Non Clustered Index Internals

Table of Contents

Before You Begin

Estimated time to complete this lab

50 minutes

Objectives:

After completing this lab, you will be able to:

- Understand the non-clustered Index internals in SQL Server
- Understand non-clustered index architecture
- Understand non-clustered index over heap
- Understand non-clustered index over clustered Index
- Understand covering index

Prerequisites

Before working on this lab, you must have:

- Basic administration experience with SQL Server

Lab scenario

In this lab we will clear our understanding on non-clustered index. This lab is divided into multiple exercises and in each exercise we will cover different aspects of non-clustered index. In the 1st exercise we will look at the B-Tree structure of the non-clustered index, we will also look at pages at different level of the non-clustered index. In the 2nd exercise we will look at non-clustered index over a heap and we will also explore RID LOOKUP. In the 3rd exercise we will look at non-clustered index over a clustered index and the concept of key/bookmark lookup. And in our last exercise of this lab we will cover a select query to minimize I/O with a non-clustered covering index.

Tips to complete this lab successfully

Following these tips will be helpful in completing the lab successfully in time

- All lab files are located in C:\vLabs\Heap_Internals folder
- The script(s) are divided into various sections marked with 'Begin', 'End' and 'Steps'. As per the instructions, execute the statements between particular sections only or for a particular step
- Read the instructions carefully and do not deviate from the flow of the lab
- In case you execute the entire script by mistake or miss a step or get confused midway, simply 'Restart' the VM from the VM control panel to restart/redo the lab.

Exercise 1: Understanding Non Clustered Index B-Tree Structure

Scenario

In this exercise, we will look at non clustered index internals, how B-Tree is formed and what kind of data resides in each level of the non-clustered index.

Tasks	Detailed Steps
Launch **SQL Server Management Studio**	1. Click **Start** | **All Programs** | **SQL Server 2012** | **SQL Server Management Studio** or Double click **SQL Server Management Studio** shortcut on the desktop 2. In the **Connect to Server** dialog box, click **Connect**
Open **1_UnderstandingNonClusteredIndexInternals.sql**	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. Navigate to C:\vLabs\ 3. Select **1_UnderstandingNonClusteredIndexInternals.sql** and click **Open**
Execute the statement(s) in the 'Setup' section to setup the database and table	The setup section performs the following: • **SQLMaestros** database is created • **SQLMaestros** schema is created • **Table1** table is created with 200000 records In **1_UnderstandingNonClusteredIndexInternals.sql**, Review and execute the statement(s) in section 'Begin: Setup' and 'End: Setup' ---------------------

```
-- Begin: Setup
---------------------

-- Create a database named SQLMaestros
USE master;
GO
IF EXISTS(SELECT * FROM sys.databases WHERE name='SQLMaestros')
ALTER DATABASE [SQLMaestros] SET  SINGLE_USER WITH ROLLBACK IMMEDIATE;
DROP DATABASE SQLMaestros;
CREATE DATABASE SQLMaestros;
GO


USE SQLMaestros;
SET NOCOUNT ON;
GO

-- Create a schema named SQLMaestros
CREATE SCHEMA [SQLMaestros] AUTHORIZATION [dbo];
GO

-- Create Table1 table in SQLMaestros database
CREATE Table [SQLMaestros].[Table1](
   Column1 INT,
   Column2 VARCHAR(8000),
   Column3 CHAR(10),
   Column4 INT);
GO

-- Insert 200000 records in Table1 table
DECLARE @COUNT INT;
SET @COUNT = 1;
DECLARE @DATA1 VARCHAR(8000)
SET @DATA1 = 'data'
WHILE @COUNT < 200001
BEGIN
DECLARE @DATA2 INT;
SET @DATA2 = ROUND(10000000*RAND(),0);
INSERT INTO [SQLMaestros].[Table1] VALUES(@COUNT,@DATA1,'random',@DATA2);
```

	`` SET @COUNT = @COUNT + 1; END GO --------------------- -- End: Setup --------------------- ```
CREATE a non-clustered index	Execute the following statement(s) to **CREATE** a non-clustered index on **Column1** column of **Table1** table ``` -- Step 1: Create non-clustered index on Column1 column of Table1 table CREATE NONCLUSTERED INDEX NCL_Table1_Column1 ON [SQLMaestros].[Table1](Column1); GO ``` **Note:** There are many clauses that we can specify during the creation of an index. If we don't specify these, SQL server will consider the **default values**. We have to use **WITH (OPTION_NAME = VALUE)** while creating or rebuilding index to specify below options. ``` | PAD_INDEX = { ON | OFF } | FILLFACTOR = fillfactor (Integer value between 0 – 100) | SORT_IN_TEMPDB = { ON | OFF } | IGNORE_DUP_KEY = { ON | OFF } | STATISTICS_NORECOMPUTE = { ON | OFF } | DROP_EXISTING = { ON | OFF } | ONLINE = { ON | OFF } | ALLOW_ROW_LOCKS = { ON | OFF } | ALLOW_PAGE_LOCKS = { ON | OFF } | MAXDOP = max_degree_of_parallelism (Integer value depending upon the no. of CPU) | DATA_COMPRESSION = { NONE | ROW | PAGE} ```
View non-clustered index details	Execute the following statement(s) to view the non-clustered index information of **Table1** table ``` -- Step 2: View index details of Table1 table EXEC sp_helpindex 'SQLMaestros.Table1'; ```

```
GO
```

Note: **sp_helpindex** is a system stored procedure that can be used to view index details for a particular table.

View statistics details

Execute the following statement(s) to view the statistics details of **Table1** table

```
-- Step 3: View statistics details of Table1 table
SELECT STATS.* FROM sys.stats AS STATS
INNER JOIN sys.objects AS OBJ
ON STATS.object_id = OBJ.object_id
WHERE OBJ.name = 'Table1';
GO
```

Editor | Results | Messages

	object_id	name	stats_id	auto_created	user_created	no_rec
1	885578193	NCL_Table1_Column1	2	0	0	0

Note: Whenever we create an index, SQL Server automatically creates statistics for that index.
If a statistics object is manually created then **user_created** column will be **'1'**.Apart from statistics being created automatically for an index, SQL Server can automatically create statistics for a column without an index if that column is used in a predicate and **AUTO_CREATE_STATISTICS** database option is 'ON'. If that is the case, **auto_created** column will show a value of '**1**'.

View non-clustered index details	Execute the following statement(s) to view non-clustered index details of **Table1** table

```
-- Step 4: View index details of non-clustered index on Table1 table
SELECT
index_id,index_type_desc,index_level,page_count,avg_record_size_in_bytes,avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table1'), 2, NULL , 'DETAILED')
ORDER BY index_level DESC;
GO
```

Results | Messages

	index_id	index_type_desc	index_level	page_count	avg_record_size_in_bytes	avg_fragmentation_in_percent
1	2	NONCLUSTERED INDEX	2	1	22	0
2	2	NONCLUSTERED INDEX	1	2	22	100
3	2	NONCLUSTERED INDEX	0	446	16	0.224215246636771

Note: **sys.dm_db_index_physical_stats()** DMF can be used to get detailed index information. Below is the complete list of parameters that we can pass to this DMF:

```
sys.dm_db_index_physical_stats
    (
        { database_id | NULL | 0 | DEFAULT }
      , { object_id | NULL | 0 | DEFAULT }
      , { index_id | NULL | 0 | -1 | DEFAULT }
      , { partition_number | NULL | 0 | DEFAULT }
      , { mode [DETAILED|SAMPLED|LIMITED] | NULL | DEFAULT }
    )
```

Observation: We are using **sys.dm_db_index_physical_stats** Dynamic Management Object to view index metadata. In the above output, **index_level** column represents the index depth. **Index_level '0'** is for leaf level and any subsequent higher value represents the intermediate level and root level. The clustered index has three levels. 1st row is for root level (**index_level = 2**), 2nd row for intermediate level (**index_level = 1**) and 3rd row for leaf level (**index_level = 0**). **page_count**, **avg_record_size_in_bytes** and **avg_fragmentation_in_percent** represents no. of pages, average row size in each page and amount of fragmentation in each level respectively.

Note: Fillfactor is only applicable for leaf level pages. If you want to define index fillfactor to intermediate and root level**,** then we have to specify that by turning **Pad_Index** option ‘**ON’** while creating or rebuilding the index.

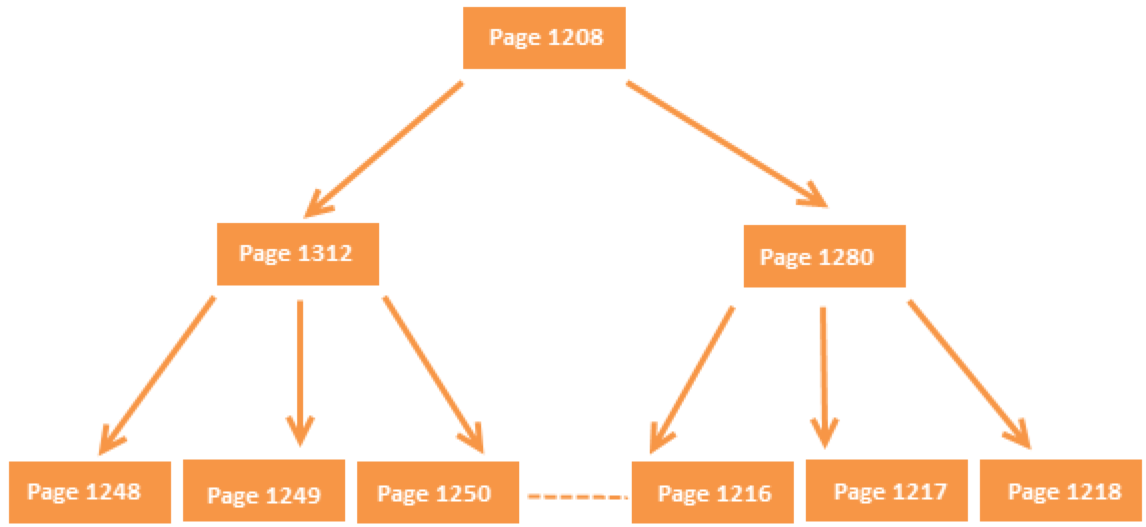
View B-Tree structure of non-clustered index

Execute the following statement(s) to view the B-Tree structure of non-clustered index on **Table1** table

```
-- Step 5: View non-clustered index architecture
SELECT allocated_page_page_id,page_type_desc,page_level,next_page_page_id,previous_page_page_id
FROM sys.dm_db_database_page_allocations(DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table1'), 2,
NULL, 'DETAILED')
WHERE page_type IN (1,2)
ORDER BY page_level DESC;
GO
```

Results | Messages

	allocated_page_page_id	page_type_desc	page_level	next_page_page_id	previous_page_page_id
1	1208	INDEX_PAGE	2	NULL	NULL
2	1280	INDEX_PAGE	1	NULL	1312
3	1312	INDEX_PAGE	1	1280	NULL
4	1344	INDEX_PAGE	0	1345	1279
5	1345	INDEX_PAGE	0	1346	1344
6	1346	INDEX_PAGE	0	1347	1345
7	1347	INDEX_PAGE	0	1348	1346
8	1348	INDEX_PAGE	0	1349	1347
9	1349	INDEX_PAGE	0	1350	1348



Note: sys.dm_db_database_page_allocations() is an undocumented DMF available only in SQL Server 2012. Below is the parameter list that can be passed into this DMF

```
sys.dm_db_database_page_allocations
      (
          { database_id | NULL | DB_ID() }
        , { object_id | NULL | OBJECT_ID() }
        , { index_id | NULL }
        , { partition_number | NULL }
        , { mode [DETAILED|LIMITED] | NULL | DEFAULT }
       )
```

View memory dump of root level page

Execute the following statement(s) to view memory dump of non-clustered index root page (Replace 1208 in the below statement with the **allocated_page_page_id** for the root page from the output of **step 5** [**Note**: Root page **page_level** is **'2'**])

```
-- Step 6: View memory dump of root page
DBCC TRACEON(3604)
DBCC PAGE('SQLMaestros',1,1208,3); -- Page ID will change in your case
GO
```

Results | Messages

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	COL1 (key)	HEAP RID (key)	KeyHashValue	Row Size
1	1	1208	0	2	1	1312	NULL	NULL	NULL	22
2	1	1208	1	2	1	1280	100001	0xEE02000001007800	NULL	22

Note: Non-Clustered index root page dose not contains any user data but only pointers to the intermediate level pages. We can observer that in the **Messages** section in the output.

Note: In order to view DBCC PAGE output in SSMS we have to enable Trace Flag 3604. DBCC PAGE() command can be used to view a page contents. Below is the complete parameter list that we can pass into DBCC PAGE() command:

```
DBCC PAGE
      (
           { database_name | database_id | DB_ID() }
         , { file_number }
         , { page_number }
          , { print_option [0 – header | 1 – header + slot array | 2 – header + whole page hex dump | 3
         -     header + complete page hex dump with row by row interpretation}
        )
```

View memory dump of intermediate level page

Execute the following statement(s) to view memory dump of a non-clustered index intermediate level page (Replace 1312 in the below statement with **allocated_page_page_id** of the intermediate level page from the output of **step 5**[**Note**: For intermediate level page **page_level** is '**1**'])

```
--Step 7: View memory dump of intermediate level page
DBCC PAGE('SQLMaestros',1,1312,3); -- Page ID will change in your case
GO
```

Editor | Results | Messages

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column1 (key)	HEAP RID (key)	KeyHashValue	Row Size
1	1	1312	0	1	1	1216	100001	0xEE02000001007800	NULL	22
2	1	1312	1	1	1	1217	100450	0xF002000001008100	NULL	22
3	1	1312	2	1	1	1218	100899	0xF202000001008A00	NULL	22
4	1	1312	3	1	1	1219	101348	0xF402000001009300	NULL	22
5	1	1312	4	1	1	1220	101797	0xF602000001009C00	NULL	22
6	1	1312	5	1	1	1221	102246	0xF80200000100A500	NULL	22
7	1	1312	6	1	1	1222	102695	0xFA0200000100AE00	NULL	22
8	1	1312	7	1	1	1223	103144	0xFC0200000100B700	NULL	22
9	1	1312	8	1	1	1224	103593	0xFE0200000100C000	NULL	22

Observation: HEAP RID (key) in the above output is used to uniquely identify each row in the page if there are no clustered index (Heap). However if there is a clustered index then non-clustered index will include the clustered index key and **UNIQUIFIER (key)** if the clustered index is not unique**.**

View memory dump of leaf level page

Execute the following statement(s) to view memory dump of non-clustered index leaf level page (Replace 1248 in the below statement with **allocated_page_page_id** of a leaf level page from the output of **step 5**[**Note**: For leaf level page **page_level** is '**0**'])

```
--Step 8: View memory dump of leaf level page
DBCC PAGE('SQLMaestros',1,1248,3); -- Page ID will change in your case
```

GO

Editor | Results | Messages

	FileId	PageId	Row	Level	Column1 (key)	HEAP RID (key)	KeyHashValue	Row Size
1	1	1248	0	0	1	0x1F01000001000000	(61208662f35e)	16
2	1	1248	1	0	2	0x1F01000001000100	(5e57c92d898d)	16
3	1	1248	2	0	3	0x1F01000001000200	(9da3f2e705c8)	16
4	1	1248	3	0	4	0x1F01000001000300	(21b957b37d2b)	16
5	1	1248	4	0	5	0x1F01000001000400	(98276e681e73)	16
6	1	1248	5	0	6	0x1F01000001000500	(a750212764a0)	16
7	1	1248	6	0	7	0x1F01000001000600	(64a41aede8e5)	16

Note: KeyHashValue is used to uniquely identify each row in case of **locking**.

Close all the query windows	Close all the query windows (X) and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, we have learnt:

- Options available during creation of an index
- B-Tree structure of non-clustered index
- Concept of HEAP RID
- Internals of pages in different level of b-tree structure of a non-clustered index

Exercise 2: Non Clustered Index on Heap

Scenario

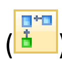
In this exercise, we will look at non-clustered index over a heap (table with no clustered index).

Tasks	Detailed Steps
Open **2_NonClusteredIndexOnHeap.sql**	1. Click **File** | **Open** | **File** or press (**Ctrl + O**) 2. Navigate to C:\vLabs\ 3. Select **2_NonClusteredIndexOnHeap.sql** and click **Open**
Execute the statement(s) in the Setup section to setup the table	The setup section performs the following: • **Table2** table is created with 100000 records In **2_NonClusteredIndexOnHeap.sql**, Review and execute the statement(s) in section 'Begin: Setup' and 'End: Setup' `---------------------` `-- Begin: Setup` `---------------------` `USE SQLMaestros;` `SET NOCOUNT ON;` `GO` `-- Create Table2 table in SQLMaestros database` `CREATE TABLE [SQLMaestros].[Table2](` `   Column1 INT,` `   Column2 CHAR(30),` `   Column3 VARCHAR(30),` `   Column4 INT,`

<table>
<tr><td></td><td>

```
   Column5 INT);
GO

-- Insert 100000 records in Table2 table
DECLARE @COUNT INT;
SET @COUNT = 1;
WHILE @COUNT < 100001
BEGIN
DECLARE @DATA2 INT;
SET @DATA2 = ROUND(10000000*RAND(),0);
DECLARE @RAND VARCHAR(30)
SET @RAND = (select
char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+6
5)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65));
INSERT INTO [SQLMaestros].[Table2] VALUES(@COUNT,'data', @RAND, @DATA2, @COUNT + 1);
SET @COUNT = @COUNT + 1;
END
GO

---------------------
-- End: Setup
---------------------
```

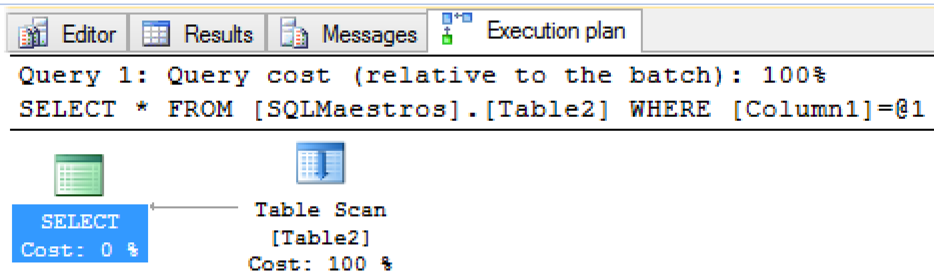
</td></tr>
<tr><td>Execute a SELECT statement</td><td>

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) against **Table2** table

```
-- Step 1: Execute below select statement with actual execution plan (Ctrl + M)
SET STATISTICS IO ON;
SELECT * FROM [SQLMaestros].[Table2] WHERE Column1 = 1000;
GO
```

Note: **SET STATISTICS IO ON** statement is used to display logical reads for the query.

</td></tr>
</table>



Observation: In the message section/tab we can see that 827 logical reads occurred. This means that SQL Server had to read 827 data pages to fetch the required result set. Observe the **Execution Plan** and we can see that a Table Scan occurred. Since there is no index on **Column1** column, SQL Server cannot perform a seek and has to scan the entire table to fetch the data where **Column1**=1000. Table scan means that SQL Server has to touch every record in the table to see if it matches the filer criteria.

CREATE a non-clustered index

Execute the following statement(s) to **CREATE** a non-clustered index on **Column1** column of **Table2** table

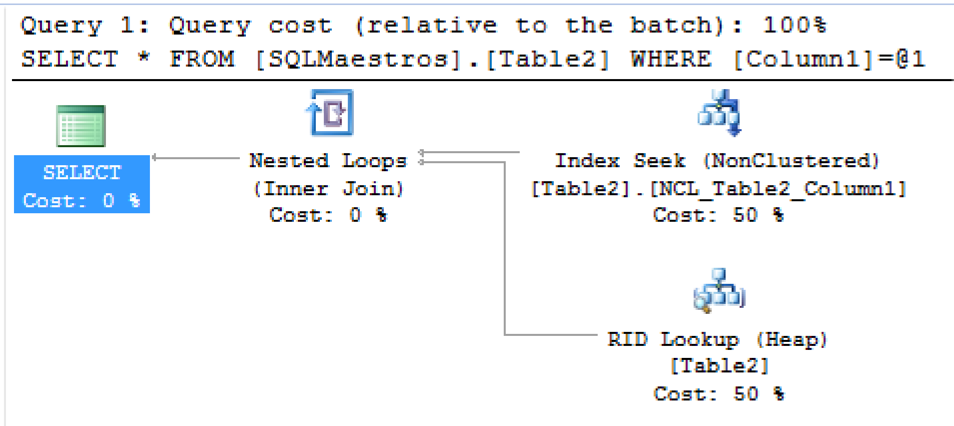
```
-- Step 2: Create a non-clustered index on Column1 column of Table2 table
CREATE NONCLUSTERED INDEX NCL_Table2_Column1 ON [SQLMaestros].[Table2](Column1);
GO
```

Execute a **SELECT** statement

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) against **Table2** table

```
-- Step 3: Execute below select statement with actual execution plan (Ctrl + M)
SELECT * FROM [SQLMaestros].[Table2] WHERE Column1 = 1000;
GO
```

Note: We can use **SET STATISTICS TIME ON** to find the CPU time of a query.



Observation: Observe in the Message section/tab that this time SQL Server required 3 logical reads (as opposed to 827 in the previous execution). I/O cost is reduced due to the presence of a non-clustered index. We can see in **Execution Plan** that this time SQL Server performs a non-clustered index seek on using the index NCL_Table2_Column1. This time SQL Server also performs RID lookup. The **SELECT** query uses the predicate **Column1 = 1000**. Non-clustered index is built on **Column1** column and thus SQL Server accesses the non-clustered index and performs a seek operation. But the non-clustered index does not contains any other column data except **Column1** and we are asking for all the columns (**SELECT ***), thus, in order to fetch the rest of the column data, SQL Server performs a **RID Lookup** where it fetches the **RID (key)** from the non-clustered index and performs a lookup on the heap.

View non-clustered index details

Execute the following statement(s) to view index details of non-clustered index on **Table2** table

```
-- Step 4: View index details of Table2 table
SELECT name,index_id FROM sys.indexes
WHERE name = 'NCL_Table2_Column1';
GO
```

View page allocations	Execute the following statement(s) to view pages allocated to non-clustered index (Replace the **<index id>** in the following statement with the value we get from the above task [**Step 4**])

```
-- Step 5: View pages allocated to non-clustered index
SELECT allocated_page_page_id,page_type_desc,page_level,next_page_page_id,previous_page_page_id
FROM sys.dm_db_database_page_allocations(DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table2'), <index_id>,
NULL, 'DETAILED')
WHERE page_type IN (1,2)
ORDER BY page_level DESC;
GO
```

Editor | Results | Messages

	allocated_page_page_id	page_type_desc	page_level	next_page_page_id	previous_page_page_id
1	2608	INDEX_PAGE	1	NULL	NULL
2	2640	INDEX_PAGE	0	2641	2575
3	2641	INDEX_PAGE	0	2642	2640
4	2642	INDEX_PAGE	0	2643	2641
5	2643	INDEX_PAGE	0	2644	2642
6	2644	INDEX_PAGE	0	2645	2643
7	2645	INDEX_PAGE	0	2646	2644
8	2646	INDEX_PAGE	0	2647	2645
9	2647	INDEX_PAGE	0	2648	2646

Explanation: The non-clustered index has two levels depicted by **page_level** in the above output. **Page_level '1'** is for the root level and **'0'** is for the leaf level.

View memory dump of root page	Execute the following statement(s) to view memory dump of non-clustered index root page (Replace 2608 in the below statement with the **allocated_page_page_id** for the root page from the output of **step 5** [**Note**: Root page **page_level** is **'1'**])

```
-- Step 6: View memory dump of root page
DBCC TRACEON(3604)
DBCC PAGE('SQLMaestros',1,2608,3); -- Page ID will change in your case
GO
```

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column1 (key)	HEAP RID (key)	KeyHashValue	Row Size
1	1	2608	0	1	1	2544	NULL	NULL	NULL	22
2	1	2608	1	1	1	2545	450	0x0805000001005600	NULL	22
3	1	2608	2	1	1	2546	899	0x0C05000001003300	NULL	22
4	1	2608	3	1	1	2547	1348	0x3307000001001000	NULL	22
5	1	2608	4	1	1	2548	1797	0x3607000001006600	NULL	22
6	1	2608	5	1	1	2549	2246	0x3A07000001004300	NULL	22
7	1	2608	6	1	1	2550	2695	0x3E07000001002000	NULL	22
8	1	2608	7	1	1	2551	3144	0x4107000001007600	NULL	22

Observation: We have created the non-clustered index on **Column1** column and the table is a heap. Therefore, the non-clustered index data structure has included **Column1 (key)** and **HEAP RID (key). HEAP RID (key)** is the address of the actual data page with the same **Column1** value. For the above **SELECT** statement, SQL Server will first navigate to the root page of the non-clustered index and find the **page_id** for **Column1 = 1000** (In this case page 2546 which contains data for **Column1** between 900 to 1347) and then will go to that leaf page (**ChildPageid**) to find the record.

View memory dump leaf level page

Execute the following statement(s)to view memory dump of the non-clustered index leaf page (Replace 2546 in the below statement with **ChildPageid** from the output of **step 7** where **Column1** value contains 1000)

```
--Step 8: View memory dump of leaf level page
DBCC PAGE('SQLMaestros',1,2546,3); -- Page ID will change in your case
GO
```

Editor | Results | Messages

	FileId	PageId	Row	Level	Column1 (key)	HEAP RID (key)	KeyHashValue	Row Size
100	1	2546	99	0	998	0x3007000001001D00	(483dec2d5eaa)	16
101	1	2546	100	0	999	0x3007000001001E00	(8bc9d7e7d2ef)	16
102	1	2546	101	0	1000	0x3007000001001F00	(dae935b244c5)	16
103	1	2546	102	0	1001	0x3007000001002000	(44b883b1d797)	16
104	1	2546	103	0	1002	0x3007000001002100	(7bcfccfead44)	16
105	1	2546	104	0	1003	0x3007000001002200	(b83bf7342101)	16

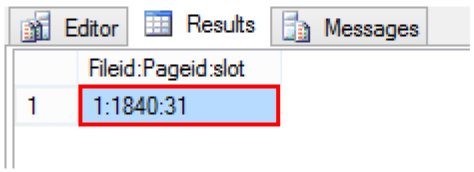
Explanation: The **SELECT** statement contains all columns (**SELECT ***). SQL Server will navigate from root page to the leaf page and extract the **HEAP RID (key)** where **Column1 (key) = 1000**. **HEAP RID (key)** defines the location of the complete record in the heap in hexadecimal format (<**FileID:PageID:Slot No**.>).Using this, SQL Server will locate the correct data page to fetch the entire record. This is called **RID LOOKUP**.

Get physical location of data from **HEAP RID(key)**

Execute the following statement(s) to find the physical location of data from **HEAP RID(key)**

```
--Step 9: View Heap Rid in 'Fileid:Pageid:slot' format
DECLARE @HeapRid BINARY(8)
SET @HeapRid = 0x3007000001001F00 -- Replace this with HEAP RID(key) from previous output WHERE Column1(key) =
1000
SELECT
       CONVERT (VARCHAR(5),
                    CONVERT(INT, SUBSTRING(@HeapRid, 6, 1)
                               + SUBSTRING(@HeapRid, 5, 1)))
     + ':'
     + CONVERT(VARCHAR(10),
                    CONVERT(INT, SUBSTRING(@HeapRid, 4, 1)
                               + SUBSTRING(@HeapRid, 3, 1)
                               + SUBSTRING(@HeapRid, 2, 1)
                               + SUBSTRING(@HeapRid, 1, 1)))
     + ':'
          + CONVERT(VARCHAR(5),
                    CONVERT(INT, SUBSTRING(@HeapRid, 8, 1)
                               + SUBSTRING(@HeapRid, 7, 1)))
                               AS 'Fileid:Pageid:slot';
```

```
GO
```



Explanation: The above query converts the **HEAP RID (key)** hexadecimal value into integer value. The record location is slot 31 of page 1840 of data file 1. (Note: your values may be different)

View memory dump of data page

Execute the following statement(s) to view memory dump of a data page (Replace 1840 with **page_id** we get in the output of **step 9**)

```
--Step 10: View memory dump of data page
DBCC PAGE('SQLMaestros',1,1840,3); -- Page ID will change in your case
GO
```

```
Slot 31 Column 1 Offset 0x4 Length 4 Length (physical) 4

Column1 = 1000

Slot 31 Column 2 Offset 0x8 Length 30 Length (physical) 30

Column2 = data

Slot 31 Column 3 Offset 0x35 Length 9 Length (physical) 9

Column3 = SCEMEIKIL

Slot 31 Column 4 Offset 0x26 Length 4 Length (physical) 4

Column4 = 2777667

Slot 31 Column 5 Offset 0x2a Length 4 Length (physical) 4

Column5 = 1001
```

	Observation: SQL Serve now fetches the entire record from the data page.
Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, we have learnt:

- How storage engine finds a record with the help of non-clustered index key and heap rid
- Concept of RID LOOKUP
- Concept of HEAP RID(key)

Exercise 3: Non Clustered Index over Clustered Index

Scenario

In this exercise, we will look at non clustered index over a clustered index.

<table>
<tr><th>Tasks</th><th>Detailed Steps</th></tr>
<tr><td>Open 3_NonClusteredIndexOverClusteredIndex.sql</td><td>1. Click File | Open | File or press (Ctrl + O)
2. Navigate to C:\vLabs\
3. Select 3_NonClusteredIndexOverClusteredIndex.sql and click Open</td></tr>
<tr><td>Execute the statement(s) in the Setup section to setup the table and a clustered index</td><td>The setup section performs the following:
• Table3 table is created with 100000 records
• Clustered index is created on Column1 column of Table1 table

In 3_NonClusteredIndexOverClusteredIndex.sql, Review and execute the statement(s) in section ‘Begin: Setup’ and ‘End: Setup’

<pre>

-- Begin: Setup

USE SQLMaestros;
SET NOCOUNT ON;
GO

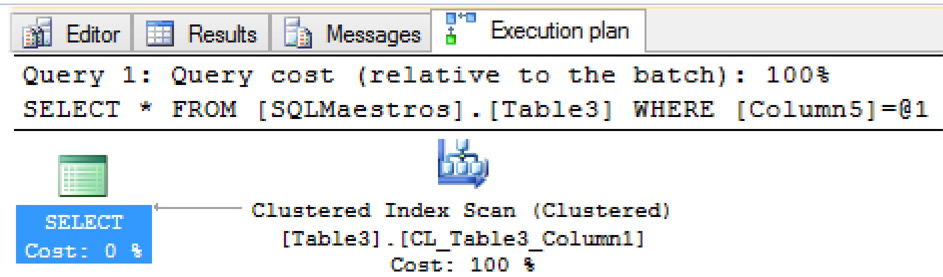
-- Create Table2 table in SQLMaestros database
CREATE TABLE [SQLMaestros].[Table3](
   Column1 INT,
   Column2 CHAR(30),
   Column3 VARCHAR(30),
</pre></td></tr>
</table>

```sql
   Column4 INT,
   Column5 INT);
GO

-- Insert 100000 records in Table3 table
DECLARE @COUNT INT;
SET @COUNT = 1;
WHILE @COUNT < 100001
BEGIN
DECLARE @DATA2 INT;
SET @DATA2 = ROUND(10000000*RAND(),0);
DECLARE @RAND VARCHAR(30)
SET @RAND = (select
char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand(
)*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65));
INSERT INTO [SQLMaestros].[Table3] VALUES(@COUNT,'data', @RAND, @DATA2, @COUNT);
SET @COUNT = @COUNT + 1;
END
GO

-- Create a clustered index on Column1 column of Table3 table
CREATE CLUSTERED INDEX CL_Table3_Column1 ON [SQLMaestros].[Table3](Column1);
GO

---------------------
-- End: Setup
---------------------
```

Execute a **SELECT** statement

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) against **Table3** table

```sql
-- Step 1: Execute below select statement with actual execution plan (Ctrl + M)
SET STATISTICS IO ON;
SELECT * FROM [SQLMaestros].[Table3] WHERE Column5 = 1000;
GO
```



Observation: If we look at the message section in the output, we can see that our **SELECT** query is taking 824 logical I/O. In the **Execution Plan** tab we can see that it's doing a clustered index scan operation. We have the clustered index on **Column1** column of **Table3** table but the search predicate is on **Column5** column. Thus SQL Server does not know the location of data where **Column5 = 1000**. In order to find the record it has to look at every clustered index data page, thus we are getting a scan instead of seek.

CREATE a non-clustered index

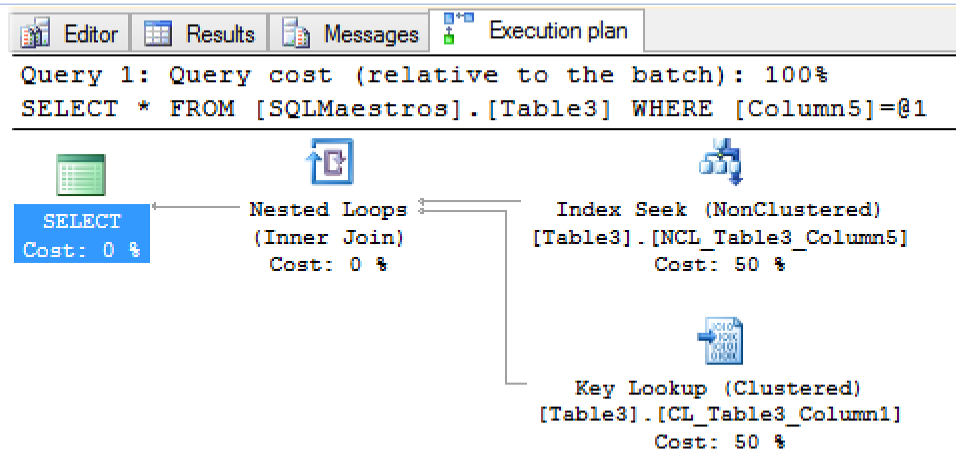
Execute the following statement(s) to **CREATE** a non-clustered index on **Column5** column of **Table3** table

```
-- Step 2: Create a non-clustered index on Column5 column of Table3 table
CREATE NONCLUSTERED INDEX NCL_Table3_Column5 ON [SQLMaestros].[Table3](Column5);
GO
```

Execute a **SELECT** statement

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) against **Table3** table

```
-- Step 3: Execute below select statement with actual execution plan (Ctrl + M)
SELECT * FROM [SQLMaestros].[Table3] WHERE Column5 = 1000;
GO
```



Observation: If we look at the message section in the output, we can see that the **SELECT** query is taking 5 logical I/O; previously it was taking 824 logical I/O. Again if we look at the **Execution Plan** tab we can see that it's doing a non-clustered index seek + Key Lookup. We have a non-clustered index on **Column5** column and the search predicate is also in **Column5** column thus SQL Server is doing a non-clustered index seek where it directly goes to non-clustered index root page and finds the leaf page from there (**Where Column5 = 1000**) and then it goes to non-clustered index leaf page. As we are selecting all the rows (**SELECT *)** thus SQL Server needs to fetch the entire row where **Column5 = 1000**. But other column data is not there in the non-clustered index leaf page. Thus in order to fetch the other columns data it has to go in the data page (In this case clustered index leaf page) and this is what called Key Lookup where SQL Server takes the clustered index key in the non-clustered index leaf page and goes to the clustered index leaf/data page and retrieves rest of the data. In the next few steps we will do the same.

Find non-clustered index **index_id**

Execute the following statement(s) to find non-clustered index **index_id**

```
-- Step 4: Execute below statement to find non-clustered index index_id
SELECT name,index_id FROM sys.indexes WHERE name = 'NCL_Table3_Column5';
GO
```

View pages allocations

Execute the following statement(s) to view all the pages allocated to the non-clustered index (Replace **<index_id>** in the below statement with **index_id** we get from **step 4**)

```
-- Step 5: View pages allocated to non-clustered index
SELECT allocated_page_page_id,page_type_desc,page_level,next_page_page_id,previous_page_page_id
FROM sys.dm_db_database_page_allocations(DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table3'),
<index_id>, NULL, 'DETAILED')
WHERE page_type IN (1,2)
ORDER BY page_level DESC;
GO
```

Editor | Results | Messages | Execution plan

	allocated_page_page_id	page_type_desc	page_level	next_page_page_id	previous_page_page_id
1	2744	INDEX_PAGE	1	NULL	NULL
2	2832	INDEX_PAGE	0	2833	2639
3	2833	INDEX_PAGE	0	2834	2832
4	2834	INDEX_PAGE	0	2835	2833
5	2835	INDEX_PAGE	0	2836	2834
6	2836	INDEX_PAGE	0	2837	2835
7	2837	INDEX_PAGE	0	2838	2836

Note: page_level column in the above output represents index level. In the above output **page_level '1'** represents root page and **page_level '0'** represents leaf level pages.

View memory dump of root page

Execute the following statement(s) to view memory dump of non-clustered index root page (Replace 2744 in the below statement with the **allocated_page_page_id** for the root page from the output of **step 5** [**Note**: Root page **page_level** is **'1'**])

```
-- Step 6: View memory dump of root page
DBCC TRACEON(3604)
DBCC PAGE('SQLMaestros',1,2744,3); -- Page ID will change in your case
GO
```

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column5 (key)	Column1 (key)	UNIQUIFIER (key)	KeyHashValue	Row Size
1	1	2744	0	1	1	2600	NULL	NULL	NULL	NULL	18
2	1	2744	1	1	1	2601	579	579	0	NULL	18
3	1	2744	2	1	1	2602	1157	1157	0	NULL	18
4	1	2744	3	1	1	2603	1735	1735	0	NULL	18
5	1	2744	4	1	1	2604	2313	2313	0	NULL	18

Observation: To perform the search operation SQL Server will first go the root page of non-clustered index and will search the **ChildPageId** where **Column5 (key)** contains 1000. In the above output **ChildPageId** 2601 contains data for **Column5** where **Column5** value lies between 579 and 1156 (Highlighted with red).

View memory dump of leaf level page

Execute the following statement(s) to view memory dump of a non-clustered index leaf page (Replace 2601 in the below statement with **ChildPageid** from the output of **step 6** which contains **Column5** value 1000)

```
--Step 7: View memory dump of leaf level page
DBCC PAGE('SQLMaestros',1,2601,3); -- Page ID will change in your case
GO
```

	FileId	PageId	Row	Level	Column5 (key)	Column1 (key)	UNIQUIFIER (key)	KeyHashValue	Row Size
420	1	2601	419	0	998	998	0	(200614c56b24)	12
421	1	2601	420	0	999	999	0	(fd012cc1e4dc)	12
422	1	2601	421	0	1000	1000	0	(032844fc31f3)	12
423	1	2601	422	0	1001	1001	0	(de2f7cf8be0b)	12

Observation: SQL Server will look at this page for records where **Column5 = 1000**. But this page only contains **Column1** and **Column5** records and we are looking for the entire data (**SELECT ***). Thus in order to find **Column2, Column3** and **Column4** data SQL Server has to go to the clustered index data page. Thus SQL Server will take the clustered index key value (**Column1 (key)** = 1000) and will look (Key Lookup) at clustered index root page and then clustered index data page.

View page allocations

Execute the following statement(s) to view pages allocated to the clustered index

```
-- Step 8: View pages allocated to clustered index
SELECT allocated_page_page_id,page_type_desc,page_level,next_page_page_id,previous_page_page_id
FROM sys.dm_db_database_page_allocations(DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table3'), 1,
NULL, 'DETAILED')
WHERE page_type IN (1,2)
ORDER BY page_level DESC;
GO
```

	allocated_page_page_id	page_type_desc	page_level	next_page_page_id	previous_page_page_id
1	2583	INDEX_PAGE	2	NULL	NULL
2	3648	INDEX_PAGE	1	NULL	3680
3	3680	INDEX_PAGE	1	3648	NULL
4	3712	DATA_PAGE	0	3713	3647
5	3713	DATA_PAGE	0	3714	3712
6	3714	DATA_PAGE	0	3715	3713

Note: Clustered index **index_id** is always **'1'**.

View memory dump of root page

Execute the following statement(s) to view memory dump of clustered index root page (Replace 2583 in the below statement with **allocated_page_page_id** from the output of **step 8** where **page_level** is **'2'**)

```
--Step 9: View memory dump of clustered index root page
DBCC PAGE('SQLMaestros',1,2583,3); -- Page ID will change in your case
GO
```

Editor | Results | Messages

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column1 (key)	UNIQUIFIER (key)	KeyHashValue	Row Size
1	1	2583	0	2	1	3680	NULL	NULL	NULL	14
2	1	2583	1	2	1	3648	50001	0	NULL	14

Observation: The clustered index has three levels and these two pages are in the intermediate level page. 1st intermediate level page contains data where **Column1(key)** is less than 50001 thus SQL Server will go to page with **ChildPageid** 3680

View memory dump of intermediate level page

Execute the following statement(s) to view memory dump of clustered index intermediate level page (Replace 3832 with **ChildPageId** from the output of **step 9** which contains **Column1 = 1000**)

```
--Step 10: View memory dump of clustered index intermediate level page
DBCC PAGE('SQLMaestros',1,3680,3); -- Page ID will change in your case
GO
```

Editor | Results | Messages

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column1 (key)	UNIQUIFIER (key)	KeyHashValue	Row Size
7	1	3680	6	1	1	3622	733	0	NULL	14
8	1	3680	7	1	1	3623	855	0	NULL	14
9	1	3680	8	1	1	3624	977	0	NULL	14
10	1	3680	9	1	1	3625	1099	0	NULL	14
11	1	3680	10	1	1	3626	1221	0	NULL	14
12	1	3680	11	1	1	3627	1343	0	NULL	14

Observation: SQL Server will look at the content of this page and will go to the row where **Column1 (key) = 1000**. Page with **ChildPageId** 3624 contains data where **Column1** is between 997 and 1098.

View memory dump of leaf level page

Execute the following statement(s) to view memory dump of clustered index leaf level page (Replace 3624 in the below statement with **ChildPageId** in the output of **step 10** where **Column1** contains 1000)

```
--Step 11: View memory dump of clustered index leaf level page
DBCC PAGE('SQLMaestros',1,3624,3); -- Page ID will change in your case
GO
```

<table>
<tr>
<td></td>
<td>

```
KeyHashValue = (4f2714d49a1a)
Slot 23 Offset 0x620 Length 64

Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 64
Memory Dump @0x000000006F95A620

0000000000000000:   30002e00 e8030000 64617461 20202020 20202020  0...è...data
0000000000000014:   20202020 20202020 20202020 20202020 20209f41                   A
0000000000000028:   4c00e803 00000600 00020037 00400053 4a535554  L.è........7.@.SJSUT
000000000000003C:   52445056                                      RDPV

Slot 23 Column 0 Offset 0x0 Length 4 Length (physical) 0
UNIQUIFIER = 0
Slot 23 Column 1 Offset 0x4 Length 4 Length (physical) 4

Column1 = 1000
Slot 23 Column 2 Offset 0x8 Length 30 Length (physical) 30

Column2 = data
Slot 23 Column 3 Offset 0x37 Length 9 Length (physical) 9

Column3 = SJSUTRDPV
Slot 23 Column 4 Offset 0x26 Length 4 Length (physical) 4

Column4 = 4997535
Slot 23 Column 5 Offset 0x2a Length 4 Length (physical) 4

Column5 = 1000
Slot 23 Offset 0x0 Length 0 Length (physical) 0
```

Observation: Finally SQL Server will go to the data row where **Column5 = 1000** and fetches all the records from there.
</td>
</tr>
<tr>
<td>Close all the query windows</td>
<td>Close all the query windows ( ) and if SSMS asks to save changes, click NO</td>
</tr>
</table>

Summary

In this exercise, you have learnt:

- How SQL Server finds data with the help of non-clustered index key and clustered index key.
- Concept of key/bookmark lookup.

Exercise 4: Covering Index

Scenario

In this exercise, we will understand the concept of covering index.

<table>
<tr><th>Tasks</th><th>Detailed Steps</th></tr>
<tr><td>Open 4_CoveringIndex.sql</td><td>1. Click File | Open | File or press (Ctrl + O)
2. Navigate to C:\vLabs\
3. Select 4_CoveringIndex.sql and click Open</td></tr>
<tr><td>Execute the statement(s) in the Setup section to setup table</td><td>The setup section performs the following:
• Table4 table is created with 100000 records.
• Clustered Index CL_Table4_Column1 is created on Column1 column of Table4 table.

In 1_CoveringIndex.sql, Review and execute the statement(s) in section ‘Begin: Setup’ and ‘End: Setup’

<pre>

-- Begin: Setup

USE SQLMaestros;
SET NOCOUNT ON;
GO

-- Create Table4 table in SQLMaestros database
CREATE TABLE [SQLMaestros].[Table4](
   Column1 INT,
   Column2 CHAR(30),
   Column3 VARCHAR(30),
</pre></td></tr>
</table>

<table>
<tr><td></td><td>

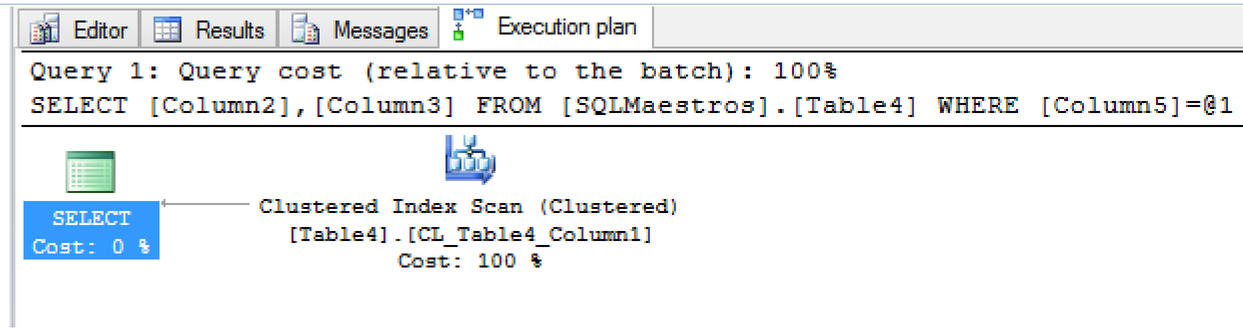
```
   Column4 INT,
   Column5 INT);
GO

-- Insert 100000 records in Table4 table
DECLARE @COUNT INT;
SET @COUNT = 1;
WHILE @COUNT < 100001
BEGIN
DECLARE @DATA2 INT;
SET @DATA2 = ROUND(10000000*RAND(),0);
DECLARE @RAND VARCHAR(30)
SET @RAND = (select
char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(ran
d()*26+65)+char(rand()*26+65)+char(rand()*26+65)+char(rand()*26+65));
INSERT INTO [SQLMaestros].[Table4] VALUES(@COUNT,'data', @RAND, @DATA2, @COUNT);
SET @COUNT = @COUNT + 1;
END
GO

-- Create a clustered index on Column1 column of Table4 table
CREATE CLUSTERED INDEX CL_Table4_Column1 ON [SQLMaestros].[Table4](Column1);
GO

---------------------
-- End: Setup
---------------------
```

</td></tr>
<tr><td>Execute a SELECT statement</td><td>

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) against **Table4** table

```
-- Step 1: Execute below select query with actual execution plan (Ctrl + M)
SET STATISTICS IO ON;
SELECT Column2,Column3 FROM [SQLMaestros].[Table4] WHERE Column5 = 1000;
GO
```

</td></tr>
</table>



Observation: If we look at the message section in the output, we will see that the **SELECT** query is taking 824 logical I/O. In the **Execution Plan** tab we can see that it's doing a clustered index scan operation. We have our clustered index on **Column1** column of **Table4** table but the search predicate is on **Column5** column. Thus SQL Server does not know the location of data where **Column5 = 1000**. In order to find the record it has to look at every clustered index data page, thus we are getting a scan instead of seek.

CREATE a non-clustered index

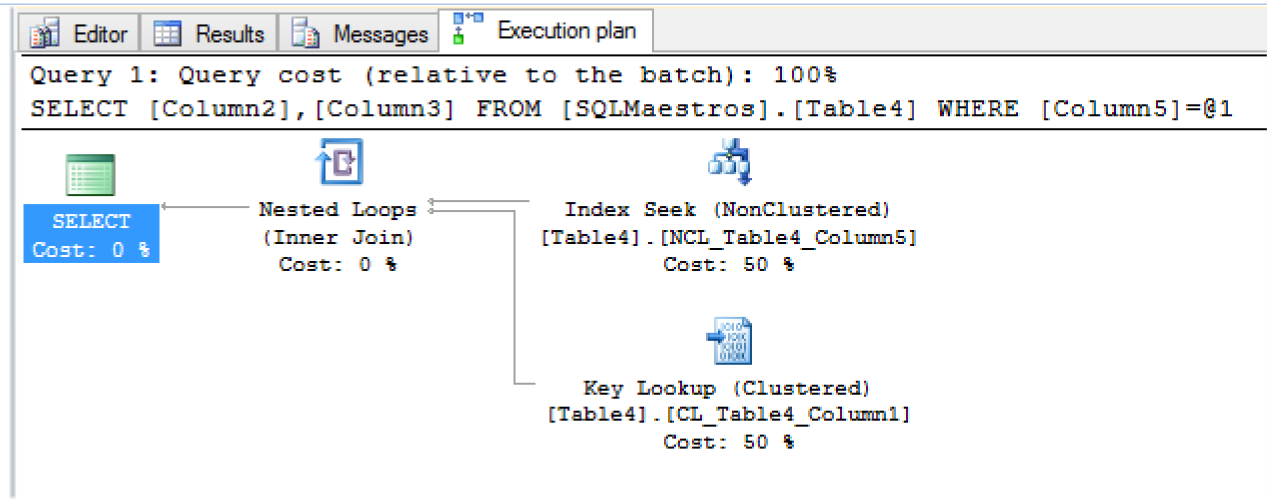
Execute the following statement(s) to **CREATE** a non-clustered index on **Column5** column of **Table4** table

```
-- Step 2: Create a non-clustered index on Column5 column of Table4 table
CREATE NONCLUSTERED INDEX NCL_Table4_Column5 ON [SQLMaestros].[Table4](Column5);
GO
```

Execute a **SELECT** statement

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) against **Table4** table

```
-- Step 3: Execute below select query with actual execution plan (Ctrl + M)
SELECT Column2,Column3 FROM [SQLMaestros].[Table4] WHERE Column5 = 1000;
GO
```



Observation: If we look at the message section in the output, we will see that the **SELECT** query is taking 5 logical I/O; previously it was taking 824 logical I/O. If we look at the **Execution Plan** tab we can see that it's doing a non-clustered index seek + Key Lookup. We have a non-clustered index on **Column5** column and the search predicate is also in **Column5** column, thus SQL Server is doing a non-clustered index seek where it directly goes to non-clustered index root page and finds the leaf page from there (Where **Column5 = 1000**) and then it goes to non-clustered index leaf page. We are selecting **Column2** and **Column4** column, thus SQL Server needs to fetch the **Column2** and **Column3** data where **Column5 = 1000**. But other column data is not there in the non-clustered index leaf page. Thus in order to fetch the other columns data it has to go in the data page (In this case clustered index leaf page) and this is what called Key Lookup where storage engine takes the clustered index key from the non-clustered index leaf page and goes to the clustered index leaf/data page and retrieves rest of the data.

CREATE a covering non-clustered index

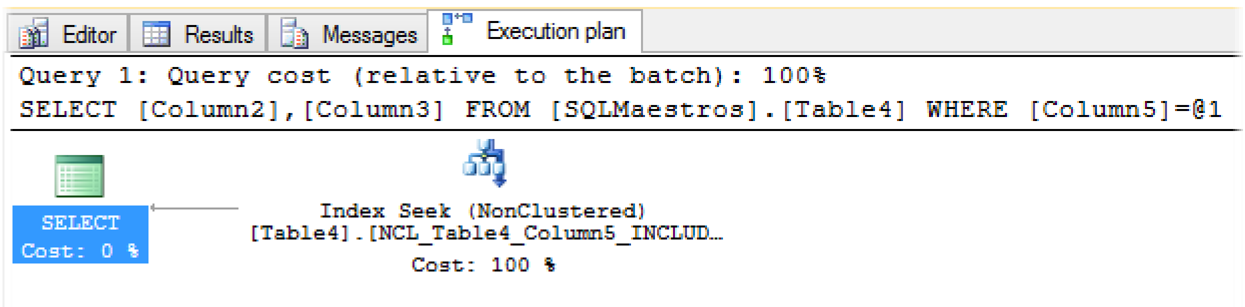
Execute the following statement(s) to **CREATE** a covering non-clustered index on **Table4** table

```
-- Step 4: Create a non-clustered index on Column5 column of Table4 table
CREATE NONCLUSTERED INDEX NCL_Table4_Column5_INCLUDE_Column2_Column3 ON [SQLMaestros].[Table4](Column5)
INCLUDE(Column2, Column3);
GO
```

Explanation: The **SELECT** query is selecting **Column2** and **Column3** where **Column5 = 1000**. In the above non-clustered index we have created the non-clustered index on **Column5** (Search Predicate) and have included **Column2** and **Column3** in that. This is what is known as covering index where we cover the entire **SELECT** query by including the columns needed to display.

Execute a **SELECT** statement

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) against **Table4** table



Observation: If we look at the message section in the output, you can see that the **SELECT** query is taking 3 logical I/O; previously it was taking 5 logical I/O. If we look at the **Execution Plan** tab we can see that it's doing a non-clustered index seek. We have a non-clustered index on **Column5** column with **Column2** and **Column3** columns as included columns and the search predicate is also in **Column5** column, thus SQL Server is doing a non-clustered index seek where it directly goes to non-clustered index root page and finds the leaf page from there (Where **Column5 = 1000**) and then it goes to non-clustered index leaf page and fetches all the required data as **Column2** and **Column3** column is covered in this index.

View index details

Execute the following statement(s) to view the covering index details

```
-- Step 6: Execute below query to find non-clustered index index_id
SELECT name,index_id FROM sys.indexes WHERE name = 'NCL_Table4_Column5_INCLUDE_Column2_Column3';
GO
```

View page allocations

Execute the following statement(s) to view pages allocated to the non-clustered index (Replace **<index_id>** in the below statement with **index_id** we get from **step 6**)

```
-- Step 7: View pages allocated to non-clustered index NCL_Table4_Column5_INCLUDE_Column2_Column3
SELECT allocated_page_page_id,page_type_desc,page_level,next_page_page_id,previous_page_page_id
FROM sys.dm_db_database_page_allocations(DB_ID(N'SQLMaestros'), OBJECT_ID(N'SQLMaestros.Table4'),
<index_id>, NULL, 'DETAILED')
WHERE page_type IN (1,2)
ORDER BY page_level DESC;
GO
```

Editor | Results | Messages | Execution plan

	allocated_page_page_id	page_type_desc	page_level	next_page_page_id	previous_page_page_id
1	2576	INDEX_PAGE	2	NULL	NULL
2	3304	INDEX_PAGE	1	NULL	3336
3	3336	INDEX_PAGE	1	3304	NULL
4	3368	INDEX_PAGE	0	3369	3303
5	3369	INDEX_PAGE	0	3370	3368
6	3370	INDEX_PAGE	0	3371	3369
7	3371	INDEX_PAGE	0	3372	3370

View memory dump of root page

Execute the following statement(s) to view the memory dump of the root page of covering index (Replace 2576 in the below statement with **allocated_page_page_id** from the output of **step 7** where **page_level** is **'2'**)

```
-- Step 8: View memory dump of root page of non-clustered index
DBCC TRACEON(3604)
DBCC PAGE('SQLMaestros',1,2576,3); -- Page ID will change in your case
GO
```

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column5 (key)	Column1 (key)	UNIQUIFIER (key)
1	1	2576	0	2	1	3336	NULL	NULL	NULL
2	1	2576	1	2	1	3304	50001	50001	0

Observation: The non-clustered index has three levels and these two pages are in the intermediate level page. The 1st intermediate level page contains data where **Column5(key)** is less than 50001, thus SQL Server will go to the page with **page_id** 3336

View memory dump of intermediate level page

Execute the following statement(s) to view memory dump of non-clustered index intermediate page (Replace 3336 in the below statement with **ChildPageId** from the output of **step 8** where **Column5** contains 1000)

```
--Step 9: View memory dump of intermediate level non-clustered index
DBCC PAGE('SQLMaestros',1,3336,3); -- Page ID will change in your case
GO
```

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Column5 (key)	Column1 (key)	UNIQUIFIER (key)	Ke
6	1	3336	5	1	1	3277	686	686	0	N
7	1	3336	6	1	1	3278	823	823	0	N
8	1	3336	7	1	1	3279	960	960	0	N
9	1	3336	8	1	1	3280	1097	1097	0	N
10	1	3336	9	1	1	3281	1234	1234	0	N
11	1	3336	10	1	1	3282	1371	1371	0	N

Observation: SQL Server will look at the content of this page and will go to the row where Column5 (key) contains 1000. In the above output **ChildPageId** 3279 contains data for **Column5** column where **Column5** lies between 960 and 1096. Thus SQL Server will go to this page.

View memory dump of leaf level page

Execute the following statement(s) to view the memory dump of non-clustered index leaf level page (Replace 3279 in the below statement with **ChildPageId** in the output of **step 9** where **Column1** contains the value 1000)

```
--Step 10: View memory dump of leaf level non-clustered index
DBCC PAGE('SQLMaestros',1,3279,3); -- Page ID will change in your case
GO
```

Editor | Results | Messages

	FileId	PageId	Row	Level	Column5 (key)	Column1 (key)	UNIQUIFIER (key)	Column2	Column3
39	1	3279	38	0	998	998	0	data	BYZVMGUOI
40	1	3279	39	0	999	999	0	data	BXPNYUYKY
41	1	3279	40	0	1000	1000	0	data	JYOSFYZWB
42	1	3279	41	0	1001	1001	0	data	LOPBYYWZG
43	1	3279	42	0	1002	1002	0	data	SDMOXZOOZ

Observation:
Since we have included **Column2** and **Column3** column in our non-clustered index, thus SQL Server will search this page and will fetch data where **Column5 (key) = 1000**. If we run the above **SELECT** query again we will get this output.

Cleanup

After completing this lab, please execute the following statement(s) to drop **SQLMaestros** database

```
--------------------
-- Begin: Cleanup
--------------------
USE [master]
GO
ALTER DATABASE [SQLMaestros] SET  SINGLE_USER WITH ROLLBACK IMMEDIATE
GO
DROP DATABASE [SQLMaestros]
GO

--------------------
-- End: Cleanup
```

Close all the query windows	Close all the query windows (X) and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, you have learnt:

- Concept of non-clustered index with included column.
- Concept of covering index.
- How covering index works.