rather unsurprisingly, the `CREATE INDEX` statement is used to create indexes. The basic structure of this command is:

```
CREATE INDEX IndexName ON TableName (Columns);
```

To create a clustered index you'd write:

```
CREATE CLUSTERED INDEX IndexName ON TableName (Columns);
```

Creating non-clustered indexes is very similar:

```
CREATE NONCLUSTERED INDEX IndexName ON TableName (Columns);
```

Including additional columns with the index is as simple as adding the `INCLUDES` keyword:

```
CREATE INDEX IndexName ON TableName (Columns) INCLUDE (Columns);
```

You can also create something called a *filtered index*, which works on a subset of data (more to come on this, in the section Filtered Indexes). The command to create a filtered index is:

```
CREATE INDEX IndexName ON TableName (Columns) WHERE (Conditions);
```

We'll delve into all these commands starting right now.

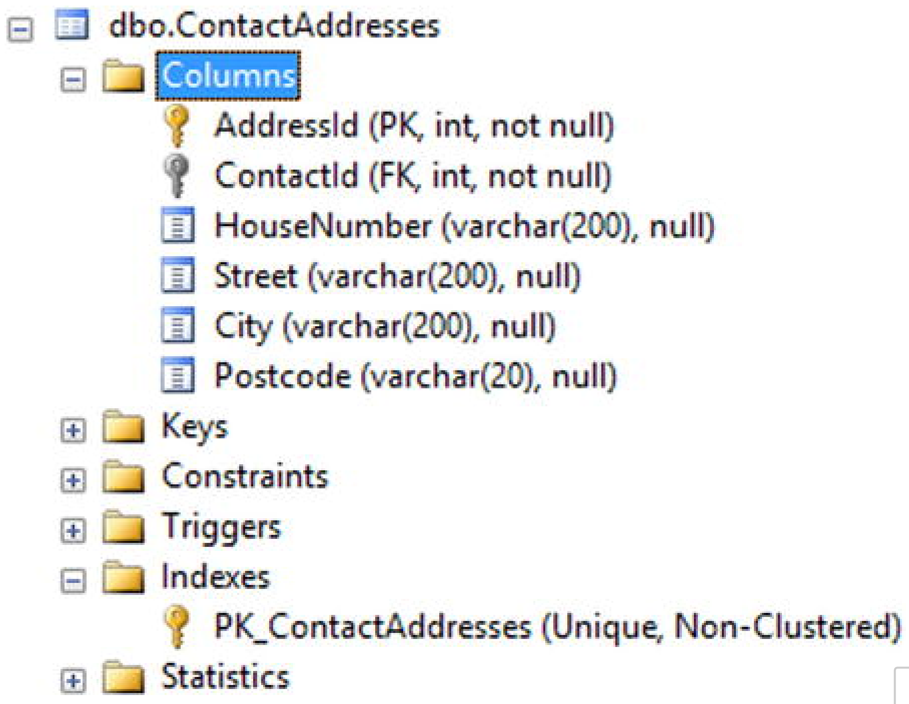
Clustered Indexes

A clustered index dictates how the data in a table is sorted on disk. As it's only possible to sort data on disk in one particular way, you can only have one clustered index per table. Clustered indexes are often the most performant kind of index because the data is returned as soon as it is located by the index. This is because the data is stored with the index.

To determine how a clustered index works, think of a telephone book (if this seems too old-fashioned for you, think of the Contacts app on your mobile phone). Say you want to look up Grace McQuillan's phone number. You open your phone book (or app) and scan for Grace McQuillan. All data is stored alphabetically. As soon as you find Grace McQuillan you have access to her phone number. The data was there as soon as you located Grace McQuillan. This is different to a non-clustered index, which will tell you where to locate the data.

Creating a Clustered Index

Most of the tables in our **AddressBook** database already have clustered indexes—we created all tables with clustered primary keys—so we cannot create additional clustered indexes on those tables. However, there is one exception: the ContactAddresses table, shown in Figure 14-5. This was created with a non-clustered primary key.

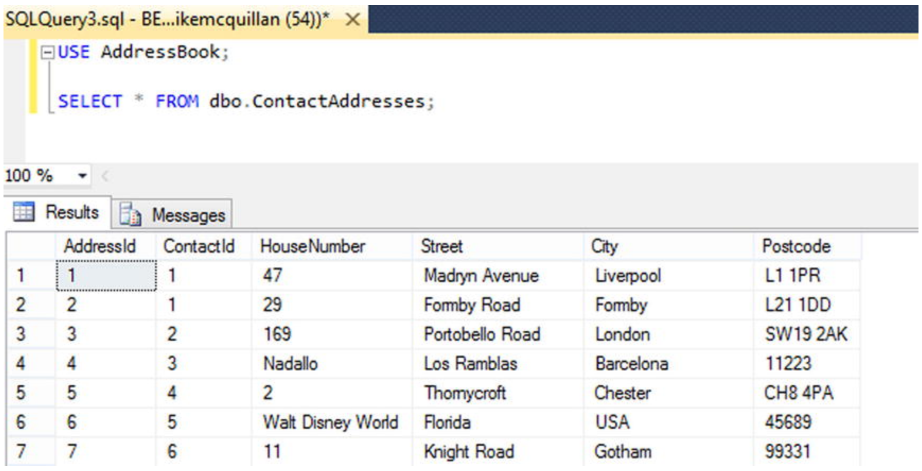


```
dbo.ContactAddresses
  Columns
    AddressId (PK, int, not null)
    ContactId (FK, int, not null)
    HouseNumber (varchar(200), null)
    Street (varchar(200), null)
    City (varchar(200), null)
    Postcode (varchar(20), null)
  Keys
  Constraints
  Triggers
  Indexes
    PK_ContactAddresses (Unique, Non-Clustered)
  Statistics
```

Figure 14-5. *`ContactAddresses` with a non-clustered primary key*

Why did we do this? The primary key on this table is `AddressId`. This column exists purely to give a unique, fast primary key to the table. As a piece of data the users are interested in, it is utterly irrelevant. Users will never see it and we are unlikely to use it to find addresses; we are more likely to use the `ContactId` or the `Postcode` when searching for addresses. It therefore makes infinitely more sense to have the data sorted by these columns—lookups will be much faster.

Before creating the index, open up a New Query Window and execute the T-SQL shown in Figure 14-6. We can see that the data is sorted by `AddressId`.



SQLQuery3.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses;
```

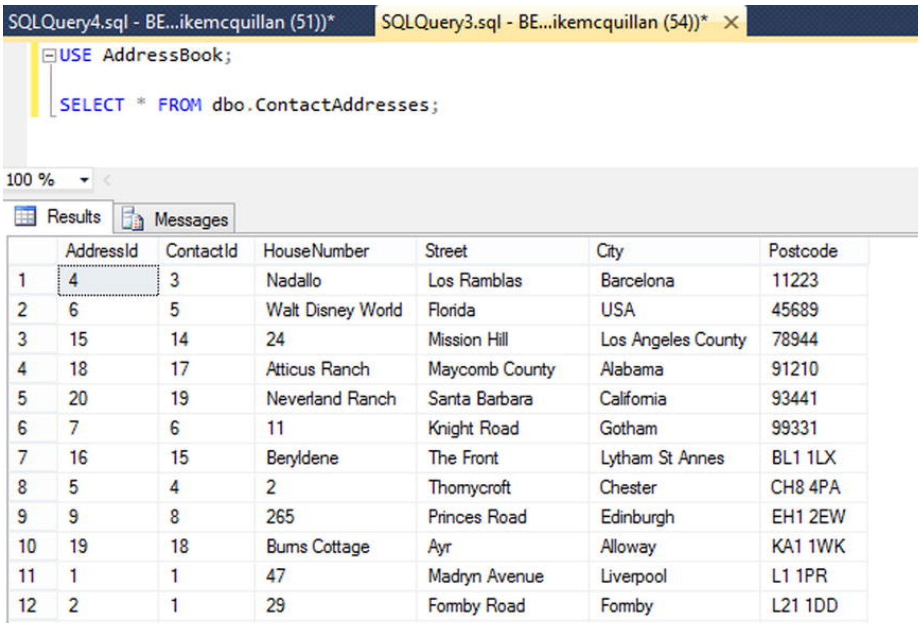
	AddressId	ContactId	HouseNumber	Street	City	Postcode
1	1	1	47	Madryn Avenue	Liverpool	L1 1PR
2	2	1	29	Formby Road	Formby	L21 1DD
3	3	2	169	Portobello Road	London	SW19 2AK
4	4	3	Nadallo	Los Ramblas	Barcelona	11223
5	5	4	2	Thornycroft	Chester	CH8 4PA
6	6	5	Walt Disney World	Florida	USA	45689
7	7	6	11	Knight Road	Gotham	99331

Figure 14-6. *Query showing how the primary key orders data*

Open up another New Query Window and enter this script (don't run it yet).

```
USE AddressBook;
CREATE CLUSTERED INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses(Postcode, Co

GO
```

The statement is pretty simple. The `CLUSTERED` keyword informs SQL Server we want to create a clustered index—if this weren't present, a `NONCLUSTERED` index would be created by default. We've given the index a descriptive name, so future developers coming to our database can easily see what its purpose is. The `IX_C` at the start indicates the object is a clustered index. We then have the name of the table for which we are creating the index, followed by the columns we are indexing. Execute this script to create the clustered index, then return to the `SELECT * FROM dbo.ContactAddresses` script and run it again (Figure 14-7).



SQLQuery4.sql - BE...ikemcquillan (51))* SQLQuery3.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses;
```

	AddressId	ContactId	HouseNumber	Street	City	Postcode
1	4	3	Nadallo	Los Ramblas	Barcelona	11223
2	6	5	Walt Disney World	Florida	USA	45689
3	15	14	24	Mission Hill	Los Angeles County	78944
4	18	17	Atticus Ranch	Maycomb County	Alabama	91210
5	20	19	Neverland Ranch	Santa Barbara	California	93441
6	7	6	11	Knight Road	Gotham	99331
7	16	15	Beryldene	The Front	Lytham St Annes	BL1 1LX
8	5	4	2	Thornycroft	Chester	CH8 4PA
9	9	8	265	Princes Road	Edinburgh	EH1 2EW
10	19	18	Burns Cottage	Ayr	Alloway	KA1 1WK
11	1	1	47	Madryn Avenue	Liverpool	L1 1PR
12	2	1	29	Formby Road	Formby	L21 1DD

Figure 14-7. *The same query after adding a clustered index*

Wow, Figure 14-7 shows us that the order has changed somewhat! `AddressId`, which was neatly ordered earlier, is now all over the place. So is `ContactId`. If you look at the Postcode column you'll see it is nicely ordered. This makes sense, as it was the first column specified in our index.

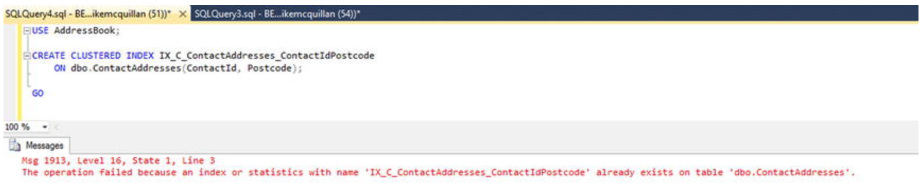
Take a look at rows 11 and 12—these are both addresses for `ContactId` 1, and they are ordered by `Postcode` in ascending order. At a glance, the data in Figure 14-7 looks oddly ordered, because the ID columns are jumbled up. But the rows are sorted according to our index. It's because of indexes like the one we've just implemented that query results sometimes seem to be arbitrarily ordered.

So we've created a clustered index. We can see the results of the index simply by executing a `SELECT *` statement. Do you think this is a good index? It's *almost* a good index. To figure out why, we need to assess how this table is likely to be used. This table is probably going to be used in two ways:

- Obtain address information when a contact is being viewed through an app of some description, so address details can be displayed alongside the contact details (this will use a join of some sort)
- Support searching for addresses via a search interface (e.g., searching by postcode)

Of these two use cases, the most common is probably going to be the first. The likely user interface will ask the operator to ask for the contact's name or date of birth, from which their record will be found and displayed on-screen. A join will then be executed to return the contact's addresses using the `ContactId`. It seems `ContactId` is likely to be used more than Postcode, so we'll make `ContactId` the first field in the index, with Postcode the second.

Return to the window containing your `CREATE INDEX` statement. Swap the field names around and execute the `CREATE INDEX` statement again. As Figure 14-8 tells us, it all goes wrong!



SQLQuery4.sql - BE...ikemcquillan (51))* SQLQuery3.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

CREATE CLUSTERED INDEX IX_C_ContactAddresses_ContactIdPostcode
    ON dbo.ContactAddresses(ContactId, Postcode);

GO
```

```
Msg 1913, Level 16, State 1, Line 3
The operation failed because an index or statistics with name 'IX_C_ContactAddresses_ContactIdPostcode' already exists on table 'dbo.ContactAddresses'.
```

Figure 14-8. *The index already exists*

We have a couple of options here.

- Modify the index using SSMS

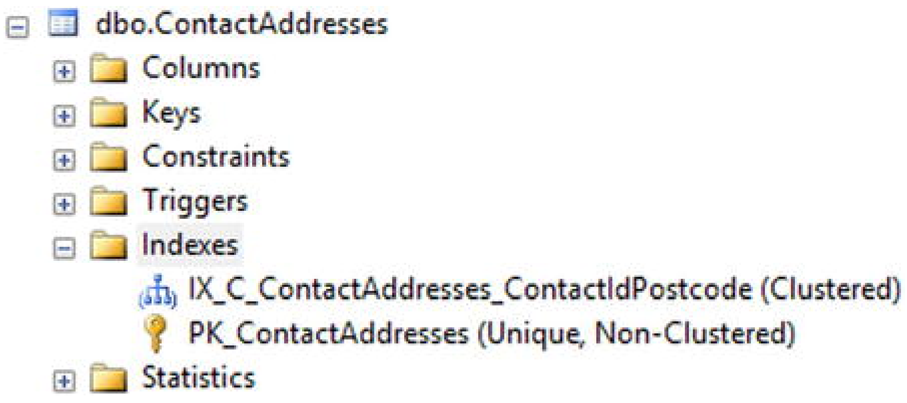
- Check if the index exists, drop it if it does, then recreate it

`ALTER INDEX` is not like other `ALTER` statements, as it doesn't allow modification of the index's definition. It is used to change index options, and we'll look at what it can do a bit later.

Modifying using SSMS isn't a good idea, as it breaks our principle of providing DBAs with a set of scripts they can execute on any environment. Still, let's take a look at what we could do here.

Modifying an Index Using SSMS

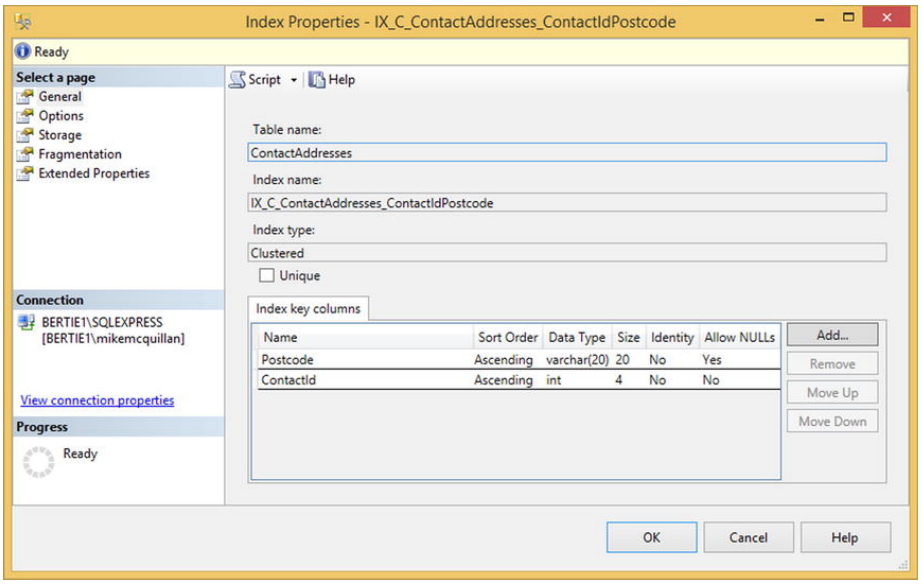
In the Object Explorer, right-click the `ContactAddresses` table and refresh it. Then expand the **Indexes** node. You should see the two indexes shown in Figure 14-9: our new index and the primary key index.



```
dbo.ContactAddresses
    Columns
    Keys
    Constraints
    Triggers
    Indexes
        IX_C_ContactAddresses_ContactIdPostcode (Clustered)
        PK_ContactAddresses (Unique, Non-Clustered)
    Statistics
```

Figure 14-9. *The indexes present for ContactAddresses*

Right-click **IX_C_ContactAddresses_ContactIdPostcode** and choose the **Properties** option from the context menu (alternatively, double-click the index name). The index properties dialog in Figure 14-10 opens up.



Index Properties - IX_C_ContactAddresses_ContactIdPostcode

Ready

Select a page: General, Options, Storage, Fragmentation, Extended Properties

Connection: BERTIE1\SQLEXPRESS [BERTIE1\mikemcquillan]

View connection properties

Progress: Ready

Script ▾ | Help

Table name: ContactAddresses

Index name: IX_C_ContactAddresses_ContactIdPostcode

Index type: Clustered

☐ Unique

Index key columns

Name	Sort Order	Data Type	Size	Identity	Allow NULLs
Postcode	Ascending	varchar(20)	20	No	Yes
ContactId	Ascending	int	4	No	No

Figure 14-10. *Index properties dialog*

Note the **Index key columns** section. You could use the **Move Up/Move Down** buttons to change the order of the index columns. There are other options on the right that allow you to customize various aspects of the index—we'll take a look at one or two of those using T-SQL later. You'd also see this screen if you chose to create a new index using SSMS.

It's worth pointing out that you can right-click an index in the Object Explorer and script it to a file or New Query Window, just as we did with tables and databases earlier.

For me, using SSMS involves a lot more work! Let's modify our script to check if the index already exists.

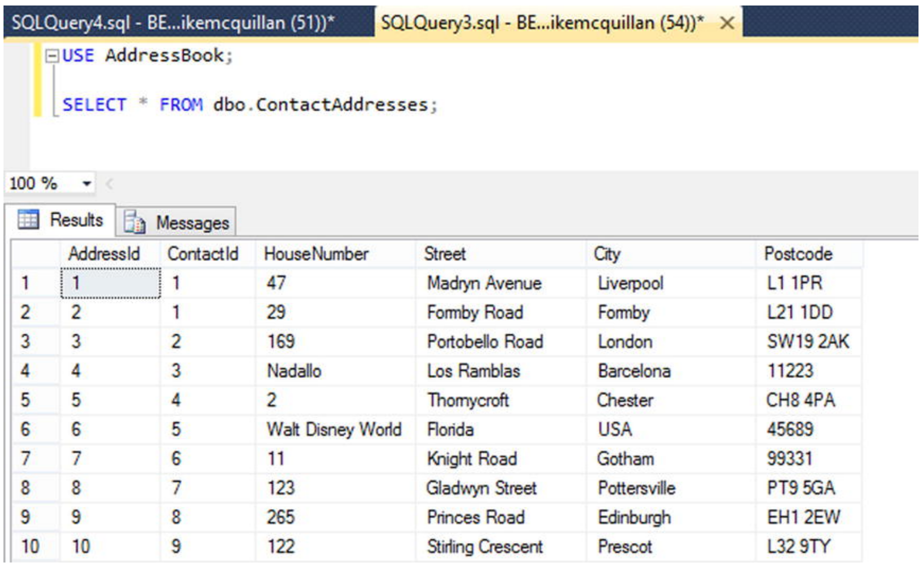
Checking If an Index Exists with T-SQL

Return to our `CREATE INDEX` script window. We'll add a check to see if the index exists, very similar to the checks we've already added for tables and views. We used `sys.tables` and `sys.views` to check if a table or view existed, so no prizes for guessing that the system table holding the indexes is called `sys.indexes`. Let's add that check!

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_C_ContactAddresses_ContactIdPostcode')
BEGIN
DROP INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses;
END;

CREATE CLUSTERED INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses(ContactId, P

GO
```

This should be pretty familiar to you now. We check if the index exists and drop it if it does, then we create the index. Make sure you check the last line carefully—we've switched the order of `ContactId` and `Postcode` around. The `DROP INDEX` statement is slightly different to other `DROP` statements we've met—we had to specify not only the index name, but the name of the table on which the index exists, too.

If you execute this script, it should run successfully. It should work no matter how many times you run it. Return to the `SELECT * FROM dbo.ContactAddresses` window and run the `SELECT` statement. The order has changed back to its original form, as you can see in Figure 14-11.


SQLQuery4.sql - BE...ikemcquillan (51))*
SQLQuery3.sql - BE...ikemcquillan (54))*
100 %
Results
Messages

```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses;
```

	AddressId	ContactId	HouseNumber	Street	City	Postcode
1	1	1	47	Madryn Avenue	Liverpool	L1 1PR
2	2	1	29	Formby Road	Formby	L21 1DD
3	3	2	169	Portobello Road	London	SW19 2AK
4	4	3	Nadallo	Los Ramblas	Barcelona	11223
5	5	4	2	Thornycroft	Chester	CH8 4PA
6	6	5	Walt Disney World	Florida	USA	45689
7	7	6	11	Knight Road	Gotham	99331
8	8	7	123	Gladwyn Street	Pottersville	PT9 5GA
9	9	8	265	Princes Road	Edinburgh	EH1 2EW
10	10	9	122	Stirling Crescent	Prescot	L32 9TY

Figure 14-11. *ContactAddresses with* `ContactId` *in the clustered index*

The fact that `AddressId` is in order is actually a coincidence—the table is now sorted by `ContactId`, then `Postcode`. If we added a new address for `ContactId` 1 it would appear in the top three rows, depending upon the postcode provided.

Save the index script as `c:\temp\sqlbasics\apply\21 - Create ContactAddresses Clustered Index.sql`. Then add a call to the bottom of the `00 - Apply.sql` script.

```
:setvar currentFile "21 - Create ContactAddresses Clustered Index.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Now that we have a handle on clustered indexes, we'll move on to creating some indexes of the non-clustered variety.

Non-Clustered Indexes

You can have pretty much all the non-clustered indexes you want on any table in your database—you can add up to 999 of them (in SQL Server 2008 and later you were previously limited to "only" 249). These are the indexes your queries will normally use. Clustered indexes are great, but they are limited because you can only have one of them. You also need to keep the key as small and efficient as possible, which can limit their effectiveness in queries.

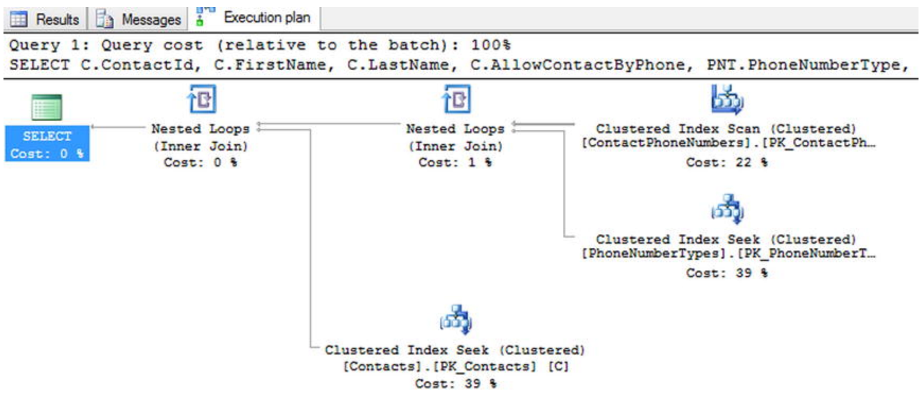
Non-clustered indexes can exist with or without a clustered index. A table will generally work better if a clustered index exists, as will its non-clustered indexes. But it isn't a deal breaker.

We've already seen how a non-clustered index differs from a clustered index, in that it doesn't dictate the order in which table data is stored. Rather, the keys in the index are stored in a separate structure, which is used to identify the rows we are interested in. Once the rows have been identified, a link held within the non-clustered index is used to obtain the appropriate data from the appropriate rows.

We'll use the phone number tables to demonstrate non-clustered indexes. We want to find all contacts who have a home phone number. Not a problem; we can write that query in a jiffy!

```
USE AddressBook;

SELECT C.ContactId, C.FirstName, C.LastName, C.AllowContactByPhone, PNT.PhoneNumberType, CPN.PhoneN
```

Running this returns four rows, and generates the execution plan you can see in Figure 14-12 (remember, Ctrl+M will toggle execution plans on/off):


Results
Messages
Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT C.ContactId, C.FirstName, C.LastName, C.AllowContactByPhone, PNT.PhoneNumberType,
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 1 %
Clustered Index Scan (Clustered)
[ContactPhoneNumbers].[PK_ContactPh...
Cost: 22 %
Clustered Index Seek (Clustered)
[PhoneNumberTypes].[PK_PhoneNumberT...
Cost: 39 %
Clustered Index Seek (Clustered)
[Contacts].[PK_Contacts] [C]
Cost: 39 %

Figure 14-12. *An execution plan, using an Index Scan for ordering*

We have two inner joins here: one for each join specified in our query. There is a Clustered Index Seek at the bottom of the plan. This is used to seek the `ContactId` values from the `Contacts` table. In the top right, we have a Clustered Index Scan over the `ContactId` column in the `ContactPhoneNumbers` table. This is because we configured the `PhoneNumberId` column as the clustered index when we created it as the primary key. In hindsight, this probably wasn't a good decision. The scan has been used because the column we are using in the join does not order the data.

Scans are used when the data is not ordered, and seeks are used when the data is ordered. The final seek is for the `PhoneNumberTypes` table, which is clustered on the `PhoneNumberTypeId` column.

CLUSTERED INDEX SEEKS AND SCANS

You are probably wondering what the difference between an index seek and scan is. A seek will use the B-Tree to locate the data it requires. It will use the search parameters provided to limit the number of pages it searc through.

A scan starts at the beginning of the index and moves through each row in order, pulling out matching rows as it finds them. Every row in the index is scanned.

You are no doubt thinking seeks are more efficient than scans, and a lot of the time you would be right. But there are instances where a scan will outperform a seek. Further complications arise by considering that a seek sometimes contains a scan!

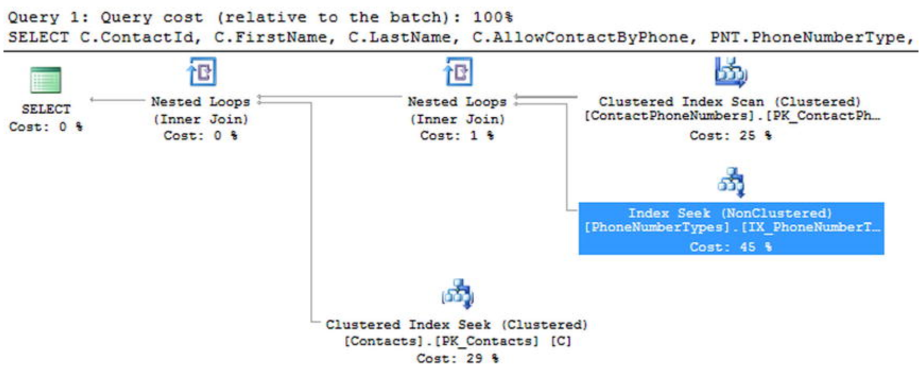
The general rule to follow is that seeks are usually better for fairly straightforward queries (e.g., queries using `JOIN`s and `WHERE` clauses), while more complicated queries may benefit from the use of a scan. If in doubt, play around with your indexes until you are satisfied with the performance level.

Nowhere in this query plan do we have an index used that supports the `PhoneNumberType` column. This is being used in our `WHERE` clause. We can create a non-clustered index on this column, which will assist our query when the number of records in our tables start to grow.

It's time for a New Query Window, into which we'll type our non-clustered index statement.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_NC_PhoneNumberTypes_PhoneNumberType')
BEGIN
DROP INDEX IX_NC_PhoneNumberTypes_PhoneNumberType ON dbo.PhoneNumberTypes;
END;

CREATE INDEX IX_NC_PhoneNumberTypes_PhoneNumberType ON dbo.PhoneNumberTypes(PhoneNumberType);

GO
```

Run this, then execute the `SELECT` statement again. Check out the execution plan now (Figure 14-13).



```
Query 1: Query cost (relative to the batch): 100%
SELECT C.ContactId, C.FirstName, C.LastName, C.AllowContactByPhone, PNT.PhoneNumberType,
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 1 %
Clustered Index Scan (Clustered)
[ContactPhoneNumbers].[PK_ContactPh…
Cost: 25 %
Index Seek (NonClustered)
[PhoneNumberTypes].[IX_PhoneNumberT…
Cost: 45 %
Clustered Index Seek (Clustered)
[Contacts].[PK_Contacts] [C]
Cost: 29 %

Figure 14-13. *The updated execution plan (with highlighted Index Seek)*

MOUSING OVER OPERATIONS IN THE EXECUTION PLAN

Top tip time: If you place your mouse over an operation within the execution plan, a detailed tool tip will appear, providing you with various statistics about that particular operation.

Aha! Now our new, non-clustered index is being used by the query. This means any queries we write in the future that use the `PhoneNumberType` column will be optimized. Top stuff.

Save this index as `c:\temp\sqlbasics\apply\22 - Create PhoneNumberTypes Index.sql`. Here's the SQLCMD code to add to `00 - Apply.sql`:

```
:setvar currentFile "22 - Create PhoneNumberTypes Index.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Execution Plan Percentages

Let's take a moment to quickly look at the percentage assigned to each item in the execution plans. The percentage tells you how much work SQL Server has to perform for each individual part of the plan. When we added our non-clustered index, it took up 45% of the plan. This is a good thing, as the aim of the index was to cause it to be used by the plan. The two clustered index items use up most of the remaining percentage. Interestingly, these figures changed when we added our non-clustered index. This is because the non-clustered index is doing some of the work these indexes were previously doing.

Execution Plans: A Quick Summary

To say this has been a crash course in execution plans is an understatement. In truth, we've only looked at them to demonstrate that our indexes are being used. But even this basic knowledge can help you. You can mouse over the items in an index plan and a pop-up will appear, telling you which columns are being used by that item. If columns you are joining on or are using in `WHERE` clauses are not mentioned, consider adding an index, then running the query again. If things don't work out as expected, you can always remove the index. Execution plans will actually suggest missing indexes it thinks you should add—they appear just above the execution plan diagram.

Execution plans—especially a good knowledge of them—is an advanced topic, but hopefully this brief introduction has whetted your appetite for more. We still need them in this chapter!

Indexed Columns vs. Included Columns

All of the indexes we've created so far have used indexed columns—that is, the columns have been declared as part of the index key. The index key consists of all columns declared within brackets after the table name. So in this index:

```
CREATE CLUSTERED INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses(ContactId, P
```

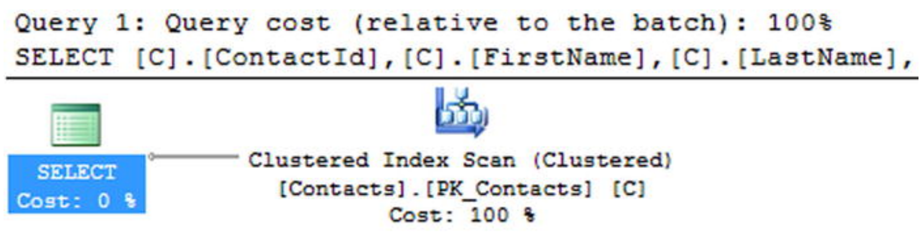
the index key is made up of `ContactId` and `Postcode`. There are no included columns.

Included columns come in useful when you want to store more data alongside the index. You don't necessarily want to make these columns part of the index key. Indeed, you may not be able to do so—the size of the key is limited to 900 bytes and 16 columns. If you had a `Postcode` column of size `VARCHAR(880)`, you'd only have 20 bytes left to play with. Included columns can work around this problem, and also the problem of non-clustered indexes needing to take an extra step.

Open a New Query Window and run this `SELECT` statement, making sure you turn on Execution Plans with Ctrl+M.

```
USE AddressBook;

SELECT C.ContactId, C.FirstName, C.LastName, C.AllowContactByPhone FROM dbo.Contacts C WHERE C.Allo
```

Ten rows are returned, and the execution plan (displayed in Figure 14-14) uses the clustered index to find the data.



Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],

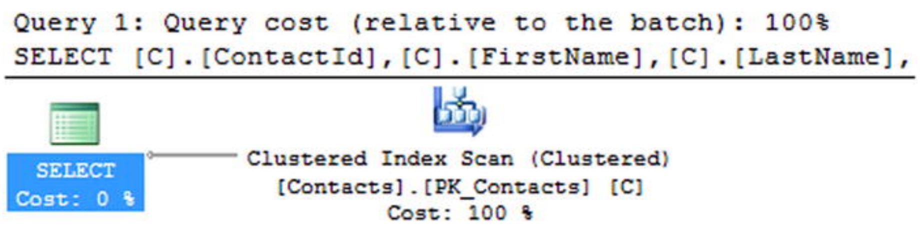
SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Contacts].[PK_Contacts] [C]
Cost: 100 %

Figure 14-14. Execution plan using the clustered index

Really, we need an index on `AllowContactByPhone`, as this is the `WHERE` clause being used. Let's go ahead and create that index in another New Query Window.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_NC_Contacts_AllowContactByPhone')
BEGIN
DROP INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts;
END;

CREATE NONCLUSTERED INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts(AllowContactByPhone);

GO
```

Execute this, then run the `SELECT` statement again. Figure 14-15 shows the query plan:



```
Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],
```

SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Contacts].[PK_Contacts] [C]
Cost: 100 %

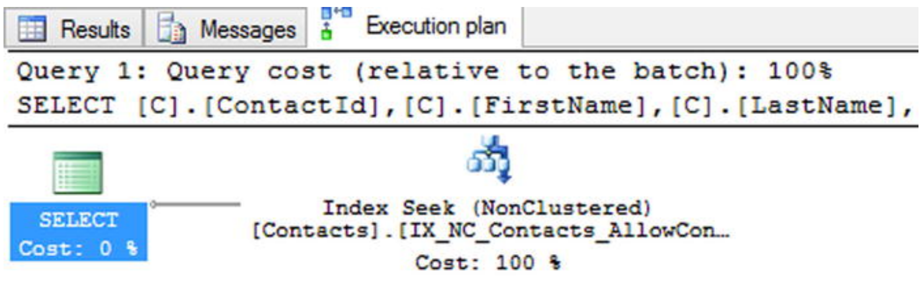
Figure 14-15. *Execution plan after adding non-clustered index*

It hasn't changed! Why is this? Well, SQL Server's Query Optimizer has decided the clustered index will execute the query more efficiently than the new non-clustered index we created. This is because the clustered index has immediate access to the data. As soon as a match is found by the clustered index, we return the data. With the non-clustered index, a match is found and then we have to skip across to the row to pull the data back.

We can remedy this situation by including the columns we are interested in as part of the index.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_NC_Contacts_AllowContactByPhone')
BEGIN
DROP INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts;
END;

CREATE NONCLUSTERED INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts(AllowContactByPhone) I

GO
```

The `INCLUDE` line is the only change. We've told SQL Server to store the `ContactId`, `FirstName`, and `LastName` columns with the index. These columns do not form part of the index—they wouldn't be used by the index to find data matching a `WHERE` clause, for example—but they are stored alongside the index, meaning an extra hop over to the row once a match is found is not required. Run the script to update the index, then run the `SELECT` statement again. A different query plan appears this time, as shown in Figure 14-16.



```
Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],
```

Results
Messages
Execution plan
SELECT
Cost: 0 %
Index Seek (NonClustered)
[Contacts].[IX_NC_Contacts_AllowCon...
Cost: 100 %

Figure 14-16. *Execution plan now using the non-clustered index!*

We've just created something called a *covering index*. This is an index that can be used to return all data for particular query. The index in Figure 14-16 utilizes all columns required to fulfil the query's requirements, so we say it *covers* the query's requirements.

Save the index script as `c:\temp\sqlbasics\apply\23 - Create Contacts AllowContactByPhone Index.sql`, and add it to the `SQLCMD file 00 - Apply.sql`.

```
:setvar currentFile "23 - Create Contacts AllowContactByPhone Index.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Think carefully when including columns as part of your indexes. They are very useful, but just remember that SQL Server has to maintain all of this information whenever you insert, update, or delete data in your table. The benefits of the index must outweigh the downside of keeping it up to date.

Filtered Indexes

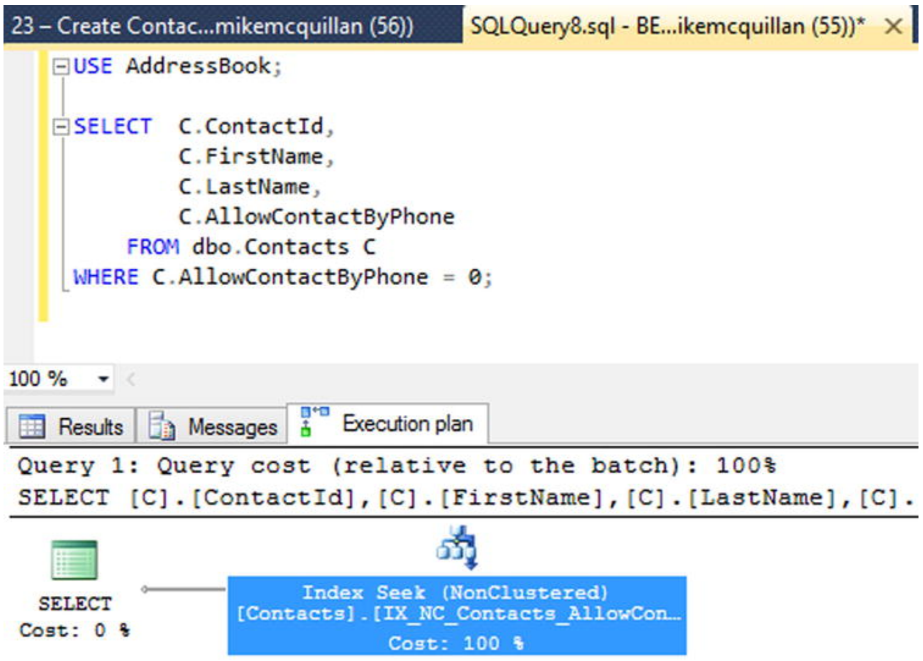
Filtered indexes were introduced in SQL Server 2008, and are severely underused in my experience. This is a shame, as they present an elegant solution to certain problems. A filtered index is the same as any other type of index you create, with one big difference: you specify a `WHERE` clause, limiting the index to certain types of data. Why would you want to do this?

- Filtered indexes are smaller than normal, full-table indexes
- Not all DML statements will cause filtered indexes to be updated, reducing the cost of index maintenance
- Less disk space is required to store a filtered index, as it only stores the rows matching the filter

We'll change the index we just created for `AllowContactByPhone` into a filtered index. Before we do that, return to the `SELECT` statement we were using to test it.

```
USE AddressBook

SELECT C.ContactId, C.FirstName, C.LastName, C.AllowContactByPhone FROM dbo.Contacts C WHERE C.Allo
```

If you run this and check the execution plan, you'll see the non-clustered index was used. Change the query to `= 0` instead of `= 1`. As Figure 14-17 proves, you'll see the non-clustered index is still used.


23 – Create Contac...mikemcquillan (56))
SQLQuery8.sql - BE...ikemcquillan (55))*
100 %
Results
Messages
Execution plan

```
USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.AllowContactByPhone
    FROM dbo.Contacts C
WHERE C.AllowContactByPhone = 0;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],[C].
```

SELECT
Cost: 0 %
Index Seek (NonClustered)
[Contacts].[IX_NC_Contacts_AllowCon...
Cost: 100 %

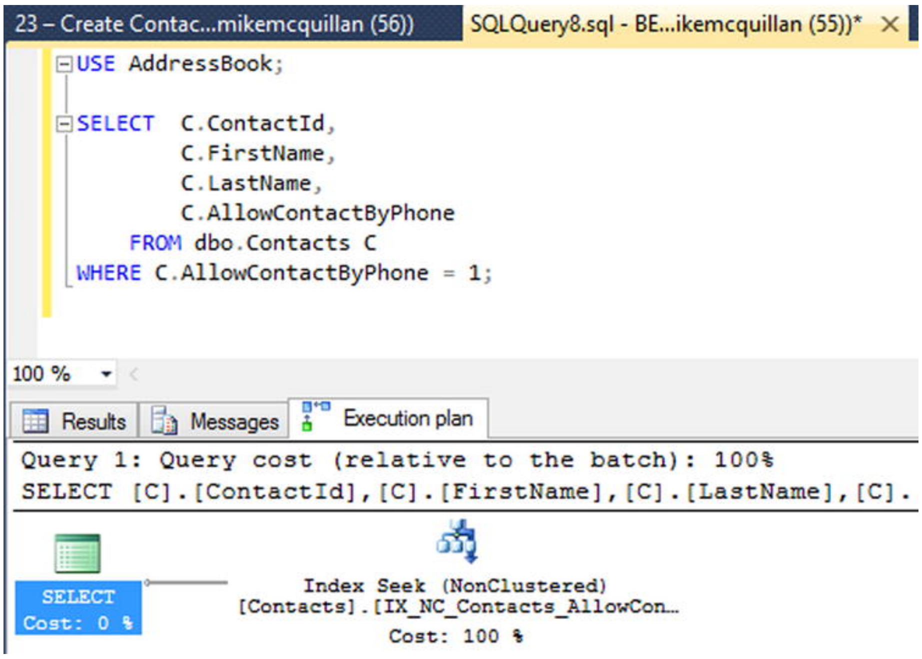
Figure 14-17*. Using a non-clustered index regardless of the value*

We'll change our index so it only applies to records where `AllowContactByPhone = 1`.

Open `c:\temp\sqlbasics\apply\23 - Create Contacts AllowContactByPhone Index.sql` and change the `CREATE INDEX` statement:

```
CREATE NONCLUSTERED INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts(AllowContactByPhone) I
```

Save and run the script to update the index. Execute the `SELECT` statement again as per Figure 14-18, setting `AllowContactByPhone = 1`.



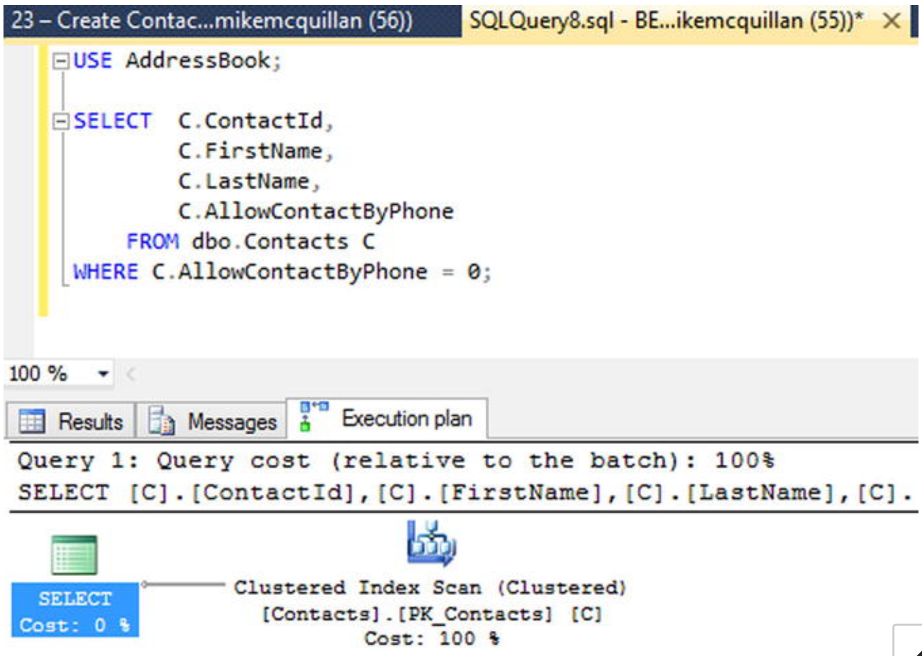
```
USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.AllowContactByPhone
    FROM dbo.Contacts C
WHERE C.AllowContactByPhone = 1;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],[C].
```

23 – Create Contac...mikemcquillan (56))
SQLQuery8.sql - BE...ikemcquillan (55))*
100 %
Results
Messages
Execution plan
SELECT
Cost: 0 %
Index Seek (NonClustered)
[Contacts].[IX_NC_Contacts_AllowCon...
Cost: 100 %

Figure 14-18. *Using a filtered index when the value matches*

Good news: our index is being used. Change the statement to use `AllowContactByPhone = 0` and run it again. This time, our index is ignored, and the clustered index is used instead (Figure 14-19).



```
USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.AllowContactByPhone
    FROM dbo.Contacts C
WHERE C.AllowContactByPhone = 0;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [C].[ContactId],[C].[FirstName],[C].[LastName],[C].
```

23 – Create Contac...mikemcquillan (56))
SQLQuery8.sql - BE...ikemcquillan (55))*
100 %
Results
Messages
Execution plan
SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Contacts].[PK_Contacts] [C]
Cost: 100 %

Figure 14-19. *Not using a filtered index when the value doesn't match*

This is perfect—we've told SQL Server the non-clustered index should only apply when the filter value for `AllowContactByPhone` is 1.

If you use them wisely, filtered indexes can really boost your queries, and can reduce the impact indexes may have on your DML statements. Keep them up your sleeve, as many SQL Server developers are not aware of filtered indexes. Knowing what features like this can do will help you stand out from the crowd.

Don't forget to add script 23 to the `00 - Apply.sql` SQLCMD script. Here is the code:

```
:setvar currentFile "23 - Create Contacts AllowContactByPhone Index.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Unique Indexes

We met unique indexes when we were talking about constraints in Chapter 7. Whenever we created a unique constraint, we were actually creating a unique index. A *unique index* is an index that prevents duplicate values from being entered into different rows across the columns it represents. A unique index on the `PhoneNumber` column in `ContactPhoneNumbers` would prevent the same phone number from being added twice, for example. Refer back to Chapter 7 if you need to refresh your knowledge.

Other Types of Index

SQL Server provides an XML data type, which can store the tiniest fragment of XML up to huge documents of 2GB in size. As you might imagine, searching all of this XML data can take a while. SQL Server has the ability to create both primary and secondary XML indexes to assist with such searching. If you have plans to create XML columns you should investigate how these indexes work—performance gains can be impressive.

A new type of index was introduced in SQL Server 2012: the Columnstore index. These are indexes aimed at bulk loads and read-only queries. It is primarily intended for use in data warehouses (essentially, flattened databases managed by SSAS), although they can be used for other purposes, too. If used in the correct manner, Microsoft claims index performance can be increased by up to 10 times that of traditional indexes.

Maintaining Indexes

Usually, your tables and other database objects require minimal maintenance. Once the object is structured how you want it and has been proven to be working correctly, you can pretty much leave it running, with the occasional check. This isn't the case with indexes. Over time, indexes will become fragmented. This means there are gaps in the index or data are not structured in the index as well as they might be.

This happens because of DML statements. `INSERT`s, `UPDATE`s, and `DELETE`s cause data to be removed from or moved about inside the index. This causes new pages to be added to the index, which can increase the number of intermediate levels created, and require more pages to be checked when searching for records in the index.

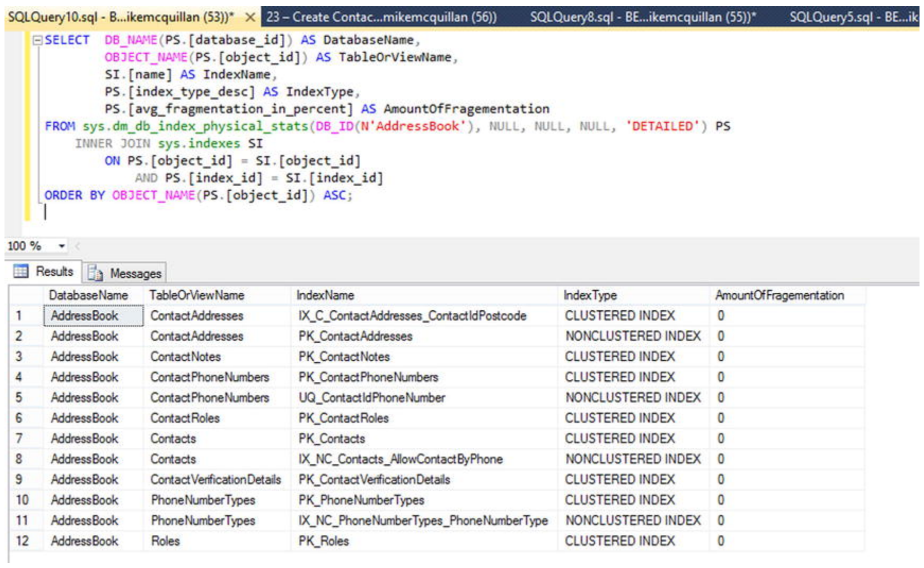
We'll take a look at some of the maintenance features SQL Server provides that help you keep on top of your indexes.

Identifying Index Fragmentation

SQL Server provides a set of Dynamic Management Views, or DMVs for short. There are lots of DMVs that provide access to all sorts of information. This query uses a DMV called `sys.dm_db_index_physical_stats` to tell you if any of your indexes are fragmented.

```
SELECT DB_NAME(PS.[database_id]) AS DatabaseName, OBJECT_NAME(PS.[object_id]) AS TableOrViewName, S
```

On the `FROM` line in Figure 14-20, note I've specified `'AddressBook'`. By substituting any database name here the query will return basic fragmentation information for your database's indexes.


SQLQuery10.sql - B...ikemcquillan (53))*
23 – Create Contac...mikemcquillan (56))
SQLQuery8.sql - BE...ikemcquillan (55))*
SQLQuery5.sql - BE...ik
100 %
Results
Messages

```
SELECT  DB_NAME(PS.[database_id]) AS DatabaseName,
        OBJECT_NAME(PS.[object_id]) AS TableOrViewName,
        SI.[name] AS IndexName,
        PS.[index_type_desc] AS IndexType,
        PS.[avg_fragmentation_in_percent] AS AmountOfFragementation
FROM sys.dm_db_index_physical_stats(DB_ID(N'AddressBook'), NULL, NULL, NULL, 'DETAILED') PS
    INNER JOIN sys.indexes SI
        ON PS.[object_id] = SI.[object_id]
            AND PS.[index_id] = SI.[index_id]
ORDER BY OBJECT_NAME(PS.[object_id]) ASC;
```

	DatabaseName	TableOrViewName	IndexName	IndexType	AmountOfFragementation
1	AddressBook	ContactAddresses	IX_C_ContactAddresses_ContactIdPostcode	CLUSTERED INDEX	0
2	AddressBook	ContactAddresses	PK_ContactAddresses	NONCLUSTERED INDEX	0
3	AddressBook	ContactNotes	PK_ContactNotes	CLUSTERED INDEX	0
4	AddressBook	ContactPhoneNumbers	PK_ContactPhoneNumbers	CLUSTERED INDEX	0
5	AddressBook	ContactPhoneNumbers	UQ_ContactIdPhoneNumber	NONCLUSTERED INDEX	0
6	AddressBook	ContactRoles	PK_ContactRoles	CLUSTERED INDEX	0
7	AddressBook	Contacts	PK_Contacts	CLUSTERED INDEX	0
8	AddressBook	Contacts	IX_NC_Contacts_AllowContactByPhone	NONCLUSTERED INDEX	0
9	AddressBook	ContactVerificationDetails	PK_ContactVerificationDetails	CLUSTERED INDEX	0
10	AddressBook	PhoneNumberTypes	PK_PhoneNumberTypes	CLUSTERED INDEX	0
11	AddressBook	PhoneNumberTypes	IX_NC_PhoneNumberTypes_PhoneNumberType	NONCLUSTERED INDEX	0
12	AddressBook	Roles	PK_Roles	CLUSTERED INDEX	0

Figure 14-20. *Returning index fragmentation details*

Altering Indexes

To manage an existing index, you use the `ALTER INDEX` statement. This `ALTER` statement works differently from the `ALTER VIEW` or `ALTER TABLE` statements we've seen so far, and also from the other `ALTER` statements we'll meet later in the book. Usually, an `ALTER` statement makes direct changes to the object concerned; `ALTER VIEW` allows you to completely change the definition of the view, for example.

`ALTER INDEX` is used for maintenance purposes. Its principal aim is to allow you to either disable, rebuild, or reorganize an index. You can change certain options for the index, but you cannot change its definition—to do that, you need to drop the index and then recreate it.

Disabling Indexes

You may occasionally need to disable an index. You might do this if you want to see how a query performs with the index and without it, but you don't want to lose the various metadata held for and about the index. Note that if you disable a clustered index you won't be able to query the table (but the data are still present; you just need to re-enable the index).

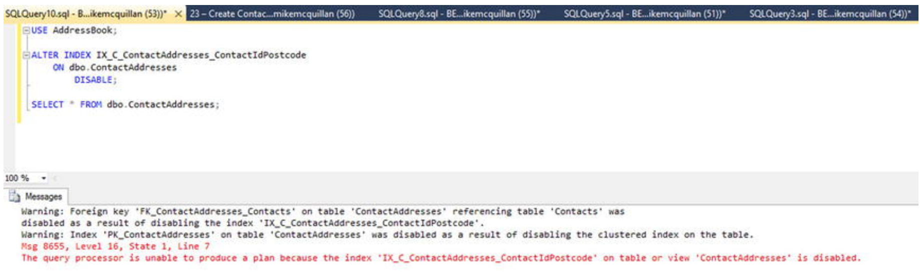
In a New Query Window, run this query.

```
USE AddressBook;

SELECT * FROM dbo.ContactAddresses;
```

All rows will be returned from the table. Now, change the script so it includes an `ALTER INDEX` statement above the `SELECT`, disabling the clustered index we created earlier.

```
USE AddressBook;

ALTER INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses DISABLE;

SELECT * FROM dbo.ContactAddresses;
```

You'll see some interesting messages, which are displayed in Figure 14-21.



```
USE AddressBook;

ALTER INDEX IX_C_ContactAddresses_ContactIdPostcode
    ON dbo.ContactAddresses
        DISABLE;

SELECT * FROM dbo.ContactAddresses;
```

```
Warning: Foreign key 'FK_ContactAddresses_Contacts' on table 'ContactAddresses' referencing table 'Contacts' was
disabled as a result of disabling the index 'IX_C_ContactAddresses_ContactIdPostcode'.
Warning: Index 'PK_ContactAddresses' on table 'ContactAddresses' was disabled as a result of disabling the clustered index on the table.
Msg 8655, Level 16, State 1, Line 7
The query processor is unable to produce a plan because the index 'IX_C_ContactAddresses_ContactIdPostcode' on table or view 'ContactAddresses' is disabled.
```

Figure 14-21. *Querying a table with a disabled index*

The two warning messages are generated by the `ALTER INDEX` statement. When a clustered index is disabled, other indexes on the table are disabled, too. This includes foreign keys. We've ended up disabling three indexes here: the index we requested, the `FK_ContactAddresses_Contacts` foreign key index, and the primary key index `PK_ContactAddresses`. These will all need to be re-enabled separately, unless we use the `ALL` keyword when rebuilding.

The error message was raised by the `SELECT` statement. Our `ALTER INDEX` was successful, but because the clustered index is disabled data can no longer be retrieved from the table. We'll have to re-enable the indexes so we can query our data.

Rebuilding Indexes

There is no option to re-enable an index. Instead, you must rebuild the index. You rebuild indexes when the index is not performing as expected, probably due to fragmentation. Rebuilding an index causes the index to be dropped and recreated, resolving any fragmentation issues. Specifying the `ALL` keyword causes every index on the table to be dropped and recreated.

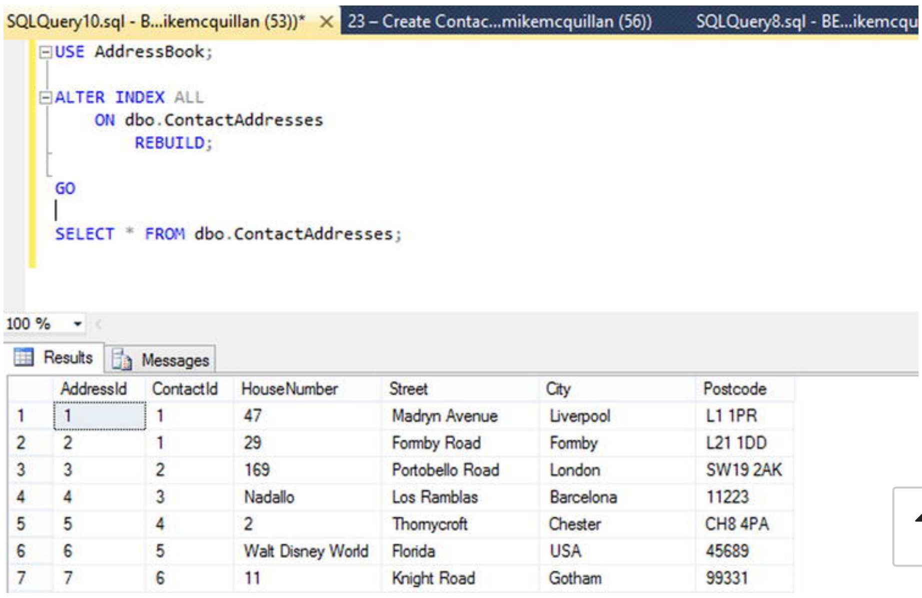
This statement will rebuild just the clustered index on the `ContactAddresses` table:

```
ALTER INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses
REBUILD;
```

But this statement will rebuild every index on the `ContactAddresses` table:

```
ALTER INDEX ALL ON dbo.ContactAddresses REBUILD;
```

Running this will make our `SELECT` statement work again. Note in Figure 14-22 that we've had to add a `GO` before the `SELECT` statement—this is needed, as the `REBUILD` has to complete in its own batch before we can query the table.



```
USE AddressBook;

ALTER INDEX ALL
    ON dbo.ContactAddresses
        REBUILD;

GO

SELECT * FROM dbo.ContactAddresses;
```

	AddressId	ContactId	HouseNumber	Street	City	Postcode
1	1	1	47	Madryn Avenue	Liverpool	L1 1PR
2	2	1	29	Formby Road	Formby	L21 1DD
3	3	2	169	Portobello Road	London	SW19 2AK
4	4	3	Nadallo	Los Ramblas	Barcelona	11223
5	5	4	2	Thornycroft	Chester	CH8 4PA
6	6	5	Walt Disney World	Florida	USA	45689
7	7	6	11	Knight Road	Gotham	99331

Figure 14-22. *Running the query after rebuilding the index*

Reorganizing Indexes

Reorganizing causes the leaf level of the index—the level that holds the data (or points to the data in a non-clustered index)—to be, well, reorganized. This eliminates fragmentation. This is similar to rebuilding, but crucially it can be done without impacting access to the table.

The indexes we are playing with here are very small and rebuild instantly. Imagine a table with millions of rows. Rebuilding an index on these tables can sometimes take hours. If this happens, rebuilding an index may not be desirable—it could prevent access to the table during the rebuild. It is for this kind of scenario that reorganization was introduced. The index is reorganized but the table is still accessible. You cannot reorganize a disabled index; the index must be active.

If we wanted to reorganize our clustered index we would specify the `REORGANIZE` keyword.

```
ALTER INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses REORGANIZE;
```

Again, we could reorganize all indexes with the `ALL` keyword:

```
ALTER INDEX ALL ON dbo.ContactAddresses REORGANIZE;
```

Altering Indexes Using SSMS

You can disable, rebuild, or reorganize using SSMS. Locate the required index in the **Indexes** node (found within the table the index is applied to), right-click it, and choose the appropriate option.

Dropping Indexes

Over time, you may decide certain indexes have outlived their usefulness. They can be removed quite easily by using the `DROP INDEX` command, which we've already met. Just for practice, we'll create three rollback scripts to drop the indexes we've created in this chapter. Create these three scripts in `c:\temp\sqlbasics\rollback`.

- `21 - Create ContactAddresses Clustered Index Rollback.sql`

```
USE AddressBook;
IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_C_ContactAddresses_ContactIdPostcode'
BEGIN
DROP INDEX IX_C_ContactAddresses_ContactIdPostcode ON dbo.ContactAddresses;
END;

GO
```

- `22 - Create PhoneNumberTypes Index Rollback.sql`

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_PhoneNumberTypes_PhoneNumberType')
BEGIN
DROP INDEX IX_PhoneNumberTypes_PhoneNumberType ON dbo.PhoneNumberTypes;
END;

GO
```

- `23 - Create Contacts AllowContactByPhone Index Rollback.sql`

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.indexes WHERE [name] = 'IX_NC_Contacts_AllowContactByPhone')
BEGIN
DROP INDEX IX_NC_Contacts_AllowContactByPhone ON dbo.Contacts;
END;

GO
```

Add these lines to the top of `00 - Rollback.sql` to ensure these scripts are executed whenever we rollback the database.

```
:setvar currentFile "23 - Create Contacts AllowContactByPhone Index Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "22 - Create PhoneNumberTypes Index Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "21 - Create ContactAddresses Clustered Index Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Statistics

I view statistics as an advanced topic, but I want to mention them briefly so you at least know they exist. There, I've mentioned them—let's move on! Only joking (it's very good for morale!).

SQL Server uses statistics to figure out how it can best process your query. Statistics include the number of records for an index, how many pages those records cover, and details of table records, such as how many records in `Contacts` have a value of 1 in the `AllowContactByPhone` column.

As an example, assume you are running the query `WHERE AllowContactByPhone = 1`. You have an index for this column, but for some reason it isn't being used. There are a couple of possible reasons for this. One is the statistics for the index are out of date. SQL Server automatically maintains statistics based on particular rules. Sometimes these rules are not met and the statistics are not updated perhaps as often as they should be.

Another reason the index may not be used is because the statistics are up to date, and they inform SQL Server that using the index will be less efficient than using a scan or an alternative index.

You can delve into statistics by expanding the **Statistics** node under a table in Object Explorer. In Figure 14-23, we can see the statistics for the `ContactAddresses` table.

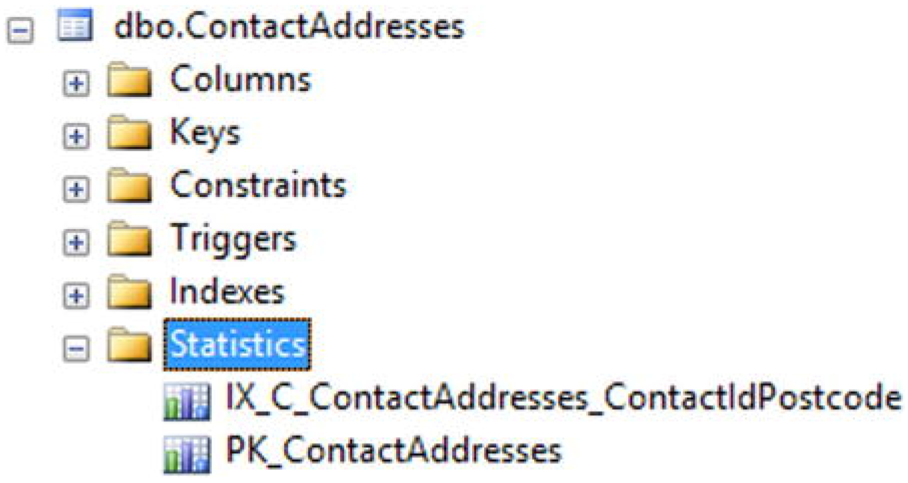


Figure 14-23. *Viewing statistics for a table*

Double-clicking one of these items will bring up more information about the statistics. As you are just starting out in your SQL Server journey, don't worry too much about statistics at the moment. I know people who've worked with SQL Server for years and don't understand them properly (or at all). But know they are there—it just might be worth delving into them in more detail.

Creating Indexed Views

The last piece of code we'll write in what has been an extremely involving chapter is to take one of our views and transform it into an indexed view. Indexed views perform better than normal views because they have a unique clustered index added to them. This means the view exists as a physical object, making it work in a manner very similar to a table. Creating an indexed view is the only way you can create an index that bridges multiple tables.

You can create both clustered and non-clustered indexes against a view, but you can only create non-clustered indexes if a unique clustered index already exists. Also, the view definition must meet certain rules. If any of these rules are not met, you cannot create a clustered index on the view:

- The view definition must include `WITH SCHEMABINDING`
- All table names must include the schema name (e.g., `dbo.Contacts`, not `Contacts`)
- All expressions in the view must be deterministic; that is, the same value is always returned (`GETDATE()` is not deterministic as it never returns the same value, but `ADDNUMBERS(1,2)` would always return 3, so it is deterministic)
- The view can only include tables, not other views
- The tables in the view must exist in the same database
- Most aggregate functions cannot be used (e.g., `COUNT()`, `MIN()`, `MAX()`)

There are more rules, such as certain `SET` options that must be configured, but these are usually set to the default values anyway, so we won't concern ourselves with them. The preceding rules represent the most common things you have to think about. A full list of rules can be found at `https://msdn.microsoft.com/en-us/library/ms191432.aspx`.

We'll create an index on the `VerifiedContacts` view. Before we can create any other type of index, we *must* create a unique, clustered index. You cannot just create a clustered index; it must be unique, too.

The statement to create an index on a view is no different from other indexes we've seen, other than the inclusion of the `UNIQUE` keyword:

```
CREATE UNIQUE CLUSTERED INDEX IX_C_VerifiedContacts_ContactIdFirstNameLastName
ON dbo.VerifiedContacts(ContactId, LastName, AllowContactByPhone);
```

The combination of columns declared in the index must be unique. `ContactId` will always be unique as it is an `IDENTITY` column. This means we can add other columns to the index, as the first column guarantees uniqueness in this case. If the first column didn't provide us with a unique value, we'd need to add more columns so the combination of values would be unique.

Running this statement is successful, and allows us to go ahead and create a non-clustered index on the view:

```
CREATE INDEX IX_NC_VerifiedContacts_DrivingLicenseNumber
ON dbo.VerifiedContacts(DrivingLicenseNumber);
```

At this point, we could create as many non-clustered indexes as we wished on the view. These indexes will really help when your database has grown and your view is retrieving lots of rows. I once reduced a three-minute query to under a second using an indexed view. This query was used all over the system, so you can imagine the gains that were made.

There's actually a lot more to indexed views than we've discussed here, so take some time to read the MSDN article—it's well worth a look.

Are Indexes Ever a Bad Idea?

We've seen lots of index-related escapades in this chapter, all universally positive. It's worth asking: Is there ever a time when using indexes could be a bad thing? The answer is simple: Heck, *yes*! Like any other piece of SQL Server technology, indexes can have a negative effect on the database when used incorrectly. The main reason indexes can cause problems is because of `INSERT`, `UPDATE`, and `DELETE` statements—the DML statements. Let's think about how indexes work again for a moment.

- If an `INSERT` or `UPDATE` statement executes, the index must be updated with the current data
- If a `DELETE` statement executes, the rows in question need to be removed from the index

The point here is when a DML statement executes, it means SQL Server does some additional work to keep the index up to date. This additional work marginally slows down your `INSERT`, `UPDATE`, and `DELETE` statements. They don't slow down so much that you'd notice, but there is an effect.

Now, imagine a couple of scenarios.

- An index that has become badly defragmented
- An index whose statistics are wildly out of date
- A covering index with multiple columns

Any of these can cause update problems. Let me tell you a story about a covering index that went bad. I should point out this is an extreme example! A company I once worked for had built an event logging system. All log requests were sent to a queue and a service then came along, picked up the requests from the queue, and inserted them into an `EventLog` table. This table had a covering index on it, allowing easy querying of the log.

For the first couple of months, the event logging system worked really well. Eventually, a developer needed the log to investigate a problem, and was surprised to discover data for the past week wasn't in the table. I took a look at the queue and saw there was a huge backlog of queue items waiting to be processed. This was strange; querying the table returned results effectively enough.

I tried manually inserting a row and was stunned to discover it took *five minutes* to insert! This was into a table of about six columns. The covering index was the problem. Removing it fixed the problem immediately, resulting in inserts of less than a second. Because the index had multiple columns and hadn't been maintained properly, it was taking forever (well, five minutes) to figure out where it should place new rows.

Remember this cautionary tale when building your indexes! They are a great thing, but make sure you use them correctly and in moderation. Don't add them for the sake of it.

Summary

My word, this has been a big chapter. We've covered just about every aspect of indexes, although we've hardly skimmed the surface of what is a huge topic. Indexes are the most impressive speed improvement you can make to your queries. Take some time to tinker with them and you'll be the toast of your department.

Our **AddressBook** database has some nice indexes now, but if we can't add data in a consistent manner, indexes won't help us at all. Our next chapter will help us in this regard, as we take a look at transactions.

PREV
Chapter 13 : Views

NEXT
Chapter 15 : Transactions