

Brad's Sure Guide to Indexes

11 June 2009

*by*[*Brad McGehee*](https://www.simple-talk.com/author/brad-mcgehee/)

In order to best appreciate the reasons for, or importance of, the various administrative tasks that are needed for indexes, it is a good idea to take a look at how the various types of indexes work. Who better to help with a clear expanation for this than Brad McGehee?

In this article, we take a brief look at the different kinds of indexes that exist in SQL Server, and how they work. To get us started on this journey, we first need to answer these two questions:

* What is a table?
* What is an index?

Once we are clear on these terms, we will then explore each of the following variations of indexes available in SQL Server 2005 and SQL Server 2008:

* Clustered
* Non-clustered
* Indexed view
* Computed
* XML
* Filtered (New with SQL Server 2008)
* Spatial (New with SQL Server 2008)
* Compressed Indexes (New to SQL Server 2008)

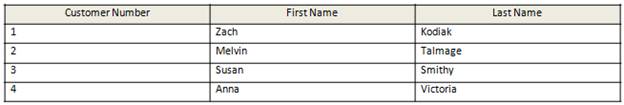
If you are already intimately familiar with all of the above, then there is no need for you to read this article. If, on the other hand, you need to start with a simple introduction to indexes for the beginner, please read Rob Sheldon's excellent [SQL Server Index Basics](http://www.simple-talk.com/sql/learn-sql-server/sql-server-index-basics/) first.

**Note:** The explanations in this article have been simplified to make the concepts easier to understand. In the real world, things are more complex. For complete coverage of SQL Server index internals, read the book *Microsoft SQL Server 2008 Internals* by Kalen Delaney et al.

# What is a Table?

Tables can be thought of in either logical or physical terms. For example, when most DBAs think of a table, they see a two-dimensional set of values arranged into rows and columns, similar in appearance to a spreadsheet.  Each column represents a different data element, and each row represents a collection of related data elements.

For example, in figure 2-1 below, we see a representation of a logical table that has three columns and four rows. This should look very familiar to DBAs as this is how most of us visualize what a table in a database looks like. It also looks similar to the results we see when we perform a SELECT statement on a table within Management Studio.

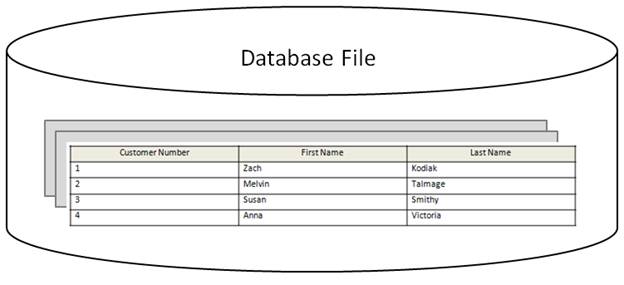


***Figure 2-1: A logical table with three data columns and four rows****.*

While a logical visualization of table is very handy for DBAs to use in their work, this is not how SQL Server sees a table. When a table is created, it has to be implemented physically inside a database.

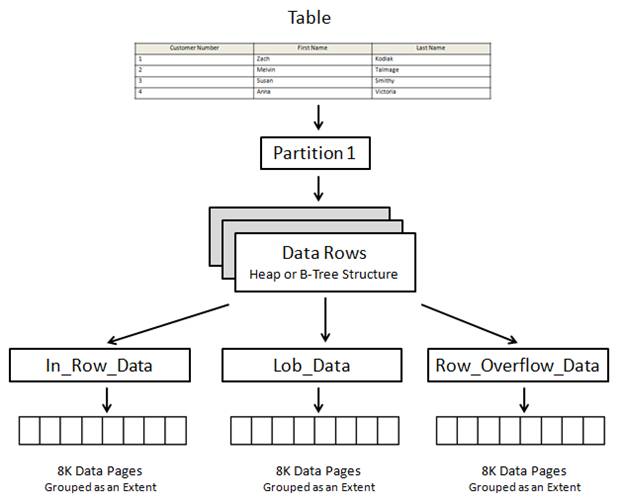
For example, the most basic structure is the database file (.mdf and .ndf files). The database file can include many objects, but for now, we want to focus on tables. As seen in figure 2.2 below, a database file will include one or more tables.

(Note: I am ignoring transaction the log file (.ldf) because it is not part of a table’s physical structure.)



***Figure 2-2: A database file is the physical container that holds tables.***

Each table in a database can be broken into multiple components. Let’s take a brief look at each of them so we better understand what makes up a physical table within SQL Server.



***Figure 2-3: A table has many components.***

At the top of figure 2-3, we start with the table, and then we see it split into many components. The first component is a partition. A partition is simply a way DBAs can sub-divide rows in a table (and the table’s indexes) into separate sections, generally for the purpose of horizontally spreading rows across more than one filegroup. This often makes it easier for DBAs to manage and scale very large tables. The topic of partitioning is outside the scope of this article, so all the following examples will assume there is a single partition (which is the default option when creating a table). In addition, only the Enterprise and Developer editions of SQL Server support partitioning.

Partitions contain data rows stored in either the physical form of a heap (a table without a clustered index) or a B-Tree structure (a table with a clustered index). We will go into a more detailed explanation of these structures in the next section.

Heap and B-Tree structures contain one or more allocation units. This is just a fancy way of subdividing different data types into three categories, which are:

* **In\_Row\_Data**: This is the most common type of allocation unit, and is used to store data and index pages, except for Large Object (LOB) data. In other words, most of your row data (and related indexes) are stored in this type of allocation unit.
* **Lob\_Data**: Used to store Large Object (LOB) data, such as text, ntext, xml, image, varchar(max), nvarchar(max), varbinary(max), and CLR user-defined data types.
* **Row\_Overflow\_Data**: Used to store data pages when the variable data types—varchar, nvarchar, varbinary, and sql\_variant data columns—that exceed the 8,060 bytes that can fit onto a single data page.

If a heap or Be-Tree structure does not have a need for all three allocations units, then they may or may not have them. Besides the data described above, each of these allocation units also include various metadata to help manage the data within each allocation unit.

Allocation units themselves are made up of extents. In figure 2-3, we see that each allocation unit has one extent. In the real world, each allocation unit would include many, many extents.

An extent is a collection of eight contiguous 8K pages (for a total of 64K). A page is the most elemental unit of data storage used by SQL Server, and will be the focus of our next discussion.

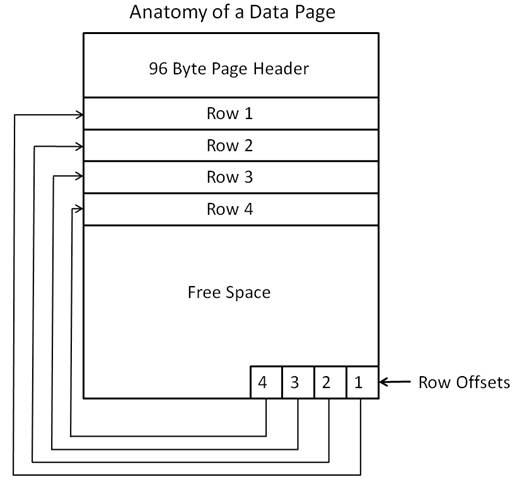
**Note:** Why are extents used to group eight 8K pages? Extents are used because it is more resource efficient for SQL Server to allocate eight, 8K pages in a single step than it is for SQL Server to create eight, 8K pages in eight separate steps. There are two types of extents: uniform and mixed extents. All eight pages of a uniform extent are all owned by the same object. In a mixed extent, each page of the extent can be owned by a different object. Why are there two different kinds of extents? This is because many objects in a database are less than 64K in size. If only uniform extends were allowed, then there would be a lot of wasted space on extents that are not fully used by an object. By allowing mixed extends, multiple, smaller objects can all fit onto a single extent, which makes more efficient use of available space. Whenever a new table or index is created, it generally is first created on a mixed extent. What happens if an object on a mixed event grows to exceed 64K? If that happens, all future page allocations are done using uniform extents.

While the basic unit of data storage in SQL Server is the page, there are eight different types of pages available for SQL Server’s use. They include:

* **Data**: Stores most data types, except for text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data (when the “text in row” option is set to on).
* **Index**: Stores indexes (root and intermediate level pages of a B-Tree structure).
* **Text/Image**: Stores LOB data types, such as text, ntext, nvarchar(max), varchar)max), xml data, and if variable length columns exceed 8K, then varchar, nvarchar, varbinary, and sql\_variant data.
* **Global Allocation Map (GAM)/Shared Global Allocation Map (SGAM)**: Stores tracking information about extents that have been allocated.
* **Page Free Space (PFS)**: Stores information about allocated pages, and the amount of free space available on them.
* **Index Allocation Map (IAM)**: Stores data about extents that are used by a table or index, per allocation unit.
* **Bulk Changed Map**: Stores data about any extents that have been changed by a bulk operation since the last transaction log backup, per allocation unit.
* **Differential Changed Map**: Stores data about any extents that have been changed since the last database backup, per allocation unit.

For the purpose of this article we will focus mostly on data, index, and Text/Image data pages, which are the most relevant to us in our goal of high performance index maintenance. When we talk about heaps later in this article, we will briefly discuss IAM pages.

Because pages are so important to us, let’s take a quick look at what a data page looks like.



***Figure 2-4: This is an idealistic depiction of a data page.***

Every SQL Server page (8,192 bytes) begins with a 96-byte header used to store metadata about the page. This includes the page’s number, page type, amount of free space on the page, the allocation unit ID of the object that owns the page, among other metadata.

**Note:** We need to learn how data pages work, as this directly affects our understanding of index fragmentation

Immediately following the page header is the data to be stored, one row after another. Each page can store a maximum of 8,060 bytes. The number of rows that can be stored on a page depends on the size of the rows. For example, the smaller the rows, the more rows that can fit on a single data page. When a page becomes full, any new data will be inserted in another data page.

Figure 2-4 is not a realistic depiction of how rows are stored on a data page. In the figure, it appears as if each row uses the identical amount of space to store its data. In the real world, each row may be a different length, which means that the beginning and ending of each row can be anywhere within the page.

If rows are packed in, one after another, and if each row can vary in size, how does SQL Server know when a row starts and when it ends? That is what row offsets are used for. Every row in a page has a row offset, which indicates how far the first byte of the row is from the start of the page. Row offset data is stored at the bottom of a page in an area reserved for this information.

For example, when the first row is added to a page, the location of the first byte of that row is stored in offset row one.  When row two is added, the location of the first byte of the second row is stored in offset row two, and so on. This way, SQL Server knows where each row begins and ends.

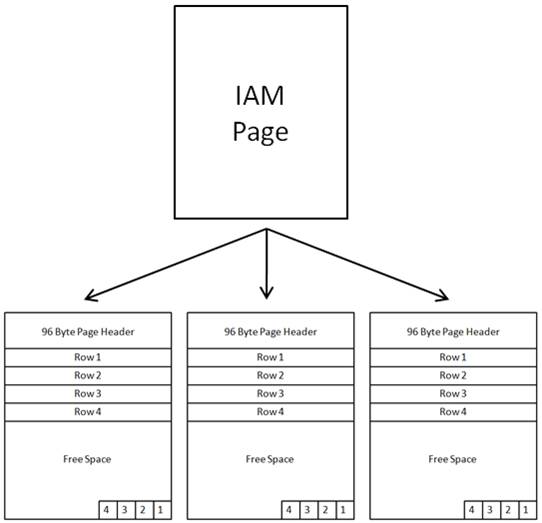
Now that you have a basic understanding of how SQL Server stored data, it’s time to talk about the heaps and B-Trees structures that we talked about briefly a little earlier.

# What is a Heap?

A heap is simply a table without a clustered index. (We will talk about clustered indexes later in this article). When rows are added to a heap, they are not stored in any particular order on a data page, and data pages are not stored in any particular sequence within a database. In other words, rows are stored wherever there is room available. This means that the data pages that contain the rows of the heap may be scattered throughout a database, in no particular order.

**Note:** A heap can have non-clustered indexes, but it does not have a clustered index.

Since a table can’t exist as a bunch of scattered pages, SQL Server provides a way to link them all together so that they act as a single table. This is done using what are called Index Allocation Map (IAM) pages. IAM pages manage the space allocated to heaps (among other tasks), and is what is used to connect the scattered pages (and their rows) into a table.



***Figure 2-5: Data pages in a heap are tracked by IAM pages. Because of this, each row in a heap must first be looked up in an IAM page so that is can be located.***

For example, let’s say that a query is executed against a heap, and a table scan has to be performed to find the data to be returned. In order to find the rows in the table that need to be scanned, the SQL Server Database Engine first goes to an IAM page, which includes pointers to the various extents and pages that contain the rows of the table that belong to the heap.  If a heap is large, numerous IAM pages and data pages have to be examined before all the rows are located and scanned.  If the data pages happen (by coincidence) to be physically contiguous, then such scans can be I/O efficient because sequential reads can be used to read the data. But in many cases, the various heap pages are scattered about, which requires less efficient random reads to scan through all the rows of the table, which can hurt performance.

While tables, taking the form of heaps, can easily be created in SQL Server, using them has many disadvantages. Some of them include:

* If non-clustered indexes are not added to a heap, then all queries against a heap will require table scans to retrieve the requested data. If the heap is large, then these queries will be very resource intensive and hurt SQL Server’s overall performance.
* Since the data in a heap is unordered, performing a table scan on a heap can cause a lot of extra I/O activity because inefficient random reads, not efficient sequential reads, are more the norm.
* While a non-clustered index can be added to a heap to speed up some queries, when the non-clustered index is non-covering, the use of a RID bookmark lookup is required. A RID lookup means that once the record(s) to be returned by the query are identified in the non-clustered index, additional reads (the RID bookmark lookup) must be made of the related rows in the heap, so that all of the data requested by the query is returned. This is not very I/O efficient, especially if many rows are returned. At some point, it may be faster for SQL Server to do a table scan than it is to use a non-clustered index when returning many rows. On the other hand, if the non-clustered index is covering, then the non-clustered can be used to immediately return the data to the user without having to lookup anything in the heap.
* If you want to create an XML index on an XML data column, a clustered index must exist on the table.
* If you want to create a spatial index on a spatial data column (GEOMETRY or GEOGRAPHY), a clustered index must exist on that table.
* If a heap has a non-clustered index on it (as the primary key), and data is inserted into the table, two writes have to occur. One write for inserting the row, and one write for updating the non-clustered index. On the other hand, if a table has a clustered index as the primary key, inserts take only one write, not two writes. This is because a clustered index, and its data, are one in the same. Because of this, it is faster to insert rows into a table with a clustered index as the primary key than it is to insert the same data into a heap that has a non-clustered index as its primary key. This is true whether or not the primary key is monotonically increasing or not.
* When data is updated in a heap, and the updated row is larger than the old row and can’t fit into the old space, a forwarding record is inserted into the original location that points to the new location of the page. If this happens a lot, then there is a lot of space wasted in a database maintaining the forwarding records. This also contributes to additional I/O activity as both the pointer, and the row, have to be read.
* Even if data updated in a heap is not larger than the old row (the updated data is smaller or the same size than the original data), updating a heap with a non-clustered primary key is slower than updating the same table that has a clustered index as the primary key. This is because updating a table with a clustered index is less write intensive than updating a heap with a non-clustered index as its primary key.
* If a row is deleted from a heap with a non-clustered index as its primary key, it is slower than deleting the same row from the same table with a clustered index as its primary key. This is because it takes more I/O to perform this task on a heap than on a clustered index.
* When data is deleted from a heap, the data on the page is not compressed (reclaimed). And should all of the rows of a heap page are deleted, often the entire page cannot be reclaimed. This not only wastes space, it contributes to fragmentation of the data pages within a database.
* If you take two identical tables, one that is a heap with a non-clustered index as its primary key, and a table that has a clustered index as its primary key, the heap with the non-clustered index will be substantially larger, wasting valuable space and increasing disk I/O.
* The ALTER INDEX rebuild and reorganize options cannot be used to defragment and reclaim space in a heap (but they can used to defragment non-clustered indexes on a heap). If you want to defragment a heap in SQL Server 2005, you have three options: 1) create a clustered index on the heap, then drop the clustered index; 2) Use SELECT INTO to copy the old table to a new table; or 3) use BCP or SSIS to move the data from the old table to a new table. In SQL Server 2008, the ALTER TABLE command has been changed so that it now has the ability to rebuild heap.

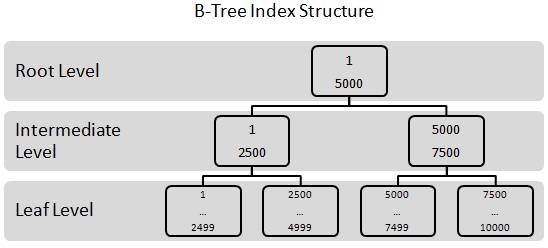
Given all of these negatives, it is highly recommended that all tables have a clustered index. In fact, for the purpose of this article, we will make the assumption that all the tables we are working with have clustered indexes (and are not heaps).

# What is an Index?

In the previous section we talked about heaps, which are tables without a clustered index. In this section, we will talk about what an index is. Later, we will talk about specific kinds of indexes that can be added to SQL Server tables.

As I have already mentioned, an index is simply a way to help queries return data faster from tables. In SQL Server, all indexes physically take the form of what is called a B-Tree.

**Note:** The “B” in B-Tree refers to “balanced,” so B-Tree is short for Balanced Tree. This is because a B-Tree index self balances, which means that every time a tree branches (splits into two pages because the original page is full), about half of the data is left on the old page, and the other half is put on the new page. One of the key benefits of having a Balanced Tree is that finding any particular row in a table requires about the same amount of SQL Server resources because the index has the same depth (number of levels) throughout its structure.



***Figure 2-6: All indexes in SQL Server use the B-Tree index structure.***

Let’s say that we want to perform a query on a table to find rows that have a Customer Number of “4”. One option would be to scan through all the rows in the table until we find one or more rows that have a customer number of “4”. (Note: For this example, we are making the assumption that the rows in this table may not be unique.) If the table is small, then a scan of all the rows in the table will be fairly quick. But if the table is very large, say over 1 million rows, then scanning every row would be time consuming. Instead, another option is to use an index to find the rows that match the query criteria much more quickly than scanning each and every row found in the table.

In figure 2-6 is a simplified example of a B-Tree index (made up of 8K index and data pages) that can be used to quickly look up all of the rows that have a customer number of “4” without having the need to scan every row in the table. For this example, we are assuming that we have a B-Tree index on the customer number column of the table, and that the data is sorted in ascending numerical order.

Notice that B-Trees have several levels. They include:

* **Root Level**: A B-Tree starts with a single index page called the root level. This is where a query begins to look for specific data within the B-Tree. In our example, our root level page only contains two values. Most root level pages have many values, each referring to a page in the intermediate level.
* **Intermediate Level**: Most B-Trees have one or more intermediate levels. The number of intermediate levels depends on the amount of data stored in the table being indexed. Our example has only one intermediate level, which includes two index pages.
* **Leaf Level**: A B-Tree has a single leaf level which may include many, many data pages, depending on the amount of the data stored as part of the leaf level. The leaf level is where the data you are looking for is stored. For example, in the leaf level page that starts with “1”, all the rows with a customer number that ranges from “1” through “2499” are located. On the leaf level page that begins with “2500”, all the rows that range from “2500” to “4999” are located, and so on.

**Note:** As we will learn later in this article, a clustered index stores all of the data of a row at the leaf level, while a non-clustered index stores only the columns included in the index, along with a pointer to the related a row in the clustered index. For the purposes of this example, we don’t care if the index is clustered or non-clustered.

Here’s how a B-Tree index works.

In a B-Tree index search, we always begin at the root level. For example, to find the rows that have a customer number of “4”, the first step is to scan the rows at the root level. In our example, there are two rows than can be scanned: a “1” and a “5000”. Each of these rows points to different pages in the intermediate level. The rows we are seeking have a customer number of “4”, but as you can see, there is no “4” at the root level. But since this index is in numerical order, we know that “4” follows “1”, but is less than “5000”. So, to find the rows that have a value of “4”, we must hop down (transverse) to the intermediate level to further narrow our search. As you can see in figure 2-6, the number “1” in the root level points to an index page in the intermediate level that also starts with the number “1”.

When we go to the intermediate level and look at the index page that begins with the number “1”, notice that it only has two records: a “1” and a “2500”. Notice that the number “4” is still nowhere to be found. Again, we know that the number “4” is greater than 1, but less than “2500”, so we need to transverse to the leaf level and see if we can find it there.

So the next step is to transverse the index to the leaf level page that begins with “1” and ends with “2499”, as we know that number “4” falls between these numbers. Once we are at the correct leaf level page, the rows in this page are examined until we find one or more records that have a customer number of “4”.

While this seems like a long way around to find the rows that have a customer number of “4”, it is actually very short. Instead of having to potentially scanning through thousands and thousands of data pages looking for the rows to be returned, the query only had to scan through several different index pages until the rows that matched the WHERE clause of the query are found and returned. This is usually a much faster way to find the rows you want to return. Of course, this example was kept simple. In the real world, there may be more intermediate levels that need to be transversed, but using a B-Tree index to retrieve data is usually much faster than a scan of every row in a table.

Now that you have a basic understanding of how B-Tree indexes work, we are ready to talk about the specific kinds of indexes available to SQL Server. Keep in mind that although each of the following indexes work slightly differently, that are all still B-Tree indexes, and work on the principles described above.

# Types of Indexes

SQL Server includes many different types of B-Tree indexes, including:

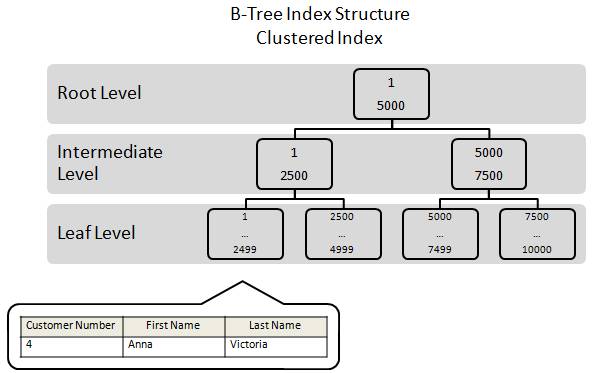
* Clustered
* Non-clustered
* Indexed view
* XML
* Filtered (New with SQL Server 2008)
* Spatial (New with SQL Server 2008)
* Compressed Indexes (New to SQL Server 2008)

**Note:** SQL Server also offers full-text indexes, but they are beyond the scope of this article

Let’s take a brief look at each of these index types.

## Clustered

We have already talked about clustered indexes several times in this article. What I haven’t done until now is to describe them in much detail. Technically speaking, a clustered index is a B-Tree data structure where all the rows in a table are stored at the leaf level of the index. In other words, a clustered index not only includes the root level and intermediate level index pages found in B-Tree indexes, it also includes all the data pages that contain every row in the entire table. In addition, the rows in a clustered table are logically ordered by the key selected for the clustered index (unlike a heap whose rows and pages are unordered). Because of this, a table can only have one clustered index.



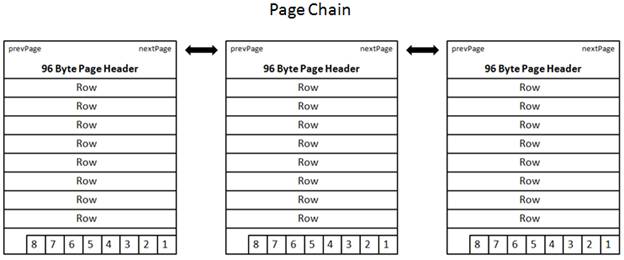
***Figure 2-7: In a clustered index, the index and the data are one in the same. Rows are stored as the leaf pages of a clustered index.***

For example, let’s say that we have a table called “Customers” that includes three columns: Customer Number, First Name, and Last Name, and that the key for the clustered index is based on the Customer Number column.

Because this table has a clustered index, the entire row would be stored on a data page as part of the leaf level of the B-Tree index. This is shown in figure 2-7, where we see that a particular row from that table is physically stored on the leaf level page that begins with the letter “1.” This page also holds additional rows, but they are not shown here in this illustration in order to keep the example as simple as possible.

Because a clustered index is both an index and the rows of the table, whenever you execute a query against the data using the key column as your selection criteria, you can quickly return the data because the data is part of the index. All the query has to do is transverse the index until it finds the row data stored at the leaf level, and return it. As we will learn in the next section, another type of index in SQL Server is called a non-clustered index. In a non-clustered index, the index and the data rows are located in separate objects, which can often contribute to additional resource use because now two objects, not just one, have to be touched by the query in order to return the appropriate results.

As you might imagine, if a table has many rows, it will take many data pages to hold all of these rows at the leaf level. To better facilitate this, data pages at the leaf level of a clustered index are kept in what is called a doubly linked list, also called a page chain. See figure 2-8. The pages in the data chain, and the rows within each page, are logically ordered based on the key.  What does this mean to us?



***Figure 2-8: Pages in a clustered index are connected to one another using a doubly-linked list (or page chain). A page’s page header includes nextPage and prevPage metadata, which is used to logically link one page to another. While this illustration shows the pages as physically contiguous, they don’t have to be.***

Earlier, I said that a clustered index orders the data stored in it logically. Often, DBAs confuse logical ordering with physical ordering. For example, what if a new row is added between two other rows, and there is no room on the existing page for the new row? When SQL Server faces this problem, here’s what it does. Once SQL Server has determined more space is needed on a page, the page will be split. This means that about half of the rows stay on the existing page and about half of the rows are added to a new page. The physical order of the rows stays intact on the two pages, with the new row being place in the proper order on either the original or the new page.

Think for a moment, if SQL Server required that all rows in a clustered index were to be physically contiguous, then as part of the page split, the new page would have to be physically inserted directly after the original page, and all of the following data pages would have to be shifted down one page. If there were 100,000 pages below the page that was split, that would mean that 100,000 pages would have to be physically moved, which would be hugely resource intensive and inefficient. Instead, what SQL Server does when a page split occurs, is to place the new page where there is room for it, which in many cases won’t be very close to the original page. In other words, the physical order of the pages aren’t maintained. But what is maintained is the logical ordering of the pages. For example, when a page is split between the original and the new page, they are logically connected using the page chain, which is used to enforce the logical ordering of the data.

While the process described above keeps data in logical order, we can see that the logical order and the physical order of the pages can get out of synch over time as data in the table changes. Is this a problem? Yes, it potentially can be a big problem, and we will discuss this in great detail in article 3: “What is Index Fragmentation.”

At this point, you should now realize that a table can take only two forms: a table without a clustered index (a heap) or a table with a clustered index. As I explained earlier, for the purposes of this article, we are going to assume that all tables have a clustered index.

## Non-Clustered

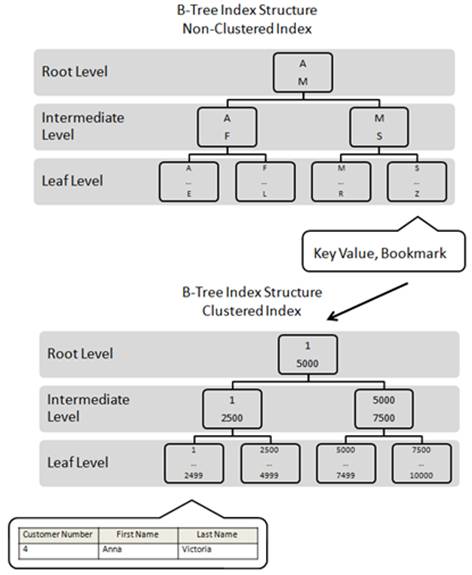
Like a clustered index, a non-clustered index is a B-Tree data structure (although it is a separate data structure from the clustered index). The main difference between the two is that while a clustered index includes all of the data of a table at the leaf level, a non-clustered index does not. Instead, at the leaf level, a non-clustered index includes the key value and a bookmark (pointer) that tells SQL Server where to find the actual row in the clustered index.

**Note:** Besides the key value and a bookmark, the leaf level of a non-clustered index can also store non-key columns, also referred to as included columns. Included columns are often added to non-clustered indexes to create covering indexes.

Because a non-clustered index doesn’t normally store the actual data at its leaf level, it is possible to have many non-clustered indexes on a table.  For example, a table can have one clustered index and up to 249 non-clustered index.

Let’s take a brief look at how a non-clustered index works. In this example, we are going expand upon our previous example of a Customer table that has three columns: Customer Number, First Name, and Last Name. As in our previous example, we are going to have a clustered index that uses the Customer Number as its key. In addition, we are going to add a non-clustered index on the Last Name column. Given this, let’s say we want to return all the rows (and all the row’s columns) in the Customer Table that have a last name of “Victoria”, and that we want to use the non-clustered index on the Last Name column to quickly find and return the data.

Here’s what happens at the index level. See figure 2-9.



***Figure 2-9: A non-clustered index must often work with a clustered index in order for all the selected data to be returned by a query.***

The first step in retrieving all of the rows with a Last Name of “Victoria” is to use the non-clustered index on the Last Name column to identify any rows that match. To find the rows, SQL Server begins at the root level of the non-clustered index, looking for any matching rows. The root page is scanned and two rows are found: an “A” and an “M”. Each of these rows points to different pages in the intermediate level. The first letter of “Victoria” is “V”, but there is no row in the root level page that begins with the letter “V”. But since this index is in alphabetical order, we know that “V” follows “M”. So, to find the letter “V”, we must transverse to the intermediate level to further narrow our search. As you can see in figure 2-9, the letter “M” points to an index page in the intermediate level that starts with the letter “M”.

When we go to the intermediate level and look at the index page that begins with the letter “M”, notice that it only has two records: an “M” and an “S”. Notice that the letter “V” is still nowhere to be found. Since we can’t find the letter “V” on this index page, we need to transverse to the leaf level and see if we can find it there.

Once we get to the leaf level page that begins with “S”, we now identify all the rows on that page that begin with “V”. Each row that begins with “V” is then compared to the last name “Victoria”. In our case, there is a single match. The last name “Victoria” is found in the Last Name non-clustered index key. Now we know that a row does exist that meets our selection criteria, the problem is that all of the data for that row (Customer Number, First Name, and Last Name) aren’t stored at the leaf level. Instead, the only data stored at the leaf level is the key “Victoria” and a bookmark that points to the corresponding record in the table—which as we know—is a clustered index. In this example, the bookmark correspondents with the Customer Number “4”.

Now that SQL Server knows that a row does exist, and that it exists in the clustered index as Customer Number 4, the actual contents of the row (the three columns) must be retrieved. So next, SQL Server must take the bookmark value, the number “4”, and then start at the root level of the clustered index, and then transverse the intermediate and leaf level until it locates the matching row. At this point, the data is returned back the original query. See figure 2-9.

As you can see, using a non-clustered index with a clustered index can be somewhat complex than using a clustered index alone. But in the end, it is often much faster to retrieve rows from a table this way than by scanning each one of them, one row at a time.

## Indexed Views

As you probably know, a standard view is essentially a stored query. When you call a view in a Transact-SQL statement, the underlying query is executed and results are returned. Often, views are referred to as virtual tables because they do not exist until they are called. While views can be handy, the underlying query that produces the view must be executed every time it is called, which can often require a lot of server resources to be expended, especially if the view performs many joins or aggregations.

One potential away of avoiding the overhead of repeatedly re-executing the same standard view over and over again is to create an indexed view. An index view is essentially a standard view that has been materialized as a permanent object in the form of a clustered index, and can be accessed just as any other table can be accessed. So once an indexed view is created, it physically exists, and when the indexed view is called, it does not have to be re-executed because it already exists. This can offer a great performance boost if the same indexed views are used over and over. If you like, non-clustered indexes can also be added to indexed views.

The downside of using an indexed view is that it stores physical data, so not only must the original table’s (or tables if the indexed view joins two or more tables) clustered index must be maintained as data is modified in them, the clustered index that makes up the indexed view must also be maintained at the same time. If there is a lot of data modification going on in the underlying tables, then maintaining an indexed view might use more resources that it saves. This is a trade-off the DBA must evaluate when creating and using indexed views.

Although we won’t spend much time on Indexed views in this article, they are included in this discussion because they can become fragmented just like any index in SQL Server, and require regular maintenance in order for them to perform optimally.

## Computed Columns

SQL Server offers what are called computed columns. This is a virtual column in a row that is computed on the fly when referenced by a query.  In some occasions, it is useful to perform a search on a computed column. While this works, the problem is that a scan has to be executed on the table because the computed values are not materialized until they are touched by the query. This can greatly slow down the performance of a query that is searching on a computed column.

If you find that you often need to perform a query on computed columns, one way to speed up performance is to add an index on the computed column. When you add an index to a computed column, all the values for the computed column become materialized as physical data. Because of this, the computed column can now be queried like any column with an index, greatly boosting performance.

An index on a computed column is the equivalent of a non-clustered index, and as such, is subject to fragmentation, and requires regular index maintenance so that it performs optimally over time.

## XML

XML data in SQL Server is generally stored in XML data type columns in the form of Large Binary Objects (LOBs) that can reach up to 2 GB in size. If you run a query against an XML data column (without an index), the data has to be shredded before the query can parse the data to see if there is a match. Shredding, which means converting the XML into a relational format, can be a time-consuming process that greatly hinders query performance.

One way to avoid on-the-fly shredding, and to speed up queries against XML data, is to index your XML data columns. SQL Server offers two different types of XML indexes.

* **Primary XML Index**: When an XML index is created, the XML is shredded and materialized in a physical form. In many ways, the resulting index is similar in concept to an indexed view. In other words, the XML stored as a BLOB is converted to a relational form that can easily be indexed and queried. If the XML data is large, adding a primary XML index can take up a lot of space because all of the XML data is physically stored in its shredded form.
* **Secondary XML Index**: There are three different types of secondary XML indexes: PATH, VALUE, and PROPERTY. These are created on the primary XML index, and in many ways, are similar to adding a non-clustered index to an indexed view.

Like indexed views, adding primary and secondary XML indexes to a table can create a lot of overhead if there is a lot of XML data modification occurring. This is because any changes to the original XML data column must be reflected in the primary and secondary XML indexes. Like other SQL Server indexes, XML indexes can become fragmented over time.

## Filtered (New with SQL Server 2008)

A filtered index is a variation of a non-clustered index. While a non-clustered index indexes all of the rows in a table, a filtered index only indexes those rows that you want indexed. In other words, you choose what rows you want indexed (using a WHERE clause in your CREATE NONCLUSTERED INDEX command), and only those rows that match your WHERE clause will be included as part of the filtered index.

The assumption you must make when creating filtered indexes is that there are queries that are run regularly against your data that can make good use of them. For example, let’s say you have a table, and that one of the columns of this table has mostly NULL as its value. For the sake of this example, let’s assume the table has 1,000,000 rows and that 900,000 of them have NULL as a value for the column. If you were to add a non-clustered index to this column, the index could potentially be very large, even though NULL is the most common value. To continue with our example, let’s also assume that you commonly run a query against this table that uses this same column as part of the WHERE clause. While a non-clustered index will certainly speed up the query, you still have a large index to deal with.

Now, let’s consider the possibility of adding a filtered index on this column instead of a non-clustered index. If you do this, then the index will only be built for the 100,000 rows that have a non-NULL value in them. Because of this, the index is much smaller, and the overhead to use and maintain it are much less.

If you run a lot of queries against that are looking for non-NULL values, then this filtered index can be very useful. On the other hand, if you create such a filtered index, and are looking for NULL values, then the filtered index would not be used. As you can imagine, you will have to carefully consider the queries you are running in order to identify those columns that could be best served by adding a filtered index.

Filtered indexes are best used on columns that can contain easily be divided into well-defined subsets. For example:

* Columns that contain few values, but many NULL, such as in the example above.
* Columns that have heterogeneous data, such as a column that stores the color of a product. For example, green, red, blue, and so on.
* Columns that contain ranges of values, such as time, date, money, and so on.
* Columns that have data that can be easily defined using comparison logic within a WHERE clause

Now, you may be asking, what are the benefits of using filtered indexes? One major benefit is that filtered indexes also use filtered statistics. Filtered statistics are more accurate than non-filtered statistics, and as a result, the quality of query execution plans often occurs, helping to boost query performance. Second, because fewer rows are indexed, when data modifications occur, there is less overhead when maintaining the filtered index, as compared to a non-clustered index. Third, smaller indexes use up less space, saving storage costs. And forth, smaller indexes take less time to rebuild.

As with any non-clustered index, filtered indexes can become fragmented over time, and need periodic maintenance.

## Spatial (New with SQL Server 2008)

Spatial indexes can be created on either geometry or geography spatial columns. Before a spatial index can be created as a B-Tree structure, spatial data has to be decomposed into a grid hierarchy. This is because a spatial index refers to finite space. For example, an index on a geometry data column could map an area on a plane, and an index on a geography column could map geographic data to a two-dimensional space.

In some ways, this “decomposing” is similar to how XML data needs to be shredded before it can be indexed (although this is a much different process). How this is done is beyond the scope of this article, but what is important to remember is that spatial indexes, like all SQL Server B-Tree indexes, need regular maintenance.

## Compressed (New with SQL Server 2008)

In SQL Server 2008, there is no special “compressed” index type. What I am referring to in this section is the ability in SQL Server 2008 (Enterprise Edition) to perform compression on clustered indexes, non-clustered indexes, and indexed views. Compression comes in two forms:

* **Row-level Data Compression**: Row-level data compression is essentially turning fixed length data types into variable length data types, freeing up empty space. It also has the ability to ignore zero and null values, saving additional space. In turn, more rows can fit into a single data page.
* **Page-level Data Compression**: Page-level data compression starts with row-level data compression, then adds two additional compression features: prefix and dictionary compression, providing for greater compression than row-level compression alone.

While two types of data compression are available, this is only available for the leaf level pages of an index. The root and intermediate levels of an index can only be compressed using row-level compression.

The reason I mention compressed indexes here is several fold. First, turning on compression can in some cases reduce fragmentation (a DELETE on a clustered index), and it other cases, it can make it worse (if an UPDATE causes a page split on clustered index) than without using compression in the first place. In addition, turning on compression can significantly add to the overhead required to maintain indexes. Because of this, if you use compressed indexes, you need to keep these implications in mind.

# Summary

In this article, we learned the basics of SQL Server indexing. Not only did we answer the questions: “What is a table?” and “What is an Index?” we described all the different types of indexes available in SQL Server. One of the most important things you should get out of this article is that all SQL Server indexes are B-Trees, no matter what kind of data they hold. In addition, each of these types of indexes are subject to fragmentation, which would be a subject for a article in its' own right

Effective Clustered Indexes

06 January 2011

*by*[*Michelle Ufford*](https://www.simple-talk.com/author/michelle-ufford/)

As a guideline, clustered Indexes should be Narrow, Unique, Static and Ever Increasing (NUSE).  Michelle Ufford Explains why.

Clustered indexes are the cornerstone of good database design. A poorly-chosen clustered index doesn't just lead to high execution times; it has a 'waterfall effect' on the entire system, causing wasted disk space, poor IO, heavy fragmentation, and more.

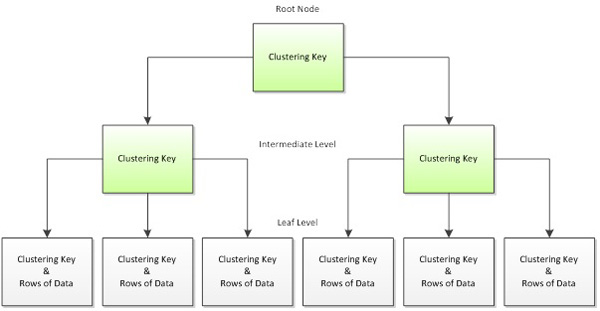
This article will present all the attributes that I believe make up an efficient clustered index key, which are:

* **Narrow** – as narrow as possible, in terms of the number of bytes it stores
* **Unique** – to avoid the need for SQL Server to add a "uniqueifier" to duplicate key values
* **Static** – ideally, never updated
* **Ever-increasing** – to avoid fragmentation and improve write performance

By explaining how SQL Server stores clustered indexes and how they work, I will demonstrate why these attributes are so essential in the design of a good, high-performance clustered index.

**How clustered indexes work**

In order to understand the design principles that underpin a good clustered index, we need to discuss how SQL Server stores clustered indexes. All table data is stored in 8 KB data pages. When a table contains a clustered index, the clustered index tells SQL Server how to order the table's data pages. It does this by organizing those data pages into a B-tree structure, as illustrated in Figure 1.



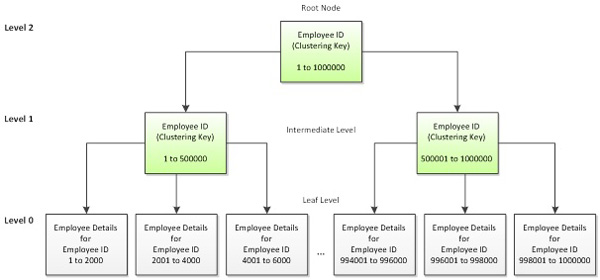
**Figure 1: The b-tree structure of a clustered index**

It can be helpful, when trying to remember which levels hold which information, to compare the B-tree to an actual tree. You can visualize the root node as the trunk of a tree, the intermediate levels as the branches of a tree, and the leaf level as the actual leaves on a tree.

The leaf level of the B-tree is always level 0, and the root level is always the highest level. Figure 1 shows only one intermediate level but the number of intermediate levels actually depends on the size of the table. A large index will often have more than one intermediate level, and a small index might not have an intermediate level at all.

Index pages in the root and intermediate levels contain the clustering key and a page pointer down into the next level of the B-tree. This pattern will repeat until the leaf node is reached. You'll often hear the terms "leaf node" and "data page" used interchangeably, as the leaf node of a clustered index contains the data pages belonging to the table. In other words, the leaf level of a clustered index is where the actual data is stored, in an ordered fashion based on the clustering key.

Let's look at the B-tree again. Figure 2 represents the clustered index structure for a fictional table with 1 million records and a clustering key on EmployeeID.



**Figure 2: A b-tree index for a 1-million row table**

The pages in Level 1 and Level 2, highlighted in green, are index pages. In Level 1, each page contains information for 500,000 records. As discussed, each of these pages stores not half a million rows, but rather half a million clustered index values, plus a pointer down into the associated page on the next level. For example, to retrieve the details for Employee 500, SQL Server would read three pages: the root page in Level 2, the intermediate page in Level 1, and the appropriate leaf level page in Level 0. The root page tells SQL Server which intermediate level page to read, and the intermediate page tells it which specific leaf level page to read.

**Index seeks and Index scans**  
When specific data is returned from data page, in this fashion, it is referred to as an index seek. The alternative is an index scan, whereby SQL Server scans all of the leaf level pages in order to locate the required data. As you can imagine, index seeks are almost always much more efficient than index scans. For more information on this topic, please refer to the *Further Reading* section at the end of this article.

In this manner, SQL Server uses a clustered index structure to retrieve the data requested by a query. For example, consider the following query against the Sales.SalesOrderHeader table in AdventureWorks, to return details of a specific order.

SELECT  CustomerID ,

        OrderDate ,

        SalesOrderNumber

FROM    Sales.SalesOrderHeader

WHERE   SalesOrderID = 44242 ;

This table has a clustered index on the SalesOrderID column and SQL Server is able to use it to navigate down through the clustered index B-tree to get the information that is requested. If we were to visualize this operation, it would look something like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Root Node | **SalesOrderID** | **PageID** |  |  |  |
|  | NULL | 750 |  |  |  |
|  | 59392 | 751 |  |  |  |
| Intermediate level  (Page 750) | **SalesOrderID** | **PageID** |  |  |  |
|  | 44150 | 814 |  |  |  |
|  | 44197 | 815 |  |  |  |
|  | 44244 | 816 |  |  |  |
|  | 44290 | 817 |  |  |  |
|  | 44333 | 818 |  |  |  |
| Leaf level  (Page 815) | **SalesOrderID** | **OrderDate** | **SalesOrderNumber** | **AccountNumber** | **CustomerID** |
|  | 44240 | 9/23/2005 | SO44240 | 10-4030-013580 | 13580 |
|  | 44241 | 9/23/2005 | SO44241 | 10-4030-028155 | 28155 |
|  | 44242 | 9/23/2005 | SO44242 | 10-4030-028163 | 28163 |

In the root node, the first entry points to PageID 750, for any values with a SalesOrderID between NULL and 59391. The data we're looking for, with a SalesOrderID of 44242, falls within that range, so we navigate down to page 750, in the intermediate level. Page 750 contains more granular data than the root node and indicates that the PageID 815 contains SalesOrderID values between 44197 and 44243. We navigate down to that page in the leaf level and, finally, upon loading PageID 815, we find all of our data for SalesOrderID 44242.

**Characteristics of an effective clustered index**

Based on this understanding of how a clustered index works, let's now examine why and how this dictates the components of an *effective* clustered index key: narrow, unique, static, and ever-increasing.

**Narrow**

The width of an index refers to the number of bytes in the index key. The first important characteristic of the clustered index key is that it is as narrow as is practical. To illustrate why this is important, consider the followingnarrow\_example table:

CREATE TABLE dbo.narrow\_example  
    (  
      web\_id      INT IDENTITY(1,1), -- unique

      web\_key     UNIQUEIDENTIFIER , -- unique

      log\_date    DATETIME , -- not unique

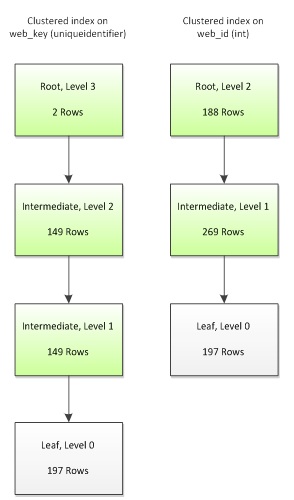
      customer\_id INT -- not unique  
    ) ;

The table has been populated with 10 million rows and table contains two columns that are candidates for use as the clustering key:

* web\_id – a fixed-length **int** data type, consuming 4 bytes of space
* web\_key – a fixed-length **uniqueidentifier** data type, consuming 16 bytes.

**TIP:**   
Use the DATALENGTH function to find how many bytes are being used to store the data in a column.

So, which column will make a better clustered index key? Let's take a look at the B-tree structure of each, shown in Figure 3.

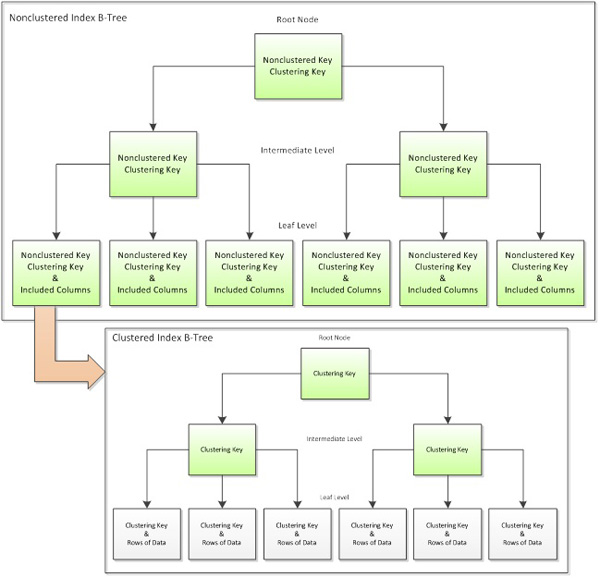


**Figure 3: The b-tree levels for clustered indexes based on int and uniqueidenitifier key**

The most obvious difference is that the uniqueidentifier key has an additional non-leaf level, giving 4 levels to its tree, as opposed to only 3 levels for the int key. The simple reason for this is that the uniqueidentifierconsumes 300% more space than the int data type, and so when we create a clustered key onuniqueidentifier, fewer rows can be packed into each index page, and the clustered key requires an additional non-leaf level to store the keys.

Conversely, using a narrow int column for the key allows SQL Server to store more data per page, meaning that it has to traverse fewer levels to retrieve a data page, which minimizes the IO required to read the data. The potential benefit of this is large, especially for range scan queries, where more than one row is required to fulfill the query criteria. In general, the more data you can fit onto a page, the better your table can perform. This is why appropriate choice of data types is such an essential component of good database design.

However, our choice of clustering key can affect the performance of not only the clustered index, but also any non-clustered indexes that rely on the clustered index. As shown in Figure 4, a non-clustered index contains the clustered index key in every level of its b-tree structure, as a pointer back into the clustered index. This happens regardless of whether or not the clustering key was explicitly included in the nonclustered index structure, either as part of the index key or as an included column. In other words, whereas in the clustered index the leaf level contains the actual data rows, in a nonclustered index, the leaf level contains the clustered key, which SQL Server uses to find the rest of the data.

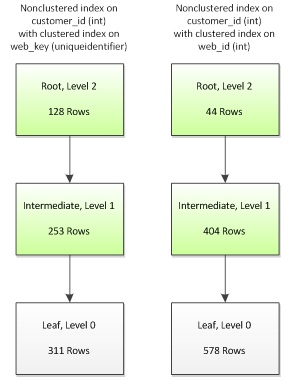


**Figure 4: Non-clustered indexes also store the clustering key in order to look up data in the clustered index**

So, let's see how our choice of clustering key impacts the potential performance of our non-clustered indexes. We'll keep the example pretty simple and create a non-clustered index on customer\_id, which is an int data type.

CREATE NONCLUSTERED INDEX IX\_example\_customerID  
ON dbo.narrow\_example (customer\_id) ;

Figure 5 shows the resulting B-tree structures of our nonclustered index, depending on whether we used theuniqueidentifier or the int column for our clustered index key.



**Figure 5**

While we have the same number of levels in each version of the index, notice that the non-clustered index based on the int clustering key stores 86% more data in each leaf-level data page than its uniqueidentifiercounterpart. Once again, the more rows you can fit on a page, the better the overall system performance: range-scan queries on the narrow int version will consume less IO and execute faster than equivalent queries on the wider, uniqueidentifier version.

In this example, I've kept the table and index structures simple in order to better illustrate the basic points. In a production environment, you'll often encounter tables that are much, much wider. It's possible that such tables will require a composite clustered index, where the clustering key is comprised of more than one column. That's okay; the point isn't to advise you to base all of your clustered keys on integer IDENTITY columns, but to demonstrate that a wide index key can have on a significant, detrimental impact on a database's performance, compared to a narrow index key. Remember, narrowness refers more to the number of bytes consumed than the number of columns. For example, a composite clustered key on three int columns would still be narrower than auniqueidentifier key (4 + 4 + 4 = 12 bytes for the former vs. 16 bytes for the latter).

**Unique**

Index uniqueness is another highly desirable attribute of a clustering key, and goes hand-in-hand with index narrowness. SQL Server does not require a clustered index to be unique, but yet it must have some means of uniquely identifying every row. That's why, for non-unique clustered indexes, SQL Server adds to every duplicate instance of a clustering key value a 4-byte integer value called a uniqueifier. This uniqueifier is added everywhere the clustering key is stored. That means the uniqueifier is stored in every level of the B-tree, in both clustered and non-clustered indexes. As you can imagine, if there are many rows using the same clustering key value, this can become quite expensive.

What's more, the uniqueifier is stored as a variable-length column. This is important because if a table does not already contain any other variable-length columns, each duplicate value is actually consuming 8-bytes of overhead: 4 bytes for the uniqueifier value and 4 bytes to manage variable-length columns on the row. The following example demonstrates this. We create a table with a non-unique clustered index, insert into it a single row, and then retrieve minimum and maximum record sizes (which currently refer to the same, single record) from the sys.dm\_db\_index\_physical\_stats DMV:

CREATE TABLE dbo.overhead ( myID INT NOT NULL ) ;  
   
CREATE CLUSTERED INDEX CIX\_overhead -- not unique!  
ON dbo.overhead(myID) ;  
   
INSERT  INTO dbo.overhead  
        ( myID )  
        SELECT  1 ;

SELECT  min\_record\_size\_in\_bytes ,

        max\_record\_size\_in\_bytes

FROM    sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID(N'dbo.overhead'),

                                       NULL, NULL, N'SAMPLED') ;

min\_record\_size\_in\_bytes   max\_record\_size\_in\_bytes

------------------------ ------------------------

11                         11

(1 row(s) affected)

Although we only have a single column in the table, there is a minimum of 7 bytes of overhead per row, in SQL Server. While this overhead may increase with the addition of NULL or variable-length columns, it will never be less than 7 bytes per row. The other 4 bytes are used to store the int column, myID.

Now let's insert a duplicate value into the table:

INSERT  INTO dbo.overhead

        ( myID )

        SELECT  1 ;

SELECT  min\_record\_size\_in\_bytes ,

        max\_record\_size\_in\_bytes

FROM    sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID(N'dbo.overhead'),

                                       NULL, NULL, N'SAMPLED') ;

min\_record\_size\_in\_bytes   max\_record\_size\_in\_bytes

------------------------ ------------------------

11                         19

(1 row(s) affected)

The duplicate value requires the addition of a uniqueifier, which consumes an extra 4 bytes. However, since a variable-length column, such as a varchar() column, does not already exist on the table, an additional 4 bytes are added by SQL Server to manage the variable-length properties of the uniqueifier. This brings the totaluniqueifier overhead to 8 bytes per row.

**TIP:**  
The sys.dm\_db\_index\_physical\_stats DMV runs in three modes: LIMITED, SAMPLED, or DETAILED. The min\_record\_size\_in\_bytes and max\_record\_size\_in\_bytes columns are only available inSAMPLED or DETAILED mode. Be careful when running this DMV in production or on large tables, as theSAMPLED mode scans 1% of pages and DETAILED modes scans all pages. Refer to Books Online for more information.

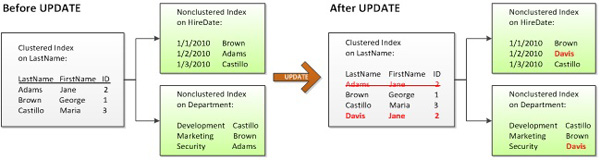
So, returning to our original narrow\_example table, let's see what would happen if the clustering key was changed to customer\_id, which is a non-unique int. Although the uniqueifier is not readily visible and cannot be queried, internally the leaf-level page might look something like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| web\_id | web\_key | log\_date | customer\_id | uniqueifier |
| 1 | 6870447C-A0EC-4B23-AE5F-9A92A00CE166 | 12/15/2010 | 1 | NULL |
| 2 | 5AB480CF-40CD-43FD-8C3D-5C625875E143 | 12/15/2010 | 1 | 1 |
| 3 | 95C312B9-83AF-4725-B53C-77615342D177 | 12/15/2010 | 1 | 2 |
| 4 | 88AA4497-9A20-4AB7-9704-1FDFAE200564 | 12/15/2010 | 2 | NULL |
| 5 | E3EA3014-FC23-48B6-9205-EE6D06D37C5B | 12/15/2010 | 2 | 1 |
| 6 | 9F6A8933-F6EC-416F-AACA-1C3FF172151C | 12/15/2010 | 3 | NULL |
| 7 | B16406A8-649B-4E7A-A234-C7B7D8FCE2D3 | 12/15/2010 | 4 | NULL |
| 8 | 443B627B-21CE-4466-AD15-1879C8749225 | 12/15/2010 | 4 | 1 |
| 9 | 2F3757DE-3799-4246-BA88-944C5DA3683E | 12/15/2010 | 4 | 2 |
| 10 | 25D9F2AA-6610-48CD-9AC4-4F1E29FDED1C | 12/15/2010 | 4 | 3 |

The uniqueifier is NULL for the first instance of each customer\_id, and is then populated, in ascending order, for each subsequent row with the same customer\_id value. The overhead for rows with a NULLuniqueifier value is, unsurprisingly, zero bytes. This is why min\_record\_size\_in\_bytes remained unchanged in the overhead table; the first insert had a uniqueifier value of NULL. This is also why it is impossible to estimate how much additional storage overhead will result from the addition of a uniqueifier, without first having a thorough understanding of the data being stored. For example, a non-unique clustered index on a datetime column may have very little overhead if data is inserted, say, once per minute. However, if that same table is receiving thousands of inserts per minute, then it is likely that many rows will share the same datetimevalue, and so the uniqueifier will have a much higher overhead.

If your requirements seem to dictate the use of a non-unique clustered key, my advice would be to look to see if there are a couple of relatively narrow columns that, together, can form a unique key. You'll still see the increase in the row size for your clustering key in the index pages of both your clustered and nonclustered indexes, but you'll at least save the cost of the uniqueifier in the data pages of the leaf level of your clustered index. Also, instead of storing an arbitrary uniqueifier value to the index key, which is meaningless in the context of your data, you would be adding meaningful and potentially useful information to all of your nonclustered indexes.

A good clustered index is also built upon static, or unchanging, columns. That is, you want to choose a clustering key that will never be updated. SQL Server must ensure that data exists in a logical order based upon the clustering key. Therefore, when the clustering key value is updated, the data may need to be moved elsewhere in the clustered index so that the clustering order is maintained. Consider a table with a clustered index on LastName, and two non-clustered indexes, where the last name of an employee must be updated.



**Figure 6: The effect of updating a clustered key column**

Not only is the clustered index updated and the actual data row moved – most likely to a new data page – but each non-clustered index is also updated. In this particular example, at least three pages will be updated. I say "at least" because there are many more variables involved, such as whether or not the data needs to be moved to a new page. Also, as discussed earlier, the upper levels of the B-tree contain the clustering key as pointers down into the leaf level. If one of those index pages happens to contain the clustering key value that is being updated, that page will also need to be updated. For now, though, let's assume only three pages are affected by the UPDATEstatement, and compare this to behavior we see for the same UPDATE, but with a clustering key on ID instead ofLastName.



**Figure 7: An UPDATE that does not affect the clustered index key**

In Figure 7, only the data page in the clustered index is changed because the clustering key is not affected by theUPDATE statement. Since only one page is updated instead of three, clustering on ID requires less IO than clustering on LastName. Also, updating fewer pages means the UPDATE can complete in less time.

Of course, this is another simplification of the process. There are other considerations that can affect how many pages are updated, such as whether an update to a variable-length column causes the row to exceed the amount of available space. In such a case, the data would still need to be moved, although only the data page of the clustered index is affected; nonclustered indexes would remain untouched.

Nevertheless, updating the clustering key is clearly more expensive than updating a non-key column. Furthermore, the cost of updating a clustering key increases as the number of non-clustered indexes increases. Therefore, it is a best practice is to avoid clustering on columns that are frequently updated, especially in systems where UPDATEperformance is critical.

**Ever-Increasing**

The last important attribute of a clustered index key is that it is ever-increasing. In addition to narrow, unique, and static, an integer identity column is an excellent example of an ever-increasing column. The identity property continuously increments by the value defined at creation, which is typically one. This allows SQL Server, as new rows are inserted, to keep writing to the same page until the page is full, then repeating with a newly allocated page.

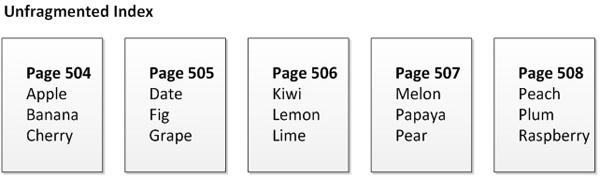
There are two primary benefits to an ever-increasing column:

1. **Speed of insert** – SQL Server can much more efficiently write data if it knows the row will always be added to the most recently allocated, or last, page
2. **Reduction in clustered index fragmentation** – this fragmentation results from data modifications and can take the form of gaps in data pages, so wasting space, and a logical ordering of the data that no longer matches the physical ordering.

However, before we can discuss the effect of the choice of clustering key on insert performance and index fragmentation, we need to briefly review the types of fragmentation that can occur.

**Internal and external index fragmentation**

There are two types of index fragmentation, which can occur in both clustered and non-clustered indexes: extent (a.k.a. external) and page (*a.k.a.* internal) fragmentation. First, however, Figure 8 illustrates an un-fragmented index.

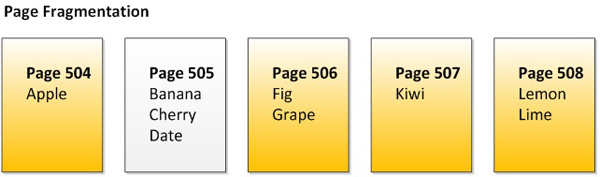


**Figure 8: Data pages in an un-fragmented clustered index**

In this simplified example, a page is full if it contains 3 rows, and in Figure 8 you can see that every page is full and the physical ordering of the pages is sequential. In extent fragmentation, also known as *external fragmentation*, the pages get out of physical order, as a result of data modifications. The pages highlighted in orange in Figure 9 are the pages that are externally fragmented. This type of fragmentation can result in random IO, which does not perform as well as sequential IO.



**Figure 9: External fragmentation in a clustered index**



**Figure 10: Internal fragmentation in a clustered index**

Figure 10 illustrates page fragmentation, also known as *internal fragmentation*, and refers to the fact that the there are gaps in the data pages, which reduces the amount of data that can be stored on each page, and so increase the overall amount of space needed to store the data. Again, the pages in orange indicate an internally fragmented page.

For example, comparing Figures 8 and 10, we can see that the un-fragmented index holds 15 data rows in 5 pages. By contrast, the index with internal fragmentation only holds 9 data rows in the same number of pages. This is not necessarily a big issue for singleton queries, where just a single record is needed to fulfill the request. However, when pages are not full and additional pages are required to store the data, range-scan queries will feel the effects, as more IO will be required to retrieve those additional pages.

Most indexes suffering from fragmentation will often have both extent *and*page fragmentation.

**How non-sequential keys can increase fragmentation**

Clustering on ever-increasing columns such as identity integers will result in an un-fragmented index, as illustrated in Figure 8. This results in sequential IO and maximizes the amount of data stored per page, resulting in the most efficient use of system resources. It also results in very fast write performance.

Use of a non-sequential key column can, however, result in a much higher overhead during insertion. First, SQL Server has to find the correct page to write to and pull it into memory. If the page is full, SQL Server will need to perform a **page split** to create more space. During a page split, a new page is allocated, and half the records are moved from the old page to the newly-allocated page. Each page has a pointer to the previous and next page in the index, so those pages will also need to be updated. Figure 11 illustrates the results of a page split.



**Figure 11: Internal and external fragmentation as a result of a page split**

Initially, we have two un-fragmented pages, each holding 3 rows of data. However, a request to insert "coconut" into the table results in a page split, because Page 504, where the data naturally belongs, is full. SQL Server allocates a new page, Page 836, to store the new row. In the process, it also moves half the data from Page 504 to the new page in order to make room for new data in the future. Lastly, it updates the previous and next pointers in both pages 504 and 505. We're left with Page 836 out of physical ordering, and both pages 504 and 836 contain free space. As you can see, not only would writes to this latter scenario be slower, but both internal and external fragmentation of the table would be much higher.

I once saw a table with 4 billion rows clustered on a non-sequential uniqueidentifier, also known as a GUID. The table had a fragmentation level of 99.999%. Defragging the table and changing the clustering key to an identityinteger resulted in a space savings of over 200 GB. Extreme, yes, but it illustrates just how much impact an ever-increasing clustering key can have on table.

I am not suggesting that you only create clustered indexes on identity integer columns. Fragmentation, although generally undesirable, primarily impacts range-scan queries; singleton queries would not notice much impact. Even range-scan queries can benefit from routine defragmentation efforts. However, the ever-increasing attribute of a clustered key is something to consider, and is especially important in OLTP systems where INSERT speed is important.

**Summary**

In this article, I've discussed the most desirable attributes of a clustered index: narrow, unique, static, and ever-increasing. I've explained what each attribute is and why each is important. I've also presented the basics of B-tree structure for clustered and non-clustered indexes. The topic of "indexing strategy" is vast topic and we've only scratched the surface. Beyond what I presented in this article, there are also many application-specific considerations when choosing a clustering key, such as how data will be accessed and the ability to use the clustered index in range-scan queries. As such, I'd like to stress that the attributes discussed in this article are not concrete rules but rather time-proven guidelines. The best thing to do if you're not sure if you've chosen the best clustering key is to test and compare the performance of different strategies.