Chapter 6
Clustered and Nonclustered Indexes

Indexes are a very important part of Microsoft SQL Server databases. We use indexes to store data in a meaningful, organized way, which makes the *retrieval* of that data **very fast**.

When we write queries, we not only want to make sure our queries return complete and accurate information, but we also want the data to be retrieved *quickly*. Many enterprise databases have **tens of millions of rows** in their tables, which is a large amount of data for SQL Server to *wade* through when it's looking for data to return for a query we've written. We can help SQL Server search through this data by introducing some helpful indexes.

There are only 2 basic types of indexes:

- **Clustered**
- **Nonclustered**

We will discuss each type of index and give you all the important details you should understand, like how these indexes use a **B-Tree** to organize and store data in a way that makes it easy to search.

Clustered Indexes

A clustered index is a data structure that defines the physical storage order of data in a table. The data is ordered according to something called the clustered index "*key*". The data is physically stored in this order on the hard disk.

The "*data structure*" I mentioned above has a name. It's called the **B-Tree**. Retrieving data from a B-tree is extremely fast, which is usually what we want when we query data. We'll go more into detail about the B-Tree in a later section.

To understand clustered indexes in SQL Server, it's best to think of a really great analogy: **A bookstore**.

Imagine you're in a bookstore, and you're looking at books on a shelf for a particular genre. Maybe it's the "*Fiction*" genre.

How are the books ordered on the bookshelf? They are ordered in a very particular way to make it easy to locate specific books. What is that ordering?

They are ordered by **author last name**.

Despite there being hundreds or maybe thousands of books in the genre, you can locate a book you're interested in very easily if you know the author's last name.

Think about how efficient this ordering is. Say you're looking for the book "*Fight Club*" by **Chuck Palahniuk**. When you go to the bookshelf, you know you can skip all the books whose author's last name starts with '*A*' through '*O*', because you know '*Palahniuk*' won't be there. The same idea is true if you look through all the '*P*' authors and you don't actually find '*Palahniuk*'. You know you don't have to look through the books whose author starts with '*Q*' through '*Z*' because you know your book won't be there!

What if the books were **not** in any kind of order at all? What if you walked into the bookstore and the books are simply sitting in a pile on the floor like this:

No ordering at all, just simply *there*. It would take you *ages* to find the book you're interested in. Sure, you may get lucky and find the book sitting near the top of the heap, but you also might get *unlucky* and find the book near the bottom! Or worse, what if you look through the entire heap and realize the book you wanted **isn't even there!**

That would be a *tragedy*. But luckily, that's not the case. Barnes and Noble understands the obvious truth that books are easier to locate if they are **organized and ordered**.

Folks, this analogy represents the idea of the **clustered index**.

In the analogy, the ordering "*key*" is the author names. A "*key*" is just the property by which the books are ordered on the shelf. The **clustered index "*key*"** is the same idea. It's the property by which the data is ordered on disk.

Understanding Execution Plans

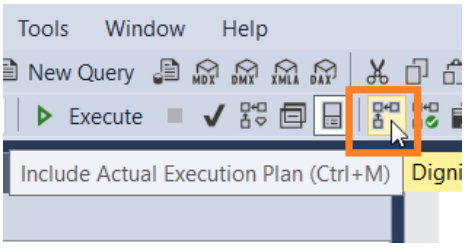
To further understand how clustered indexes work in SQL Server, we need to see what SQL Server is doing behind the scenes when a query is executed. After all, SQL is a *declarative* language, meaning we tell it what we want, and SQL Server determines the best way to get the information. To see how SQL Server gathers the data we asked for, we can look at the **query execution plan**.

Let's re-create our **Books** table and populate it with some data to demonstrate:

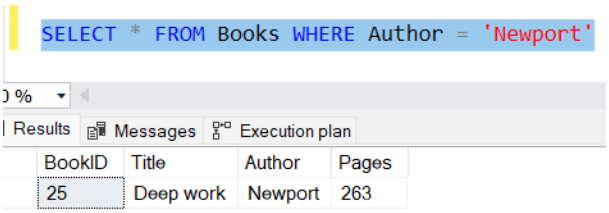
```
DROP TABLE IF EXISTS Books

CREATE TABLE Books
(
 BookID INT IDENTITY(10,5) NOT NULL,
 Title VARCHAR(50) NOT NULL,
 Author VARCHAR(15),
 Pages INT
)
GO
```

```
INSERT INTO Books (Title, Author, Pages)
VALUES ('As a man thinketh', 'Allen', 45),
('Eat that frog', 'Tracy', 108),
('The war of art', 'Pressfield', 165),
('Deep work', 'Newport', 263),
('The Pragmatic Programmer', 'Thomas', 283),
('The Education of a Bodybuilder', 'Schwarzenegger', 256),
('Debt Free Degree', 'O''Neil', 224)
```

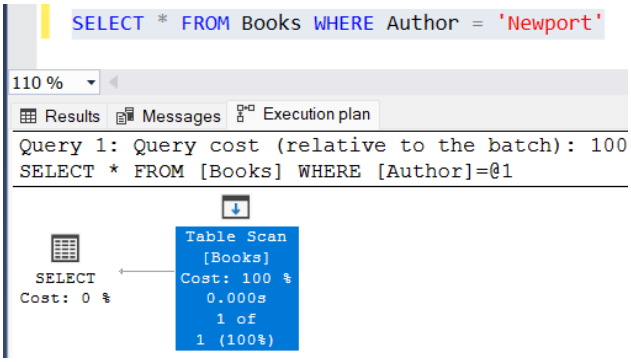
You can see the execution plan of a query by enabling an option in SQL Server Management Studio called "*Include Actual Execution Plan*", seen here:



Let's run a quick query to gather the details we have for a book in our inventory written by "*Newport*":



Notice the "**Execution Plan**" tab that appears in the result set window. If we click this tab, we can see the execution plan that SQL Server used to gather the data we asked for:



Notice SQL Server decided to do a full **table scan** to get the data we asked for. Folks, this is like trying to look for a book when all the books are in a giant heap on the floor. A table scan means that there isn't a nice ordering established to make the task of looking for the data easier, so SQL Server simply had to look through the **entire table** to find the information we asked for, which was all details we have for a book written by "*Newport*".

It should go without saying that looking through a *heap*, i.e. performing a *table scan*, is **not ideal**.

> **Note:** A table without a clustered index is referred to as a "***heap***".

What we ought to do is establish a clustered index on this table and make the Author column the clustered

index key (just like we would establish an ordering of books on a bookshelf where the ordering key is the author last names). This will make queries of this type much faster and more efficient!

Creating a clustered index

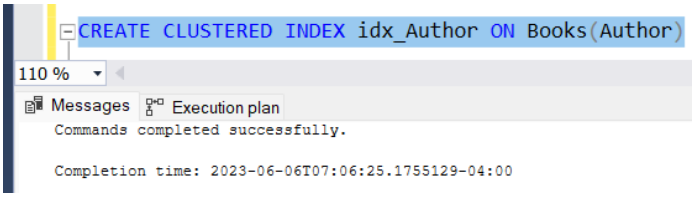
Let's add a clustered index to the Books table. The syntax follows this pattern:

```
CREATE CLUSTERED INDEX index_name ON table_name(key_column_name)
```

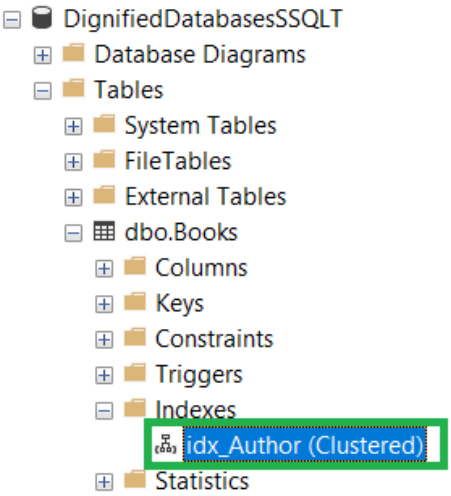
So for our example, we want to create a clustered index on the Books table, where the clustered index key is the Author column. Our statement would look like this:

```
CREATE CLUSTERED INDEX idx_Author ON Books(Author)
```

I chose to call the index "***idx_Author***", but understand that you can name it whatever you want. Let's run this code and see that it executes successfully:

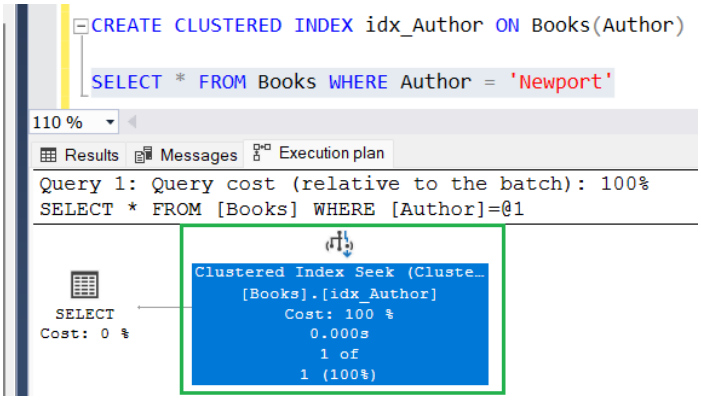


We can actually see details for this clustered index in the object explorer. We look within our database under **Tables | <table name> | Indexes**, like this:



I like how it tells us the index is **Clustered**. We'll learn that there is also a *nonclustered* type of index in SQL Server. SQL Server makes that distinction here in the object explorer.

Now that this clustered index exists, let's execute our query against the Books table again and see what the execution plan looks like:



Fantastic! Now it looks like SQL Server sees that we have established a very helpful structure and ordering to make the job of locating books by Author very easy. The "**Clustered Index Seek**" operation means we are utilizing the clustered index to basically ***zero-in*** on the book we're looking for in our query. Just like in the bookstore, we can *zero-in* on a book if we know that books are ordered on the shelf by author!

Creating a clustered index within a CREATE TABLE statement

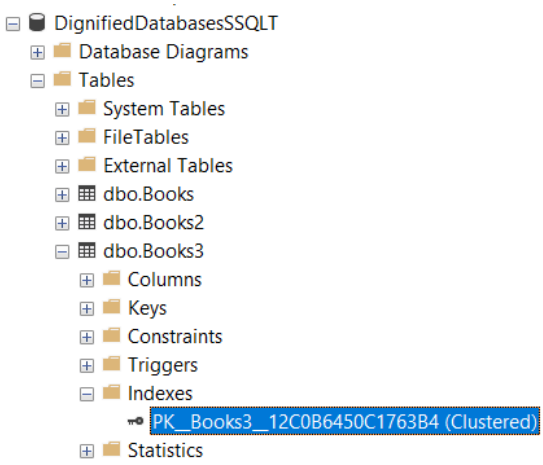
In the previous example, we created our clustered index *after the fact*. We created it after the table had already been created and populated with data. This is fine, but if we know right away that we want to create a clustered index on a table, we can include a bit of details in our CREATE TABLE statement to do it. Here's an example of creating a **Books2** table where we establish that the Author column is the clustered index key:

```
CREATE TABLE Books2
(
   BookID INT IDENTITY(10,5) NOT NULL,
   Title VARCHAR(50),
   Author VARCHAR(15) INDEX idx_Author CLUSTERED,
   Pages INT
)
```

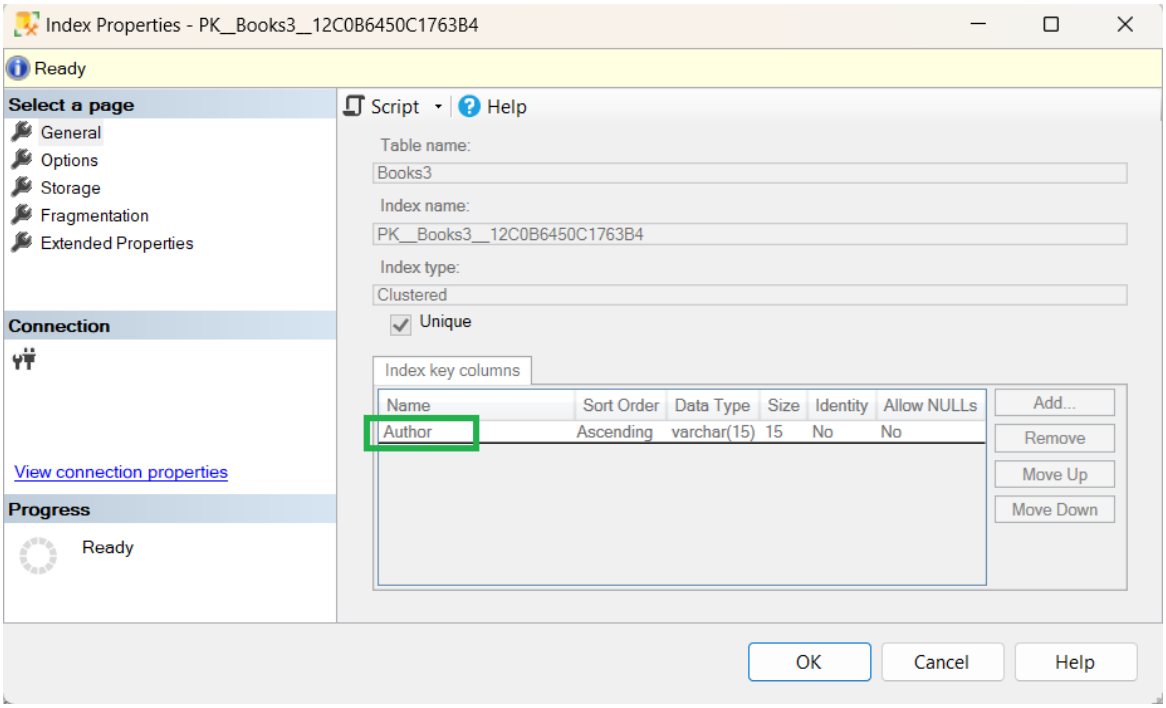
Also, you should know that if we establish a column to be the primary key column, SQL Server will automatically create a clustered index for that table where the clustered index key is that primary key column. Here's an example of a separate **Books3** table where I don't outline a clustered index key at all, but instead make the Author column the primary key:

```
CREATE TABLE Books3
(
   BookID INT IDENTITY(10,5) NOT NULL,
   Title VARCHAR(50),
   Author VARCHAR(15) PRIMARY KEY NOT NULL,
   Pages INT
)
```

If we run that statement, we can navigate to the indexes folder in the object explorer and see that a clustered index was created:



The name of that index is not very intuitive, but maybe that's ok. (Also, the name of *your* index will likely be different). If we look at the properties of this index, we can see that the Author column is indeed the key:



Fantastic.

Let's actually drop both our testing tables for cleanup:

```
DROP TABLE IF EXISTS Books2
DROP TABLE IF EXISTS Books3
```

A clustered index can have a compound key

It's possible to specify two or more columns as your clustered index key. Going back to the bookstore analogy, we've already discussed how books will be in order on the shelf by author last name, but what about the *sub ordering*? It is very common for an author to write more than one book, so how should we order the books within each author? For example, Steven King has written *dozens* of books. All his books will be together on the bookshelf, of course, but what should the *sub ordering* be within his collection of books? Maybe it would be good to order them by **title**. Once again, this helps us locate a particular Steven

King book very easily.

We can set up something like this in an index. The syntax is very similar to what we saw earlier:

```
CREATE CLUSTERED INDEX idx_Author ON Books(Author,Title)
```

All you do is outline your comma-separated list of columns in the parentheses. You just need to make sure you list your columns in the proper ordering. For example, we want our books in order by Author, *and then by Title*, so that's how we list them in the parentheses.

There can only be one clustered index on a table

Folks, you can only order things **one way**. Going back to the bookstore analogy, we can't order our books on the shelf by Author *and* by Title. *That doesn't make sense*. It must be one or the other. If we rearrange our books to be in alphabetical order by Title, then they won't be in alphabetical order by Author anymore, will they?

Now, like we just discussed with compound index keys, it's possible to order by Author *and* ***then*** *by Title*, but that's different. That's referring to primary ordering and secondary ordering. What I'm saying is that you cannot have two **primary** orderings. That doesn't make sense!

The downside of a clustered index

Folks, clustered indexes aren't *flawless*. We need to think back to our bookstore analogy to understand a problem with clustered indexes.

Imagine you are an employee at the bookstore, and you've been asked to add a new book to the Fiction shelf. Maybe the book is "*The Road*" by Cormac McCarthy. If we're trying to keep our books in order by author last name, this means a book written by "McCarthy" will likely need to go somewhere in the middle of our bookshelf.

This means we need to shift several books over to fit the new book on the shelf. If we say our books are *tightly packed* on the bookshelf, this might mean we need to shift *thousands* of books over to properly place our new "McCarthy" book.

I suppose if we had a new book whose author name started with 'Z', maybe this wouldn't be such a big deal. We would likely only need to shift a few books, or maybe none at all if this new book would be placed at the very end.

What if the new book author started with the letter 'A'? Well, we'll need to shift possibly *all* our books over just to fit this one new book!

Folks, it's the same idea in SQL Server. If we add or modify data, it might mean SQL Server needs to do some data shifting to make sure everything stays in the proper order. This data shifting on disk can take some time, especially if there are **millions of rows** in a table!

Understand that when it comes to indexes, there is always a *downside*. As a data professional, you will need to decide if the benefits of an index are great enough to outweigh the costs.

A solution for data shifting

I need to point out a way we can add/modify data and make sure that data shifting will not be necessary or

perhaps extremely unlikely. Let's create an entirely new table called **Orders** and populate it with data:

```
CREATE TABLE Orders
(
 OrderID INT IDENTITY(100,10) NOT NULL,
 BookID INT,
 Qty TINYINT,
 Orderdate DATETIME
)
GO

INSERT INTO Orders(BookID, Qty, Orderdate)
VALUES (15, 1, '2021-06-01'),
(10, 2, '2021-06-06'),
(25, 1, '2021-06-13'),
(40, 1, '2021-07-01'),
(30, 1, '2021-07-06'),
(35, 1, '2021-07-14'),
(35, 1, '2021-07-18'),
(15, 1, '2021-07-24'),
(30, 1, '2021-08-06'),
(20, 1, '2021-08-06'),
(40, 3, '2021-09-01')
GO
```

Here's what the content should look like:

```
SELECT * FROM Orders
```

esults Messages

OrderID	BookID	Qty	Orderdate
100	15	1	2021-06-01 00:00:00.000
110	10	2	2021-06-06 00:00:00.000
120	25	1	2021-06-13 00:00:00.000
130	40	1	2021-07-01 00:00:00.000
140	30	1	2021-07-06 00:00:00.000
150	35	1	2021-07-14 00:00:00.000
160	35	1	2021-07-18 00:00:00.000
170	15	1	2021-07-24 00:00:00.000
180	30	1	2021-08-06 00:00:00.000
190	20	1	2021-08-06 00:00:00.000
200	40	3	2021-09-01 00:00:00.000

This table stores some basic information about Orders that have been placed for our bookstore. In this table, we can see the ID of the book that was purchased. We can also see other helpful details about the quantity they purchased and the date the order was placed.

We need to focus on the **OrderID** column. If we're looking to gather details about a specific order, the best column we can filter on is of course the OrderID column. This is likely the column we would want to filter on in a query, which means it's probably the best column on which to create a clustered index.

Remember, if we were to create a clustered index on the OrderID column, this means rows from this table will be physically arranged on disc by OrderID.

Once again, if we write a query where we're filtering on OrderID, SQL Server ought to be able to find the

order **very easily** since rows are arranged on disk by OrderID!

What happens if we get a new order? Well, that order will be given a new OrderID of course, and that OrderID will be put at the **end** of the clustered index data structure.

Remember, data shifting only happens when new data needs to go ***somewhere in the middle***. In the case of OrderID's, we've set up our table to where a new order will always have a higher OrderID than any orders before it. For example, our OrderID values appear to be generated in increments of 10. The most recent order had an ID of 200, which means the next ought to have an ID of **210**.

Going back to our bookshelf analogy, it's like saying that all new books would always go at the very *end* of the bookshelf, meaning no book-shifting needs to occur!

You might think "*Well, what if an OrderID needs to change? Wouldn't that mean the new ID may need to go somewhere in the middle, i.e. data shifting would need to happen?*"

Well, why would an ID need to change? *It's just an ID*. Who cares what the ID is as long as it's *unique* (and a **primary key constraint** is a great way to ensure column uniqueness, thus ensuring we never need to update an ID because of a uniqueness problem).

Folks, this is the reason why the best clustered index key is an **ever-increasing integer**. This ensures that new numbers always go at the *end* of the clustered index data structure. Integers also take up a relatively small amount of space in memory. The INT data type, for example, only uses **4 bytes** to store a value.

Sometimes it makes sense to create a clustered index on an ever-increasing integer column, but sometimes it doesn't. Thinking back to the bookstore, the best way to uniquely identify a book is usually by the author (or the title, which we'll talk about in the next section). Our table has a BookID column, but we probably aren't going to run too many queries where we filter on *that* column. It makes more sense to establish indexes on columns that will be the **primary means of filtering**.

Let's drop the Orders table for cleanup:

```
DROP TABLE IF EXISTS Orders
```

Nonclustered Indexes

A nonclustered index is a data structure that defines the order of rows according to a column you specify, known as the nonclustered index "*key*". It stores these rows **in memory** and will store a pointer to the row that exists in the clustered index (if the table has a clustered index). If the table doesn't have a clustered index, the nonclustered index will store a pointer to the physical address of the row on disk instead.

A nonclustered index provides a fast way to find a row when filtering on a column that is **not** the clustered index key column.

We need to revisit our bookstore analogy to better understand nonclustered indexes. Let's say you walk into the bookstore, and you are looking for a book with the title "*Debt Free Degree*", but you **don't know who wrote it**. Well, we understand that books are in order on the shelf by *author*, but that doesn't help us. We're essentially reduced to looking through **all the books** to find the one book we're interested in. Again, this will take some time. We may get lucky and find the book towards the front of the shelf, but we might also get *unlucky* and find the book towards the end (or maybe not find the book at all!)

This is exactly the same issue we encountered before. But as we discussed, books can only be ordered on the shelf in one way, so how are we supposed to easily locate a book if all we know is the title?
...
...
...
...