**Why can’t a table have two clustered indexes?**

The short answer? A clustered index *is* the table. When you define a clustered index on a table, the database engine sorts all the rows in the table, in ascending or descending order, based on the columns identified in the index definition (the key columns). The clustered index is not a separate entity like it is with other index types, but rather a mechanism for sorting the table and facilitating quick data access.

Suppose you have a table that contains data about the company’s sales transactions. The **Sales** table includes such information as the order ID, line item ID, product number, quantity, order and shipping dates, and so on. You create a clustered index based on the **OrderID** and **LineID** columns, sorted in ascending order, as shown in the following T-SQL code:

CREATE UNIQUE CLUSTERED INDEX ix\_oriderid\_lineid

ON dbo.Sales(OrderID, LineID);

When you run the statement, all rows in the table are physically sorted, first by the **OrderID** column and then by the**LineID** column, but the data itself remains a single logical unit, which is the table. For this reason, you cannot create two clustered indexes. There can be only one table and that table can be sorted in only one order.

**Given the many benefits of clustered tables, why even bother with heaps?**

You’re right. Clustered tables are great, and most of your queries will probably perform best of your tables are configured with clustered indexes. But in some cases you might want to leave the table in its natural state, that is, as a heap, and create only nonclustered indexes to support your queries.

A heap, as you’ll recall, stores data in an unspecified order. Normally, the database engine adds the data in the order the rows are inserted into the table, although the engine likes to move rows around on occasion to store them more efficiently. As a result, you have no way to predict how the data will be ordered.

If the query engine must find data without the benefit of a nonclustered index, it does a full table scan to locate the target rows. On a very small table, this is usually not a big deal, but as a heap grows in size, performance is likely to quickly degrade. A nonclustered index can help, of course, by using a pointer that directs the query engine to the file, page, and row where the data is stored—normally a far better alternative to a table scan. Even so, it’s still hard to beat the benefits of a clustered index when weighing query performance.

Yet heaps can help improve performance in certain situations. Consider the table that has a lot of insert activity, but few updates and deletes, if any. For example, a table that stores log data is likely restricted mostly to insert operations, until perhaps the data is archived. On a heap, you won’t see the type of page splits and fragmentation you would with a clustered index (depending on the key columns) because rows are simply added to the end of the heap. Too much page splitting can have a significant effect on performance, and not in a good way. In general, heaps make insert operations relatively painless, and you don’t have to contend with the storage or maintenance overhead you find with clustered indexes.

But the lack of updates and deletions should not be the only considerations. The way in which data is retrieved is also an important factor. For example, you should not use a heap if you frequently query ranges of data or the queried data must often be sorted or grouped.

What all this means is that you should consider using a heap *only* when you’re working with ultra-light tables or your DML operations are limited to inserts and your queries are fairly basic (and you’re still using nonclustered indexes). Otherwise, stick with a well-designed clustered index, that is, one defined on a simple ascending key, such as the ubiquitous **IDENTITY** column.

**How do I override the default fill factor when creating an index?**

Overriding the default fill factor is one thing. Understanding how the default fill factor works is another. But first, a step back. The fill factor refers to the amount of space an index uses on a leaf node before flowing over to a new page. If the fill factor is set to **90**, for example, the index will use up to 90% of the page and then flow on to the next.

By default, the fill factor on a SQL Server instance is set to **0**, which is the same as setting the fill factor to **100**. As a result, all new indexes will automatically inherit that setting, unless you specifically override the behavior or you change the default. You can change the default in SQL Server Management Studio by modifying the server properties or by running the **sp\_configure** system store procedure. For example, the following T-SQL sets the default fill factor to **90** (after switching to the advanced options):

EXEC sp\_configure 'show advanced options', 1;

GO

RECONFIGURE;

GO

EXEC sp\_configure 'fill factor', 90;

GO

RECONFIGURE;

GO

Once you’ve reset the fill factor, you must restart the SQL Server service. You can then verify whether the default has been updated by again running **sp\_configure**, only this time without specifying the fill factor value:

EXEC sp\_configure 'fill factor'

GO

The command should return a value of **90**. As a result, all new indexes will now use this fill factor. You can verify the change by creating an index and then retrieving its fill factor:

USE AdventureWorks2012;

GO

CREATE NONCLUSTERED INDEX ix\_people\_lastname

ON Person.Person(LastName);

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

  AND name='ix\_people\_lastname';

In this case, we’re creating a nonclustered index on the **Person** table in the **AdventureWorks2012** database. After we create the index, we can retrieve its fill factor from the **sys.indexes** table. The **SELECT** statement should return**90**.

However, suppose we drop the index and re-create it, only now we provide a specific fill factor:

CREATE NONCLUSTERED INDEX ix\_people\_lastname

ON Person.Person(LastName)

WITH (fillfactor=80);

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

  AND name='ix\_people\_lastname';

This time around, we add a **WITH** clause and **fillfactor** option to our **CREATE** **INDEX** statement and specify the fill factor as **90**. As you would expect, the **SELECT** statement now returns the value **90**.

So far, this should all be fairly straightforward. Where you could get stung in this whole process is if you create an index that uses the default fill factor, assuming that you know how it’s been set. For example, someone could have mucked around with the server and done something dumb like set the fill factor to **20**. In the meantime, you continue to create indexes assuming that the default is still **0**. Unfortunately, you have no way of knowing the fill factor when you create the index unless you specifically retrieve that value afterwards, like we did in our examples. Otherwise, you have to wait until the performance starts steadily degrading and you realize something is wrong.

Another fill factor issue you should be aware of has to do with rebuilding indexes. As with creating an index, you can specify a fill factor when you rebuild it. However, unlike index creation, the rebuild does not use the server default, despite how it might appear. Rather, if you don’t specify a fill factor, SQL Server uses the index’s fill factor as it existed before the rebuild. For example, the following **ALTER** **INDEX** statement rebuilds the index we just created:

ALTER INDEX ix\_people\_lastname

ON Person.Person REBUILD;

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

  AND name='ix\_people\_lastname';

 When we retrieve the fill factor this time, it shows as **80**, because that’s what we assigned to the index the last time we created it. The server default is out of the picture.

As you can see, overriding the default fill factor is no big deal. The bigger trick is knowing what the default is and when it’s applied. If you always specify the fill factor when creating and rebuilding indexes, than you always know exactly what you’re getting. Then only thing you have to worry about is someone once again mucking around with the server and creating a maintenance place that rebuilds all the indexes with a ridiculously low fill factor.

**Can you create a clustered index on a column with duplicate values?**

Yes and no. Yes, you can create a clustered index on key columns that contain duplicate values. No, the key columns cannot remain in a non-unique state. Let me explain. If you create a non-unique clustered index on a column, the database engine adds a four-byte integer (a *uniquifier*) to duplicate values to ensure their uniqueness and, subsequently, to provide a way to identify each row in the clustered table.

For example, you might decide to create a clustered index on the **LastName** column of a table that contains customer data. The column includes the values **Franklin**, **Hancock**, **Washington**, and **Smith**. You then insert the values **Adams**, **Hancock**, **Smith**, and **Smith**. Because the values in the key column must ultimately be unique, the database engine will modify the duplicates so that the values look something like this: **Adams**, **Franklin**, **Hancock**,**Hancock1234**, **Washington**, **Smith**, **Smith4567**, and **Smith5678**.

On the surface, this might seem an okay approach, but the integer increases the size of the key values, which could start becoming an issue if you have a lot of duplicate values and those values are being referenced by foreign keys and nonclustered indexes. For this reason, you should try to create unique clustered indexes whenever possible. If not possible, at least go for columns that have a high percentage of unique values.

**How is a table stored if a clustered index has not ben defined on the table?**

SQL Server essentially supports two types of tables: a clustered table, that is, one on which a clustered index has been defined, and a heap table, or just plain *heap*. Unlike a clustered table, data within a heap is not ordered in any way. It is essentially a pile of data. If you add a row to the table, the database engine simply tacks it at the end of the page. When the page fills, data is added to a new page.

In most cases, you’ll want to create a clustered index on a table to take advantage of the sorting capabilities and query benefits they can deliver. (Consider what it would be like to find a number in a phone book if it were not sorted in any way.) However, if you choose not to create a clustered index, you can still create nonclustered indexes on the heap. In such cases, each row in the index includes a pointer that identifies the row being referenced in the heap. The pointer includes the data file ID, page number, and row number for the targeted data.

**What is the relationship between unique and primary key constraints and a table’s indexes?**

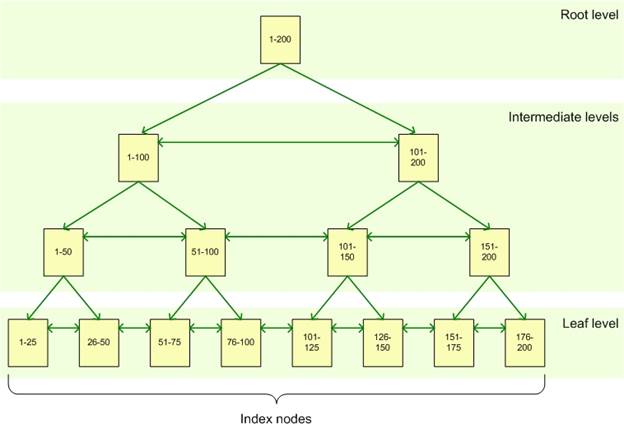
Primary key and unique constraints ensure that the values in the key columns are unique. You can define only one primary key on a table and it cannot contain null values. You can create multiple unique constraints on a table and each one can contain a single null value.

When you create a primary key constraint, the database engine also creates a unique clustered index, if a clustered index doesn’t already exist. However, you can override the default behavior and specify that a nonclustered index be created. If a clustered index does exist when you create the primary key, the database engine creates a unique nonclustered index.

When you create a unique constraint, the database engine creates a unique nonclustered index. However, you can specify that a unique clustered index be created if a clustered index does not already exist. For all practical purposes, a unique constraint and unique index are one in the same.

**Why are SQL Server clustered and nonclustered indexes considered B-tree indexes?**

A basic SQL Server index, whether clustered or nonclustered, is spread across a set of pages, referred to as the*index nodes*. These pages are organized into a hierarchical B-tree structure. At the top sits the root node, at the bottom, the leaf nodes, with intermediate nodes in between, as shown in the following illustration:



The root node represents the main entry point for queries trying to locate data via the index. From that node, the query engine negotiates the hierarchy down to the appropriate leaf node, where the actual data resides.

For example, suppose your query is looking for the row that contains the key value **82**. The query engine starts at the root node, which points to the correct intermediate node, in this case, the **1-100** node. From the **1-100** node, the engine proceeds to the **51-100** node, and from there, goes to the **76-100** leaf node. If this is a clustered index, the leaf node will contain the entire row of data associated with the key value **82**. If this is a nonclustered index, the leaf node will point to the clustered table or heap where the row exists.

**How can an index improve performance if the query engine has to negotiate through all those index nodes?**

First off, indexes do not always improve performance. Too many of the wrong type of indexes can bog down a system and make query performance worse. That said, if indexes have been carefully implemented, they can provide a significant performance boost.

Think of a big fat book about SQL Server performance tuning (the printed version, not the ebook variety). Imagine that you want to find information about configuring the Resource Governor. You can thumb through the book one page at a time, or you can go to the index and find the exact page number where the information is located (assuming the book has been properly indexed). Undoubtedly, this could save you a considerable amount of time, despite the fact that you must refer to an entirely different structure (the index) to get the information you need from the primary structure (the book).

Just like a book’s index, a SQL Server index lets you perform targeted queries instead of scanning all of a table’s data. For small tables, a full scan is usually not a big deal, but large tables spread across many data pages can result in excessively long-running queries if no index exists to point the query engine in the right direction. Imagine being lost on the LA freeways at rush hour without a map, and you get the idea.

**If indexes are so great, why not just create them on every column?**

No good deed goes unpunished. At least that’s how it works with indexes. Sure, they’re great as long as all you run are **SELECT** statements against the database, but throw in a lot of **INSERT**, **UPDATE**, and **DELETE** statements, and the landscape quickly changes.

When you issue a **SELECT** statement, the query engine finds the index, navigates the B-tree structure, and locates the desired data. What could be simpler? But that all changes if you issue a data modification statement, such an**UPDATE**. True, for the first part of the **UPDATE** operation, the query engine can again use the index to locate the row to be modified. That’s the good news. And if it’s a simple update and no key values are involved, chances are the process will be fairly painless. But if the update forces a page split or key values change and get moved to different nodes, the index might need to be reorganized, impacting other indexes and operations and resulting in slower performance all around.

Same with a **DELETE** statement. An index can help locate the data to be deleted, but the deletion itself might result in page reshuffling. And as for the **INSERT** statement, its the sworn enemy of all indexes. You start adding a lot of data and your indexes have to be modified and reorganized and everybody suffers.

So the way in which your database is queried must be uppermost in your thinking when determining what sort of indexes to add and how many. More is not necessarily better. Before you throw another index at a table, consider the costs, not only on query performance, but also on disk space, index maintenance, and the domino effects on other operations. Your index strategy is one of the most important aspects of a successful database implementation and should take into account a number of considerations, from index size to the number of unique values to the type of queries being supported.

**Does a clustered index have to be created on the primary key column?**

You can create a clustered index on any qualified columns. True, a clustered index and primary key constraint isusually a match made in heaven, so well suited in fact that when you define a primary key, a clustered index is automatically created, if one doesn’t already exist. Still, you might decide that the clustered index would be better matched elsewhere, and often your decision would be justified.

The main purpose of a clustered index is to sort all the rows in your table, based on the key columns in your index definition, and provide quick and easy access to the table’s data. The table’s primary key can be a good choice because it uniquely identifies every row in the table, without the need for additional data. In some cases, a surrogate primary key can be an even better choice because, in addition to being unique, the values are small and added sequentially, making the nonclustered indexes that reference those values more efficient as well. The query optimizer also loves such an arrangement because joins can be processed faster, as can queries that in some other way reference the primary key and its associated clustered index. As I said, a match made in heaven.

In the end, however, your clustered index should take into account a number of factors, such as how many nonclustered indexes will be pointing to the clustered index, how often the clustered key values will change and how large those key columns are. When the values in a clustered index change or the index doesn’t perform well, all of the table’s other indexes can be impacted. A clustered index should be based on relatively stable columns that grow in an orderly fashion, as opposed to growing randomly. The index should also support the queries most commonly accessing the table’s data so they can take full advantage of the data being sorted and available in the leaf nodes. If the primary key fits this scenario, then use it. Otherwise, use a different set of columns.

**If you index a view is it still a view?**

A view is a virtual table made up a data from one or more other tables. It is essentially a named query that retrieves the data from the underlying tables when you call that view. You can improve a view’s performance by creating clustered and nonclustered indexes on that view, just like you create indexes on a table, the main caveat being that you must create a unique clustered index before you can create a nonclustered one.

When creating an indexed view (also referred to as a *materialized view*), the view definition itself remains a separate entity. It is, after all, merely a hard-coded **SELECT** statement stored in the database. The indexes are a different story. Whether you create a clustered or nonclustered index, the data is persisted to disk, just like a regular index. In addition, when the data in the underlying tables change, the indexes are automatically updated (which means you might want to avoid indexing views where the underlying data changes frequently). In any case, the view itself still remains a view, but one that just happens to have indexes associated with it.

Before you can create an index on a view, it must meet a number of restrictions. For example, the view can reference only base tables, not other views, and those tables must be within the same database. There are lots more restrictions, of course, so be sure to refer to the SQL Server documentation for all the sordid details.

**Why would you use a covering index instead of a composite index?**

First, let’s make sure we understand the differences between them. A composite index is simply one in which you include more that one key column. Multiple key columns can be useful for uniquely identifying a row, as can be the case when a unique cluster is defined on a primary key, or you’re trying to optimize an often-used query that references multiple columns. In general, however, the more key columns an index contains, the less efficient that index, which means composite indexes should be used judiciously.

That said, there are times when a query would benefit greatly if all the referenced columns were located on the same leaf nodes as the index. This is not an issue for clustered indexes because all data is already three. (That’s why it’s so important to give plenty of thought to how you create your clustered indexes.) But a nonclustered index includes only the key column values in the leaf nodes. For all other data, the optimizer must take additional steps to retrieve that data from elsewhere, which could represent significant overhead for your most common queries.

That’s where the covering index comes in. When defining your nonclustered index, you can include columns in addition to the key columns. For example, suppose one of your application’s primary queries retrieves data from both the **OrderID** and **OrderDate** columns in the **Sales** table:

SELECT OrderID, OrderDate

FROM Sales

WHERE OrderID = 12345;

You can create a composite nonclustered index on both columns, but the **OrderDate** column only adds overhead to the index and serves no purpose as a key column. A better solution is to create a covering index with **OrderID** as the key column and **OrderDate** as the included column:

CREATE NONCLUSTERED INDEX ix\_orderid

ON dbo.Sales(OrderID)

INCLUDE (OrderDate);

This way, you avoid the disadvantages of indexing a column unnecessarily, while still benefiting your query. The included column is not part of the key, but the data is still stored in the leaf nodes. This can improve performance without incurring more overhead. Plus, there are fewer restrictions on columns used as included columns, compared to those used as key columns.

**Does it matter how many duplicate values a key column contains?**

Whenever you create an index, you should try to minimize the number of duplicate values contained in your key columns, or more precisely, try to keep the ratio of duplicate values as low as possible, when compared to the entire set of values.

If you’re working with a composite index, that duplication refers to the key columns as a whole. The individual columns can contain lots of duplicates, but duplication across the columns should be at a minimum. For example, if you create a composite nonclustered index on the **FirstName** and **LastName** columns, you can have multiple **John**values and multiple **Doe** values, but you want to have as few **John** **Doe** values as possible, or better still, only one**John** **Doe**.

The ratio of unique values within a key column is referred to as *index selectivity*. The more unique the values, the higher the selectivity, which means that a unique index has the highest possible selectivity. The query engine loves highly selective key columns, especially if those columns are referenced in the **WHERE** clause of your frequently run queries. The higher the selectivity, the faster the query engine can reduce the size of the result set. The flipside, of course, is that a column with relatively few unique values is seldom a good candidate to be indexed.

**Can you create a nonclustered index on a subset of data in your key column?**

By default, a nonclustered index contains one row for every row in the table. Of course, you can say the same about a clustered index, given that the index *is* the table, but in terms of a nonclustered index, the one-to-one relationship is an important concept because, since SQL Server 2008, you’ve been able to create *filtered indexes* that limit the rows included in the index.

A filtered index can improve query performance because it is smaller and includes filtered statistics, which are more accurate than full-table statistics, resulting in better execution plans. A filtered index also reduces storage and maintenance costs. The index is updated only when the applicable underlying data changes.

In addition, a filtered index is simple to create. In your **CREATE** **INDEX** statement, simply add a **WHERE** clause that defines the filter conditions. For example, you can filter out all null values from your index, as shown in the following statement:

CREATE NONCLUSTERED INDEX ix\_trackingnumber

ON Sales.SalesOrderDetail(CarrierTrackingNumber)

WHERE CarrierTrackingNumber IS NOT NULL;

You can, in fact, filter out any data not relevant to your most critical queries. Be aware, however, that SQL Server places a number of restrictions on filtered indexes, such as not being able to create a filtered index on a view, so be sure to check out the SQL Server documentation.

Also, it might have occurred to you that you can achieve similar results by creating an indexed view. However, a filtered index has several advantages, such as being able to reduce maintenance costs and improving the quality of your execution plans. Filtered indexes also permit online index rebuilds. Try getting that with an indexed view.