Benchmarking SQL Server 2005 Covering Indexes

Source	http://www.sql-server-performance.com/da_benchmark_covering_indexes.asp

What is a Covering Index?

A covering index is a form of a non-clustered composite index, which includes all of the columns referenced in the SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data the query is looking for and SQL Server does not have to look up the actual data in the table, reducing logical and/or physical I/O, and boosting performance.

One way to help determine if a covering index could help a query's performance is to create a graphical query execution plan in SQL Server 2005 Management Studio for the query in question and see if there are any Bookmark Lookups being performed. Essentially, a Bookmark Lookup tells you that the Query Processor had to look up the row columns it needs from a table or a clustered index, instead of being able to read it directly from a non-clustered index. Bookmark Lookups can reduce query performance because they produce extra disk I/O to retrieve the column data.

Bookmark lookups are a mechanism to navigate from a non-clustered index row to the actual data row in the base table (clustered index) and can be very expensive when dealing with a large number of rows. When a small number of rows are requested by a query, the SQL Server optimizer will try to use a non-clustered index on the column or columns contained in the WHERE clause to retrieve the data requested by the query. If the query requests data from columns not contained in the non-clustered index, SQL Server must go back to the data pages to obtain the data in those columns. It doesn't matter if the table contains a clustered index or not, the query will still have to return to the table or clustered index to retrieve the data.

One way to avoid a Bookmark Lookup is to create a covering index. This way, all the columns from the query are available directly from the non-clustered index, which means that Bookmark Lookups are unnecessary, which reduces disk I/O and helps to boost performance of your queries.

Impact

Covering indexes are used to boost query performance because the index includes all the columns in the query. Non-clustered indexes include a row with an index key value for every row in a table. In addition, SQL Server can use these entries in the index's leaf level to perform aggregate calculations. This means that SQL Server does not have to go to the actual table to perform the aggregate calculations, which can boost performance.

While covering indexes boost retrieval performance, they can slow down INSERT, DELETE, and UPDATE queries. This is because extra work is required for these procedures to maintain a covering index. This is generally not a problem, unless your database is subject to a very high level of INSERTs, DELETEs, and UPDATEs. You may have to experiment with covering indexes to see if they help more than they hurt performance before you implement them in your production systems.

While introducing a covering index provides both positive and negative performance issues, as discussed above, the focus of this article is to find out what will happen under the following conditions when running a query under SQL Server 2005:

1. **Performance without any indexes.**
2. **Performance with non-clustered indexes.**
3. **Performance with covering indexes.**

Approach

In this section, we take a look at how we will be testing the above three index conditions. First, we create a table with the following format.

```
CREATE TABLE [dbo].[OrderDetails](
     [OrderNo] [int] NOT NULL,
     [ItemCode] [varchar](50) NOT NULL,
     [Qty] [int] NULL,
     [Price] [float] NULL,
     [Status] [char](1) NULL
) ON [PRIMARY]
```

Then, we insert data into the table OrderDetails. This table needs a large volume of data so that SELECT statements without a covering index force a table scan, index scan, or bookmark lookup, which will result in a sufficiently long enough timeframe so we can better compare results. So we have selected a table with more than 2,000,000 records for our testing.

We will execute the following query and note the query execution plan, Execution Time, CPU Cost, and I/O Cost. A query execution plan outlines how the SQL Server query optimizer actually ran (or will run) a specific query. This information is very valuable when it comes to finding out why a specific query is running slow. We will analyze the execution plans for each case and identify how performance has increased or decreased. Execution time is the time taken to execute a query, and the smaller the execution time, the better should be the performance. When you have smaller I/O or CPU cost, this indicates that fewer server resources will be used, indicating better performance.

Below is the query that we are going to use throughout our testing. We have selected this query so that it will use a Bookmark Lookup as specified previously.

```
SELECT OrderNo,
     ItemCode,
     Qty,
     Price
FROM dbo.OrderDetails
WHERE ItemCode = 'A2-K137-FF1931'
     AND (OrderNo BETWEEN 250000 and 300000)
```

Our first test is to see what the performance is when there are no indexes on the table.

Then, our next step is to create two non-clustered indexes, as follows:

```
CREATE NONCLUSTERED INDEX IX_Order_Details_ItemCode ON OrderDetails
(ItemCode)
GO
CREATE NONCLUSTERED INDEX IX_Order_Details_OrderNo ON OrderDetails
(OrderNo)
```

And last, we apply a covering index. As per the definition of the covering index, we have to apply the index for the all the columns in the query. So we will apply a covering index to the OrderNo, ItemCode, Qty, and Price columns of the OrderDetails table, like this:

```
CREATE NONCLUSTERED INDEX IX_Order_Details_Coverindex ON OrderDetails
(
     OrderNo,
     ItemCode,
     Qty,
     Price)
```

We will then re-execute the query and note the above parameters again to obtain performance numbers for data retrieval when the covering index exists.

For each of the above cases the following INSERT query will be executed and the same data will be returned.

```
INSERT INTO OrderDetails
VALUES (
     3124567,
     '123456',
     1,
     0.35,
     'N')
```

NOTE: CHECKPOINT and DBCC DROPCLEANBUFFERS will be executed after every operation, which clears data from the cache. The DBCC DROPCLEANBUFFERS command is used to remove all the test data from SQL Server's data cache (buffer) between tests to ensure fair testing. Keep in mind that this command only removes clean buffers, not dirty buffers. Because of this, before running the DBCC DROPCLEANBUFFERS command, you may first want to run the CHECKPOINT command first. Running CHECKPOINT will write all dirty buffers to disk. And then when you run DBCC DROPCLEANBUFFERS, you can be assured that all data buffers are cleaned out, not just the clean ones.

Results

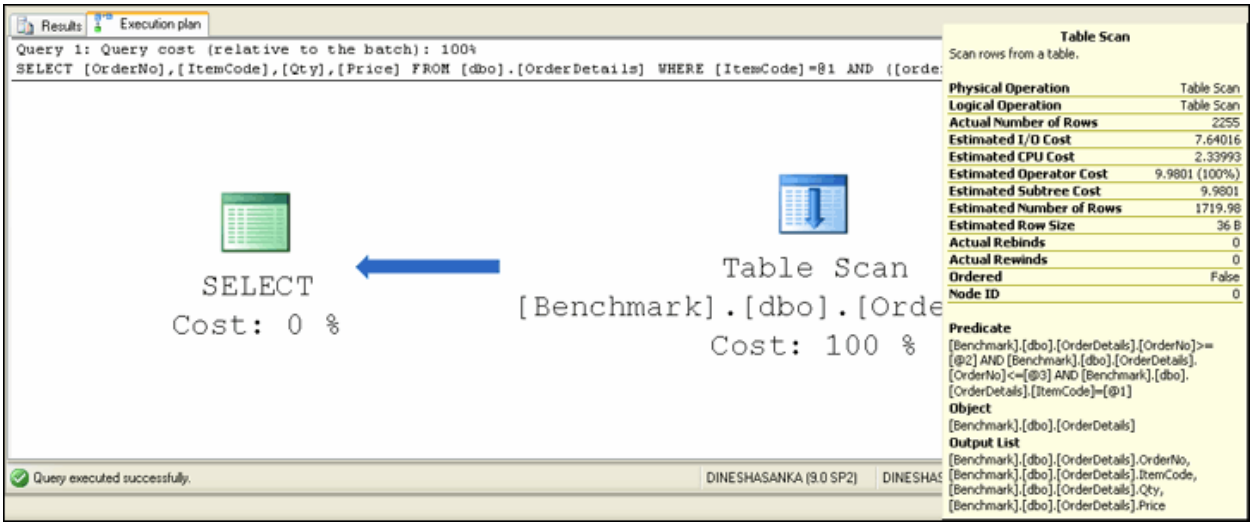
Following are the results gathered from the Execution Plans of each query. Execution time is in seconds, and CPU Cost and I/O Cost were calculated/measured from the Execution plans. All the values for the parameters were noted down for three cases as mentioned above against the same SELECT queries and the same INSERT queries.

	SELECT			INSERT		
	Without any Indexes	With Non-Covering Index	With Covering Index	Without any Indexes	With Non-Covering Index	With Covering Index
Execution Time (Sec.)	12	4	3	1	4	5
CPU Cost	2.339937	0.4381462	0.312734	0.000001	0.000003	0.000004
I/O Cost	7.64016	2.6854338	1.12757	0.01	0.03	0.04

Analysis

Case 1: Without Any Indexes

Where there are no indexes on the table, there is no other way of returning data except to perform a table scan. Your query will run through the entire table, row by row, to fetch the record(s) that matches your query conditions.

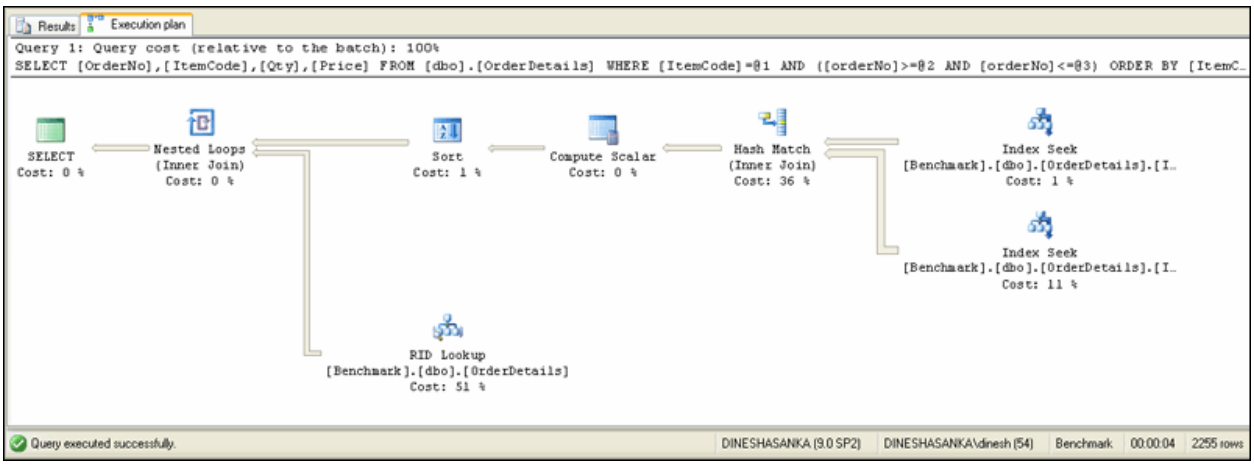


Needless to say, this is an extremely insufficient way of fetching data from your tables. You can see that it took twelve seconds to fetch the data.

Case 2: With Non-Covering Indexes

Now we add two non-clustered indexes to these two columns—ItemCode and Order Number—in our table. This time, the query optimizer uses a RID Lookup to get this

information. A RID Lookup is a bookmark lookup on a heap using a supplied row identifier (RID). The Argument column contains the bookmark label used to look up the row in the table and the name of the table in which the row is looked up. As you can see below, the RID Lookup has taken 51% of the entire cost.

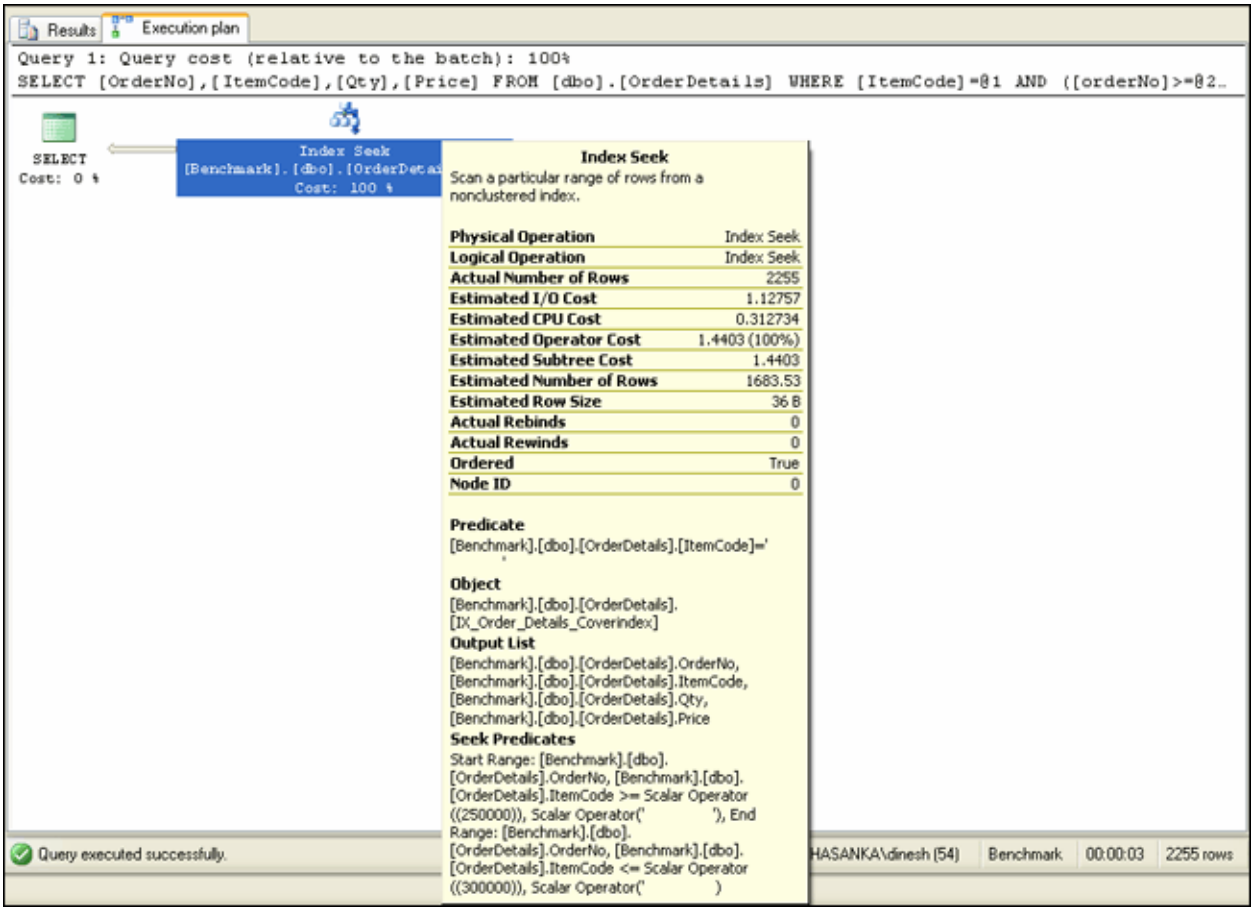


However, because of the usage of indexes, this time the script was executed in 4 seconds, which is 300% faster than the normal. CPU Costs were reduced by 81%, while I/O Costs were reduced by 64%.

However, INSERT statements now take more time and resources than when there was no index on the table.

Case 3: With Covering Indexes

Our next scenario includes a covering index. As a covering index includes all the information for the query, SQL Server will retrieve the data faster and with less resource utilization. In addition, with a covering index, you won't get as complex an Execution Plan.



With a covering index, the execution time of the SELECT query has been reduced to 3 seconds. When you compare this result to not using any indexes, you can see that it has an improvement of 400%, while with the non-clustered index, it has a 75% improvement in performance. CPU cost and I/O Cost also improved, which means that after the covering index was introduced, the query uses fewer resources for SELECT queries.

Like in the previous case, INSERTs have taken more time, as well as additional resources. You can see INSERT statement execution time has gone up for 5 seconds, as compared to the 1-second timeframe when no indexes were added to the table.

Conclusion

As the above statistics suggest, covering indexes offer both advantages and disadvantages. It is your job as the DBA to determine whether the advantages outweigh the disadvantages, and whether implementing a covering index is best for your specific needs.

~~~ End of Article ~~~