Indexes in MS SQL Server

Keshav Singh, 2 May 2011 CPOL

This article focuses on how MS SQL Server uses indexes to read and write data.

Introduction

I bought a book from crossword; he packed the book and added two bookmarks into my pack. A thought came to my mind. Why do I need a bookmark? I can easily memorize the page number and the next time resume from the same page when I resume reading, or read them all over to reach to the point where I stopped reading. But not all have a blessed memory; moreover, there are better things to remember, my grandpa would rather bookmark and rely on it to help him resume reading. It's a kind of simple index, isn't it?

This article focuses on how MS SQL Server uses indexes to read and write data. Data is arranged by SQL Server in the form of *extents* and *pages*. Each extent is of size 64 KB, having 8 pages of 8KB sizes. An extent may have data from multiple or same table, but each page holds data from a single table only. Logically, data is stored in record sets in the table. We have fields (columns) identifying the type of data contained in each of the record sets. A table is nothing but a collection of record sets; by default, rows are stored in the form of heaps unless a clustered index has been defined on the table, in which case, record sets are sorted and stored on the clustered index. The heaps structure is a simple arrangement where the inserted record is stored in the next available space on the table page.

Heaps seem a great option when the motive is simply storing data, but when data retrieval steps in, this option back fires. An index acts as a fire fighter in this scenario. Indexes are arranged in the form of a B-Tree where the leaf node holds the data or a pointer to the data. Since the stored data is in a sorted order, indexes precisely know which record is sitting where. Hence an index optimizes and enhances the data retrieval immensely.

But everything comes at a cost; the price we pay for having an index on the table is, each time there is an Insert/Update/Delete, SQL Server updates the active indexes on the table where these DML are operated. Hence simply creating indexes madly for the sake of better data retrieval will not serve the purpose. If there are 20 indexes on a table, each time a DML is done on the table, all these 20 indexes shall be updated so that they can uniquely figure out the location of the record. Let's dive deep into the indexes.

Setup: All code has been tested on MS SQL Server 2008 R2.

Clustered Index (CI)

A clustered index is something that reorganizes the way records in the table are physically stored. Therefore a table can have only one clustered index. The leaf nodes of a clustered index contain the data pages, by which I mean the key-value pair in the clustered index has the index key and the actual data value. Also remember, a clustered index will be created on a table by default the moment a primary key is created on the table. A clustered index is something like your train ticket B4/24, you know that you need to board coach B4 and sit on seat number 24. So this index physically leads you to your actual seat.

We will follow this up closely with an example:

```
USE TestDB
GO

CREATE TABLE Sales(
 ID INT IDENTITY(1,1)
,ProductCode VARCHAR(20)
,Price FLOAT(53)
,DateTransaction DATETIME);
```

I have created a table Sales, and then created a Stored Procedure to insert 2,00,000 records into the Sales table. This sizable chunk of data will help us to notice the differences very clearly.

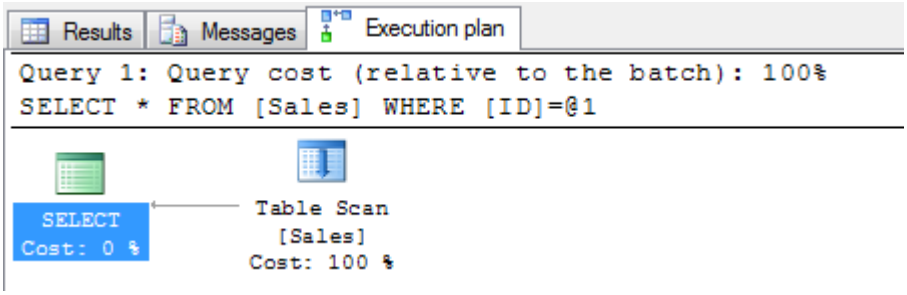
```
CREATE PROCEDURE InsertIntoSales
AS
SET NOCOUNT ON
BEGIN
DECLARE @PC VARCHAR(20)='A12CB'
DECLARE @Price INT = 50
DECLARE @COUNT INT = 0
      WHILE @COUNT<200000
      BEGIN
      SET @PC=@PC+CAST(@COUNT AS VARCHAR(20))
      SET @Price=@Price+@COUNT
      INSERT INTO Sales VALUES (@PC,@Price,GETDATE())
      SET @PC='A12CB'
      SET @Price=50
      SET @COUNT+=1
      END
END

EXEC InsertIntoSales
```

Now we have created the table and inserted 2,00,000 records into it, but there is no index defined on any column.

Press Control+M. This will “Include the Actual Execution Plan” in the results. Let's run the below query.

```
SET STATISTICS IO ON
SELECT * FROM Sales WHERE ID=189923

ID          ProductCode      Price      DateTransaction
----------- ---------------- ---------- -----------------------
189923      A12CB189922      189972     2011-03-21 12:07:48.310

(1 row(s) affected)

Table 'Sales'. Scan count 1, logical reads 1129, physical reads 0,
```

```
      read-ahead reads 0, lob logical reads 0,
      lob physical reads 0, lob read-ahead reads 0.


(1 row(s) affected)
```



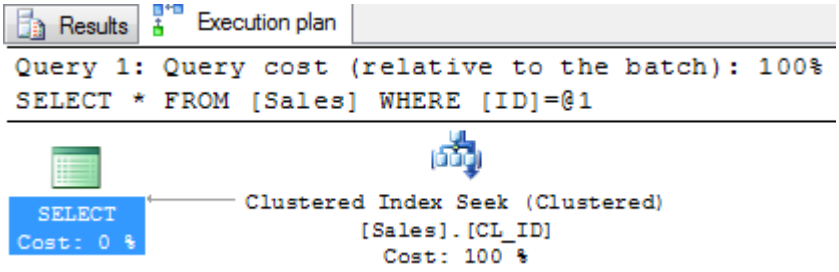
The Execution plan tab on the results show that the record has been retrieved on a table scan and the logical reads are 1129.

Now let's build a clustered index on the ID column of the Sales table.

```
CREATE CLUSTERED INDEX CL_ID ON SALES(ID);
```

Let us press CTRL+M and rerun the same query:

```
SET STATISTICS IO ON
SELECT * FROM Sales WHERE ID=189923

ID          ProductCode      Price        DateTransaction
----------- ---------------- ------------ -----------------------
189923      A12CB189922      189972       2011-03-21 12:07:48.310

(1 row(s) affected)

Table 'Sales'. Scan count 1, logical reads 3, physical reads 0,
      read-ahead reads 0, lob logical reads 0,
      lob physical reads 0, lob read-ahead reads 0.


(1 row(s) affected)
```



The Execution plan tab on the results shows that the record has been retrieved on Index seek and the logical reads are 3. After the clustered index creation, SQL Server has been able to reduce the logical reads dramatically and the query has been optimized. Clearly the index knows where to look for the record.

Non-Clustered Index (NCI)

A non-clustered index is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk. The leaf node of a non-clustered index does

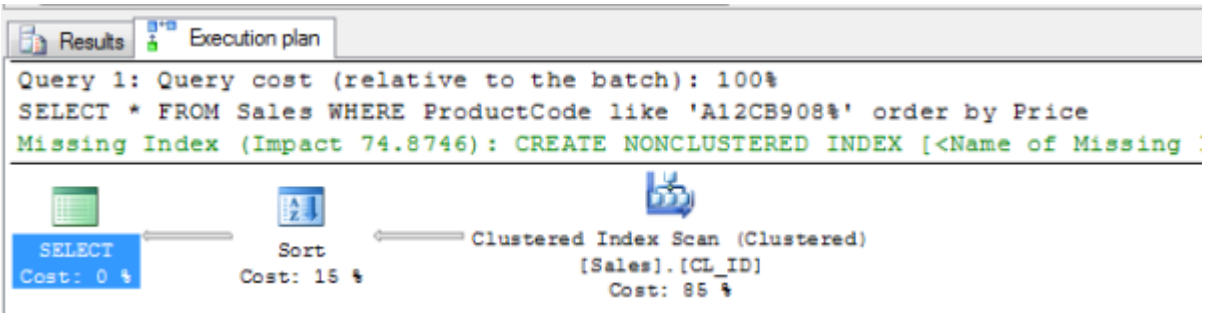
not consist of the data pages but a pointer to it. That goes to say that a non-clustered index can't survive on its own - it needs a base to live on. A non-clustered index uses a clustered index (if defined) or the heap to build itself.

When a non-clustered index uses the heap, the leaf node (or the pointer) is a physical location of the data. When it uses a clustered index, the leaf node (or the pointer) is the clustered index key value and this key value in turn points to the actual data.

Part I: When NCI Uses a CI

Getting back to Sales, we already have a CI (CL_ID) on the ID column, now if we have a query something like:

```
SET STATISTICS IO ON
SELECT * FROM Sales WHERE ProductCode like 'A12CB908%' order by Price
Press Control+M and execute the query
There are arround 111 records retrived
-----------------------------------------------------
(111 row(s) affected)

Table 'Sales'. Scan count 1, logical reads 1130, physical reads 0,
      read-ahead reads 0, lob logical reads 0,
      lob physical reads 0, lob read-ahead reads 0.

(1 row(s) affected)
```



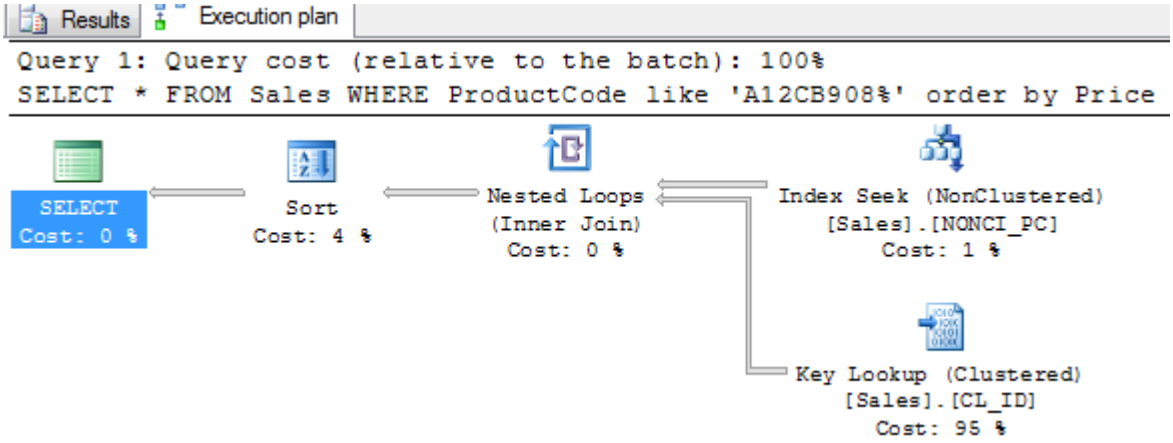
We find that the query first uses the clustered index to get 111 records and then uses a sort operation; the logical reads are as high as 1130. There is also a missing index suggestion.

Let's consider SQL Server's advice and create a non-clustered index (NONCI_PC) on the ProductCode column. Since we have a CI already, this NCI would be built on the CI.

```
CREATE NONCLUSTERED INDEX NONCI_PC ON SALES(ProductCode);
```

Press Control+M and rerun the same query, this time around, we can see the data fetch plan change.

```
SET STATISTICS IO ON
SELECT * FROM Sales WHERE ProductCode like 'A12CB908%' order by Price
-------------------------------
(111 row(s) affected)

Table 'Sales'. Scan count 1, logical reads 351, physical reads 0,
      read-ahead reads 7, lob logical reads 0,
      lob physical reads 0, lob read-ahead reads 0.

(1 row(s) affected)
```



The logical reads have been minimized and the revised execution plan is as in the figure. This was the example where a ***non-clustered index used a clustered index***.

Part II: When NCI Uses a Heap

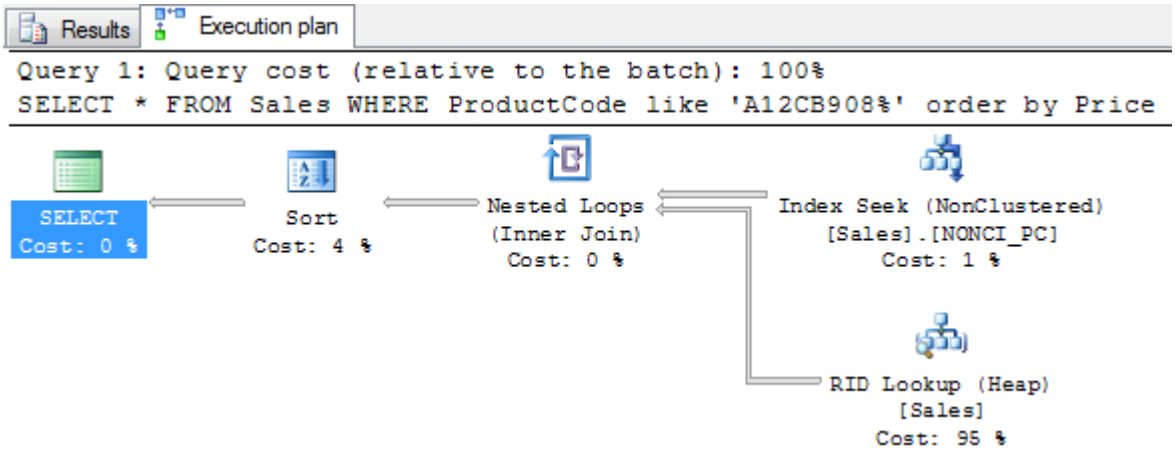
When there is no clustered index built on a table and a non-clustered index is built, it uses the heap for data retrieval. The indexed column or columns are sorted along with a pointer to the physical location of the data.

The big question is, how do I know if I should create an NCI on a CI or on a heap?

The answer is in the query, if data is queried typically on one particular column, it would be beneficial to build NCI upon a CI, but when the case is something like what we have in the present Sales example where we will be building a one-column NC index on a heap, the NCI would be merely a two-column table containing the key-value pair (index key and physical location, i.e., value). This would be the best optimization in this scenario. To follow this up, let's review the example closely.

With respect to the sales example, let's delete the clustered index CL_ID created on the ID column and re-evaluate.

```
DROP INDEX Sales.CL_ID;

SET STATISTICS IO ON
SELECT * FROM Sales WHERE ProductCode like 'A12CB908%' order by Price
-------------------------------------
(111 row(s) affected)

Table 'Sales'. Scan count 1, logical reads 114, physical reads 0,
      read-ahead reads 0, lob logical reads 0,
      lob physical reads 0, lob read-ahead reads 0.

(1 row(s) affected)
```



The logical reads have been further optimized and the execution plan also has been revised. In this case, the query uses the non-clustered index to be run on the heap.

We have been able to create indexes so that our queries work with a minimal performance overhead. So now the next big question is, *Wouldn't it be great if there was someone to help us out in prompting on the indexes to be built based on our queries?*

Yes there is, DTA reports work for helping us out with this..

Using DTA

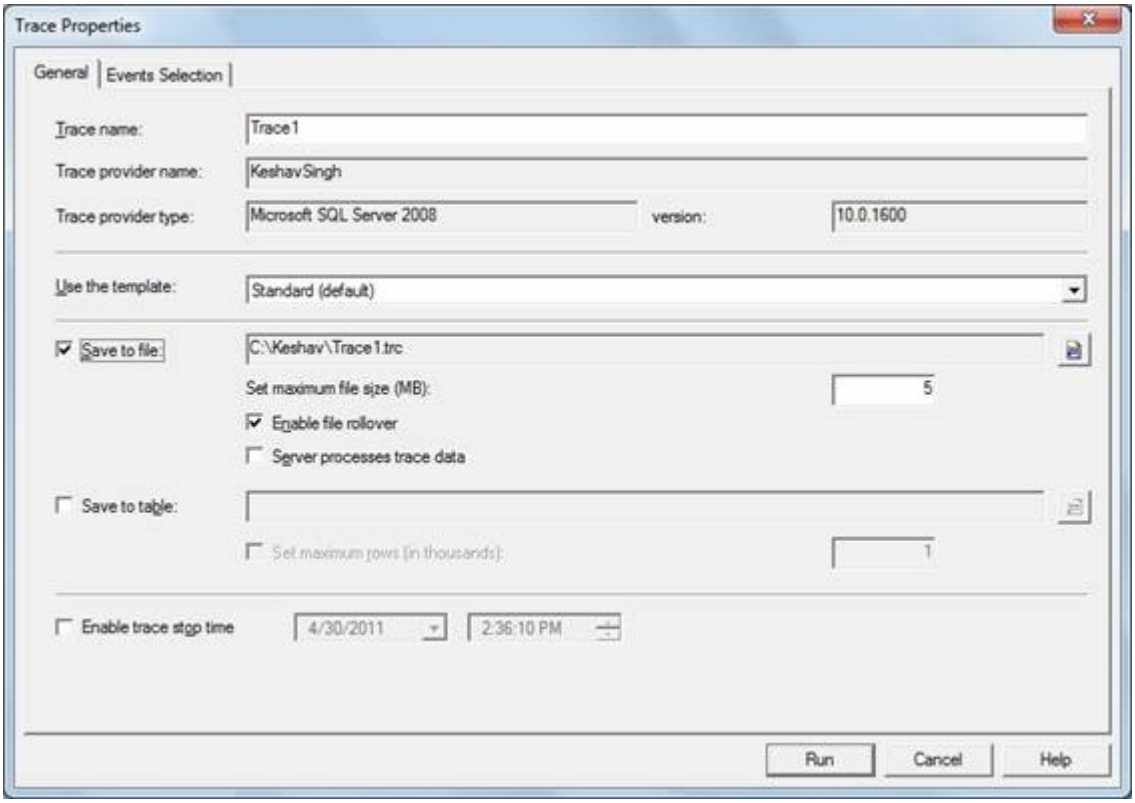
Now we we will see how we can get the advices from DTA and optimize our queries.

To be able to do this, let's drop all the indexes from our Sales table that we have created so far.

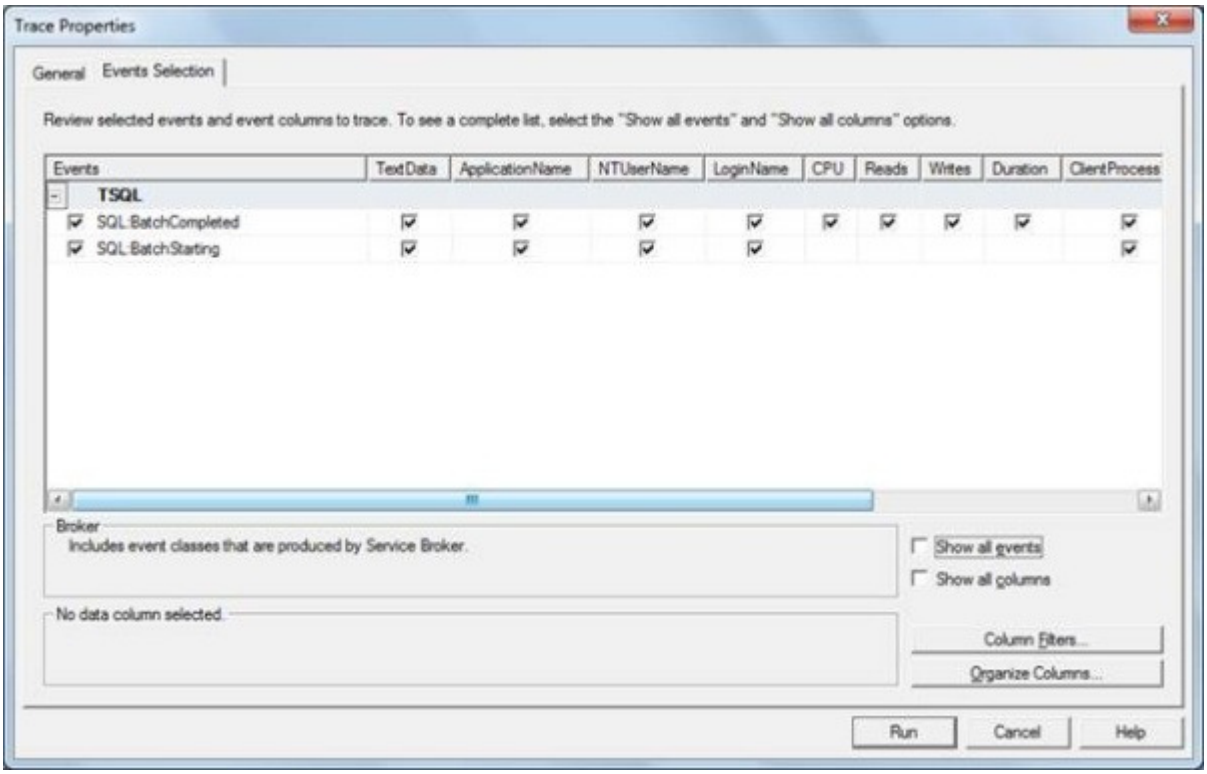
```
DROP INDEX CL_ID ON Sales;
DROP INDEX NONCI_PC ON Sales;
```

Great! Next we will create two workloads for DTA. Go to Start>All Programs>MS SQL Server 2008>Performance Tools>SQL Server Profiler.

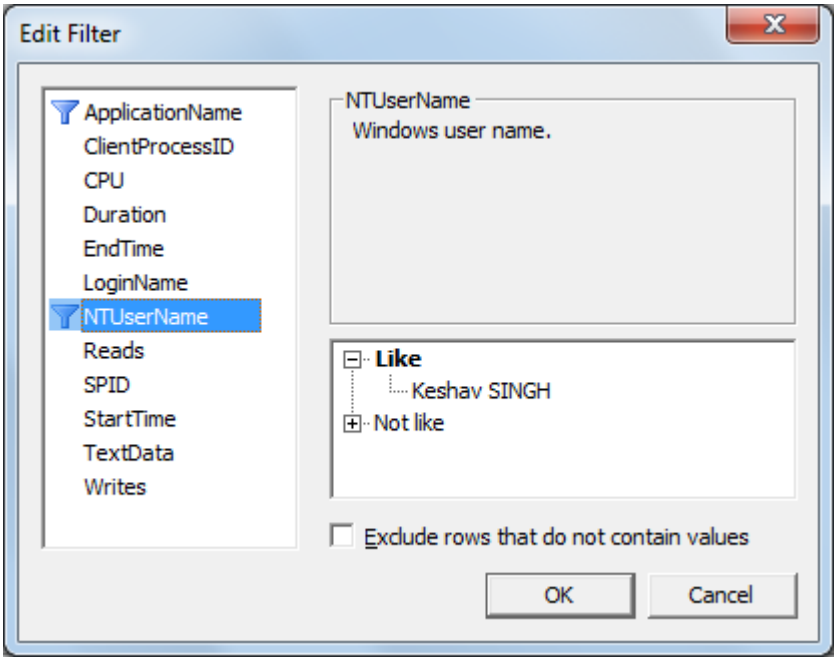
Once it loads, on the top left corner, click on New trace, connect to the server, and then the below screen pops up. In the General tab, fill in the trace name (I have named it 'Trace1') and check 'Save to file' and provide a path for saving your *Trace1.trc* file.



Next comes the Event Selection tab. Here we need to provide the events for which we want the trace to log the entries. I have simply kept the 'T-SQL' active and unchecked the rest.



In the Column Filters button, I have filtered by *NTUserName like 'Keshav SINGH'* which is my login.. and pressed OK.

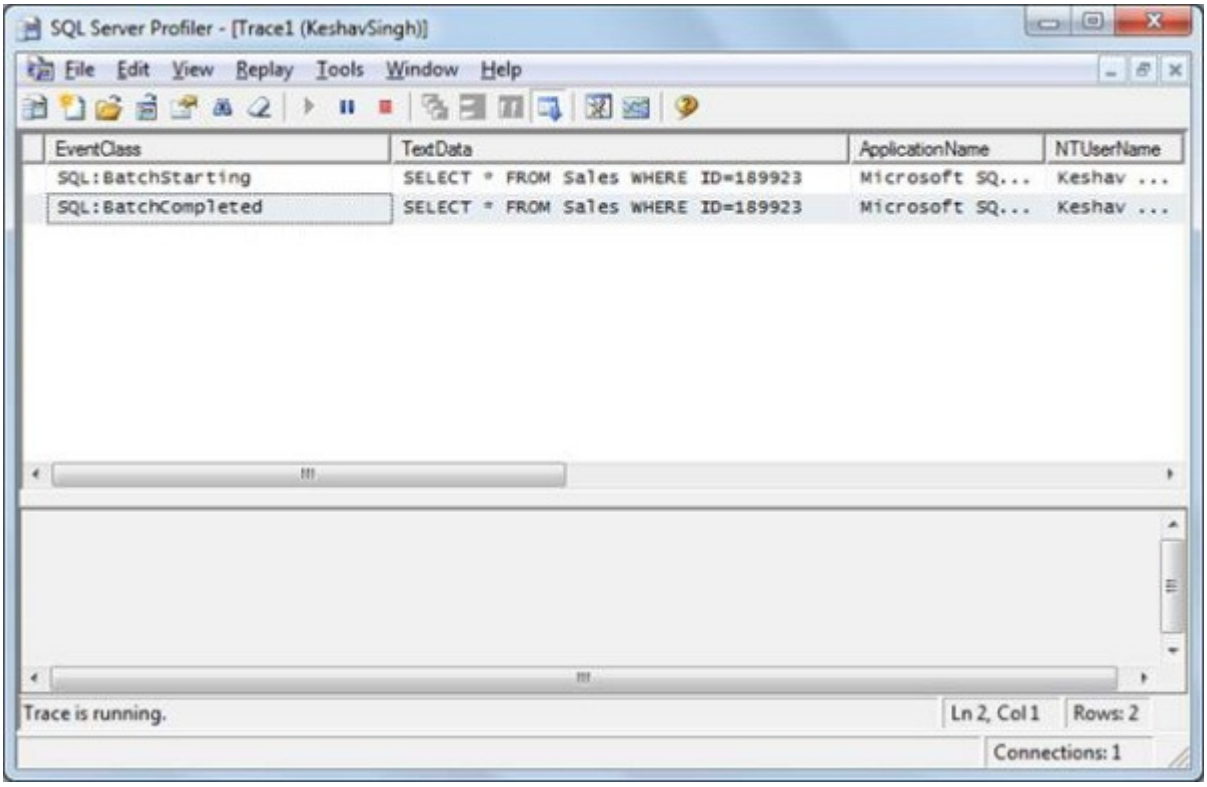


Looks like we are all set. Press Run and begin the trace.

Next we go to the SSMS login and execute the query:

```
SELECT * FROM Sales WHERE ID=189923
```

Go back to the profiler, and we will find the SQL query has been captured. Stop the trace and save it.

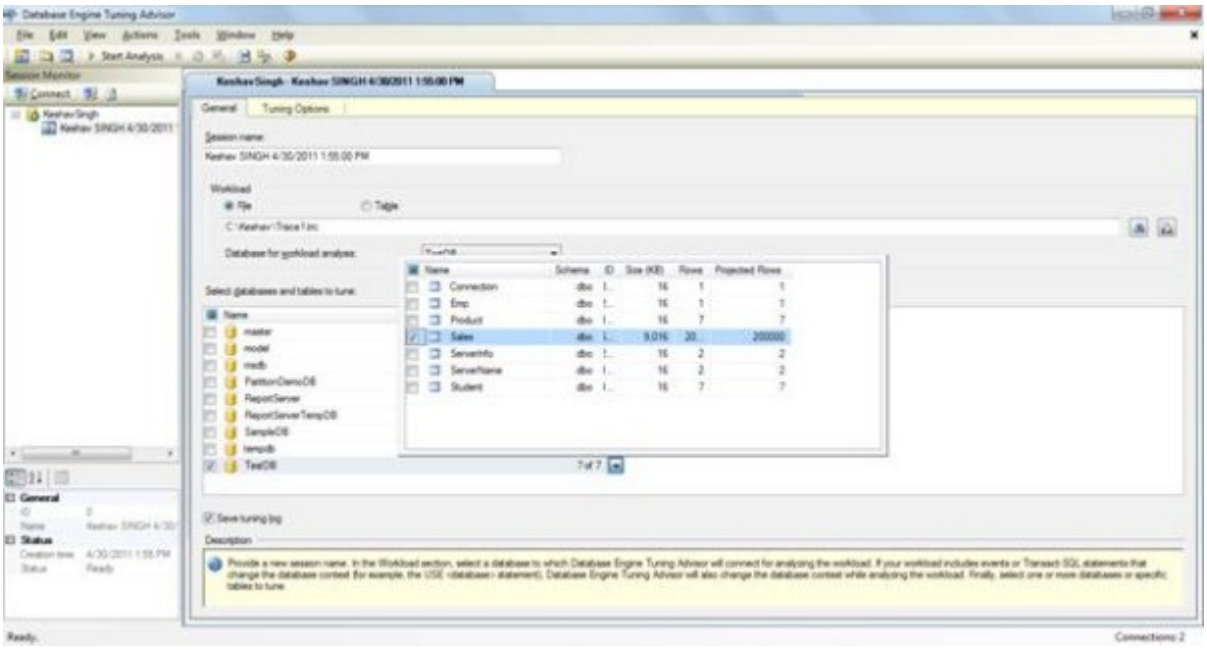


Okay, we have been able to prepare the workload. Now it's time we seek help from DTA..

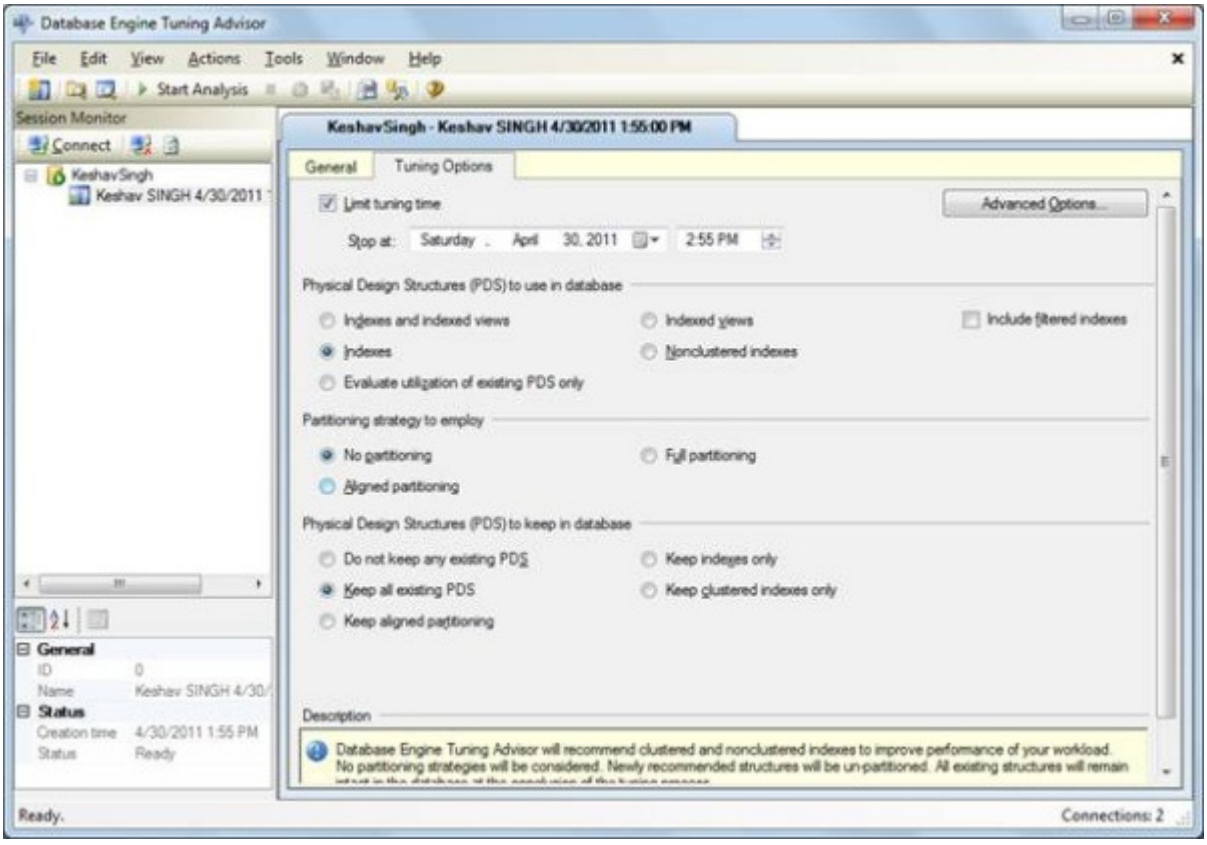
Go to Start>All Programs>MS SQL Server 2008>Performance Tools>Database Engine Tuning Advisor.

The DTA looks like below. Connect to the same server hosting the TestDB database which has

our Sales table. In the General tab for Workload selection, check the file radio button and browse through to *Trace1.trc* that we captured from the Profiler. The database for Workload Analysis is TestDB and under *Select databases and tables to tune*, select TestDB and check Sales Table.



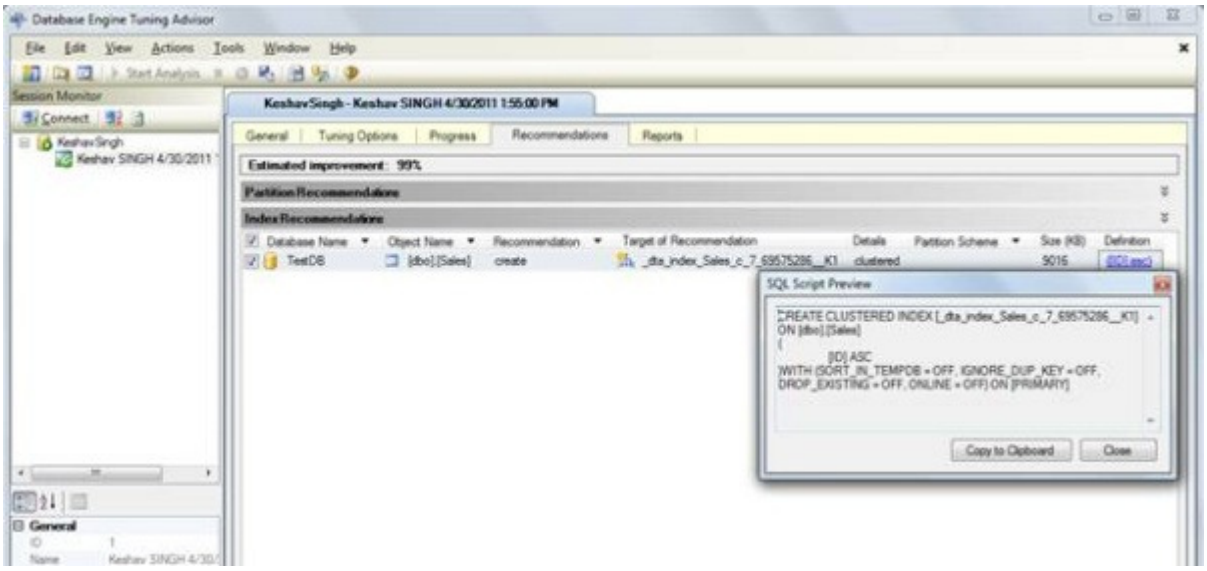
Next, in the Tuning Option tab, keep the default selection which is for Indexes tuning.



Press Start Analysis (top left with the green triangle) and DTA starts the job.

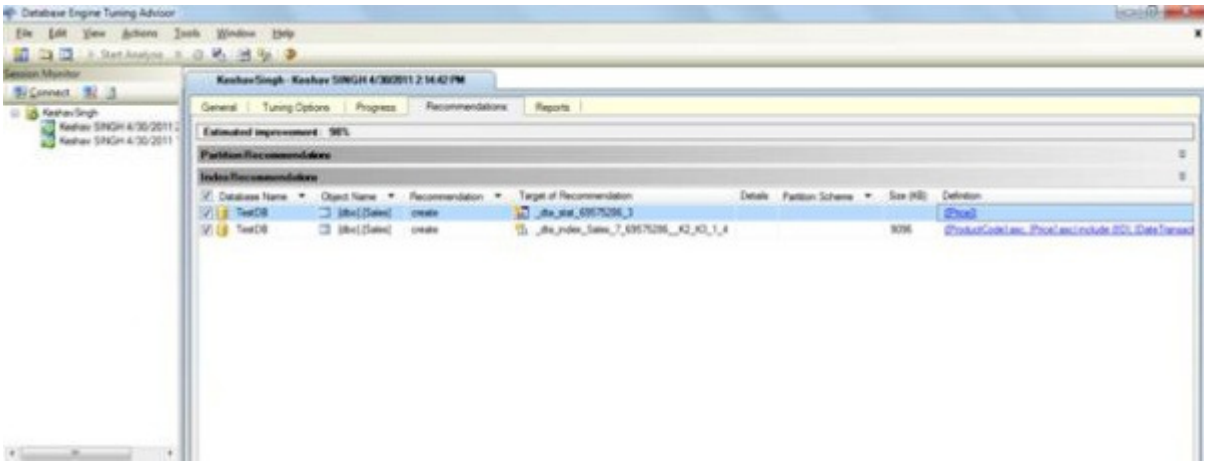
Once the job is complete, you will find three more tabs: progress, recommendations, and reports. Let's take a look at the recommendations. DTA suggests for the query that we should be creating a clustered index on the ID column of the Sales table and also gives the estimated size/cost of the index. If you click on the definition, it also provides the T-SQL query you need to execute. Copy and simply connect to SSMS and execute the query and you shall gain an estimated

improvement of 99 %.



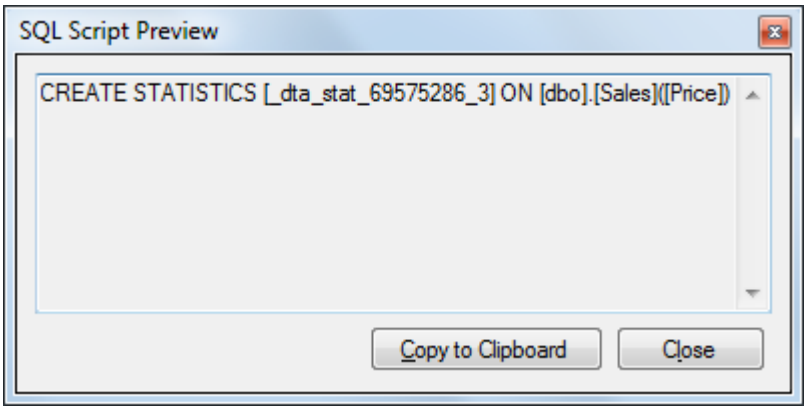
Okay, so now we have created a CI on Sales (ID) as per the DTA's advice. Next we shall follow the same guidelines for the below query. Create a workload in the profiler and use DTA to optimize the query.

```
SELECT * FROM Sales WHERE ProductCode like 'A12CB908%' order by Price
```

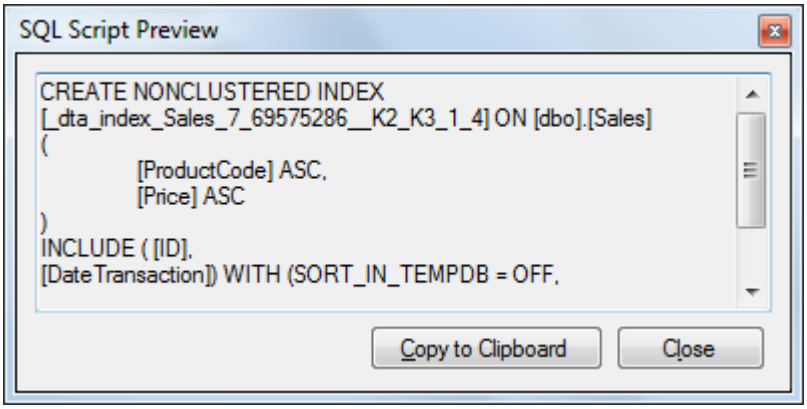


On completion, we get two recommendations from DTA and as we can see, an estimated improvement of 98 %.

The first advice is to create a statistics:



And the second being a non-clustered index on ProductCode and Price.



Summary

Building indexes is totally based on the criteria of querying. There is no hard and fast rule on the number of non-clustered indexes that can be created on a table. The columns on which the DMLs are executed frequently qualify for indexing. SQL Server allows at most one clustered index in any version. As far as non-clustered indexes are concerned, 2005 allows 249 of them to be created while 2008 allows 999 non-clustered indexes.

To add to that, it is not always the case that the query can be optimized further by simply creating an index on a column. And as there are no free services, indexes too charge considerable fees. Every time there is a DML (insert/update/delete) fired on an indexed table, SQL Server updates the index to be able to identify the record. Hence if there are more indexes, it's liable that the DMLs will take a longer time to execute. Hence simply creating a large number of indexes doesn't serve the purpose. That is why it's advised to drop all indexes before a BCP or BULK INSERTs, and rebuild them upon completion of the activity.

Also, indexes lead to defragmentation of the tables and they charge the costly time of DBAs for their maintenance. If used judiciously, they enhance the performance of queries many folds.

FAQ

In an interview, I was asked: *Once we declare a primary key, a clustered index is created on the column by default; what if I wish to create a clustered index and a primary key on two different columns? Is it possible?*

It is very much possible to have two different columns as primary key and clustered indexes. But remember, if I create a Primary Key on a table first, a CI will also be created. Now, in case I need them on two different columns, drop the Primary Key constraint and the CI shall automatically vanish. Now create a CI on column A and declare column B as Primary Key, and column B will have a NCI created by default on it instead of a CI. This way, we can have two columns as Primary Key and CI declared on them.

License

Share

About the Author

Keshav Singh
Database Developer
India

I am a Microsoft certified Technology Specialist in MS SQL Server 2008 and 2005.I have fair amount of experience and expertise in MS database development, administration and modeling and MS BI. Started my career in 2007 and primarily into MS databases, and later diversified into MS BI especially SSIS. I also have a little exposure in Oracle 10g while working on one of the migration projects. But MS SQL Server is my passion!

Firm believer in knowledge grows from sharing, I really like reading books, trying new features and sharing the little that I know. Unless we're willing to have a go, fail miserably, have another go, success won't happen. The best thing I have discovered about myself is the "never say die" attitude. The best comment I have ever received is when my manager at my first job said "when this guy works he throws away his watch".

Comments and Discussions

25 messages have been posted for this article Visit **http://www.codeproject.com/Articles/190263/Indexes-in-MS-SQL-Server** to post and view comments on this article, or click **here** to get a print view with messages.