SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 1

January 4, 2013 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/" \o "Posts by pinaldave)

Index never stops amazing me, there are so much to learn about Index that I never feel that there is enough knowledge out about this subject. If you are interested you can watch my [**Indexing Course on Pluralsight**](http://bit.ly/sql-index1) for further learning on this subject. On my production server I personally use [**embarcadero DB Optimizer**](http://bit.ly/emb-apr-13)for all performance tuning and routine health check up. I will be interested to know what is your feedback about the product.

Instead of going on the theory overload – let us start with this blog post as a conversation between two individuals – Jon and Mike. These are just random names. Jon is senior and experienced SQL Server Expert and Mike is beginner with SQL Server.

Mike – What is Redundant Index?

Jon – Indexes are redundant when they have similar columns as a part of a definition. Additionally, the indexes are considered redundant when their first few columns are in the same position with same order by direction are also considered as a redundant.

Mike – Would you please explain it with examples?

Jon – Sure, Let us assume we have two indexes:

**Index 1:** Col1, Col2, Col3  
**Index 2:** Col1, Col2, Col3

Now if you look at them – they have the same columns as a part of their definition, so they are indeed redundant indexes. However, look at the following scenario:

**Index 3:** Col1, Col2  
**Index 4:** Col1, Col2, Col3

In this case they are also considered as a redundant because the position of the Col1, Col2 are same in both of the index. It is quite commonly considered that if the initial positions of the columns are the same, they are redundant.

However, there is one more concept here to be looked at as well before we make certain about their redundancy. Look at the following indexes:

**Index 5:** Col1 ASC, Col2 DESC  
**Index 6:** Col1 DESC, Col2 DESC, Col3 ASC

In this case if the initial positions are the same, they are not redundant as the order of the column is not the same.

There are lot more to discuss but this is just to give you an initial idea. There is one more concept we should consider before calling any index redundant is Included Index. Here is the simple scenario for it.

**Index 7:** Col1 ASC Included (col2)  
**Index 8:** Col1 ASC Included (col3)

You can notice they have same initial column but the Included columns are totally different.

Mike – Thanks, I got it. It seems that Redundant Indexes are not good and they should be dropped correct.

In case of Index 1 and Index 2 I think we should drop either of the one.

In case of Index 3 and Index 4 I believe Index 4 has more columns and covering, so we should keep it and drop the other one.

In case of Index 5 and Index 6, they are both different indexes so we should keep both.

In case of Index 7 and Index 8, again they are both different index in this case. They can be redundant if the included columns are overlapping to each other.

Am I correct to say this?

Jon – Very good analysis. You are very close to the understanding. Generally, redundant indexes are not good and they should be absolutely addressed. **In most cases, redundant cases should be dropped.**

Mike – Ahha, so in the most cases indexes should be dropped. Ok, so is there any script or guidance to detect redundant indexes for the most cases.

Jon – Sure, [**here is the script**](http://www.pinaldave.com/sql-downloads/script-downloads/sql-server-missing-index-unused-index-and-duplicate-index-script-2/) which does that – however, this query just addresses the scenario of the Index 3 and Index 4. It does not talk about Included Columns or Index Order (ASC or DESC). Just use that for a start but do your analysis on this subject before you drop your indexes. You still have to check for order of the index and included columns as well.

Mike – Perfect, I understand that the script is just for a quick start and not the complete solution. Now you mentioned “In Most Cases” – what are the special cases. What are the cases when an Index which absolutely qualify for the redundant index but should not be dropped. Would you please explain the special cases?

Jon – Absolutely – there are always special cases. For example the width of the column matters.

Mike – Okey I would love to learn more about this – would you please explain.

Jon – Absolutely – I have a working example of it

SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 2

January 7, 2013 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/" \o "Posts by pinaldave)

Before you start reading this blog post, I strongly suggest you to read the [part 1 of this series](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/). It talks about [**What is Redundant Index**](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/). The story is a conversation between two individuals – Jon and Mike. They are different but have single goal learn and explore SQL Server. Their initial conversation sets the ground for this blog post. They earlier discussed what is a Redundant Index as well, discussed what are the special cases for the same. It is a general assumption (or common best practices) is to drop Redundant Indexes. Later Mike asks for special case where even though the index is clearly a Redundant Index, why it should not be removed. Jon promises to explain with a demo where a Redundant Index is useful and should not be dropped. Here is their conversation continued from earlier. On my production server I personally use [**embarcadero DB Optimizer**](http://bit.ly/emb-apr-13)for all performance tuning and routine health check up. I will be interested to know what is your feedback about the product.

**Mike** – Today is Monday. You promised me a demo today where a Redundant Index is useful Index.

**Jon**- Absolutely. We will create two tables and will create absolutely same table. We will notice that on the first table Redundant Index will be useless and needs to be dropped whereas on the second table Redundant Index will useful and should not be dropped for performance.

Let us start with a demo.

Let us create two tables. One with all the column INT and second with a wider column CHAR (800).

USE tempdb  
GO  
-- Create table  
CREATE TABLE SampleTable1 (ID INT, Col1 INT, Col2 INT, Col3 INT)  
GO  
CREATE TABLE SampleTable2 (ID INT, Col1 INT, Col2 INT, Col3 CHAR(800))  
GO

Now let us create indexes on both the tables. We will make sure that created indexes are same on both the table.

Clustered Indexes are just created for reference – the results of this test will not be affected by its presence or absence of it. I have just created to rule out few doubts I anticipate by their absence. The important part is non clustered indexes. One of the non-clustered index is created on Col1 & Col2 and another one is created on Col1, Col2, and Col3.

**Table1: SampleTable1**  
Index on SampleTable1: IX\_ST\_Col1\_Col2  
Index on SampleTable1: IX\_ST\_Col1\_Col2\_Col3

**Table2: SampleTable2**  
Index on SampleTable2: IX\_ST\_Col1\_Col2  
Index on SampleTable2: IX\_ST\_Col1\_Col2\_Col3

-- Create Indexes on Sample Table1  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable1 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable1 (Col1, Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable1 (Col1, Col2, Col3)  
GO  
-- Create Indexes on Sample Table2  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable2 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable2 (Col1, Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable2 (Col1, Col2, Col3)  
GO

Now let us populate both the tables. Both the tables have absolutely same data.

-- Populate tables  
INSERT INTO SampleTable1 (ID, Col1, Col2, Col3)  
SELECT RAND()\*10000, RAND()\*1000, RAND()\*100, RAND()\*10  
GO 100000  
INSERT INTO SampleTable2 (ID, Col1, Col2, Col3)  
SELECT \*  
FROM SampleTable1  
GO

Now is the most interesting part. For this[**first enable the execution plan in SSMS**](http://blog.sqlauthority.com/2007/08/28/sql-server-actual-execution-plan-vs-estimated-execution-plan/) (shortcut key – CTRL + M).

We will be doing two tests. Let describe our first test.

**Test 1: Select a smaller set of the data**

In this test we will run a same query on both the tables. Both the times we will apply our path of the index on the each table. As there are 2 tables and each have 2 indexes we will have a total of 4 indexes.

**Table1: SampleTable1**  
Index on SampleTable1: IX\_ST\_Col1\_Col2  
Index on SampleTable1: IX\_ST\_Col1\_Col2\_Col3

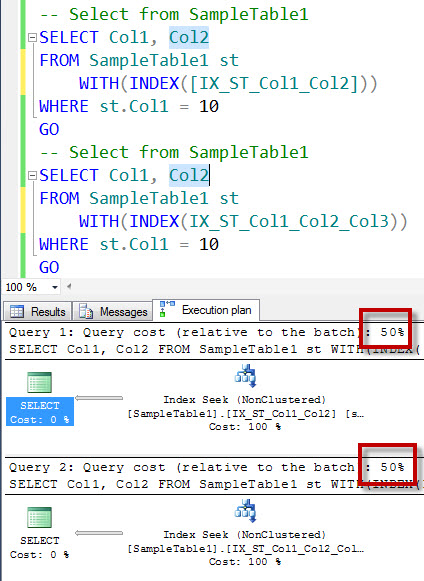
**Table2: SampleTable2**  
Index on SampleTable2: IX\_ST\_Col1\_Col2  
Index on SampleTable2: IX\_ST\_Col1\_Col2\_Col3

Now let us run following script with keeping the Actual Execution Plan on (shortcut key CTRM: + M)

Let us first run two scripts for SampleTable1

-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st WITH(INDEX([IX\_ST\_Col1\_Col2]))  
WHERE st.Col1 = 10  
GO  
-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st WITH(INDEX(IX\_ST\_Col1\_Col2\_Col3))  
WHERE st.Col1 = 10  
GO

Let us check the execution plan:



You can notice from the execution plan that in the case of the SampleTable1 it does not matter if we use either of the index the performance of the both the query is same. Both the queries are using the same amount of the resources. In this case, Col3 is an integer and for SQL Server the width of the column does not make much difference. I can clearly say in this case Indexes are redundant as they are giving the same performance.

(**Note:** If you are going to ask me to change the SELECT statement to also include Col3, it will become totally different scenario as it will require to do a key lookup for IX\_ST\_Col1\_Col2. If your SELECT statement has Col1, Col2, Col3 – the optimal index here is IX\_ST\_Col1\_Col2\_Col3, there is no further discussion in that case).

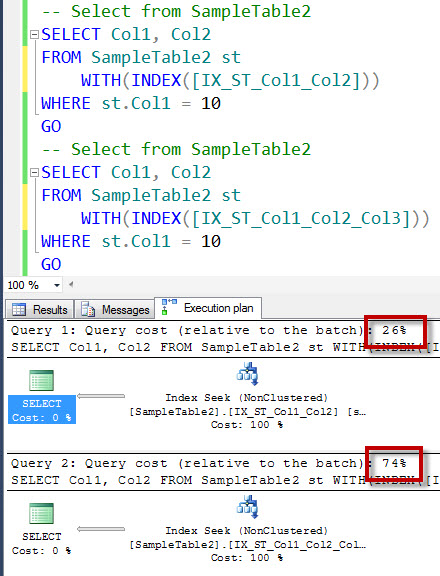
(**A Quick Tip:** When we compare execution plan – the higher cost compared to the batch explains higher usage of the resources and expensive query.)

Now looking at both the indexes indexes IX\_ST\_Col1\_Col2 is subset of IX\_ST\_Col1\_Col2\_Col3 and as mentioned in an earlier note if there is an additional column (col3) is in the SELECT statement that index will be more suitable. We can easily remove IX\_ST\_Col1\_Col2 index in this particular special case (note this does not apply all the time).

Now let us run similar script for SampleTable2

-- Select from SampleTable2  
SELECT Col1, Col2  
FROM SampleTable2 st WITH(INDEX([IX\_ST\_Col1\_Col2]))  
WHERE st.Col1 = 10  
GO  
-- Select from SampleTable2  
SELECT Col1, Col2  
FROM SampleTable2 st WITH(INDEX([IX\_ST\_Col1\_Col2\_Col3]))  
WHERE st.Col1 = 10  
GO

Let us check the execution plan:



You can notice from the execution plan that in the case of the SampleTable2 it matters a lot about which index is used for the query as the performance differences between those queries is huge.  One of the query is using very little resources and another one is taking a huge amount of the resources. In this case, Col3 is a CHAR (800) datatype which is fixed length string datatype. In this case, for SQL Server the width of the column does make a big difference. We can clearly say that here in our SELECT statement IX\_ST\_Col1\_Col2 is the most optimal indexes.

If that is the case, what is the use of IX\_ST\_Col1\_Col2\_Col3. Well, the answer of this question is also interesting. If you change the SELECT statement to also include Col3, it will become totally different scenario as it will require to do a key lookup for IX\_ST\_Col1\_Col2. If your SELECT statement has Col1, Col2, Col3 – the optimal index here is IX\_ST\_Col1\_Col2\_Col3. The need of the both the indexes is different and they achieve a specific task. If you think IX\_ST\_Col1\_Col2 is redundant as a more inclusive index IX\_ST\_Col1\_Col2\_Col3 exists, it will be not the optimal thinking. Even though, IX\_ST\_Col1\_Col2\_Col3 includes all the columns, when SQL Server only needs Col1 and Col2 it finds IX\_ST\_Col1\_Col2 more suitable for performance.

I hope this is now clear to you that how to identify if the redundant index is useful or useless now.

**Mike**- Thanks Jon, this is a great explanation. Let me quickly **summarize** it.

Even though Indexes look redundant there may be some queries which may find them useful. This usually happens when the data type of the of any column is much wider than other columns. Before dropping the indexes one should properly validate the usage patterns of the indexes and query workloads. One should properly test everything before taking any actions.

**Jon** – Good summary. Test before you Act! However, you should notice that this is not the only case when redundant indexes are useful. There may be other cases too!

**Mike** – I understand. However, before you continue further, I see that you called this as a Test 1. Is there any test 2 with the larger dataset? Does it also validate my earlier summary.

**Jon** – Another good question – Let us see the Test 2 in Friday’s blog post. You can clean up your database by dropping your test tables.

-- Clean Up  
DROP TABLE SampleTable1  
GO  
DROP TABLE SampleTable2  
GO

« [SQLAuthority News – Speaking at Southeast Asia SharePoint Conference 2013 – Singapore](http://blog.sqlauthority.com/2013/01/10/sqlauthority-news-speaking-at-southeast-asia-sharepoint-conference-2013-singapore-2/)

[SQL SERVER – Weekly Series – Memory Lane – #011](http://blog.sqlauthority.com/2013/01/12/sql-server-weekly-series-memory-lane-011/) »

SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 3

January 11, 2013 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/" \o "Posts by pinaldave)

Before you start reading this blog post, I strongly suggest you to read the [part 1 of this series](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/).It talks about [What is Redundant Index](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/). The story is a conversation between two individuals – Jon and Mike. They are different but have single goal learn and explore SQL Server. Their initial conversation sets the ground for this blog post. They earlier discussed what is a Redundant Index as well, discussed what are the special cases for the same. It is a general assumption (or common best practices) is to drop Redundant Indexes. On my production server I personally use [embarcadero DB Optimizer](http://bit.ly/emb-apr-13) for all performance tuning and routine health check up. I will be interested to know what is your feedback about the product.

Later Mike asks for special case where even though the index is clearly a Redundant Index, why it should not be removed. Jon promises to explain with a demo where a Redundant Index is useful and should not be dropped. In [part 2 of this series](http://blog.sqlauthority.com/2013/01/07/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-2/) – we talked about in detail about [Kind of Redundant Indexes are Useful](http://blog.sqlauthority.com/2013/01/07/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-2/). We demonstrated that a column width makes a big difference in the performance of the index with a query. The query used in the demonstration was returning a very small result set.

Mike had a question for Jon that if the query is returning a larger result set will it still have the same conclusion which they had arrived in an earlier blog post. Let us continue the conversation here.

Mike – Today is Friday. You promised me a demo today where a Redundant Index is useful Index, even though the query were returning a larger result set.

Jon - Absolutely. Our demo is absolutely same as the earlier demo – the only difference today we will see is that our where condition will be different from the earlier query. This time we will have a much larger resultset returning by our query. We will observe the output and come to the conclusion if our finding is still the same as we [discovered earlier](http://blog.sqlauthority.com/2013/01/07/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-2/).

We will create two tables and will create absolutely same table. We will notice that on the first table Redundant Index will be useless and needs to be dropped whereas on the second table Redundant Index will useful and should not be dropped for performance. One with all the column INT and second with a wider column CHAR (800).

USE tempdb  
GO  
-- Create table  
CREATE TABLE SampleTable1 (ID INT, Col1 INT, Col2 INT, Col3 INT)  
GO  
CREATE TABLE SampleTable2 (ID INT, Col1 INT, Col2 INT, Col3 CHAR(800))  
GO

Now let us create indexes on both the tables. We will make sure that created indexes are same on both the table.

Clustered Indexes are just created for reference – the results of this test will not be affected by its presence or absence of it. I have just created to rule out few doubts I anticipate by their absence. The important part is non clustered indexes. One of the non-clustered index is created on Col1 & Col2 and another one is created on Col1, Col2, and Col3.

Table1: SampleTable1   
Index on SampleTable1: IX\_ST\_Col1\_Col2  
Index on SampleTable1: IX\_ST\_Col1\_Col2\_Col3

Table2: SampleTable2  
Index on SampleTable2: IX\_ST\_Col1\_Col2  
Index on SampleTable2: IX\_ST\_Col1\_Col2\_Col3

-- Create Indexes on Sample Table1  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable1 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable1 (Col1, Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable1 (Col1, Col2, Col3)  
GO  
-- Create Indexes on Sample Table2  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable2 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable2 (Col1, Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable2 (Col1, Col2, Col3)  
GO

Now let us populate both the tables. Both the tables have absolutely same data.

-- Populate tables  
INSERT INTO SampleTable1 (ID, Col1, Col2, Col3)  
SELECT RAND()\*10000, RAND()\*1000, RAND()\*100, RAND()\*10  
GO 100000  
INSERT INTO SampleTable2 (ID, Col1, Col2, Col3)  
SELECT \*  
FROM SampleTable1  
GO

Now is the most interesting part. For this[first enable the execution plan in SSMS](http://blog.sqlauthority.com/2007/08/28/sql-server-actual-execution-plan-vs-estimated-execution-plan/) (shortcut key – CTRL + M).

We will be doing two tests. Let describe our first test.

Test 2: Select a larger set of the data

In this test we will run a same query on both the tables. Both the times we will apply our path of the index on the each table. As there are 2 tables and each have 2 indexes we will have a total of 4 indexes.

Table1: SampleTable1   
Index on SampleTable1: IX\_ST\_Col1\_Col2  
Index on SampleTable1: IX\_ST\_Col1\_Col2\_Col3

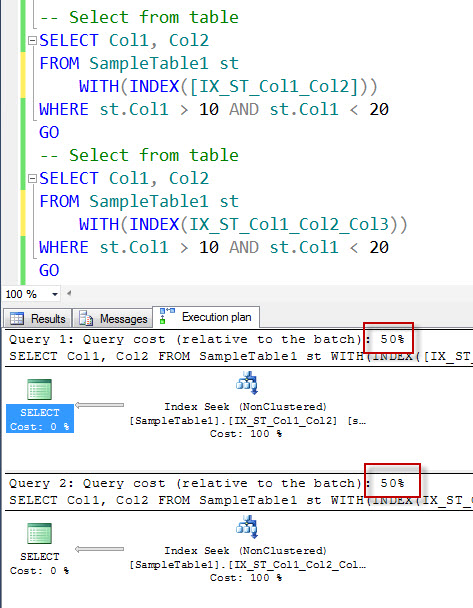
Table2: SampleTable2  
Index on SampleTable2: IX\_ST\_Col1\_Col2  
Index on SampleTable2: IX\_ST\_Col1\_Col2\_Col3

Now let us run following script with keeping the Actual Execution Plan on (shortcut key CTRM: + M)

Let us first run two scripts for SampleTable1

-- Select from table  
SELECT Col1, Col2  
FROM SampleTable1 st WITH(INDEX([IX\_ST\_Col1\_Col2]))  
WHERE st.Col1 > 10 AND st.Col1 < 20  
GO  
-- Select from table  
SELECT Col1, Col2  
FROM SampleTable1 st WITH(INDEX(IX\_ST\_Col1\_Col2\_Col3))  
WHERE st.Col1 > 10 AND st.Col1 < 20  
GO

Let us check the execution plan:



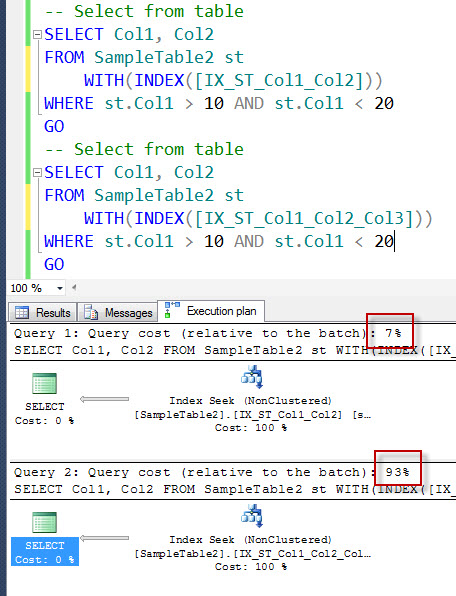
You can notice from the execution plan that in the case of the SampleTable1 it does not matter if we use either of the index the performance of the both the query is same. Both the queries are using the same amount of the resources. In this case, Col3 is an integer and for SQL Server the width of the column does not make much difference. I can clearly say in this case Indexes are redundant as they are giving the same performance.

Now looking at both the indexes indexes IX\_ST\_Col1\_Col2 is subset of IX\_ST\_Col1\_Col2\_Col3 and as mentioned in an earlier note if there is an additional column (col3) is in the SELECT statement that index will be more suitable. We can easily remove IX\_ST\_Col1\_Col2 index in this particular special case (note this does not apply all the time).

Now let us run similar script for SampleTable2

-- Select from table  
SELECT Col1, Col2  
FROM SampleTable2 st WITH(INDEX([IX\_ST\_Col1\_Col2]))  
WHERE st.Col1 > 10 AND st.Col1 < 20  
GO  
-- Select from table  
SELECT Col1, Col2  
FROM SampleTable2 st WITH(INDEX([IX\_ST\_Col1\_Col2\_Col3]))  
WHERE st.Col1 > 10 AND st.Col1 < 20  
GO

Let us check the execution plan:



You can notice from the execution plan that in the case of the SampleTable2 it matters a lot about which index is used for the query as the performance differences between those queries is huge.  One of the query is using very little resources and another one is taking a huge amount of the resources. In this case, Col3 is a CHAR (800) datatype which is fixed length string datatype. In this case, for SQL Server the width of the column does make a big difference. We can clearly say that here in our SELECT statement IX\_ST\_Col1\_Col2 is the most optimal indexes.

If that is the case, what is the use of IX\_ST\_Col1\_Col2\_Col3. Well, the answer of this question is also interesting. If you change the SELECT statement to also include Col3, it will become totally different scenario as it will require to do a key lookup for IX\_ST\_Col1\_Col2. If your SELECT statement has Col1, Col2, Col3 – the optimal index here is IX\_ST\_Col1\_Col2\_Col3. The need of the both the indexes is different and they achieve a specific task. If you think IX\_ST\_Col1\_Col2 is redundant as a more inclusive index IX\_ST\_Col1\_Col2\_Col3 exists, it will be not the optimal thinking. Even though, IX\_ST\_Col1\_Col2\_Col3 includes all the columns, when SQL Server only needs Col1 and Col2 it finds IX\_ST\_Col1\_Col2 more suitable for performance.

I hope this is now clear to you that how to identify if the redundant index is useful or useless now.

Even this second test, proved our earlier findings correct that it does not matter if resultset had more rows or less, the Redundent Index can be helpful and before dropping them we should evaluate them properly.

One more thing to notice is that when we ran the query for SampleTable2, the resource utilization is even larger than what we have seen in the earlier tests done when we had smaller resultset. With the larger resultset in this example the redudent index which has less number of column have greater (optimal) performance.

Mike - Ahha – I guess in this case the summary should stay the same. Let me quickly summarize it.

*Even though Indexes look redundant there may be some queries which may find them useful. This usually happens when the data type of the of any column is much wider than other columns. Before dropping the indexes one should properly validate the usage patterns of the indexes and query workloads. One should properly test everything before taking any actions.*

Jon – Good summary. It is always a good idea to test various combinations before making a decision. One should not drop an index because it was marked as a Redundant Index.

Mike – This is interesting! Jon. Do you have any more goodies related to this subject.

Jon – I do have many more goodies. However, let us do a quick clean up.

-- Clean Up  
DROP TABLE SampleTable1  
GO  
DROP TABLE SampleTable2  
GO

SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 4

January 17, 2013 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/" \o "Posts by pinaldave)

This is the fourth part of the series regarding Redundant Indexes. If have not read earlier part – there is quite a good chance that you will miss the context of this part. I quickly suggest you to read earlier three parts. On my production server I personally use [**embarcadero DB Optimizer**](http://bit.ly/emb-apr-13)for all performance tuning and routine health check up. I will be interested to know what is your feedback about the product.

[**Part 1: What is Redundant Indexes?**](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/)Conversation between Mike and Jon – where they discuss about the fundamentals of Redundant Indexes.

[**Part 2: Demo – What kind of Redundant Indexes are Useful with a smaller result set**](http://blog.sqlauthority.com/2013/01/07/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-2/)Here Jon demonstrates a script where even though Redundant Indexes exist they are extremely useful and should not be dropped. The result set contains the very small result set.

[**Part 3: Demo – What kind of Redundant Indexes are Useful with a larger result set**](http://blog.sqlauthority.com/2013/01/11/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-3/)Here Jon demonstrates a script where even though Redundant Indexes exist they are extremely useful and should not be dropped. The result set contains the very large result set.

Once again if you have not read the earlier blog post, there are good chances you will miss the core point of this blog post. I suggest you at least read the conversation between Mike and Jon to get familiar with their personalities.

Jon and Mike continue their conversation from last week.

**Mike** – Jon, In very[**first day**](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/)we discussed that we should also pay attention to the Order By clause of the columns used in the Indexes. Do you have any working demonstration example where we can understand that concept in depth.

**Jon** – Sure, Mike. I think you are referring to following example which we have discussed in the conversation.

**Index 5:** Col1 ASC, Col2 DESC  
**Index 6:** Col1 DESC, Col2 DESC, Col3 ASC

Am I correct?

**Mike** – Yes, would you please explain how ASC and DESC works with the case of Redundant Indexes.

In normal case, we will not see the direction of the Columns used and consider Indexes as a redudant. For example when I see Index 5 and Index 6 I would see them as following.

**Index 5:** Col1 ASC, Col2 DESC  
**Index 6:** Col1 DESC, Col2 DESC, Col3 ASC

I will just see that Index 5 has two columns (Col1 and Col2), whereas Index 6 have three columns (Col1, Col2, Col3) and they are redundant (read the [very first blog](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/) post for why they are considered as a redundant).

However, from our conversation I do understand that the columns used to build indexes have a direction and we should consider them as an important factor before we make these indexes redundant.

**Jon** – Great Point Mike! You are absolutely correct. Let us understand the same using example.

We will create a single table with two different Indexes. Later we will run two different but similar looking queries on the table. We will observe how their index usage pattern is.

First create a table called SampleTable1.

USE tempdb  
GO  
-- Create table  
CREATE TABLE SampleTable1 (ID INT, Col1 INT, Col2 INT, Col3 INT)  
GO

Now let us create indexes on Sample Table.

Clustered Indexes are just created for reference – the results of this test will not be affected by its presence or absence of it. I have just created to rule out few doubts I anticipate by their absence. The important part is non clustered indexes. One of the non-clustered index is created on Col1 & Col2 DESC and another one is created on Col1 DESC, Col2 DESC, and Col3.

**Table1: SampleTable1**  
Index on SampleTable1: IX\_ST\_Col1\_Col2  
Index on SampleTable1: IX\_ST\_Col1\_Col2\_Col3

-- Create Indexes on Sample Table1  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable1 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable1 (Col1 ASC, Col2 DESC)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable1 (Col1 DESC, Col2 DESC, Col3 ASC)  
GO

Now let us populate both the tables. Both the tables have absolutely same data.

-- Populate tables  
INSERT INTO SampleTable1 (ID, Col1, Col2, Col3)  
SELECT RAND()\*10000, RAND()\*1000, RAND()\*100, RAND()\*10  
GO 100000

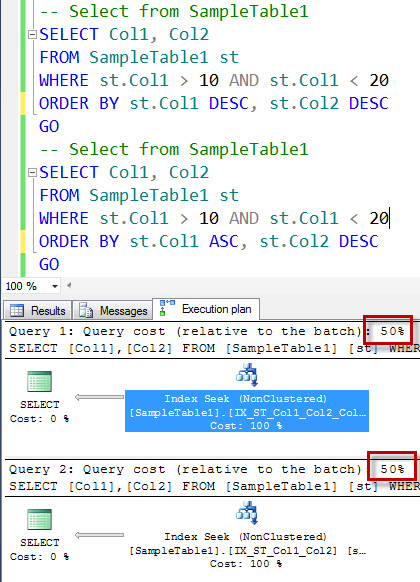
Now is the most interesting part. For this[**first enable the execution plan in SSMS**](http://blog.sqlauthority.com/2007/08/28/sql-server-actual-execution-plan-vs-estimated-execution-plan/) (shortcut key – CTRL + M).

In this test we will run two different queries on both the tables.

Let us first run two scripts for SampleTable1

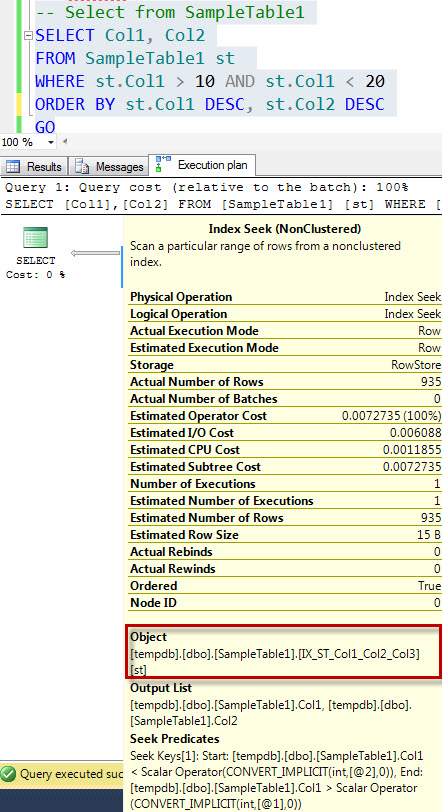
-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 DESC, st.Col2 DESC  
GO  
-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 ASC, st.Col2 DESC  
GO

Let us check the execution plan:



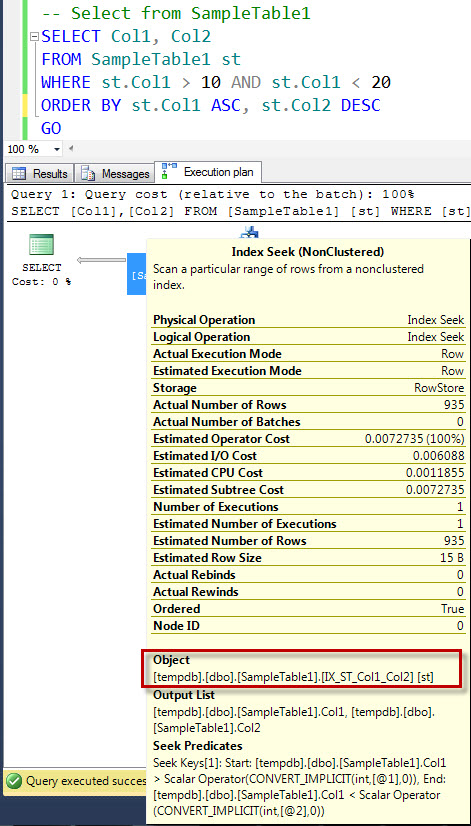
Now when you see the execution plan, you will notice that both the queries are taking an equal amount of the resources to execute both the queries. They each take 50% of the time compared to the each other. However, we need to pay extra attention to the queries and see which indexes each of the query is using.

Let us examine very first query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col2\_Col3.

Now let us examine second query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col2.

Now let us put a everything together quickly.

**When WHERE Conditions are st.Col1 DESC, st.Col2 DESC they use the Index with Columns Col1 DESC, Col2 DESC, Col3 ASC.**  
**When WHERE Conditions are st.Col1 ASC, st.Col2 DESC they use the Index with ColumnsCol1 ASC, Col2 DESC.**

I think by for you must have guessed that there is a relationship between the order of the columns with the ORDER by Clause used in the WHERE condition.

Even though, Index IX\_ST\_Col1\_Col2 looks like a Redundant Index, it is not a Redundant Index of all. They are matter of the fact very different Indexes. They should be both kept if both are used in the queries.

There are a lot more things we can discuss over here – but this is just a starter. Any question so far.

**Mike** – Thanks Jon, I understood so far. Here is the question in my mind – If I am following two Indexes they are considered as a different Index am I correct?

**Index AA:** Col1 DESC, Col2 ASC  
**Index BB**: Col1 ASC, Col2 ASC

Am I correct?

**Jon** – Yes, they are different Indexes for sure.

**Mike** – Oh okey I perfectly got it. Let me quickly **summarize** it.

Even though Indexes look redundant there may be some queries which may find them useful. ORDER of the columns used in the Indexes is very important when considering the redundancy of the indexes. If the column order is different the Indexes are also considered as a different indexes. Before dropping the indexes one should properly validate the usage patterns of the indexes and query workloads. One should properly test everything before taking any actions.

**Jon** – Good summary. Let us do a quick clean up.

-- Clean Up  
DROP TABLE SampleTable1  
GO

**Mike** – Perfect, I think I know a lot now.

**Jon** – Do you really think you know everything?

**Mike** – Well, sort of. Right?

**Jon** – If you think so answer me following question -

Are following two indexes should be considered as a redundant indexes or are they different indexes?

**Index AA:** Col1 DESC, Col2 DESC  
**Index BB:** Col1 ASC, Col2 ASC

**Mike** – From what I understood today – they are Not Redundant Index.

**Jon** – Glad I asked this question. **You are WRONG!**

**Mike** – What? How come?

**Jon** – Let us talk about this **future blog posts** – I think I need to teach you a lot of things!

**Mike** – Thanks, I am indeed looking forward to learn more. So what I understand is that it is quite possible that the indexes may look redundant but they may not be redundant indexes. What I understand is that before dropping any indexes I need to identify if the indexes are used or not. ***Is there any professional tool which can help me?***

**Jon** – Absolutely, I suggest that you try [**embarcadero DB Optimizer**](http://bit.ly/emb-apr-13). It helps finding users the precise details related to index usage. It also indicates if Indexes are useful or useless. This tool is very easy to use and help in various aspects of the performance tuning. Here is detailed articles on this subject [**Part 1**](http://blog.sqlauthority.com/2012/04/19/sql-server-performance-tuning-part-1-of-2-getting-started-and-configuration/)and[**Part 2**](http://blog.sqlauthority.com/2012/04/20/sql-server-performance-tuning-part-2-of-2-analysis-detection-tuning-and-optimizing/)**.** This tool can help you with writing proper queries, do indexes maintenance and optimize overall health of the server.

**Mike** – Thanks Jon!

« [SQL SERVER – What is Semantics Model – A Simple Explanation](http://blog.sqlauthority.com/2013/02/11/sql-server-what-is-semantics-models-a-simple-explanation/)

[SQL SERVER – Get SQL Server Version and Edition Information – SQL in Sixty Seconds #043 – Video](http://blog.sqlauthority.com/2013/02/13/sql-server-get-sql-server-version-and-edition-information-sql-in-sixty-seconds-043-video/) »

SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1 and Included Columns Col2 and Col3 – Part 5

February 12, 2013 by [pinaldave](http://blog.sqlauthority.com/author/pinaldave/" \o "Posts by pinaldave)

This is the fifth part of the series regarding Redundant Indexes. If have not read earlier part – there is quite a good chance that you will miss the context of this part. I quickly suggest you to read earlier four parts. On my production server I personally use [embarcadero DB Optimizer](http://bit.ly/emb-apr-13) for all performance tuning and routine health check up. I will be interested to know what is your feedback about the product.

[Part 1: What is Redundant Indexes?](http://blog.sqlauthority.com/2013/01/04/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-1/)Conversation between Mike and Jon – where they discuss about the fundamentals of Redundant Indexes.

[Part 2: Demo – What kind of Redundant Indexes are Useful with a smaller result set](http://blog.sqlauthority.com/2013/01/07/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-2/)Here Jon demonstrates a script where even though Redundant Indexes exist they are extremely useful and should not be dropped. The result set contains the very small result set.

[Part 3: Demo – What kind of Redundant Indexes are Useful with a larger result set](http://blog.sqlauthority.com/2013/01/11/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-3/)Here Jon demonstrates a script where even though Redundant Indexes exist they are extremely useful and should not be dropped. The result set contains the very large result set.

[Part 4: An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 4](http://blog.sqlauthority.com/2013/01/17/sql-server-an-interesting-case-of-redundant-indexes-index-on-col1-col2-and-index-on-col1-col2-col3-part-4/)  
In this part Jon demonstrates a script where the direction or order clause of the index is different making the index behaves very differently. To quality any index redundant the order of the index should also be considered.

Once again if you have not read the earlier blog post, there are good chances you will miss the core point of this blog post. I suggest you at least read the conversation between Mike and Jon to get familiar with their personalities.

Jon and Mike continue their conversation from last week.

Mike – Jon, you have been asking me many difficult questions. Let me simply ask you following question and instead of long answer – I would like you to give me a direct question answer.

Jon - Well, we can for sure try that way as well. What is your question?

Mike - When I see following columns as part of the index key, they are duplicate index correct?

Index A1: Col1 ASC  
Index B1: Col1 ASC

In this case, you can see I have both the indexes are same and their order of the column is same as well. I think this means that they are redundant and duplicate index correct?

Jon - Mike, indeed a great question. If the included columns are also the same, they are indeed a same index. Now you have a simple answer.

Mike - No it is not a simple answer. You inserted the curveball of included column index – what is that now?

Jon - Ha ha – well SQL Server is not simple – here is the article you should read about Included Column Index it explains the concept quite well: [Understanding Included Column Index along with Clustered Index and Non-clustered Index](http://blog.sqlauthority.com/2007/04/23/sql-server-understanding-new-index-type-of-sql-server-2005-included-column-index-along-with-clustered-index-and-non-clustered-index/).

Once you read it lets us run a quick demo script. The script is very similar as earlier script but this time the index is set up as follows:

Index A1: Col1 ASC INCLUDE Col2 ASC  
Index A1: Col1 ASC INCLUDE Col3 ASC

You can see that the key column (Col1) is same in both the indexes. However, the Included Columns are different. Now let us run following script.

First create a table called SampleTable1.

USE tempdb  
GO  
-- Create table  
CREATE TABLE SampleTable1 (ID INT, Col1 INT, Col2 INT, Col3 INT)  
GO

Now let us create indexes on Sample Table.

Clustered Indexes are just created for reference – the results of this test will not be affected by its presence or absence of it. I have just created to rule out few doubts I anticipate by their absence. The important part is non clustered indexes. One of the non-clustered index is created on Col1 ASC INCLUDE Col2 ASC and another one is created on Col1 ASC INCLUDE Col2 ASC

Table1: SampleTable1   
Index on SampleTable1: Col1 ASC INCLUDE Col2 ASC  
Index on SampleTable1: Col1 ASC INCLUDE Col3 ASC

-- Create Indexes on Sample Table1  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable1 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable1 (Col1)  
INCLUDE (Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col3]  
ON SampleTable1 (Col1)  
INCLUDE (Col3)  
GO

Now let us populate the table. The tables have absolutely same data.

-- Populate tables  
INSERT INTO SampleTable1 (ID, Col1, Col2, Col3)  
SELECT RAND()\*10000, RAND()\*1000, RAND()\*100, RAND()\*10  
GO 100000

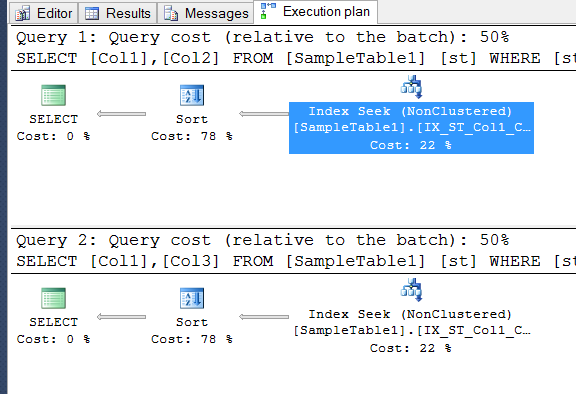
Now is the most interesting part. For this[first enable the execution plan in SSMS](http://blog.sqlauthority.com/2007/08/28/sql-server-actual-execution-plan-vs-estimated-execution-plan/) (shortcut key – CTRL + M).

In this test we will run two different queries on both the tables.

Let us first run two scripts for SampleTable1

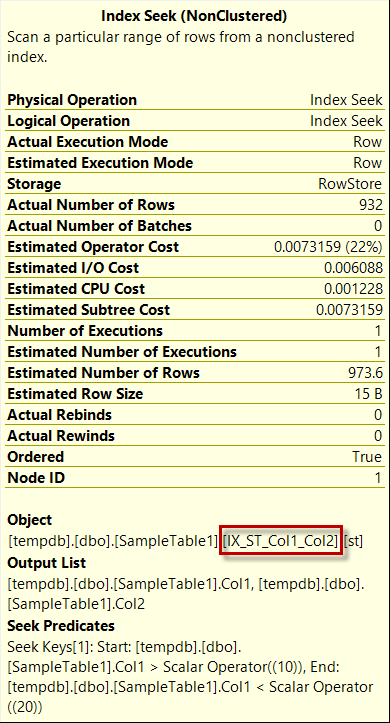
-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 DESC, st.Col2 DESC  
GO  
-- Select from SampleTable1  
SELECT Col1, Col3  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 ASC, st.Col3 DESC  
GO

Let us check the execution plan:



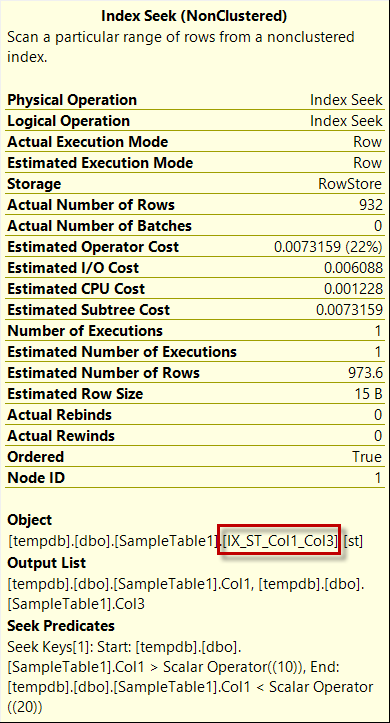
Now when you see the execution plan, you will notice that both the queries are taking an equal amount of the resources to execute both the queries. They each take 50% of the time compared to the each other. However, we need to pay extra attention to the queries and see which indexes each of the query is using.

Let us examine very first query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col2.

Now let us examine second query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col3.

Now you can see how the included columns impact the usage of the index. Even though the key column is the same – included columns are integral part of the index and they need to be evaluated.

Now you can clean up using the following script:

-- Clean Up  
DROP TABLE SampleTable1  
GO

Mike – Jon this is getting indeed complicated for me. I think this subject looks very easy but indeed it keeps on getting confusing. I think I will go back to the professional efficiency tool which you mentioned last time. I have yet to download it, can you provide me more details about it.

Jon – Absolutely, it is [embarcadero DB Optimizer](http://bit.ly/emb-apr-13). It helps finding users the precise details related to index usage. It also indicates if Indexes are useful or useless. This tool is very easy to use and help in various aspects of the performance tuning. This tool can help you with writing proper queries, do indexes maintenance and optimize overall health of the server.

Mike – Thanks Jon! I still have one more question but I will ask you about it next time.

SQL SERVER – An Interesting Case of Redundant Indexes – Index on Col1, Col2 and Index on Col1, Col2, Col3 – Part 6

Mike – Jon, I think I am now more clear with the redundant indexes scenario. I have understood that following indexes are not redudant indexes even though they have key column exactly same becasue of the included columns are different from each other. Am I correct to understand the same?

Table1: SampleTable1   
Index on SampleTable1: Col1 ASC INCLUDE Col2 ASC  
Index on SampleTable1: Col1 ASC INCLUDE Col3 ASC

Jon – Excellent Mike, I think you got this one correct. It makes me happy that we are not the right path now. Let me know if you have any further question.

Mike - The matter of the fact, I do have a question. I do understand that both the above indexes are different kind of index and they don’t duplicate but is there any possibility to create  a single index which can be as effective as both the indexes. I think it should be possible as both of the above index have the same keys.

Jon – Indeed a great question, actually you can do that as well.

Mike – Really! If that is the case, would you please explain me how to do that.

Jon – Sure, it is actually very easy to do so. Instead of above two indexes, you can just create a single index which has both the columns included in the INCLUDED clause. Here is the example:

Table1: SampleTable1   
Index on SampleTable1: Col1 ASC INCLUDE Col2, Col3

Here you can see that you have created a single index which is effective for Col1 key column and Col2 and Col3 in an included column. Remember, order of the column always mattered however, it does not matter when we are discussing INCLUDED Columns. In this case, it would have not matter if we have written INCLUDE Col3, Col2 instead of INCLUDE Col2, Col3 in the create index.

Mike – Oh, so this is interesting to know that column order does not matter in the Included clause. Would it be possible to see a demo where you demonstrate that this single Index can be replacement of the above two indexes.

Jon – Absolutely – let us check.

First create a table called SampleTable1.

USE tempdb  
GO  
-- Create table  
CREATE TABLE SampleTable1 (ID INT, Col1 INT, Col2 INT, Col3 INT)  
GO

Now let us create indexes on Sample Table.

Clustered Indexes are just created for reference – the results of this test will not be affected by its presence or absence of it. I have just created to rule out few doubts I anticipate by their absence. The important part is non clustered indexes. One of the non-clustered index is created on Col1 ASC INCLUDE Col2 ASC and another one is created on Col1 ASC INCLUDE Col2 ASC

Table1: SampleTable1   
Index on SampleTable1: Col1 ASC INCLUDE Col2 ASC  
Index on SampleTable1: Col1 ASC INCLUDE Col3 ASC

-- Create Indexes on Sample Table1  
CREATE CLUSTERED INDEX [CX\_ST]  
ON SampleTable1 (ID)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2]  
ON SampleTable1 (Col1)  
INCLUDE (Col2)  
GO  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col3]  
ON SampleTable1 (Col1)  
INCLUDE (Col3)  
GO

Now let us populate the table. The tables have absolutely same data.

-- Populate tables  
INSERT INTO SampleTable1 (ID, Col1, Col2, Col3)  
SELECT RAND()\*10000, RAND()\*1000, RAND()\*100, RAND()\*10  
GO 100000

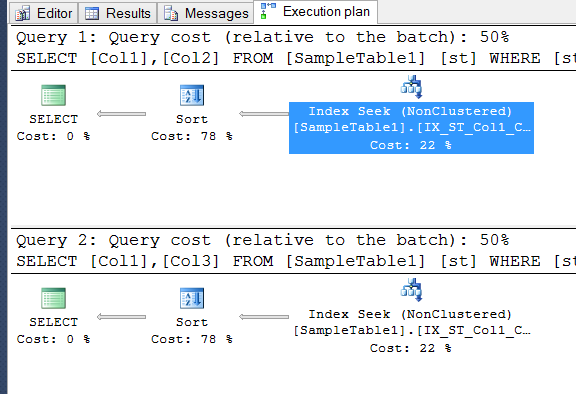
Now is the most interesting part. For this[first enable the execution plan in SSMS](http://blog.sqlauthority.com/2007/08/28/sql-server-actual-execution-plan-vs-estimated-execution-plan/) (shortcut key – CTRL + M).

In this test we will run two different queries on both the tables.

Let us first run two scripts for SampleTable1

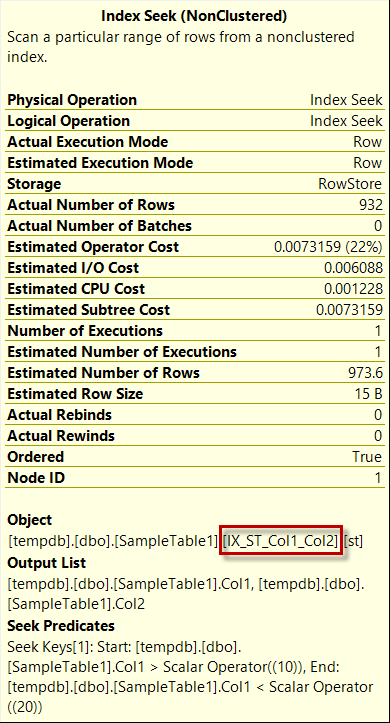
-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 DESC, st.Col2 DESC  
GO  
-- Select from SampleTable1  
SELECT Col1, Col3  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 ASC, st.Col3 DESC  
GO

Let us check the execution plan:



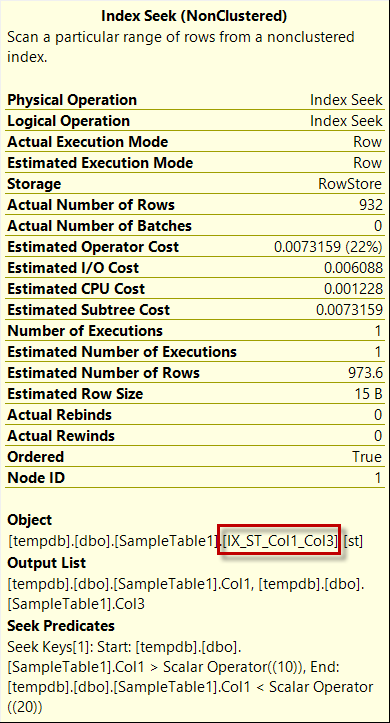
Now when you see the execution plan, you will notice that both the queries are taking an equal amount of the resources to execute both the queries. They each take 50% of the time compared to the each other. However, we need to pay extra attention to the queries and see which indexes each of the query is using.

Let us examine very first query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col2.

Now let us examine second query individually.



We can see from the execution plan – it is using the index – IX\_ST\_Col1\_Col3.

Now you can see how the included columns impact the usage of the index. Even though the key column is the same – included columns are integral part of the index and they need to be evaluated.

Creating Included Index

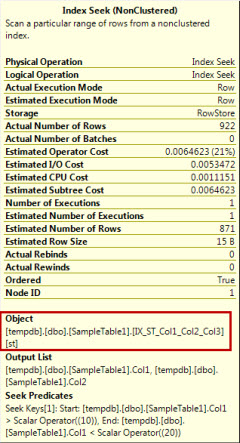
So far everything is very similar to our earlier blog posts. Now let us create a new included column index where we have Col2 and Col3 are included indexes.

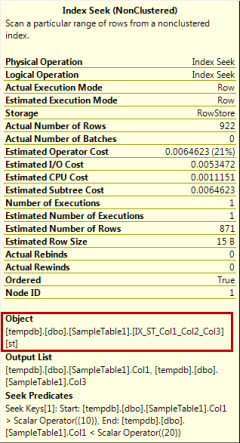
-- Create Included Index  
CREATE NONCLUSTERED INDEX [IX\_ST\_Col1\_Col2\_Col3]  
ON SampleTable1 (Col1)  
INCLUDE (Col2,Col3)  
GO

As discussed you can change the order of the columns in the Included clause and it will not matter. Now let us run the SELECT statement which we have run earlier and see which index each of the SELECT statement is using.

-- Select from SampleTable1  
SELECT Col1, Col2  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 DESC, st.Col2 DESC  
GO  
-- Select from SampleTable1  
SELECT Col1, Col3  
FROM SampleTable1 st  
WHERE st.Col1 > 10 AND st.Col1 < 20  
ORDER BY st.Col1 ASC, st.Col3 DESC  
GO

You can notice that both the SELECT statement is using newly created index IX\_ST\_Col1\_Col2\_Col3 as this single index now satisfies the need of the both the indexes created earlier. Now when you create this new index, your earlier two indexes IX\_ST\_Col1\_Col2 and IX\_ST\_Col1\_Col3 are not useful and they should be dropped eventually.





Now you can clean up using the following script:

-- Clean Up  
DROP TABLE SampleTable1  
GO

Mike – Wow! Jon, more I learn about indexes, I feel more I do not know about them. I think I am going to rely on [embarcadero DB Optimizer](http://bit.ly/emb-apr-13) for most of my performance tuning needs.

Jon – Absolutely,  [embarcadero DB Optimizer](http://bit.ly/emb-apr-13) is your best friend.  It helps finding users the precise details related to index usage. It also indicates if Indexes are useful or useless. This tool is very easy to use and help in various aspects of the performance tuning. This tool can help you with writing proper queries, do indexes maintenance and optimize overall health of the server.

Mike – Thanks Jon!