**Indexing (Chapter – 1)**

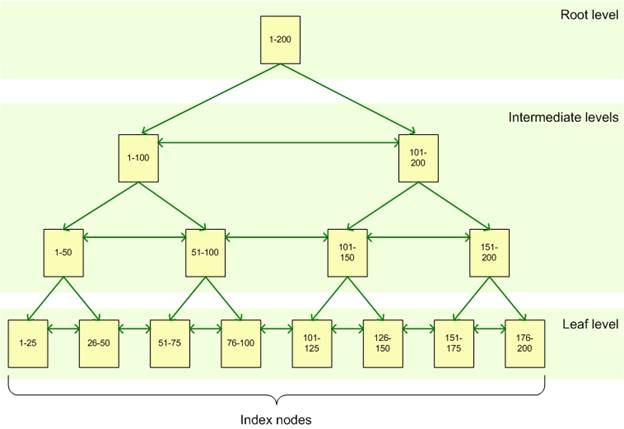
One of the most important routes to high performance in a SQL Server database is the index. Indexes speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book’s index helps you find information quickly within that book. In this article, I provide an overview of SQL Server indexes and explain how they’re defined within a database and how they can make the querying process faster. Most of this information applies to indexes in both SQL Server 2005 and 2008; the basic structure has changed little from one version to the next. In fact, much of the information also applies to SQL Server 2000. This does not mean there haven’t been changes. New functionality has been added with each successive version; however, the underlying structures have remained relatively the same. So for the sake of brevity, I stick with 2005 and 2008 and point out where there are differences in those two versions.

## Index Structures

Indexes are created on columns in tables or views. The index provides a fast way to look up data based on the values within those columns. For example, if you create an index on the primary key and then search for a row of data based on one of the primary key values, SQL Server first finds that value in the index, and then uses the index to quickly locate the entire row of data. Without the index, a table scan would have to be performed in order to locate the row, which can have a significant effect on performance.

You can create indexes on most columns in a table or a view. The exceptions are primarily those columns configured with large object (LOB) data types, such as **image**, **text,** and **varchar(max)**. You can also create indexes on XML columns, but those indexes are slightly different from the basic index and are beyond the scope of this article. Instead, I’ll focus on those indexes that are implemented most commonly in a SQL Server database.

An index is made up of a set of pages (index nodes) that are organized in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom, as shown in Figure 1.

****

**Figure 1: B-tree structure of a SQL Server index**

When a query is issued against an indexed column, the query engine starts at the root node and navigates down through the intermediate nodes, with each layer of the intermediate level more granular than the one above. The query engine continues down through the index nodes until it reaches the leaf node. For example, if you’re searching for the value 123 in an indexed column, the query engine would first look in the root level to determine which page to reference in the top intermediate level. In this example, the first page points the values 1-100, and the second page, the values 101-200, so the query engine would go to the second page on that level. The query engine would then determine that it must go to the third page at the next intermediate level. From there, the query engine would navigate to the leaf node for value 123. The leaf node will contain either the entire row of data or a pointer to that row, depending on whether the index is clustered or nonclustered.

### Clustered Indexes

A clustered index stores the actual data rows at the leaf level of the index. Returning to the example above, that would mean that the entire row of data associated with the primary key value of 123 would be stored in that leaf node. An important characteristic of the clustered index is that the indexed values are sorted in either ascending or descending order. As a result, there can be only one clustered index on a table or view. In addition, data in a table is sorted only if a clustered index has been defined on a table.

**Note:**A table that has a clustered index is referred to as a *clustered table*. A table that has no clustered index is referred to as a *heap*.

### Nonclustered Indexes

Unlike a clustered indexed, the leaf nodes of a nonclustered index contain only the values from the indexed columns and row locators that point to the actual data rows, rather than contain the data rows themselves. This means that the query engine must take an additional step in order to locate the actual data.

A row locator’s structure depends on whether it points to a clustered table or to a heap. If referencing a clustered table, the row locator points to the clustered index, using the value from the clustered index to navigate to the correct data row. If referencing a heap, the row locator points to the actual data row.

Nonclustered indexes cannot be sorted like clustered indexes; however, you can create more than one nonclustered index per table or view. SQL Server 2005 supports up to 249 nonclustered indexes, and SQL Server 2008 support up to 999. This certainly doesn’t mean you should create that many indexes. Indexes can both help and hinder performance, as I explain later in the article.

In addition to being able to create multiple nonclustered indexes on a table or view, you can also add *included columns* to your index. This means that you can store at the leaf level not only the values from the indexed column, but also the values from non-indexed columns. This strategy allows you to get around some of the limitations on indexes. For example, you can include non-indexed columns in order to exceed the size limit of indexed columns (900 bytes in most cases).

## Index Types

In addition to an index being clustered or nonclustered, it can be configured in other ways:

* **Composite index:** An index that contains more than one column. In both SQL Server 2005 and 2008, you can include up to 16 columns in an index, as long as the index doesn’t exceed the 900-byte limit. Both clustered and nonclustered indexes can be composite indexes.
* **Unique Index:** An index that ensures the uniqueness of each value in the indexed column. If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. For example, if you were to create an index on the FirstName and LastName columns in a table, the names together must be unique, but the individual names can be duplicated.

A unique index is automatically created when you define a primary key or unique constraint:

* + **Primary key:** When you define a primary key constraint on one or more columns, SQL Server automatically creates a unique, clustered index if a clustered index does not already exist on the table or view. However, you can override the default behavior and define a unique, nonclustered index on the primary key.
  + **Unique:** When you define a unique constraint, SQL Server automatically creates a unique, nonclustered index. You can specify that a unique clustered index be created if a clustered index does not already exist on the table.
* **Covering index:** A type of index that includes all the columns that are needed to process a particular query. For example, your query might retrieve the FirstName and LastName columns from a table, based on a value in the ContactID column. You can create a covering index that includes all three columns.

## Index Design

As beneficial as indexes can be, they must be designed carefully. Because they can take up significant disk space, you don’t want to implement more indexes than necessary. In addition, indexes are automatically updated when the data rows themselves are updated, which can lead to additional overhead and can affect performance. As a result, index design should take into account a number of considerations.

### Database

As mentioned above, indexes can enhance performance because they can provide a quick way for the query engine to find data. However, you must also take into account whether and how much you’re going to be inserting, updating, and deleting data. When you modify data, the indexes must also be modified to reflect the changed data, which can significantly affect performance. You should consider the following guidelines when planning your indexing strategy:

* For tables that are heavily updated, use as few columns as possible in the index, and don’t over-index the tables.
* If a table contains a lot of data but data modifications are low, use as many indexes as necessary to improve query performance. However, use indexes judiciously on small tables because the query engine might take longer to navigate the index than to perform a table scan.
* For clustered indexes, try to keep the length of the indexed columns as short as possible. Ideally, try to implement your clustered indexes on unique columns that do not permit null values. This is why the primary key is often used for the table’s clustered index, although query considerations should also be taken into account when determining which columns should participate in the clustered index.
* The uniqueness of values in a column affects index performance. In general, the more duplicate values you have in a column, the more poorly the index performs. On the other hand, the more unique each value, the better the performance. When possible, implement unique indexes.
* For composite indexes, take into consideration the order of the columns in the index definition. Columns that will be used in comparison expressions in the WHERE clause (such as WHERE FirstName = ‘Charlie’) should be listed first. Subsequent columns should be listed based on the uniqueness of their values, with the most unique listed first.
* You can also index computed columns if they meet certain requirements. For example, the expression used to generate the values must be deterministic (which means it always returns the same result for a specified set of inputs). For more details about indexing computed columns, see the topic “[Creating Indexes on Computed Columns](http://msdn.microsoft.com/en-us/library/ms189292.aspx)” in SQL Server Books Online.

### Queries

Another consideration when setting up indexes is how the database will be queried. As mentioned above, you must take into account the frequency of data modifications. In addition, you should consider the following guidelines:

* Try to insert or modify as many rows as possible in a single statement, rather than using multiple queries.
* Create nonclustered indexes on columns used frequently in your statement’s predicates and join conditions.
* Consider indexing columns used in exact-match queries.

## Index Basics

In this article, I’ve tried to give you a basic overview of indexing in SQL Server and provide some of the guidelines that should be considered when implementing indexes. This by no means is a complete picture of SQL Server indexing. The design and implementation of indexes are an important component of any SQL Server database design, not only in terms of what should be indexed, but where those indexes should be stored, how they should be partitioned, how data will be queried, and other important considerations. In addition, there are index types that I have not discussed, such as XML indexes as well as the filtered and spatial indexes supported in SQL Server 2008. This article, then, should be seen as a starting point, a way to familiarize yourself with the fundamental concepts of indexing. In the meantime, be sure to check out SQL Server Books Online for more information about the indexes described here as well as the other types of indexes.

**Chapter - 2**

SQL Indexes are used in relational databases to quickly retrieve data. They are similar to indexes at the end of the books whose purpose is to find a topic quickly. SQL provides Create Index, Alter Index, and Drop Index commands that are used to create a new index, update an existing index, and delete an index in SQL Server.

1. Data is internally stored in a SQL Server database in “pages” where the size of each page is 8KB.

2. A continuous 8 pages is called an “Extent”.

3. When we create the table then one extent will be allocated for two tables and when that extent is computed it

is filled with the data then another extent will be allocated and this extent may or may not be continuous to the first extent.

**Table Scan**

In SQL Server there is a system table with the name sysindexes that content information about indexes are available on tables into the database. A table even has no index, there will be one row in the sysindexes table related to that table indicating there is no index on the table. When you write a select statement with a condition where clause than the first SQL Server will refer to the “indid” (index id).

Columns of the “Sysindex” table determine whether or not the column on which you write the conditions has an index. When that indid columns, to get an address of the first extent of the table and then searches each and every row of the table for the given value. This process checks the given condition with each and every row of the table called a table scan. A drawback for tablescan is that if there is no increase in rows in the table then the time taken to retrieve the data will increase and that effects performance.

**Type of Indexes**

SQL Server supports two types of indexes:

1. Clustered Index

2. Non-Clusterd Index.

**1. Clustered Index**  
  
A B-Tree (computed) clustered index is the index that will arrange the rows physically in the memory in sorted order.  
  
An advantage of a clustered index is that searching for a range of values will be fast. A clustered index is internally maintained using a B-Tree data structure leaf node of btree of clustered index will contain the table data; you can create only one clustered index for a table.

**Retrieving data with clustered Index**  
When you write a select statement with a condition in a where clause then the first SQL Server will refer to the “indid” columns of the “Sysindexes” table and when this column contains the value “1”.  
  
Then it indexes the table with a clustered index and in this case it refers to the columns .”The Root” node of the B-tree of clustered index and searches in the b-tree to find the leaf node that contains the first row that satisfies the given conditions and retrieves all the rows that satisfy the given condition that will be in sequence.

**Insert and Update with Clustered Index**

* Since a clustered index arranges the rows physically in the memory in sorted order, insert and update will become slow because the row must be inserted or updated in sorted order.
* Finally, the page into which the row must be inserted or updated and if free space is not available in the page then creating the free space and then performing the insert, update and delete.
* To overcome this problem while creating a clustering index specify a fill factor and when you specify a fill factor as 70 then in every page of that table 70% will fill it with data and remaining 30% will be left free.
* Since free space is available on every page, the insert and update will fast.

**2. Non-clustered Index**

* A non-clustered index is an index that will not arrange the rows physically in the memory in sorted order.
* An advantage of a non-clustered index is searching for the values that are in a range will be fast.
* You can create a maximum of 999 non-clustered indexes on a table, which is 254 up to SQL Server 2005.
* A non-clustered index is also maintained in a B-Tree data structure but leaf nodes of a B-Tree of non-clustered index contains the pointers to the pages that contain the table data and not the table data directly.

**Retrieving data with non-clustered index**

* When you write a select statement with a condition in a where clause then SQL Server will refer to “indid” columns of sysindexes table and when this columns contains the value in the range of 2 to 1000 then it indicates that the table has a non –clustered index and in this case it will refer to the columns root of sysindexes table to get two addresses.

Of the root node of a B-Tree of a non-clustered index and then search in the B-Tree to find the leaf node that contains the pointers to the rows that contains the value you are searching for and retrieve those rows.

**Insert and Update with a Non-clustered Index**

* There will be no effect of insert and update with a non-clustered index because it will not arrange the row physically in the memory in sorted order.
* With a non-clustered index, rows are inserted and updated at the end of the table

**Difference Between Clustered and Non-Clustered Index**

|  |  |
| --- | --- |
| Clustered Index | Non- Clustered Index |
| This will arrange the rows physically in the memory in sorted order | This will not arrange the rows physically in the memory in sorted order. |
| This will fast in searching for the range of values. | This will be fast in searching for the values that are not in the range. |
| Index for table. | You can create a maximum of 999 non clustered indexes for table. |
| Leaf node of 3 tier of clustered index contains, contains table data. | Leaf nodes of b-tree of non-clustered index contains pointers to get the contains pointers to get that contains two table data, and not the table data directly. |

Creating Indexes in SQL Server

To create an index, use the create index command that has the following system.  
  
create [unique][clustered /non clusted] index :

1. <indexname> **on** <object **name**>(<**column** list>)
2. [include(<columnlst>)]
3. [**with** fillfactor=<n>]

**By default an index is non-clustered.**  
  
For example, the following examples create a non-clustered index on department\_no of emp tables.

1. **create** **index** DNoINdex **on** Emp(DeptNo)

Simple & Composite Indexes

* Based on the number of columns on which an index is created, indexes are classified into simple and composite indexes.
* When indexes are created on single columns then it is called a simple index and when the index is created in combination with multiple columns then it's called a composite index.

For example, the following example creates a non-clustered index in combination with department number and job columns of the emp table.

1. **create** **index** dnotedxi **on** emp(deptno **asc**,job **desc**)

**Unique Index**

* When an index is created using the keyword unique then it is called a unique index, whom you create unique index on columns then along with an index unique constraint will be created on the columns.
* If the columns on which you are creating a unique index contains duplicate values then a unique index will not be created and you will get an error.

Altering an Index

To alter an index use an alter index command that has the following syntax:

1. **Alter** **index** <ind **Name**> **on** <object **Name**>
2. rebuild/Recognize/Disable.

The Rebuild option will recreate the computer index, the recognize option will reorganize leaf nodes of the b-tree to index and the disable option will disable the index when the index is eligible then to enable it. alter the index using the rebuild option.  
  
For example, the following example alters the index “dnoidx” available on the department number of columns on the emp table.

1. **alter** **index** DNOiDX **on** EMp rebuild

Getting list of indexes

To get compute a list of indexes available on a table this stored procedure is used.  
  
sp\_helpindex 'Stud'

Deleting indexes

Use the drop index command that has the following syntax:

1. **drop** **index** <indexname> **on** <object **name**>

For example, the following example deletes the index dnoidex available on the department number columns of the emp table.

1. **drop** **index** doindex **on** student

**Chapter - 3**

# **Top 10 questions and answers about SQL Server Indexes**

## Introduction

Without a doubt, few technologies in SQL Server cause as much confusion and the spread of misinformation than indexes. This article looks at some of the most asked questions and a few that should be asked but often are not. We’ll be using SQL Server 2016 for the examples and a free tool, for SQL Server query execution plan analysis, [ApexSQL Plan](https://www.apexsql.com/sql_tools_plan.aspx), to explore the effects of indexes on a typical business problem: A table of customers.

## Question 1: What is an index?

Once upon a time, the most common examples of where indexes are used were dictionaries and telephone books. In today’s connected society with available online resources that would have been dismissed as little as twenty years ago as pure fantasy, it’s entirely possible that you have never held either one in your hands! So, let’s look at an online resource: The list of endangered species maintained by the World Wildlife Fund on their website at [www.worldwildlife.org](http://www.worldwildlife.org/). The list begins like this:



It continues and has two pages as of this writing. A quick glance at the list tells you that the species are arranged alphabetically by their common name. Imagine though, that you are a biologist and are used to using the Latin names. How would you find the entry for the Thunnus and Katsuwonus species? Well, you’d need to read right to the very end of the list, since that has the common name tuna (like the one in your sandwich!) and is the last one in the alphabetical list by common name. “Fine!” you think. It’s not so hard to read through two pages to find what I’m looking for.

Now, imagine that this list holds all the known species in the world (about 8.7 million) similarly arranged. Reading through that list to find a species by its Latin name would not be much fun for you and would need a lot of I/O on a computer. In a database, I/O is often (if not always) the single greatest bottleneck.

If we can reduce the I/O required to find the information we want, we can effectively give the bottle a bigger neck.

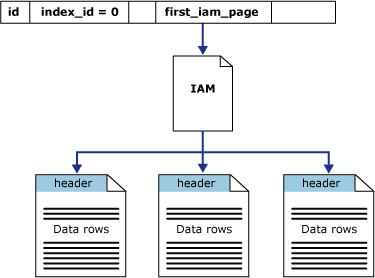
This reduction will also lead to secondary benefits, such as reduced CPU time, waits, cache use and more.

The goal of an index on a database table then is to reduce I/O. To understand how this works, we need to first look at a table that has no indexes.

## Question 2: What does a table look like without indexes?

SQL Server keeps all data in all its files for all databases in 8K pages. There are at least two files for every database: one for the data, which has the default file type .mdf, and one for the log, which uses .ldf for the default file type. Each table in the database has one or more pages. To keep track of those pages, SQL Server uses a special set of pages, called IAM (for Index Allocation Map) pages. In spite of the word “Index” in the name, IAMs are used for non-indexed tables as well. These are called heaps.

A heap is pretty much like what its name implies: an unordered pile of stuff. That could be your dirty laundry, or leftover construction materials, or the mess left on the beach by a high tide or, as in this case, a bunch of pages for a table in SQL Server. Nothing is organized in any way, except the IAM pages, which are chained together so SQL Server can find all the data pages for the table. Graphically it looks a bit like this:



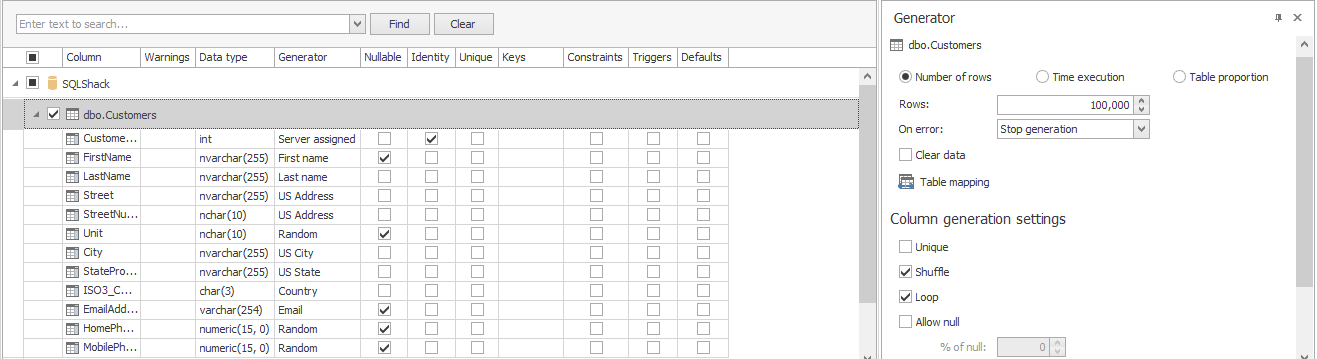
Source: ©[Microsoft.com](https://i-technet.sec.s-msft.com/dynimg/IC22444.gif)

All the data is there, but the only way to find anything is to read it starting at the beginning. For a very large table, this will be terribly inefficient. Let’s see what happens with just such a table.

We’ll start off by creating a table of customers. First, here’s the DDL for the customer table:

|  |
| --- |
| CREATE TABLE [dbo].[Customers](  [CustomerID] [int] IDENTITY(1,1) NOT NULL,  [FirstName] [nvarchar](255) NULL,  [LastName] [nvarchar](255) NOT NULL,  [Street] [nvarchar](255) NOT NULL,  [StreetNumber] [nchar](10) NOT NULL,  [Unit] [nchar](10) NULL,  [City] [nvarchar](255) NOT NULL,  [StateProvince] [nvarchar](255) NOT NULL,  [ISO3\_Country] [char](3) NOT NULL,  [EmailAddress] [varchar](254) NULL,  [HomePhone] [numeric](15, 0) NULL,  [MobilePhone] [numeric](15, 0) NULL  ) ON [PRIMARY] |

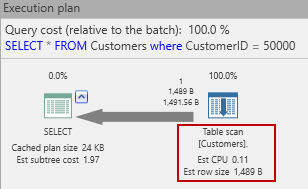
Next, let’s fill it up with data using [ApexSQL Generate](https://www.apexsql.com/sql_tools_generate.aspx):

[](https://www.sqlshack.com/wp-content/uploads/2017/10/word-image-44.png)

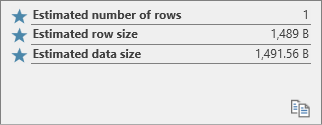
Now, let’s use ApexSQL Plan to explore this simple query:

|  |
| --- |
| SELECT \* FROM Customers where CustomerID = 50000; |

When we view the estimated execution plan we see:

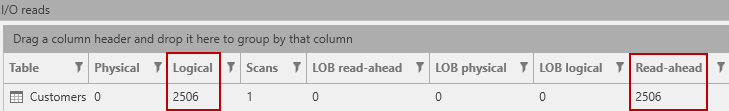


The main thing SQL Server will do is a table scan. That means it will read all the rows of this table until it finds one with a customer id of 50000. Interestingly, if we hover over the big right-to-left arrow in the estimated plan, we see this:



Estimated rows = 1! That’s because SQL Server expects just one customer with the matching id. There could be more (since at this point there’s nothing to prevent it) but there’s just no way to know for sure.

OK, now let’s run this query to see what we can learn. The most interesting thing to see is the I/O reads tab:



There are 2506 logical reads – one for every page in the table. Also, there is an equal number of read-ahead reads. These are physical reads that SQL Server does in anticipation of their being needed. So, how much of the table is that? This query tells us how many pages are used by this table:

|  |
| --- |
| SELECT      t.NAME AS TableName,      p.rows AS RowCounts,      SUM(a.total\_pages) AS TotalPages,      SUM(a.used\_pages) AS UsedPages,      (SUM(a.total\_pages) - SUM(a.used\_pages)) AS UnusedPages  FROM      sys.tables t  INNER JOIN      sys.indexes i ON t.OBJECT\_ID = i.object\_id  INNER JOIN      sys.partitions p ON i.object\_id = p.OBJECT\_ID AND i.index\_id = p.index\_id  INNER JOIN      sys.allocation\_units a ON p.partition\_id = a.container\_id  WHERE      t.NAME = 'Customers'      AND t.is\_ms\_shipped = 0      AND i.OBJECT\_ID > 255  GROUP BY      t.Name, p.Rows  ORDER BY      t.Name |

And this query tells us:



That’s right! SQL Server had to read all but one of the pages used in the table! Perhaps you thought that it would find the matching row about half-way through? That would be the expected runtime, right? O(n/2)? But since SQL Server is doing read ahead operations, to minimize the number of read operations, it winds up reading all the data pages. (The other page contains metadata.)

If this table was part of a busy OLTP system and had millions of rows and thousands of simultaneous searches, this would put a nasty drag on system resources. Even if you could hold the whole table in the buffer pool, that would make those buffers unavailable for other queries. Either way, this will make for a sickly server. Appropriate indexing is the cure!

## Question 3: What types of indexes are available in SQL Server?

SQL Server supports indexing for a variety of needs. Borrowing from the Microsoft documentation, these are available in SQL Server 2016:

|  |  |
| --- | --- |
| **Type** | **Description** |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the index key. This type of index is implemented as a B-tree structure that supports fast retrieval of the rows, based on their key values. |
| Nonclustered | A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each row in the index contains the key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the key values, but the data rows are not guaranteed to be in any particular order unless they are in a clustered index. |
| Unique | A unique index ensures that the key contains no duplicate values and therefore every row in the table or view is in some way unique. |
| Index with included columns | A nonclustered index that is extended to include nonkey columns in addition to the key columns. |
| Full-text | A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. |
| Spatial | A spatial index provides the ability to perform certain operations more efficiently on spatial objects (spatial data) in a column of the **geometry** data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. |
| Filtered | An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column. |

Source: ©2017 Microsoft

Full-text, Spatial and XML indexes are outside the scope of this article. We’ll concentrate on clustered and non-clustered indexes and look at some of their extensions including unique and filtered indexes and the usefulness of included columns. Although not mentioned above, we will also not be looking at columnstore indexes in this article, nor in-memory tables.

As described above, a clustered index affects how the data is actually stored. In a heap, the data rows are stored in no particular order. They are written wherever they fit while putting the minimum load on SQL Server resources, such as the buffer pool and the I/O subsystem. On the other hand, when you create a clustered index on a table, the organization of the data is changed so that it is now in order according to the keys specified. The entire index is organized as a B-tree (“B” stands for balanced), where the leaf nodes are the actual data pages and one or more levels of index nodes are built on top of the leaf nodes up to the single, root node. The result provides some guarantees regarding the asymptotic performance of the index. Most operations (search, insert and delete) operate in O (log n) where n is the number of entries in the index.

A nonclustered index shares the B-tree concept for the index nodes with the same performance guarantees. However, such indexes do not affect the organization of the data pages, which may be clustered or not. Some optional features of nonclustered indexes are:

* Unique indexes – where the index entries must be unique and SQL Server makes sure that they are
* Filtered indexes – which are indexes built with a WHERE clause to limit what gets included in the index
* Included columns – which can carry a subset of non-key columns as part of the index.

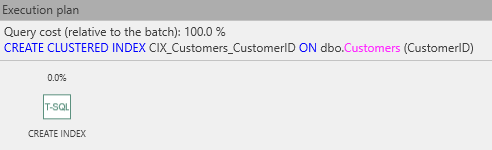
If you think through the implications of these descriptions, you should be able to see that a table is either a heap or a clustered index. Other implications are that since the leaf node of a clustered index is a data page, there is no need for included columns (since all columns are in the data page) or filtered indexes (since the clustered index is the whole table, by definition). A more subtle insight is this: Since B-Tree theory does not stipulate key uniqueness, a clustered index may have rows with duplicate keys. In that case, SQL Server will add a hidden uniqueifier (a 4-byte integer) to the index to enforce uniqueness.

## Question 4: What happens when you create a clustered index?

As explained in the answer to question 3, above, when you create a clustered index, the order of the rows in the data pages is changed. Adding a clustered index to our working example, the command is a simple one:

|  |
| --- |
| CREATE CLUSTERED INDEX CIX\_Customers\_CustomerID      ON dbo.Customers (CustomerID); |

We can view and run this in ApexSQL Plan. First, let’s see the estimated execution plan:

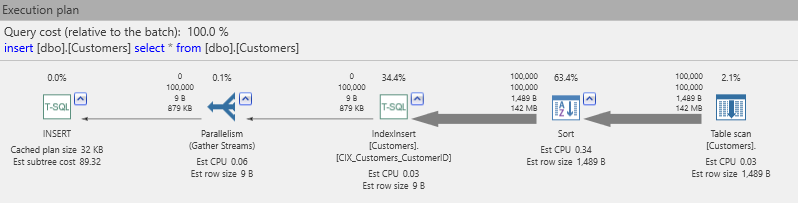




The estimated plan shows just an operation to create an index. That means that it is handled internally and/or SQL Server prefers not to expose the details. However, it is easy to deduce what must happen:

1. Read all the pages of the table
2. Sort the rows by the key specified
3. Fill new pages with the sorted rows (up to the fill factor)
4. Harden the new pages to persistent storage
5. Free up the pages used by the data rows before the create index.

If we actually execute this, we should be able to see some of this action:

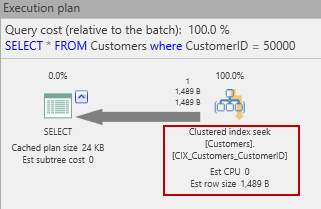
[](https://www.sqlshack.com/wp-content/uploads/2017/10/word-image-50.png)

This is pretty much as expected! Reading from right-to-left, the table contents are read (table scan), 100,000 rows are sent for sorting, then the sorted data set is sent to an IndexInsert operation, which creates the index nodes. The parallelism operator indicates provision for a parallel IndexInsert operation, which can be run in many parallel streams depending on the number of available CPUs. Finally, the new rows are INSERTed into the table. We don’t see an operator for releasing the now-unused pages, but then that is a background operation that you will never see in an execution plan.

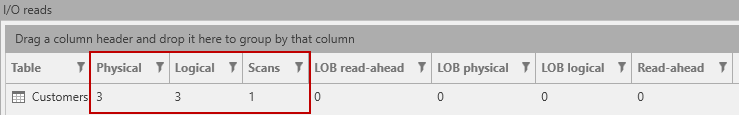
Now that we have a clustered index in place, will that affect our original query?

|  |
| --- |
| SELECT \* FROM Customers where CustomerID = 50000; |

Now the estimated plan looks like this:



The original table scan has been replaced with a clustered index seek. Will that help reduce the I/O used to get that row? Let’s see!



That’s a huge difference! The original 2500 reads have been reduced to just three. Since we know that there is only one data page for this row (since the row has a total length < 8090 – a topic for a different article), we can infer that the other two I/Os are for pages containing the index nodes. We can confirm this with a simple query:

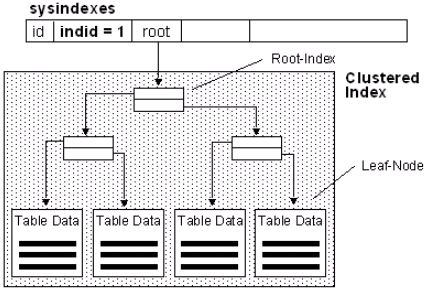
|  |
| --- |
| SELECT      INDEXPROPERTY(OBJECT\_ID('dbo.Customers'),          'CIX\_Customers\_CustomerID','IndexDepth') AS [Index Depth] |

Which returns:



matching our expectations, and the I/O reads.

Graphically, our clustered index now has this structure:



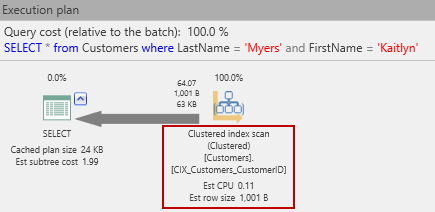
Source: ©[Microsoft.com](https://i-technet.sec.s-msft.com/dynimg/IC155040.gif)

## Question 5: What about nonclustered indexes?

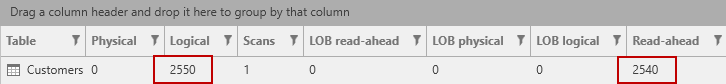
So far, we’ve only been looking for customers by their customer id. What if we wanted to look up a customer by name instead? Something like this, perhaps:

|  |
| --- |
| SELECT \* FROM Customers WHERE LastName = 'Myers' AND FirstName = 'Kaitlyn'; |

The estimated execution plan is now:



Oh no! We’re back to a scan operation. (This time, it’s a clustered index scan, since as mentioned before, a table is either a heap or a clustered index, and we just made the Customers table into the latter. But it get’s worse! The actual I/O reads have gone up!

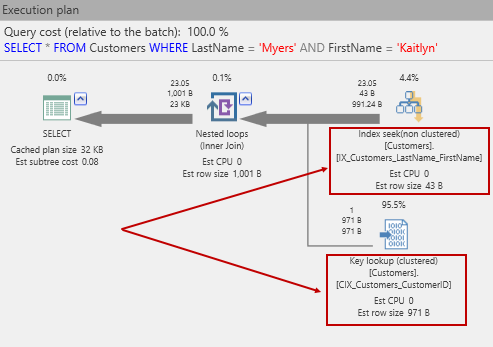


That’s because SQL Server is now also using the index nodes as it scans the data pages. What can be done about this situation? Add a nonclustered index, of course!

Let’s use this DDL to create the index:

|  |
| --- |
| CREATE NONCLUSTERED INDEX IX\_Customers\_LastName\_FirstName      ON dbo.Customers (LastName, FirstName); |

With that in place, let’s retry that SELECT. This time the plan looks like this:

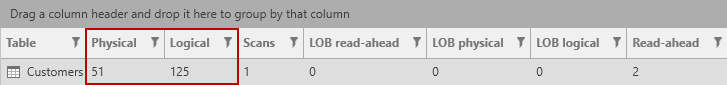


Now, we can see that the does an index seek, using the new index, followed by a key lookup. The key lookup is necessary since we’ve written:

|  |
| --- |
| SELECT \* FROM Customers … |

and SQL Server has to go back to data pages to get the other columns in the row. The reason is a Key lookup because of how SQL Server builds nonclustered indexes. You would be forgiven for thinking that the nonclustered index contains a pointer to the pages containing the data row. Actually, that’s not how it works. Instead of a pointer (e.g. a sector number on disk), a nonclustered index keeps the clustered index key if there is a clustered index. Of course, before we got here, we put a clustered index on the Customers table! As a consequence, we get a key lookup. If you guessed that this means a separate lookup into the clustered index, you’d be right!

So, does the nonclustered index reduce our I/Os? Let’s do an Actual run and see:



As shown, there is a substantial reduction in I/O. Perhaps you were expecting just 3 reads as in the clustered index example we used before. However, there is no guarantee that there is just one Kaitlyn Myers in the table. Let’s check that:

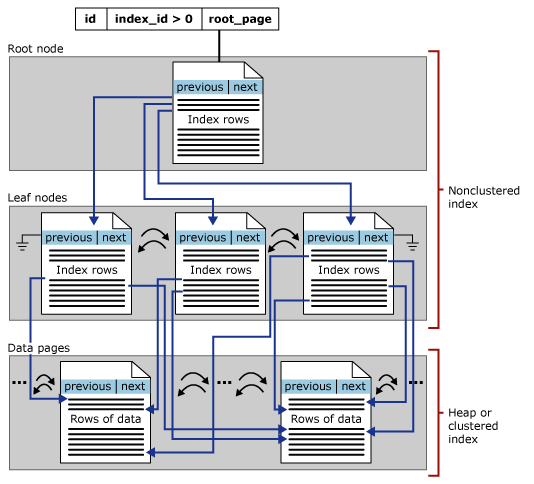
|  |
| --- |
| SELECT COUNT(\*) AS Kaitlyns FROM Customers  WHERE LastName = 'Myers' AND FirstName = 'Kaitlyn'; |

returns:



So, the I/O reads are actually quite realistic.

The name “nonclustered” stems from the simple fact that this type of index is not a clustered index. What it is, is a B-tree built atop a table (which may be clustered or a heap). So, if there is also a clustered index, the nonclustered index lives alongside the clustered index, and its entries point to the leaf level of that index – the data pages. A nonclustered index has a structure like this:



Source: ©[Microsoft.com](https://i-technet.sec.s-msft.com/dynimg/IC88960.gif)

## Question 6: What about included columns? How can they help?

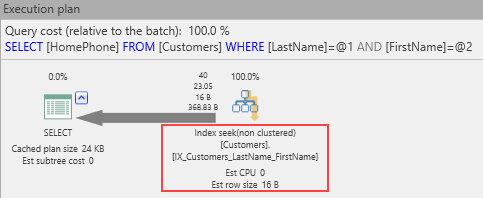
You saw that a query using a nonclustered index had to lookup other columns with a key lookup into the clustered index. What if you frequently queried just a subset of the other columns? What if you were often looking up the customer’s phone number? That’s where included columns come in. If I modify the nonclustered index like this:

|  |
| --- |
| CREATE NONCLUSTERED INDEX IX\_Customers\_LastName\_FirstName      ON dbo.Customers (LastName, FirstName) INCLUDE(HomePhone); |

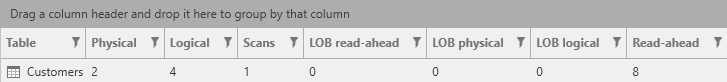
then run this query:

|  |
| --- |
| SELECT HomePhone FROM Customers      WHERE LastName = 'Myers' AND FirstName = 'Kaitlyn'; |

The execution plan changes to this:



No more key lookup! The I/O reads are reduced as well:



So, we can see the positive benefit of included columns. This naturally raises the question, “Why not just add the included columns to the index in the first place?” To understand why, first consider this: While key columns are stored at all levels of the index, nonkey columns are stored only at the leaf level. Lest there be any confusion, although a nonclustered index points to the data pages, its leaves are part of the B-tree itself.

Why not put the included columns in the index keys? There are a few reasons:

* Storage cost. If the included columns are not needed for other types of searches, keeping them out of the key list makes for shorter index entries and a flatter B-tree. That results in less index I/O.
* SQL Server limits. Currently, you can have no more than 16 key columns in an index, and altogether those key columns cannot exceed the maximum index size of 900 bytes.
* Included columns can be data types that are not allowed as index columns. For example, nvarchar(max), varbinary(max), xml and others cannot be key columns but can be included columns.

When an index contains all the columns needed to satisfy a query, it is called a covering index. Including strategic columns in a nonclustered index can ensure that the most frequent queries can be satisfied entirely from the index without the need for key lookups into the clustered index.

Though we haven’t mentioned it, you can put a nonclustered index on a heap – that is a table without a clustered index. In that case, any lookups into the data pages are RID lookup. The “R” stands for Row and RID consists of a virtual address that includes the file number and page number of the data page and the row slot within the data page.

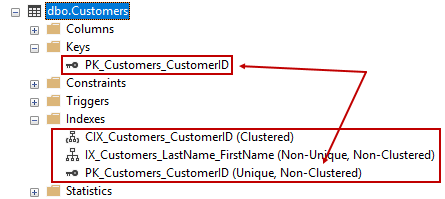
## Question 7: What about primary keys (and keys in general)?

So far, we’ve avoided any discussion of primary keys. That’s not an accident. Too often clustered indexes and primary keys are conflated as if they are one and the same thing. They are not! That’s why we’ve left them until now. It’s best to get a solid idea about types of indexes before introducing keys.

To be a relation in the formal, relational algebraic sense of the word, a table (what most RDBMSs call relations) must have a key – some column or collection of columns that, taken together, uniquely identify a row in a table. However, a key is not an index. It may be (and usually is) supported by an index, but at its heart, a key is a constraint – a condition that the database must maintain to preserve referential integrity. You can see the difference by creating a primary key on the Customers table in the example we’re using:

|  |
| --- |
| ALTER TABLE dbo.Customers      ADD CONSTRAINT PK\_Customers\_CustomerID PRIMARY KEY (CustomerID); |

Notice that the statement adds a constraint, not an index. It adds just such a constraint on the CustomerID column. Now, since we already have both a clustered index and a nonclustered index, this produces an interesting result. In SSMS Object Explorer, we now see three indexes!



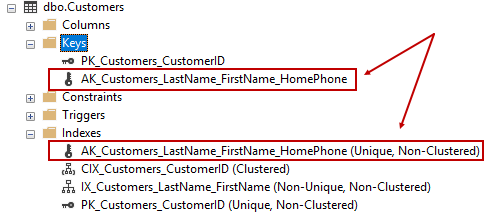
The new one has been created as a backing index to implement the constraint. Note that it is nonclustered. If you thought that primary keys have to be clustered indexes, think again. It is interesting that SQL Server did not recognize that there is already a covering clustered index and use that one. I suspect that building indexes the way we’re doing it here is good for instruction but not optimal for proper design, so the SQL Server development team has not yet worked on this edge case and may never do so.

Before we added the primary key constraint, it was possible and permissible to add new customers with the same customer IDs as existing customers. Go ahead, try it! Now that the PK is in place, though, that is no longer possible.

Although there can be only one primary key, there can be other keys, if other combinations of columns are truly unique. Creating them uses similar syntax:

|  |
| --- |
| ALTER TABLE dbo.Customers      ADD CONSTRAINT AK\_Customers\_LastName\_FirstName\_HomePhone     UNIQUE (LastName, FirstName, HomePhone); |

This operation creates a unique constraint and also adds a backing index:



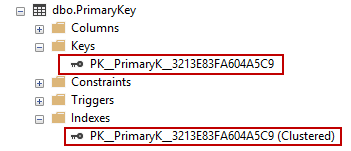
Primary keys are necessary to make a table conform to relational theory. In practice, they are highly useful (as are unique keys) for referential integrity and a good way to get the database to check your assumptions about incoming data.

## Question 8: Should my primary key also be the clustered index key?

By default, SQL Server creates a clustered index when you create a new table with a primary key:

|  |
| --- |
| CREATE TABLE PrimaryKey      (id INT IDENTITY PRIMARY KEY,  name VARCHAR(50)      ); |

This command creates a key backed by a clustered index (with a system-supplied name):



However, just because it’s the default action, that doesn’t mean its always the best choice. While a primary key needs to be backed by an index to implement the constraint, the index does not need to be clustered, as we saw above. Furthermore, the goals of primary keys and clustered indexes may not be the same.

Fundamentally, a primary key constraint enables SQL Server to maintain the important property that a table may not have duplicate rows. In fact, unique constraints do exactly the same thing. Because they are backed by indexes, keys are also helpful for searches, since

1. There can be only one row for any given key
2. The index allows for a seek operation to be used, which normally takes less I/O.

On the other hand, clustered indexes can provide a performance advantage when reading the table in index order. This allows SQL Server to better use read ahead reads, which are asymptotically faster than page-by-page reads. Also, a clustered index does not require uniqueness. If two or more rows have the same clustered index key, a uniqueifier is added to ensure that all rows in the index are unique.

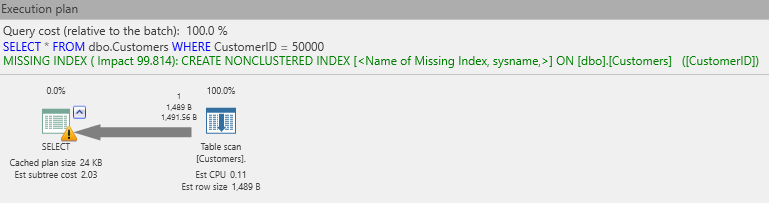
Going back to our Customer table, this is a perfectly valid index/key setup:

|  |
| --- |
| CREATE CLUSTERED INDEX IX\_Customers\_LastName\_FirstName      ON dbo.Customers(LastName, FirstName);    ALTER TABLE dbo.Customers      ADD CONSTRAINT PK\_Customers\_CustomerID     PRIMARY KEY NONCLUSTERED(CustomerID); |

Here, we’re assuming that most reads are done by last name, first name (the clustered index) and that most searches are done by CustomerID (the primary key). (SQL Server will add a uniquifier to the clustered index since it doesn’t know if there will be key collisions or not.) The point here is to carefully consider which arrangement is best for a new table. Consult with business users about how the table will be used. Remember that most reporting is sequential in nature. Choose your indexing accordingly, then test it. Use real workloads and see if actual usage is still a good match for the indexing scheme. If not, adjust it to suit.

## Question 9: What are those “missing indexes” in an execution plan

Sometimes, a message will be included with the execution plan that an index is missing. For example, if I remove the indexes from the Customer table and try a search by CustomerID, the resulting plan looks like this:

[](https://www.sqlshack.com/wp-content/uploads/2017/10/word-image-65.png)

The text in green, above, is just such a message. Basically, SQL Server is saying that if there were an index on the CustomerID column, the cost of the overall query could be reduced by 99.8%! That would reduce the cost to almost nothing. Instead of a table scan, it could use an index seek operation.

The question then becomes, should you always implement a missing index? As is often the case, “it depends!” is the only true answer. Is this a once-a-year query that runs in 10 minutes without the index? Perhaps you can leave that missing index alone. Is it a query that gets executed thousands of times a second from a busy e-commerce server? Probably you should implement the recommendation! Don’t assume that mommy knows best! You need to think it through and understand the usage patterns and the cost of an index before you do anything. Come to think about it…

## Question 10: Can I have too many indexes?

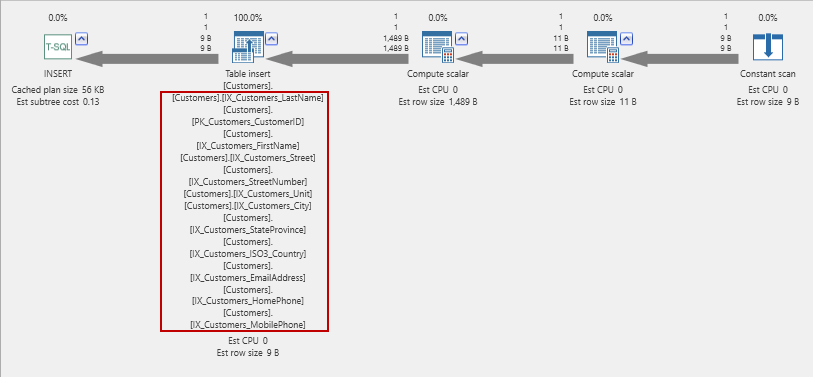
There is actually advice out in the wild stating that you should just index every column in your table so that no execution plan will ever have a missing index. Is this good advice? Once again, it depends.

Data warehouse databases often have a calendar table. That table will have a row for every date and columns for every conceivable use: MonthNumber, QuarterNumber, HalfYearNumber, MonthText, FullDateText, Is\_Holiday, Is\_BusinessDay, and many, many more. Such a table is static. Often the primary key (and clustered index) is an integer or date column with the date matching the other columns in the row. This table never (or rarely) changes. Also, it is not too big. (How many rows would you need for a century?) On such a table, indexing every column can make sense. You can have filtered indexes for the various flags, ascending and descending indexes, most of which will be nonclustered.

Now consider our Customer table. If we index every column, what happens when we insert a new customer?

|  |
| --- |
| INSERT INTO [dbo].[Customers]             ([FirstName]             ,[LastName]             ,[Street]             ,[StreetNumber]             ,[Unit]             ,[City]             ,[StateProvince]             ,[ISO3\_Country]             ,[EmailAddress]             ,[HomePhone]             ,[MobilePhone])       VALUES             ('Ford',              'Prefect',              'The Resraurant',              '42',              'N/a',              'End of the Universe',              'Improbable',              'FPP',              'Ford@HeartOfGold.com',              2125551212,              3141592653              )  GO |

The actual execution plan for this is:

[](https://www.sqlshack.com/wp-content/uploads/2017/10/word-image-66.png)

All those indexes must be maintained! This work can add up quickly, so unless you have a static table (such as a calendar table), indexing every column is usually a bad idea!

## Summary

Properly indexing a SQL Server table is key to providing consistent, optimal performance. There are many things to consider when designing an index structure. Almost none of those considerations can be done without consulting the business users who understand the data and how it will be used, though it may well turn out that, after the new table has been in use for some time, that other indexing options will present themselves.

This article has attempted to answer some of the most important questions about indexing in general and specifically how it is done in SQL Server. We’re really only scratched the surface though. Each question answered here could be its own complete article and possibly a series of articles! Also, we’ve not addressed other types of indexes at all (e.g. XML indexes) nor have we looked at in-memory objects and how those change the picture for newer editions of SQL Server.