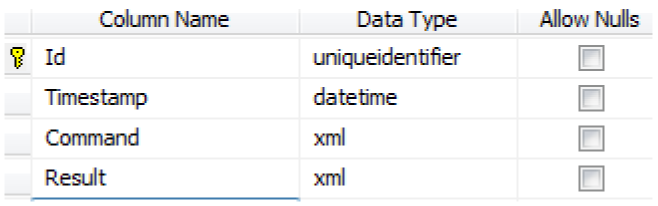
Understanding SQL Server 2008 R2 Execution Plan – Part 1

Daniel Frederico Lins Leite – daniel@machinaaurum.com.br

Brazil, Rio de Janeiro - 2011/11/12.

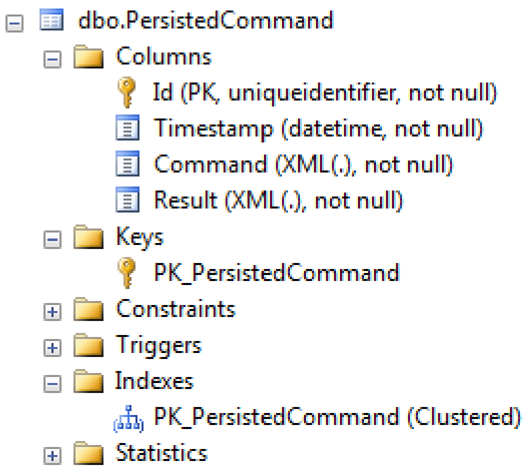
In this post we are going to understand the SQL Server 2008 R2 Execution Plan and the differences between Clustered and Non-Clustered Index Seek and Scan, Included Columns and others index-things in SQL Server.

Our first step is to create a table that will contain all the commands that my application will receipt, so I will name it “PersistedCommands”.



Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
Timestamp	datetime	☐
Command	xml	☐
Result	xml	☐

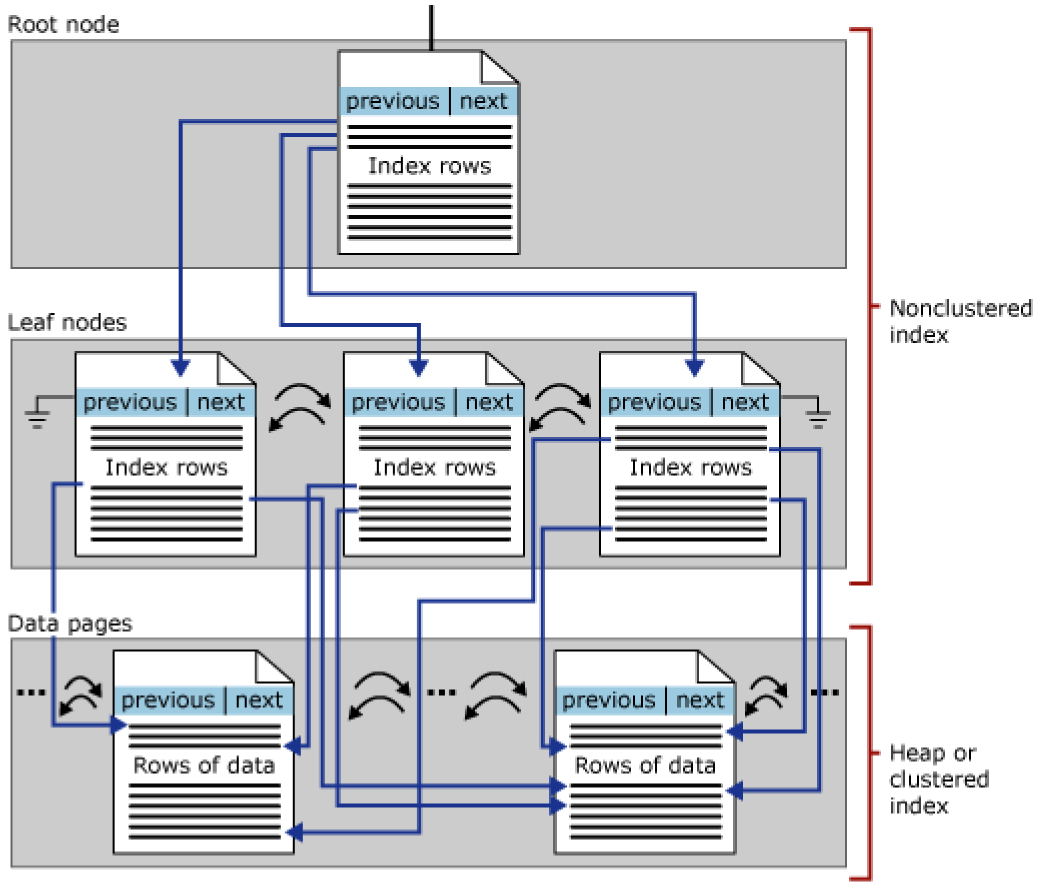
Using the default configuration of tables in SQL Server 2008 R2 we will get: a clustered index in the “Id” column.



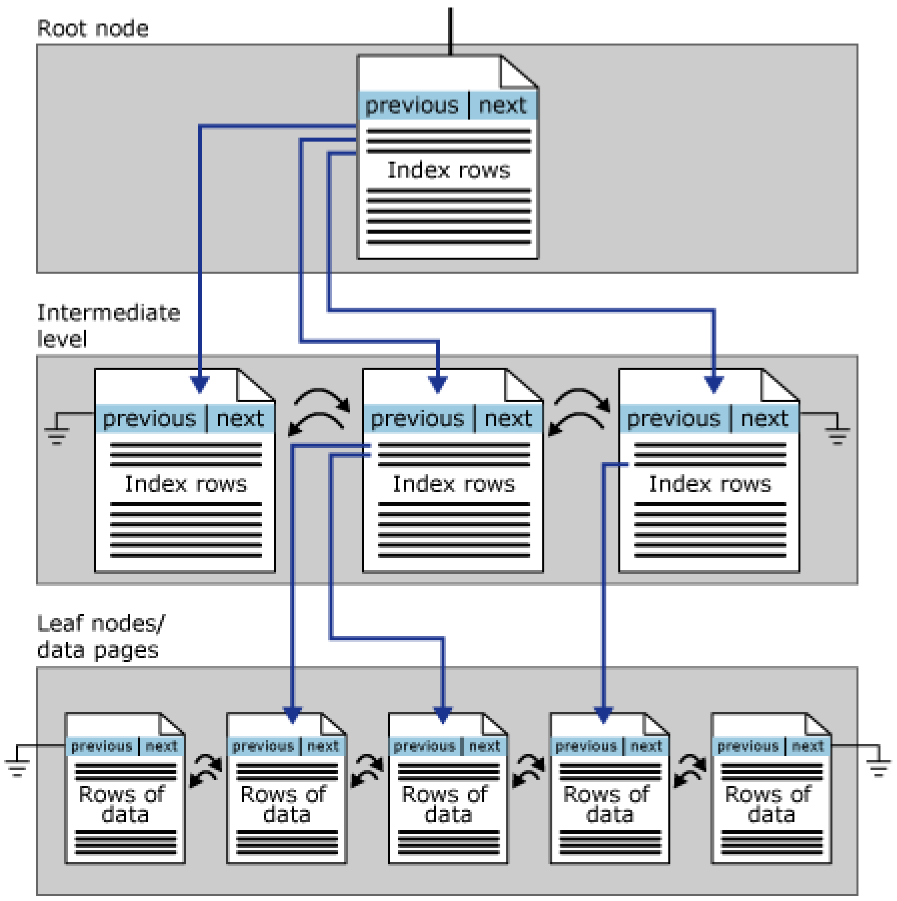
```
dbo.PersistedCommand
    Columns
        Id (PK, uniqueidentifier, not null)
        Timestamp (datetime, not null)
        Command (XML(.), not null)
        Result (XML(.), not null)
    Keys
        PK_PersistedCommand
    Constraints
    Triggers
    Indexes
        PK_PersistedCommand (Clustered)
    Statistics
```

First, we need to understand what an index is. To understand an index, we must know what a B-Tree is (http://en.wikipedia.org/wiki/B-tree). Then, we must understand what a B+Tree is (http://en.wikipedia.org/wiki/B+tree). “Normal” tables in SQL Server are B+Trees. There are two possible implementations of the leaf nodes in a B+Tree.

In the first implementation, the leaf nodes do not contain the data of the tree. The leaf node contains a pointer to the data. This data may be in the heap memory or in the HD. So when you transverse the tree searching a node you do not find the data, you find a pointer to the data. You will have to make another operation to find the data. This implementation is the non-clustered index in SQL Server 2008 R2. The image below illustrates this:



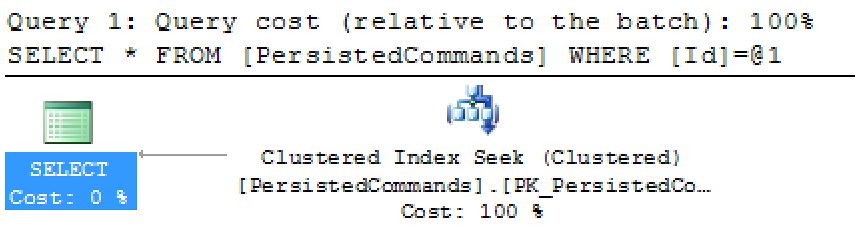
The second implementation maintains the data in the leaf node. So after finding the right node you have the data in hand. The image below illustrates this:



So now we can query our table.

The first query is to try to find a PersistedCommand knowing its Id. Remember that we have a Clustered Index on the Id column. This is the fastest possible query that we can make. Below are the query and the execution plan:

```sql
SELECT *
FROM PersistedCommands
WHERE Id = '13e5b25c-10af-4cc9-ab1a-02876a2eea75'
```



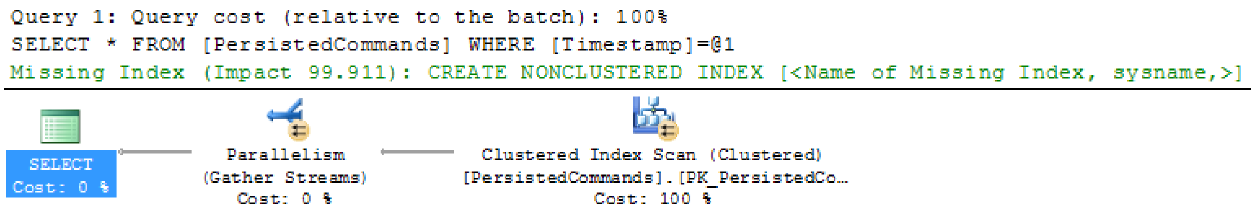
```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Id]=@1
```

SELECT
Cost: 0 %
Clustered Index Seek (Clustered)
[PersistedCommands].[PK_PersistedCo…
Cost: 100 %

We can see that 100% of the work of the query was on the “Clustered Index Seek”. The icon can help us here. We can see through it that SQL Server used the clustered index on Id column to find the right node on the leaf node. It is a clustered index, so the data was already on the leaf node. No more work is need to SQL Server return the data. This query ran immediately (0 seconds). The “Clustered Index Seek” is the fastest search because use the B+Tree index and find the data right in the leaf node.

Suppose that we do not want a specific command but one that happened in a specific date:

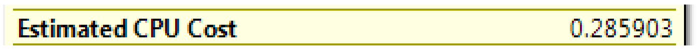
```sql
SELECT *
FROM PersistedCommands
WHERE Timestamp = '1927-07-22 00:00:00.000'
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Timestamp]=@1
Missing Index (Impact 99.911): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
```

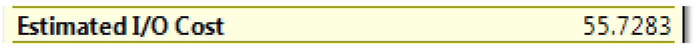
SELECT
Cost: 0 %
Parallelism
(Gather Streams)
Cost: 0 %
Clustered Index Scan (Clustered)
[PersistedCommands].[PK_PersistedCo…
Cost: 100 %

Ok. Now we have a problem, this query ran for 23 seconds. SQL Server had to make a “Clustered Index Scan” to find our commands. In a “Clustered Index Scan” you go through ALL leaf nodes filtering the rows. Comparing a date for equality is incredible fast. We can see this hovering the “Clustered Index Scan” icon and seeing the CPU Cost:



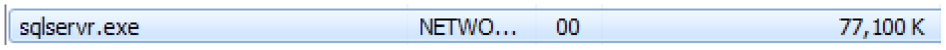
Estimated CPU Cost	0.285903

The problem here is that we gone through all the rows, so the real cost was the I/O cost. When the query ran, the major part of the rows was not in memory, so SQL Server had to gather the rows of the HD. We can see this in the “I/O Cost”:



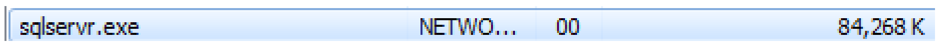
Estimated I/O Cost	55.7283

One can easily see the effect in the server of the two queries analyzing the “Memory (Private Working Set)” of the SQL Server. First the memory right after the server had been restarted:



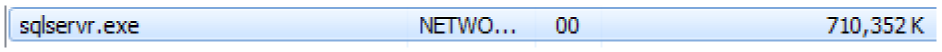
sqlservr.exe	NETWO…	00	77,100 K

Now, after the “Clustered Index Seek”:



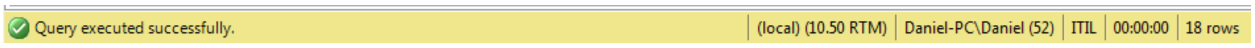
sqlservr.exe	NETWO…	00	84,268 K

Maybe SQL Server brought the entire index to memory. This table has +500.000 rows, so not a killer increase. Now let us see what happen in the “Clustered Index Scan”.



sqlservr.exe	NETWO…	00	710,352 K

Oh! One query and now we spent almost 700MB of the server. To be sure that 99% of the query time was reading the rows in the HD we can run the query again and see the query time:


Query executed successfully. | (local) (10.50 RTM) | Daniel-PC\Daniel (52) | ITIL | 00:00:00 | 18 rows

We were right. 99% of the time was reading the HD. The query now ran immediately because all the rows were already in the server memory and the CPU cost of the query is low.

So, what can we do in this case? Let us read the green text on the execution plan: “Missing Index”.

SQL Server is telling us that it had to make a “Clustered Index Scan” because it was missing an index. So let us create a non-clustered index on the timestamp column, restart the server and analyze it again.

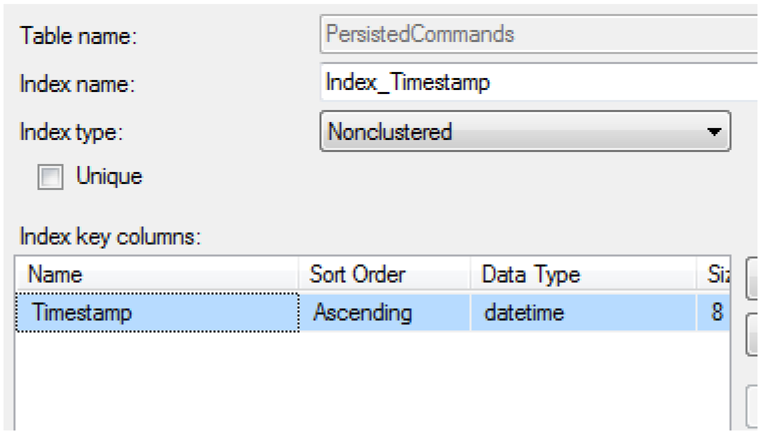


Table name: PersistedCommands
Index name: Index_Timestamp
Index type: Nonclustered
Unique

Index key columns:

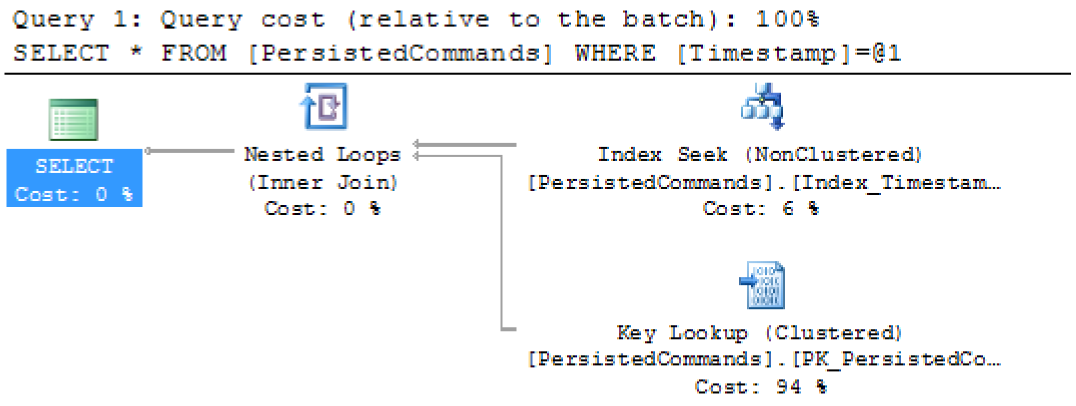
Name	Sort Order	Data Type	Si
Timestamp	Ascending	datetime	8

Ok. Index created; let us restart the SQL Server. OK, memory back to 80MB. Let us run the Timestamp-query again. Let us see the differences:

The first query ran in 23 seconds, this one ran in 0 seconds. Much Better!

00:00:00 | 18 rows

Although the query ran much faster, the execution plan is much more complex. Ok, let us analyze it.


```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Timestamp]=@1
```
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam...
Cost: 6 %
Key Lookup (Clustered)
[PersistedCommands].[PK_PersistedCo...
Cost: 94 %

We are searching rows using an index now, using our “Timestamp” index. We can see that the index was used because of the “Index Seek (Non-clustered)” operation. Perfect. But the “Index Seek” only took 6%

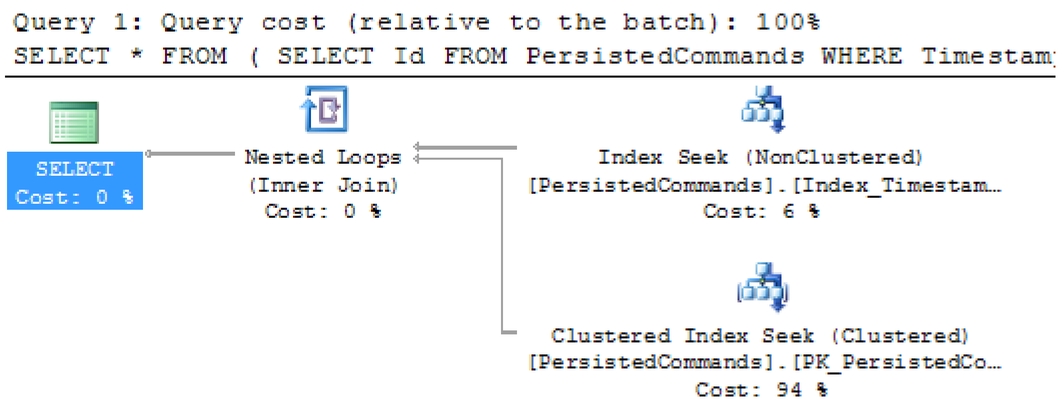
of time query time and now we have two strange operations: “Key Lookup (Clustered)” and the “Nested Loops (Inner Join)”.

Analyzing our query, we see that we do not have any “inner join”, so why SQL Server are doing an inner join here?

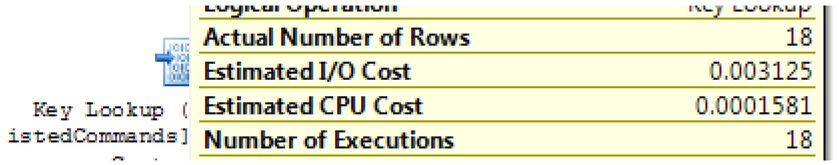
We are searching through the “Timestamp” index here. This index is a non-clustered index, so the leaf nodes do not contains the data; it contains pointers to the data, in this case the Id column.

So SQL Server is doing something like this:

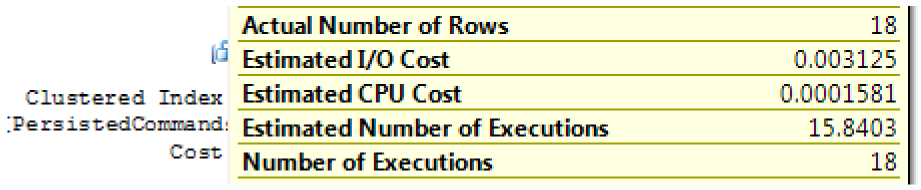
```
SELECT *
FROM
(
    SELECT Id
    FROM PersistedCommands
    WHERE Timestamp = '1927-07-22 00:00:00.000'
) as CLUSTEREDINDEXSEEK
INNER JOIN PersistedCommands on CLUSTEREDINDEXSEEK.Id = PersistedCommands.Id
```


```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM ( SELECT Id FROM PersistedCommands WHERE Timestam
```
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam…
Cost: 6 %
Clustered Index Seek (Clustered)
[PersistedCommands].[PK_PersistedCo…
Cost: 94 %

The execution plan of this query is not exactly equal of the original query, but we can see that even the percentages of the operations are equal. There are other things equal in this query:



Logical Operation	Key Lookup
Actual Number of Rows	18
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	18



Actual Number of Rows	18
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	15.8403
Number of Executions	18

We can see that both, the “Key Lookup (Clustered)” of the original query and the “Clustered Key Index Seek” of the modified query are equal on the following attributes: “Actual Number of Rows”, “Estimated I/O Cost”, “Estimated CPU Cost” and “Number of Executions”.

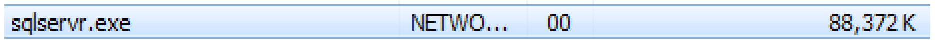
The last attribute is the most important to me. It seems that the “Nested Loops” are traversing the B+Tree of the clustered index on the Id column 18 times. That happens because we have 18 commands with Timestamp column equals to “1927-07-22 00:00:00.000”.

Perfect, it seems that we now understand what SQL Server is doing. The question now is “why he is doing this?”

The use of the non-clustered index on the “Timestamp” column seems fine to me. We have to remember that a non-clustered index does not have the data on the leaf node; in this case the leaf node contains 18 rows containing the “Id” value of each command. To gather the data of the command SQL Server have to make 18 seeks in the clustered index and it does this seeks making a nested loop in each row returned.

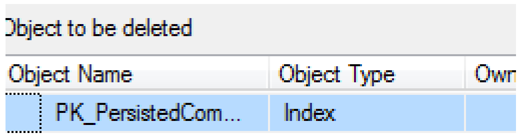
So internally it does not make an inner join, it is the case that the “inner join” also uses the “nested loop” operation. The searches of a non-clustered seek uses the “Key Lookup” to return the data and the inner join uses a “Clustered Index Seek”.

Both “Key Lookup” and “Clustered Index Seek” are much optimized seeks so we can that the memory on the server is still low:



sqlservr.exe	NETWO...	00	88,372 K

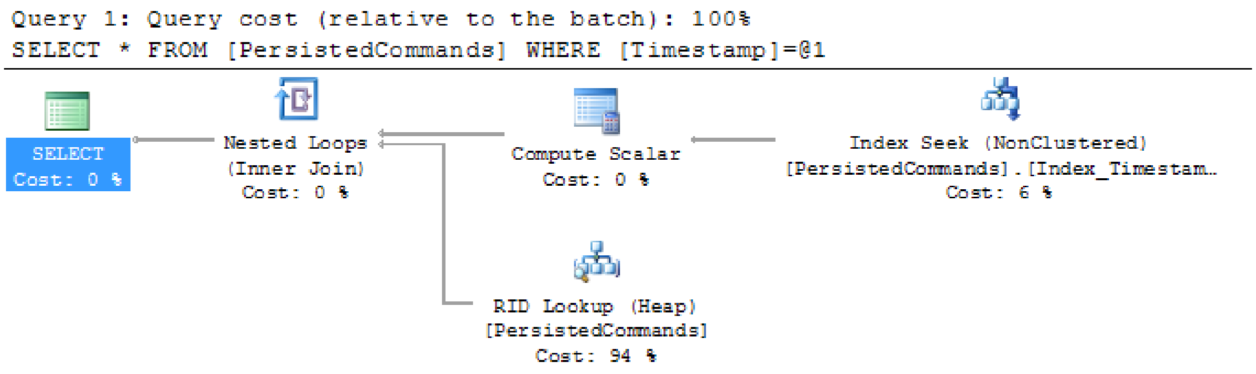
But, what would happen if we did not had an index on the “Id” column. So let us delete that index.



Object to be deleted

Object Name	Object Type	Own
PK_PersistedCom...	Index	

Let us run the query and analyze the execution plan.



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Timestamp]=@1
```

Ok. Now, SQL Server is just cheating. He made the non-clustered index seek just like before. But now he has a set with the commands Ids but does not have any index on Ids. The cheat here is, although he does not have the Ids he has a map that maps the Ids to the internal row ids. So he maps the commands Ids to its internal row ids and use this set in the "Nested Loops". The nested loops will not make a "Key Lookup" because it does not have an index on the primary key, nor will do a clustered index seek because it does not have a clustered index. The loop will use the internal infrastructure of the SQL Server to find the rows directly by its internal row ID using the "RID Lookup". So now SQL Server it is still doing a Lookup which is fast and does not bring everything to memory.

Before creating our clustered index on "Id" column again, let us see what happen if we create a "Non-clustered Index" on the "Id" column.

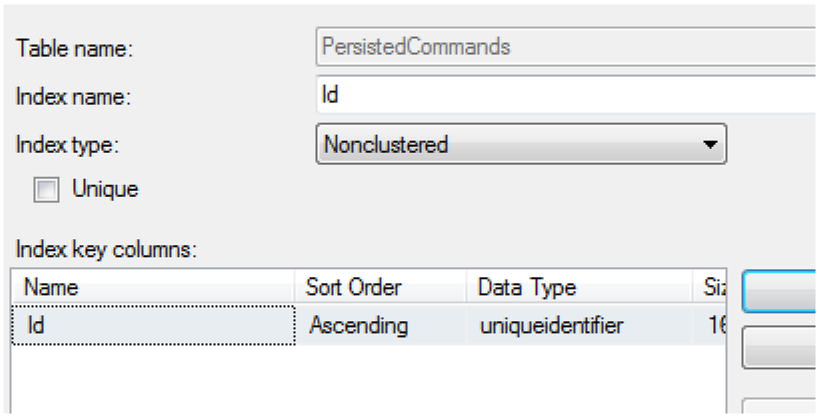


Table name: PersistedCommands

Index name: Id

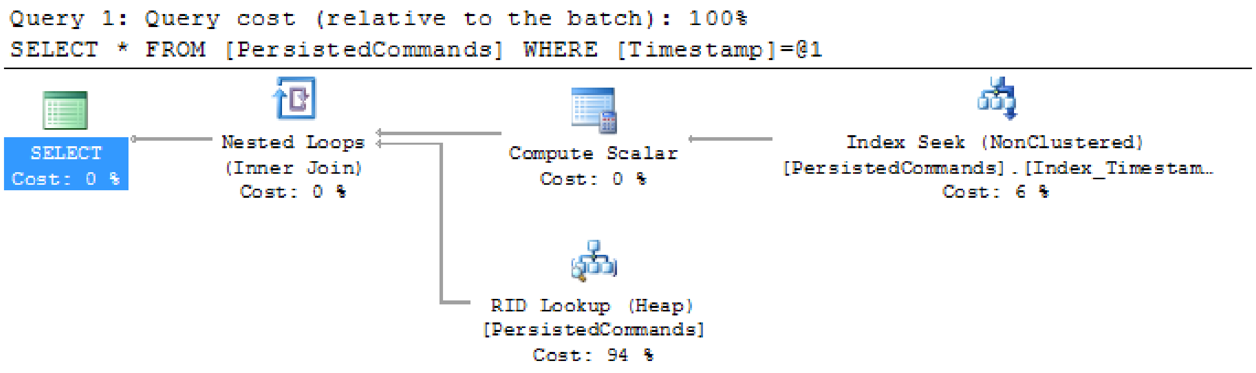
Index type: Nonclustered

Unique

Index key columns:

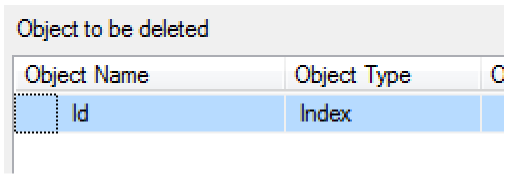
Name	Sort Order	Data Type	Si
Id	Ascending	uniqueidentifier	16

The interesting part is that this does not change anything!



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Timestamp]=@1
```

Deleting this index and creating our "Clustered Index" on "Id" column:



Object to be deleted

Object Name	Object Type	C
Id	Index	

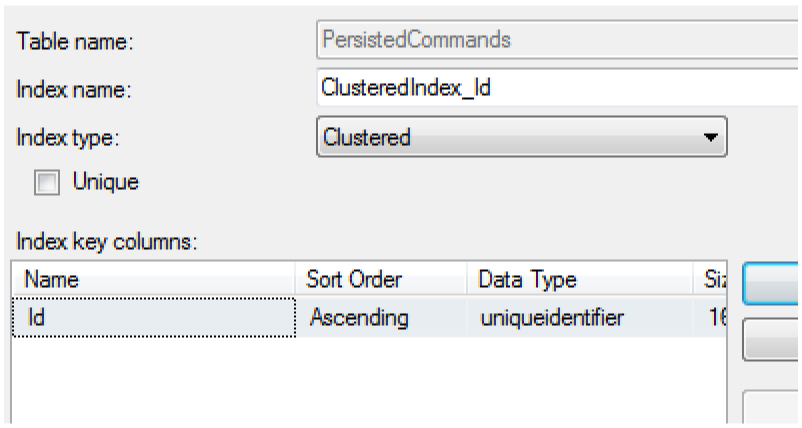
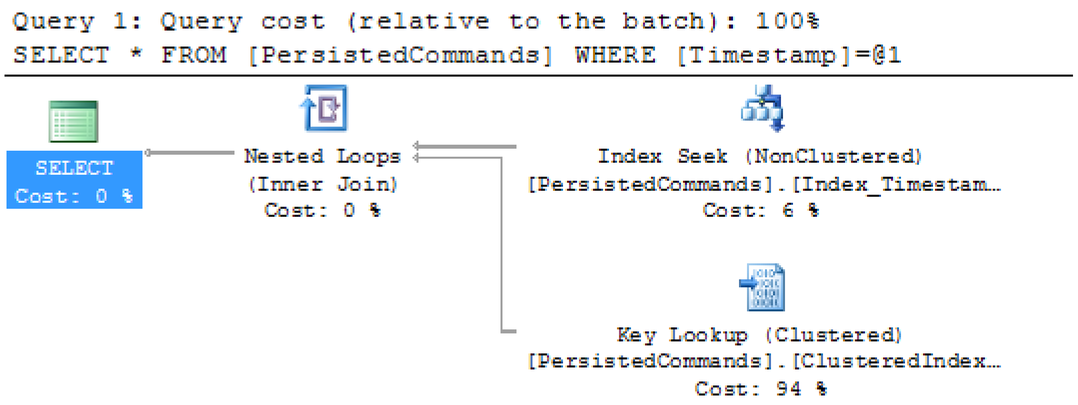


Table name: PersistedCommands
Index name: ClusteredIndex_Id
Index type: Clustered
Unique
Index key columns:

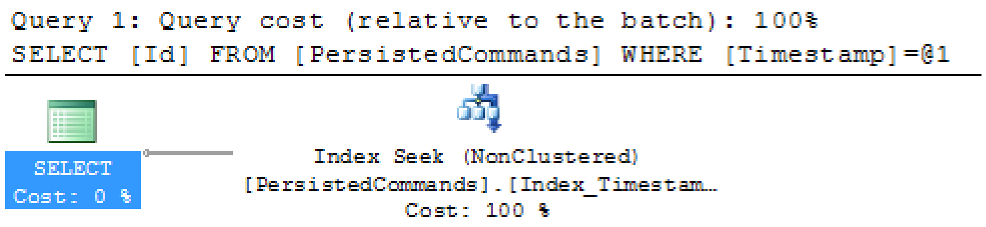
Name	Sort Order	Data Type	Si
Id	Ascending	uniqueidentifier	16

And running the query again:


```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [PersistedCommands] WHERE [Timestamp]=@1
```
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam...
Cost: 6 %
Key Lookup (Clustered)
[PersistedCommands].[ClusteredIndex...
Cost: 94 %

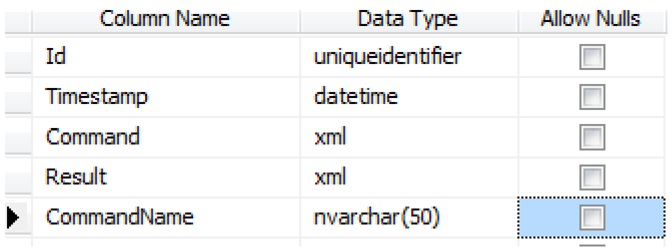
Ok, back where we were. But let us advance. Suppose that we do not want every column of the table. Suppose that for some reason we just want the Ids of the rows. Let us run the query.

```
SELECT Id
FROM PersistedCommands
WHERE Timestamp = '1927-07-22 00:00:00.000'
```


```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id] FROM [PersistedCommands] WHERE [Timestamp]=@1
```
SELECT
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam...
Cost: 100 %

Hum, the "Nested Loop" and the 18 "Key Lookup" are gone! Remember that in the leaf pages of the non-clustered index we do not have the row data, but we do have the Id. So, when we asked for the whole row data, SQL Server had to make 18 seeks to find the data. Now, the non-clustered index gave us all the information we asked, so no seeks are need. So SQL Server is smart enough to see if all data that the SELECT wants exists on the index, this is called "Covering Index", because the index itself cover all data needs.

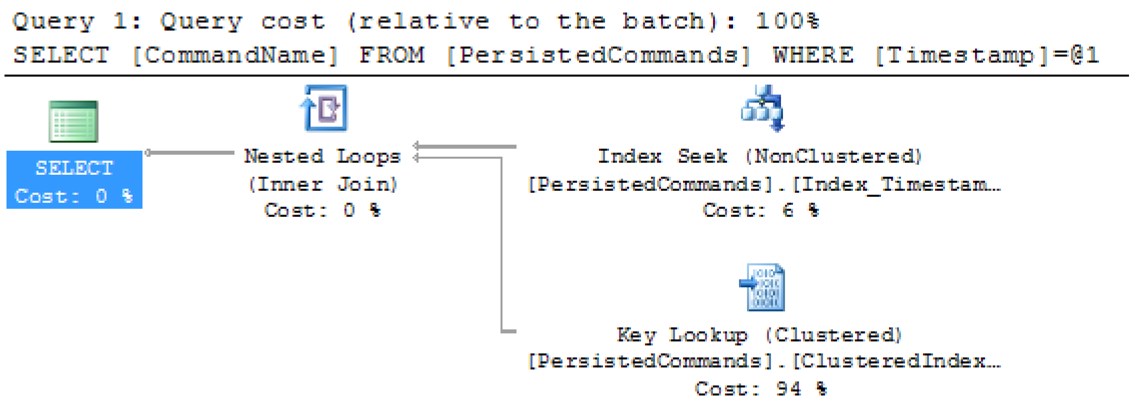
To make this example more interesting, let us create another column:



Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
Timestamp	datetime	☐
Command	xml	☐
Result	xml	☐
CommandName	nvarchar(50)	☐

Now let us analyze the following query:

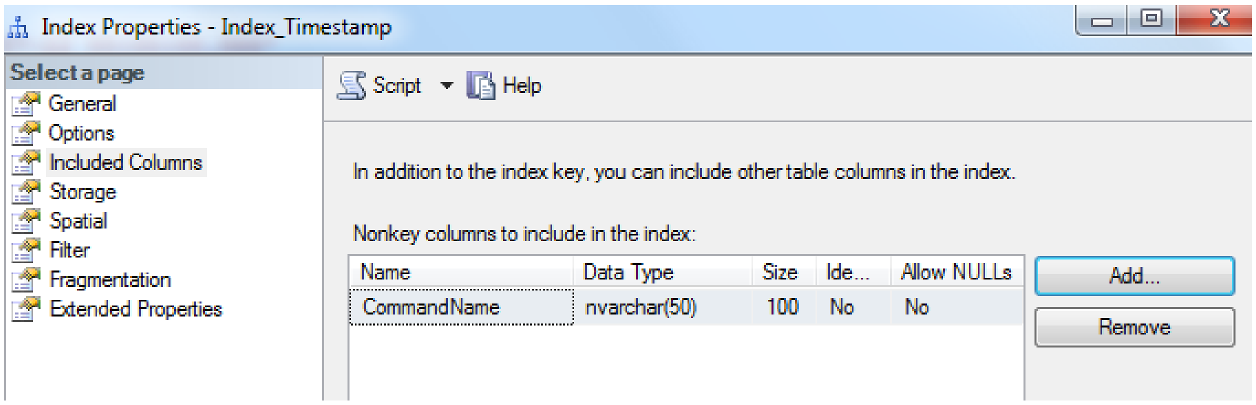
```
SELECT CommandName
FROM PersistedCommands
WHERE Timestamp = '1927-07-22 00:00:00.000'
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT [CommandName] FROM [PersistedCommands] WHERE [Timestamp]=@1
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam...
Cost: 6 %
Key Lookup (Clustered)
[PersistedCommands].[ClusteredIndex...
Cost: 94 %

OK. We area back! We need information that does not exist in the non-clustered index. But now we are going to use "Included Columns" option to kill the 18 key lookups. We just need to go to the properties of the "Index_Timestamp" in the "Include Column" tab and insert the "CommandName" column. With this we are telling SQL Server that we want that it saves the CommandName value in the leaf node of the non-clustered index of the "Timestamp" column.


Index Properties - Index_Timestamp

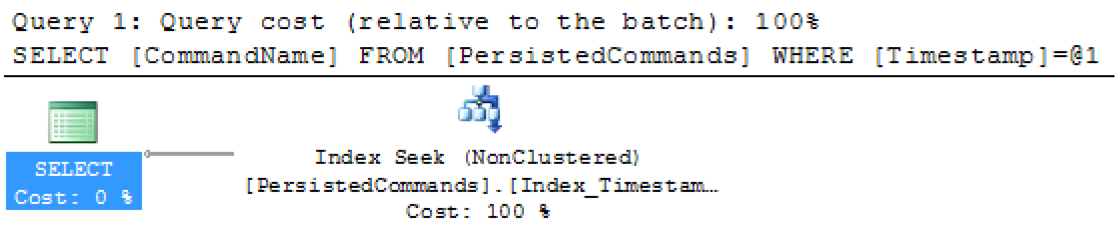
In addition to the index key, you can include other table columns in the index.

Nonkey columns to include in the index:

Name	Data Type	Size	Ide...	Allow NULLs
CommandName	nvarchar(50)	100	No	No

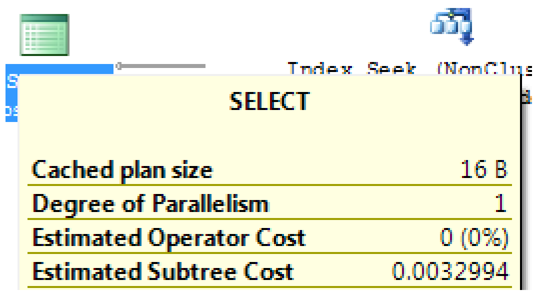
Now, every leaf node in this index will have the Id and the CommandName value without any work. Let us analyze our query now.

```
SELECT CommandName
FROM PersistedCommands
WHERE Timestamp = '1927-07-22 00:00:00.000'
```


Query 1: Query cost (relative to the batch): 100%
SELECT [CommandName] FROM [PersistedCommands] WHERE [Timestamp]=@1

SELECT
Cost: 0 %
Index Seek (NonClustered)
[PersistedCommands].[Index_Timestam…
Cost: 100 %

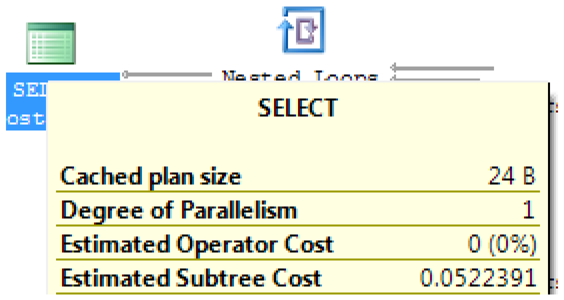
Perfect! We just killed all the lookup operations and optimized the query. The estimated cost of the whole query is 0.0032994:



SELECT

Cached plan size	16 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0032994

Without the “Included Columns” the estimated cost of the whole query is 0.0522391:



SELECT

Cached plan size	24 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0522391

Something like 16 times faster using the "Included Columns"! Remember that the "Key Lookup" was 96% of the query. With this optimization we removed 96% of the work. But we have to remember that an index with included columns is more expensive to build and update than a simple index.

Read more:

http://msdn.microsoft.com/en-us/library/ms177443.aspx

http://msdn.microsoft.com/en-us/library/ms177484.aspx

http://technet.microsoft.com/en-us/library/ms190400.aspx

http://technet.microsoft.com/en-us/library/ms175184.aspx

http://technet.microsoft.com/en-us/library/ms190376.aspx

http://technet.microsoft.com/en-us/library/ms178038.aspx

http://technet.microsoft.com/en-us/library/bb326635.aspx

http://technet.microsoft.com/en-us/library/ms187871.aspx

http://technet.microsoft.com/en-us/library/ms190696.aspx

http://technet.microsoft.com/en-us/library/ms178082.aspx