http://www.sqlservercentral.com/articles/Performance+Tuning/clusterthatindex/952/
Printed 2009/01/05 03:57AM

Cluster That Index!

By Christoffer Hedgate, 2003/03/30

One topic that is sometimes discussed in SQL Server communities is whether or not you should always have clustered indexes on your tables. Andy Warren discussed this briefly in one of his articles in the Worst Practices-series (Not Using Primary Keys and Clustered Indexes), here I will give my view on this matter. I will show you why I think you should always have clustered indexes on your tables, and hopefully you might learn something new about clustered indexes as well.

What is a clustered index
First off, we'll go through what a clustered index is. SQL Server has two types of indexes, clustered indexes and non-clustered indexes. Both types are organized in the same way with a b-tree structure. The difference between them lies in what the leaf-level nodes -- the lowest level of the tree -- contains. In a clustered index the leaf-level **is the data**, while the leaves of a non-clustered index contains **bookmarks to the actual data**. This mean that for a table that has a clustered index, the data is actually stored in the order of the index. What the bookmarks of the non-clustered index point to depends on if the table also has a clustered index or not. If it does have a clustered index then the leaves of non-clustered indexes will contain the clustering key -- the specific value(s) of the column(s) that make up the clustered index -- for each row. If the table does not have a clustered index it is known as a heap table and the bookmarks in non-clustered indexes are in RID format (File#:Page#:Slot#), i.e. direct pointers to the physical location the row is stored in. Later in this article we will see why this difference is important. To make sure that everyone understands the difference between a clustered index and a non-clustered index I have visualized them in these two images (clustered | non-clustered). The indexes correspond to those of this table:

```
CREATE TABLE EMPLOYEES
(
  empid int NOT NULL CONSTRAINT ix_pkEMPLOYEES PRIMARY KEY NONCLUSTEREI
  , name varchar(25) NOT NULI
  , age tinyint NOT NULI
)


CREATE CLUSTERED INDEX ixcEMPLOYEES ON EMPLOYEES (name)


INSERT INTO EMPLOYEES (empid, name, age) VALUES (1, 'David', 42)
INSERT INTO EMPLOYEES (empid, name, age) VALUES (2, 'Tom', 31)
INSERT INTO EMPLOYEES (empid, name, age) VALUES (3, 'Adam', 27)
INSERT INTO EMPLOYEES (empid, name, age) VALUES (4, 'John', 22)


SELECT * FROM EMPLOYEES WHERE name = 'John'
SELECT * FROM EMPLOYEES WHERE empid = 1
```

In the real indexes these four rows would fit on the same page, but for this discussion I've just put one row on each page. So, to return results for the first query containing `WHERE name = 'John'` SQL Server will traverse the clustered index from the root down through the intermediate node levels until it finds the leaf page containing John, and it would have all the data available to return for the query. But to return results for the second query, it will traverse the non-clustered index until it finds the leaf page containing empid 1, then use the clustering key found there for empid 1 (David) for a lookup in the clustered index to find the remaining data (in this case just the column age is missing). You can see this for yourself by viewing the execution plan for the queries in Query Analyzer (press Ctrl-K too see the plan).

Disadvantages of having a clustered index
Although my general opinion is that you should always have a clustered index on a table, there are a few minor disadvantages with them that in some special circumstances might remedy not having one. First of all, the lookup operation for bookmarks in non-clustered indexes is of course faster if the bookmark contain a direct pointer to the data in RID format, since looking up the clustering key in a clustered index requires extra page reads. However, since this operation is very quick it will only matter in some very specific cases.

The other possible disadvantage of clustered indexes is that inserts might suffer a little from the page splits that can be necessary to add a row to the table. Because the data is stored in the order of the index, to insert a new row SQL Server must find the page with the two rows between which the new row shall be placed. Then, if there is not room to fit the row on that page, a split occurs and some of the rows get moved from this page to a newly created one. If the table would have been a heap -- a table without a clustered index -- the row would just have been placed on any page with enough space, or a new page if none exists. Some people see this as a big problem with clustered indexes, but many of them actually misunderstand how they work. When we say that the data in clustered indexes are stored in order of the index, this doesn't mean that all the data pages are physically stored in order on disk. If it actually was this way, it would mean that in order to do a page split to fit a new row, all following pages would have to be physically moved one 'step'. As I said, this is of course not how it works. By saying that data is stored in order of the index we only mean that the data on each page is stored in order. The pages themselves are stored in a doubly linked list, with the pointers for the list (i.e. the page chain) in order. This mean that if a page split does occur, the new page can still be physically placed anywhere on the disk, it's just the pointers of the pages prior and next to it that need to be adjusted. So once again, this is actually a pretty small issue, and as you will see later in the article there are possible problems of not having a clustered index that can have much more significance than these minor disadvantages.

Advantages of having a clustered index

Apart from avoiding the problems of not having a clustered index described later in this article, the real advantage you can get from a clustered index lies in the fact that they sort the data. While this will not have any noticeable effect on some queries, i.e. queries that return a single row, it could have a big effect on other queries. You can normally expect that apart from the disadvantages shown above, a clustered index will not perform worse than non-clustered indexes. And as I said, in some cases it will perform much better. Lets see why.

Generally, the biggest performance bottleneck of a database is I/O. Reading data pages from disk is an expensive operation, and even if the pages are already cached in memory you always want to read as few pages as possible. Since the data in a clustered index is stored in order this mean that the rows returned by range searches on the column(s) that are part of the clustered index will be fetched from the same page, or at least from adjacent pages. In contrast, although a non-clustered index could help SQL Server find the rows that satisfy the search condition for the range search, since the rows might be placed on different pages many more data pages must be fetched from disk in order to return the rows for the result set. Even if the pages are cached in memory each page needs to be read once for every bookmark lookup (one for each hit in the non-clustered index), probably with each page read several times. You can see this for yourself in Script 1.

As you can see, a carefully placed clustered index can speed up specific queries, but since you can only have one clustered index per table (since it actually sorts the data) you need to think about which column(s) to use it for. Unfortunately the default index type when creating a primary key in SQL Server is a clustered index, so if you're using surrogate keys with an auto-incrementing counter make sure you specify non-clustered index for those primary keys as you will probably not do range searches on them. Also please note that ordering a result set is a great example of where a clustered index can be great, because if the data is already physically stored in the same order as you are sorting the result set the sort operation is (generally) free! However, make sure you don't fall into the trap of depending on the physical ordering of the data. Even though the data is physically stored in one order this does not mean that the result set will be returned in the same order. If you want an ordered result set, you must always explicitly state the order in which you want it sorted.

Problems with not having a clustered index

I have now shown the minor disadvantages that might occur from having a clustered index, plus shown how they can speed up some queries very much. However, neither of these facts are what really makes me recommend you to always have a clustered index on your tables. Instead it is the problems that you can run into when not having a clustered index that can really make a difference. There are two major problems with heap tables, fragmentation and forward-pointers. If the data in a heap table becomes fragmented there is no way to defragment it other than to copy all the data into a new table (or other data source), truncate the original table and then copy all data back into it. With a clustered index on the table you would simply either rebuild the index or better yet, simply run DBCC INDEXDEFRAG which is normally better since it is an online operation that doesn't block queries in the same way as rebuilding it. Of course in some cases rebuilding the index completely might actu ally suit your needs better.

The next problem, forward-pointers, is a bit more complicated. As I mentioned earlier, in a non-clustered index on a heap table the leaf nodes contains bookmarks to the physical location where the rows are stored on disk. This mean that if a row in a heap table must be moved to a different location (i.e. another data page), perhaps because the value of a column of variable length was updated to a larger value and no longer fits on the original page, SQL Server now has a problem. All non-clustered indexes on this table now have incorrect bookmarks. One solution would be to update all bookmarks for the affected row(s) to point at the new physical location(s), but this could take some time

and would make the transaction unnecessary long and would therefore hurt concurrency. Therefore SQL Server use forward-pointers to solve this problem.

What forward-pointers mean is that instead of updating the bookmarks of non-clustered indexes to point to the new physical location, SQL Server places a reference message at the old location saying that the row has been moved including a pointer to the new location. In this way the bookmarks of non-clustered indexes can still be used even though they point to the old location of the row. But, it also mean that when doing a bookmark lookup from a non-clustered index for a row that has been moved, an extra page read is necessary to follow the forward-pointer to the new page. When retrieving a single row this probably won't even be noticed, but if you're retrieving multiple rows that have been moved from their original location it can have a significant impact. Note that even though the problem stem from the fact that SQL Server can't update the non-clustered index bookmarks, it is not limited to queries using the indexes. The worst case scenario is a query where SQL Server need to do a table scan of a heap table containing lots of forward-pointers. For each row that has been forwarded SQL Server needs to follow the pointer to the new page to fetch the row, then go back to the page where the forward-pointer was (i.e. the page where row was originally located). So for every forwarded row, SQL Server need two extra page reads to complete the scan. If the table would have had a clustered index the bookmarks of all non-clustered indexes would have been clustering keys for each row, and physically moving a row on disk would of course not have any effect on these. An extreme example of this is shown in Script 2. Even though this example may be a bit extreme forward-pointers are likely to become a problem in tables where rows are sometimes moved, because there is no way in SQL Server to remove forward-pointers from a table.

Summary

In this article I have described what a clustered index is and how they differ from non-clustered indexes, and I have also tried to show you why I think that you should always have a clustered index on every table. As I said there are of course exceptions, but these are so uncommon that I always check that all tables have clustered indexes as one of the first things I do when performing a database review. Please post your thoughts on this matter in the feedback section, and don't forget to vote!
