Just as using an index in a book to find a particular bit of information is often much faster than reading all pages, so SQL Server indexes can make finding a particular row in a table dramatically faster by cutting down the number of read operations.

This section first discusses the two types of indexes supported by SQL Server, clustered and non-clustered. It also goes into included columns, a feature of non-clustered indexes. After that, we'll look at when to use each type of index.

Clustered Index

Take the following table:

CREATE TABLE [dbo].[Book](

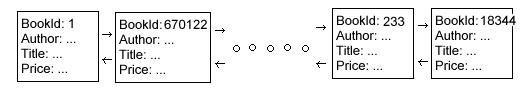
[BookId] [int] IDENTITY(1,1) NOT NULL,

[Title] [nvarchar](50) NULL,

[Author] [nvarchar](50) NULL,

[Price] [decimal](4, 2) NULL)

Because this table has no clustered index, it is called a heap table. Its records are unordered, and to get all books with a given title, you have to read all the records, which is just not efficient. It has a very simple structure:



Let's see how long it takes to locate a record in this table. That way, we can compare against the performance of a table with an index. To do that in a meaningful way, first insert a million records into the table.

INSERT INTO BOOK VALUES('Title1','Author1',10.45)

go 1000000

Tell SQL Server to show I/O and timing details of each query we run:

SET STATISTICS IO ON

SET STATISTICS TIME ON

Also, before each query, flush the SQL Server memory cache:

CHECKPOINT

DBCC DROPCLEANBUFFERS

Now run the query below with a million records in the Book table:

SELECT Title, Author, Price FROM dbo.Book WHERE BookId = 5000

The results on my machine - reads: 9564, CPU time: 109 ms, elapsed time: 808 ms. SQL Server stores all data in 8KB pages. This shows it read 9564 pages - the entire table.

(1 row(s) affected)

Table 'Book'. Scan count 3, logical reads 6494, physical reads 28, read-ahead reads 3114, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

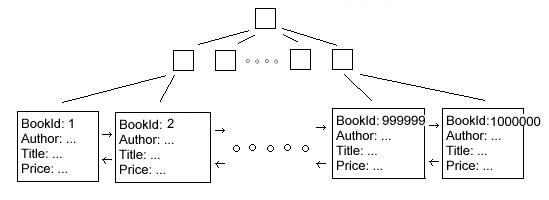
CPU time = 125 ms, elapsed time = 1846 ms.

Now add a clustered index:

ALTER TABLE Book

ADD CONSTRAINT [PK\_Book] PRIMARY KEY CLUSTERED ([BookId] ASC)

This puts the index on column BookId, making WHERE and JOIN statements on BookId faster. It sorts the table by BookId, and adds a structure called a B-tree to speed up access:



BookId is now used the same way as a page number in a book. Because the pages in a book are sorted by page number, finding a page by page number is very fast.

Now run the same query again to see the difference:

SELECT Title, Author, Price FROM dbo.Book WHERE BookId = 5000

The results - reads: 2, CPU time: 0 ms, elapsed time: 32 ms. The number of reads of 8KB pages has gone from 9564 to 2, CPU time from 109ms to less than 1 ms, and elapsed time from 808 ms to 32 ms. That's a dramatic improvement.

(1 row(s) affected)

Table 'Book'. Scan count 0, logical reads 3, physical reads 2, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 182 ms.

Non-clustered Index

Now let's select by Title instead of BookId:

SELECT Title, Author FROM dbo.Book WHERE Title = 'Don Quixote'

The results - reads: 9146, CPU time: 156 ms, elapsed time: 1653 ms. These results are pretty similar to what we got with the heap table. Which is no wonder, seeing that there is no index on Title.

(1 row(s) affected)

Table 'Book'. Scan count 3, logical reads 6267, physical reads 70, read-ahead reads 6225, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

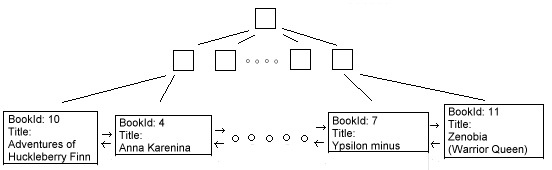
CPU time = 187 ms, elaps

**The solution obviously is to put an index on Title.**

**However, because a clustered index involves sorting the table records on the index field, there can be only one clustered index. We've already sorted on BookId, and the table can't be sorted on Title at the same time.**

The solution is to create a non-clustered index. This is essentially a duplicate of the table records, this time sorted by Title. To save space, SQL Server leaves out the other columns, such as Author and Price. You can have up to 249 non-clustered indexes on a table.

Because we still want to access those other columns in queries though, we need a way to get from the non-clustered index records to the actual table records. The solution is to add the BookId to the non-clustered records. Because BookId has the clustered index, once we have found a BookId via the non-clustered index, we can use the clustered index to get to the actual table record. This second step is called a key lookup.



*Why go through the clustered index? Why not put the physical address of the table record in the non-clustered index record?* The answer is that when you update a table record, it may get bigger, causing SQL Server to move subsequent records to make space.

If non-clustered indexes contained physical addresses, they would all have to be updated when this happens. It’s a tradeoff between slightly slower reads and much slower updates.

**If there is no clustered index then non-clustered index records do have the physical address. If there is a clustered index but it is not unique, then SQL Server does use the clustered index key in the non-clustered index records, but it adds a uniquifier to each record to distinguish records with the same clustered key.**

To see what a non-clustered index will do for us, first create it:

CREATE NONCLUSTERED INDEX [IX\_Title] ON [dbo].[Book]([Title] ASC)

Now run the same query again:

SELECT Title, Author FROM dbo.Book WHERE Title = 'Don Quixote'

The results - reads: 4, CPU time: 0 ms, elapsed time: 46 ms. The number of reads has gone from 9146 to 4, CPU time from 156 ms to less than 1 ms, and elapsed time from 1653 ms to 46 ms.

(1 row(s) affected)

Table 'Book'. Scan count 1, logical reads 6, physical reads 3, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

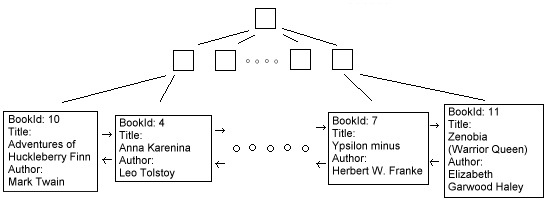
CPU time = 0 ms, elapsed time = 31 ms.

This means that having a non-clustered index is not quite as good as having a clustered index, but still dramatically better than having no index at all.

Included Columns

You can squeeze a bit more performance out of a non-clustered index by cutting out the key lookup - the second step where SQL Server uses the clustered index to find the actual record.

Have another look at the test query - it simply returns Title and Author. Title is already in the non-clustered index record. If you were to add Author to the non-clustered index record as well, there would be no longer any need for SQL Server to access the table record, enabling it to skip the key lookup. It would look like this:



This can be done by including Author in the non-clustered index:

CREATE NONCLUSTERED INDEX [IX\_Title] ON [dbo].[Book]([Title] ASC)

INCLUDE(Author)

WITH drop\_existing

Now run the query again:

SELECT Title, Author FROM dbo.Book WHERE Title = 'Don Quixote'

The results - reads: 2, CPU time: 0 ms, elapsed time: 26 ms. The number of reads has gone from 4 to 2, and elapsed time from 46 ms to 26 ms. That's almost a 50% improvement. In absolute terms, the gain isn't all that great, but for a query that is executed very frequently, this may be worthwhile. Don't overdo this - the bigger you make the non-clustered index records, the fewer fit on an 8KB page, forcing SQL Server to read more pages.

Selecting columns to give an index

Because indexes do create overhead, you want to carefully select the columns to give indexes. Before starting the selection process, keep in mind that:

* Putting a Primary Key on a column [by default results in it having a clustered index](http://msdn.microsoft.com/en-us/library/ms181043(v=SQL.100).aspx) (you can give it a unique non-clustered index instead). So you may already have many columns in your database with an index. As you'll see later in the section "When to use a clustered index", putting the clustered index on the ID column of a record is almost always a good idea.
* If you made one or more columns unique (with the UNIQUE constraint), SQL Server [will already have created a UNIQUE index](http://msdn.microsoft.com/en-us/library/ms177420(v=SQL.100).aspx) to enforce the uniqueness requirement of the UNIQUE constraint.
* Putting an index on a table column [can slow down queries that modify that table](http://www.codeproject.com/Articles/292747/Speeding-up-database-access-Part-Fixing-missing#updatedoften) (UPDATE, INSERT, DELETE). Don't focus on just one query.
* Before introducing an index on your live database, test the index in development to make sure it really does improve performance.

Let's look at when and when not to use an index, and when to use a clustered index.

**When to use an index**

You can follow this decision process when selecting columns to give an index:

* Start by looking at the most expensive queries. You identified those in [Part 1: "Pinpointing missing indexes and expensive queries"](http://www.codeproject.com/KB/database/speedingupdatabaseaccess1.aspx). There you also saw indexing suggestions generated by the Database Engine Tuning Advisor.
* Look at putting an index on [foreign keys](http://msdn.microsoft.com/en-us/library/ms175464.aspx), especially if they are used in JOINs. This may help SQL Server to identify matching records quicker.
* Consider columns used in ORDER BY and GROUP BY clauses. If there is an index on such a column, then SQL Server doesn't have to sort the column again - because the index already keeps the column values in sorted order.
* Consider columns used in WHERE clauses, especially if the WHERE will select a small number of records.
* The MIN and MAX functions benefit from working on a column with an index. Because the values are sorted, there is no need to go through the entire table to find the minimum or maximum.
* Think twice before putting an index on a column that takes a lot of space. If you use a non-clustered index, the column values will be duplicated in the index. If you use a clustered index, the column values will be used in all non-clustered indexes. The increased sizes of the index records means fewer fit in each 8KB page, forcing SQL Server to read more pages. The same applies to including columns in non-clustered indexes.
* A WHERE clause that applies a function to the column value can't use an index on that column to find records, because the output of the function is not in the index. Take for example:

SELECT Title, Author FROM dbo.Book WHERE LEFT(Title, 3) = 'Don'

Putting an index on just the Title column won't make this query any faster. However, if you use a non-clustered index that includes both the Title and Author columns, the SQL query is able to scan that index instead of the table itself - using the index to access the data rather than locating records in the table itself. In [Part 4: "Fixing expensive queries"](http://www.codeproject.com/KB/database/speedingupdatabaseaccess4.aspx), you'll see how this may be quicker when the index records are smaller than the table records.

* Likewise, SQL Server can't use an index to locate the records if you use LIKE in a WHERE clause with a wild card at the start of the search string, such as this:

SELECT Title, Author FROM dbo.Book WHERE Title LIKE '%Quixote'

However, if the search string starts with a constant text instead of a wild card, an index can be used to locate records:

SELECT Title, Author FROM dbo.Book WHERE Title LIKE 'Don%'

**When not to use an index**

Having too many indexes can actually hurt performance. Here are the main reasons not to use an index on a column:

* The column gets updated often.
* The column has low specificity - meaning it has lots of duplicate values.

Let's look at each reason in turn.

**Column updated often**

When you update a column without an index, SQL Server needs to write one 8KB page to disk - provided there are no page splits. However, if the column has a non-clustered index, or if it is included in a non-clustered index, SQL Server needs to update the index as well - so it has to write at least one additional page to disk. It also has to update the B tree structure used in the index, potentially leading to more page writes.

If you update a column with a clustered index, the non-clustered index records that use the old value need to be updated too, because the clustered index key is used in the non-clustered indexes to navigate to the actual table records. Secondly, remember that the table records themselves are sorted based on the clustered index - if the update causes the sort order of a record to change, that may mean more writes. Finally, the clustered index needs to keep its B-tree up to date.

This doesn't mean you cannot have indexes on columns that get updated - just be aware that indexes slow down updates. Test the effect of any index you add.

If an index is critical but rarely used, for example only for overnight report generation, consider dropping the index and recreating it when it is needed.

**Low specificity**

Even if there is an index on a column, the query optimizer won't always use it. Think of the index in a book - great if you are trying to find a word that is used on only a few pages, but not so great if you're trying to find all occurrences of a commonly used word such as "the". You'd be better off going through each page, rather than going back and forth to the index. In this context, it is said that "the" has low specificity.

You can use a simple query to determine the average selectivity of the values in a column. For example, to find the average selectivity of the Price column in the Book table, use:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/292747/Speeding-up-database-access-Part-Fixing-missing)

SELECT

COUNT(DISTINCT Price) AS 'Unique prices',

COUNT(\*) AS 'Number of rows',

CAST((100 \* COUNT(DISTINCT Price) / CAST(COUNT(\*) AS REAL))

AS nvarchar(10)) + '%' AS 'Selectivity'

FROM Book

If every book has a unique price, selectivity will be 100%. However, if half the books are $20 and the other half $30, then average selectivity will be only 50%. If the selectivity is 85% or less, an index is likely to incur more overhead than it would save.

Some prices may occur a lot more often than other prices. To see the specificity of each individual price, you would run:

DECLARE @c real

SELECT @c = CAST(COUNT(\*) AS real) FROM Book

SELECT

Price,

COUNT(BookId) AS 'Number of rows',

CAST((1 - (100 \* COUNT(BookId) / @c))

AS nvarchar(20)) + '%' AS 'Selectivity'

FROM Book

GROUP BY Price

ORDER BY COUNT(BookId)

The query optimizer is unlikely to use a non-clustered index for a price whose specificity is below 99%. It figures out the specificity of each price by keeping statistics on the values in the table.

In the section on [included columns](http://www.codeproject.com/Articles/292747/Speeding-up-database-access-Part-Fixing-missing#includedcolumns), we saw how SQL Server not only uses indexes to find records, but also to get table data right out of the index. SQL Server only looks at specificity when deciding whether to use an index for finding records. It could profitably get the data out of an index even if that index has very bad specificity.

When to use a clustered index

You saw that there are two types of indexes, clustered and non-clustered. And that you can have only one clustered index. How do you determine the lucky column that will have the clustered index?

To work this out, let's first look at the characteristics of a clustered index against a non-clustered index.

| **Characteristic** | **Clustered index compared to a non-clustered index** |
| --- | --- |
| Reading | Faster - Because there is no need for key lookups. No difference if all required columns are included in the non-clustered index. |
| Updating | Slower - Not only the table record, but also all non-clustered index records need potentially be updated. |
| Inserting / Deleting | Faster - With a non-clustered index, inserting a new record in the table means inserting a new record in the non-clustered index as well. With a clustered index, the table is effectively part of the index, so there is no need for the second insert. The same goes for deleting a record.  On the other hand, when the record is inserted at any place in the table but the very end, the insert may cause a page split where half the content of the 8KB page is moved to another page. Having a page split in a non-clustered index is less likely, because its records are smaller (they normally don't have all columns that a table record has), so more records fit on a page.  When the record is inserted at the end of the table, there won't be a page split. |
| Column size | Needs to be kept short and fast - Every non-clustered index contains a clustered index value, to do the key lookup. Every access via a non-clustered index has to use that value, so you want it to be fast for the server to process. That makes a column of type int a lot better to put a clustered index on than a column of type nvarchar(50). |

If only one column requires an index, this comparison shows that you'll want to always give it the clustered index. If multiple columns need indexes, you'll probably want to put the clustered index on the primary key column:

* Reading - The primary key tends to be involved in a lot of WHERE and JOIN clauses, making read performance important.
* Updating - The primary key should never or rarely get updated, because that would mean changing referring foreign keys as well.
* Inserting / Deleting - Most often you'll make the primary key an IDENTITY column, so each new record is assigned a unique, ever increasing number. This means that if you put the clustered index on the primary key, new records are always added at the end of the table without page splits.
* Size - Most often the primary key is of type int - which is short and fast.

Indeed, when you set the Primary Key on a column in the SSMS table designer, SSMS by default gives that column the clustered index unless another column already has the clustered index.

Maintaining indexes

Do the following to keep your indexes working efficiently:

* Defragment indexes. Repeated updates cause indexes and tables to become fragmented, decreasing performance. To measure the level of fragmentation and to see how to defragment indexes, refer to [parts 2](http://www.codeproject.com/KB/database/speedingupdatabaseaccess2.aspx) and [7](http://www.codeproject.com/KB/database/speedingupdatabaseaccess7.aspx).
* Keep statistics updated. SQL Server maintains statistics to figure out whether to use an index for a given query. These statistics are normally kept up to date automatically, but this can be switched off. If you did, make sure statistics are kept up to date.
* Remove unused indexes. As you saw, indexes speed up read access, but slow down updates. In [Part 1: "Pinpointing missing indexes and expensive queries"](http://www.codeproject.com/KB/database/speedingupdatabaseaccess1.aspx), you saw how to identify unused indexes.