**An Introduction to Clustered and Non-Clustered Index Data Structures**

By [G. Vijayakumar](http://www.sql-server-performance.com/author/g.vijayakumar/)

When I first started using SQL Server as a novice, I was initially confused as to the differences between clustered and non-clustered indexes. As a developer, and new DBA, I took it upon myself to learn everything I could about these index types, and when they should be used. This article is a result of my learning and experience, and explains the differences between clustered and non-clustered index data structures for the DBA or developer new to SQL Server. If you are new to SQL Server, I hope you find this article useful.

As you read this article, if you choose, you can cut and paste the code I have provided in order to more fully understand and appreciate the differences between clustered and non-clustered indexes.

Part I: Non-Clustered Index

**Creating a Table**

To better explain SQL Server non-clustered indexes; let’s start by creating a new table and populating it with some sample data using the following scripts. I assume you have a database you can use for this. If not, you will want to create one for these examples.

Create Table DummyTable1  
(  
EmpId Int,  
EmpName Varchar(8000)  
)

When you first create a new table, there is no index created by default. In technical terms, a table without an index is called a “heap”. We can confirm the fact that this new table doesn’t have an index by taking a look at the sysindexes system table, which contains one for this table with an of indid = 0. The sysindexes table, which exists in every database, tracks table and index information. “Indid” refers to Index ID, and is used to identify indexes. An indid of 0 means that a table does not have an index, and is stored by SQL Server as a heap.

Now let’s add a few records in this table using this script:

Insert Into DummyTable1 Values (4, Replicate (‘d’,2000))  
GO

Insert Into DummyTable1 Values (6, Replicate (‘f’,2000))  
GO

Insert Into DummyTable1 Values (1, Replicate (‘a’,2000))  
GO

Insert Into DummyTable1 Values (3, Replicate (‘c’,2000))  
GO

Now, let’s view the contests of the table by executing the following command in Query Analyzer for our new table.

Select EmpID From DummyTable1  
GO

|  |
| --- |
| **Empid** |
| 4 |
| 6 |
| 1 |
| 3 |

As you would expect, the data we inserted earlier has been displayed. Note that the order of the results is in the same order that I inserted them in, which is in no order at all.

Now, let’s execute the following commands to display the actual page information for the table we created and is now stored in SQL Server.

dbcc ind(dbid, tabid, -1) – This is an undocumented command.

DBCC TRACEON (3604)  
GO

Declare @DBID Int, @TableID Int  
Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable1′)

DBCC ind(@DBID, @TableID, -1)  
GO

This script will display many columns, but we are only interested in three of them, as shown below.

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26408 | 0 | 10 |
| 26255 | 0 | 1 |
| 26409 | 0 | 1 |

Here’s what the information displayed means:

PagePID is the physical page numbers used to store the table. In this case, three pages are currently used to store the data.

IndexID is the type of index,

Where:

0 – Datapage

1 – Clustered Index

2 – Greater and equal to 2 is an Index page (Non-Clustered Index and ordinary index),

PageType tells you what kind of data is stored in each database,

Where:

10 – IAM (Index Allocation MAP)

1 – Datapage

2 – Index page

Now, let us execute DBCC PAGE command. This is an undocumented command.

DBCC page(dbid, fileno, pageno, option)

Where:

dbid = database id.

Fileno = fileno of the page. Usually it will be 1, unless we use more than one file for a database.

Pageno = we can take the output of the dbcc ind page no.

Option = it can be 0, 1, 2, 3. I use 3 to get a display of the data. You can try yourself for the other options.

Run this script to execute the command:

DBCC TRACEON (3604)  
GO

DBCC page(@DBID, 1, 26408, 3)  
GO

The output will be page allocation details.

DBCC TRACEON (3604)   
GO

dbcc page(@DBID, 1, 26255, 3)  
GO

The data will be displayed in the order it was entered in the table. This is how SQL stores the data in pages. Actually, 26255 & 26409 both display the data page.

I have displayed the data page information for page 26255 only. This is how MS SQL stores the contents in data pages as such column name with its respective value.

Record Type = PRIMARY\_RECORD

EmpId = 4

EmpName = ddddddddddddddddddddddddddddddddddddddddddddddddddd  
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd

Record Type = PRIMARY\_RECORD

EmpId = 6

EmpName = ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Record Type = PRIMARY\_RECORD

EmpId = 1

EmpName = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

This displays the exact data storage in SQL, without any index on table. Now, let’s go and create a unique non-clustered index on the EmpID column.

**Creating a Non-Clustered Index**

Now, we will create a unique non-clustered index on the empid column to see how it affects the data, and how the data is stored in SQL Server.

CREATE UNIQUE NONCLUSTERED INDEX DummyTable1\_empid  
ON DummyTable1 (empid)  
GO

Now, execute the DBCC ind (dbid, tabid, -1)

DBCC TRACEON (3604)   
GO

Declare @DBID Int, @TableID Int  
Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable1′)  
DBCC ind(@DBID, @TableID, -1)  
GO

Here are the results:

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26408 | 0 | 10 |
| 26255 | 0 | 1 |
| 26409 | 0 | 1 |
| 26411 | 2 | 10 |
| 26410 | 2 | 2 |

Now, we see two more rows than before, which now contains index page details. Page 26408 displays the page allocation details, and pages 26255 and 26409 display the data page details, as before.

In regard to the new pages, page 26411 displays the page allocation details of an index page and page 26410 displays the index page details.

MS SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Let us see what would be the output for page 26411, that is page type = 10

IAM: Single Page Allocations @0x308A608E

—————————————–

Slot 0 = (1:26410)

Let us view page 26410 to see the index page details.

DBCC TRACEON (3604)  
GO

DBCC page(10, 1, 26410, 3)  
GO

SQL populates the index column data in order. The last column (?) is pointed to the row locator.

Here are the results, using two different methods:

**Method I**

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **EMPID** | **?** |
| 1 | 26410 | 1 | 0x8F66000001000200 |
| 1 | 26410 | 3 | 0×2967000001000000 |
| 1 | 26410 | 4 | 0x8F66000001000000 |
| 1 | 26410 | 6 | 0x8F66000001000100 |

The row location display in one of two ways:

* If the table does not have a clustered index, the row locator will be combination of fileno, pageno and the no of rows in a page.
* If the table does have clustered index, the row location will be clustered index key value.

Non-clustered indexes are particularly handy when we want to return a single row from a table.

For example, to search for employee ID (empid = 3) in a table that has a non-clustered index on the empid column, SQL Server looks through the index to find an entry that lists the exact page and row in the table where the matching empid can be found, and then goes directly to that page and row. This greatly speeds up accessing the record in question.

Select EmpID, EmpName From DummyTable1 WHERE EMPID = 3

Now, let’s insert some more rows in our table and view the data page storage of our non-clustered index.

Insert Into DummyTable1 Values (10, Replicate (‘j’,2000))  
GO  
  
Insert Into DummyTable1 Values (2, Replicate (‘b’,2000))  
GO

Insert Into DummyTable1 Values (5, Replicate (‘e’,2000))  
GO

Insert Into DummyTable1 Values (8, Replicate (‘h’,2000))  
GO

Insert Into DummyTable1 Values (9, Replicate (‘i’,2000))  
GO

Insert Into DummyTable1 Values (7, Replicate (‘g’,2000))  
GO

Now, let’s view the data in our table.

Execute:

Select EmpID From DummyTable1

Here are the results:

|  |
| --- |
| **EmpID** |
| 4 |
| 6 |
| 1 |
| 3 |
| 10 |
| 2 |
| 5 |
| 8 |
| 9 |
| 7 |

As you may notice above, the data is still in the order we entered it, and not in any particular order. This is because adding the non-clustered index didn’t change how the data was stored and ordered on the data pages.

Now, let’s view the results of the DBCC IND command. In order to find out what happened when the new data was added to the table.

DBCC TRACEON (3604)   
GO

Declare @DBID Int, @TableID Int  
Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable1′)  
DBCC ind(@DBID, @TableID, -1)  
GO

Here are the results:

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26408 | 0 | 10 |
| 26255 | 0 | 1 |
| 26409 | 0 | 1 |
| 26412 | 0 | 1 |
| 26413 | 0 | 1 |
| 26411 | 2 | 10 |
| 26410 | 2 | 2 |

Let us execute the page 26410 again and get the index page details.

DBCC TRACEON (3604)   
GO

dbcc page(10, 1, 26410, 3)  
GO

SQL Server populates the index column data in order. The last column (?) is pointed to the row locator.

Here are the results:

**Method I**

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **EMPID** | **?** |
| 1 | 26410 | 1 | 0x8F66000001000200 |
| 1 | 26410 | 2 | 0x2C67000001000000 |
| 1 | 26410 | 3 | 0×2967000001000000 |
| 1 | 26410 | 4 | 0x8F66000001000000 |
| 1 | 26410 | 5 | 0x2C67000001000100 |
| 1 | 26410 | 6 | 0x8F66000001000100 |
| 1 | 26410 | 7 | 0x2D67000001000000 |
| 1 | 26410 | 8 | 0x2C67000001000200 |
| 1 | 26410 | 9 | 0×2967000001000200 |
| 1 | 26410 | 10 | 0×2967000001000100 |

As I explained earlier, there are two types of row locations. We have seen Method I. Now, let’s try Method II with the help of a clustered and non-clustered index in a table. DummyTable1 already has a non-clustered index. Let’s now add a new column to the DummyTabl1 table and add a clustered index on that column.

Alter Table DummyTable1 Add EmpIndex Int IDENTITY(1,1)  
GO

This will link the clustered index key value, instead of **the row locator, and be will the combination of fileno, pageno and no of rows in a page.**

**This adds the Empindex column to DummyTable1. I have used an identity column so that we will not have null values on that column.**

**You can execute the DBCC ind and DBCC page to check if there any change after the new column is added to the table. If you don’t want to check this yourself, I can tell you that adding the new column did not affect the total number of pages currently allocated to the table by SQL Server.**

**Now, let’s add a unique clustered index on the empindex column and then view the differences in page 26410.**

**First, we execute the DBCC ind command. This displays a new set of pages for dummytable1.**

**DBCC TRACEON (3604)   
GO**

**Declare @DBID Int, @TableID Int  
Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable1′)  
DBCC ind(@DBID, @TableID, -1)  
GO**

**Here are the results:**

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26415 | 1 | 10 |
| 26414 | 0 | 1 |
| 26416 | 1 | 2 |
| 26417 | 0 | 1 |
| 26418 | 0 | 1 |
| 26420 | 2 | 10 |
| 26419 | 2 | 2 |

**Pages 26415 and 26420 have page allocation details. Pages 26414, 26417 and 26418 have data page details.**

**Now, let’s view pages 26416 and 26419 and see the output.**

**DBCC TRACEON (3604)   
GO**

**DBCC page(10, 1, 26416, 3)  
GO**

Here are the results:

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **ChildPageID** | **EMPID** |
| 1 | 26416 | 26414 | 0 |
| 1 | 26416 | 26417 | 5 |
| 1 | 26416 | 26418 | 9 |

This displays the output of the clustered index page, which has got a link to data page (ChildPageID). EMPID is an index column that contains the starting row of the page.

DBCC TRACEON (3604)   
GO

DBCC page(10, 1, 26419, 3)  
GO

Here are the results:

**Method II**

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **EMPID** | **EMPIndex** |
| 1 | 26419 | 1 | 1 |
| 1 | 26419 | 2 | 2 |
| 1 | 26419 | 3 | 3 |
| 1 | 26419 | 4 | 4 |
| 1 | 26419 | 5 | 5 |
| 1 | 26419 | 6 | 6 |
| 1 | 26419 | 7 | 7 |
| 1 | 26419 | 8 | 8 |
| 1 | 26419 | 9 | 9 |
| 1 | 26419 | 10 | 10 |

It is interesting to see the differences now. There is a difference between **Method I** and **Method II**. **Method II** is now linked to a clustered index key.

The main difference between **Method I** and **Method II** is the link to a row in a data page.

Part II: Clustered Index

**Creating a Table**

To better explain how SQL Server creates clustered indexes; let’s start by creating a new table and populating it with some sample data using the following scripts. You can use the same sample database as before.

Create Table DummyTable2

(  
EmpId Int,  
EmpName Varchar(8000)  
)

As in the previous example, when you first create a new table, there is no index created by default, and a heap is created. As before, we can confirm the fact that this new table doesn’t have an index by taking a look at the sysindexes system table, which contains one for this table with an of indid = 0. The sysindexes table, which exists in every database, tracks table and index information. “Indid” refers to Index ID, and is used to identify indexes. An indid of 0 means that a table does not have an index, and is stored by SQL Server as a heap.

Now let’s add a few records in this table using this script:

Insert Into DummyTable2 Values (4, Replicate (‘d’,2000))  
GO

Insert Into DummyTable2 Values (6, Replicate (‘f’,2000))  
GO

Insert Into DummyTable2 Values (1, Replicate (‘a’,2000))  
GO

Insert Into DummyTable2 Values (3, Replicate (‘c’,2000))  
GO

Now, let’s view the contents of the table by executing the following command in Query Analyzer for our new table.

Select EmpID From DummyTable2  
GO

|  |
| --- |
| **Empid** |
| 4 |
| 6 |
| 1 |
| 3 |

As you would expect, the data we inserted has been displayed. Note that the order of the results is in the same order that I inserted them in, which is in no order at all.

Now, let’s execute the following commands to display the actual page information for the table we created and is now stored in SQL Server.

DBCC ind(dbid, tabid, -1) – It is an undocumented command.

DBCC TRACEON (3604)   
GO

Declare @DBID Int, @TableID Int  
Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable2′)  
DBCC ind(@DBID, @TableID, -1)  
GO

This script will display many columns, but we are only interested in three of them, as shown below.

Here are the results:

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26408 | 0 | 10 |
| 26255 | 0 | 1 |
| 26409 | 0 | 1 |

Here’s what the information displayed means:

PagePID is the physical page numbers used to store the table. In this case, three pages are currently used to store the data.

IndexID is the type of index,

Where:

0 – Datapage

1 – Clustered Index

2 – Greater and equal to 2 is an Index page (Non-Clustered Index and ordinary index)

PageType tells you what kind of data is stored in each database

Where:

10 – IAM (Index Allocation MAP)

1 – Datapage

2 – Index page

Now, let us execute DBCC PAGE command.

DBCC page(dbid, fileno, pageno, option)

Where:

dbid = database id.

Fileno = fileno of the page. Usually it will be 1, unless we use more than one file for a database.

Pageno = we can take the output of the dbcc ind page no.

Option = it can be 0, 1, 2, 3. I use 3 to get a display of the data. You can try yourself for the other options.

Run this script to execute the command:

DBCC TRACEON (3604)   
GO

DBCC page(@DBID, 1, 26408, 3)  
GO

The output will be page allocation details.

DBCC TRACEON (3604)   
GO

DBCC page(@DBID, 1, 26255, 3)  
GO

The output will display the data however it was entered in the table. This is how SQL stores the data in pages. Actually, 26255 & 26409 will display the data page.

I have displayed the data page information for page 26255 only. This is how MS-SQL stores the contents in data pages as such column name with its respective value.

Record Type = PRIMARY\_RECORD

EmpId = 4

EmpName = dddddddddddddddddddddddddddddddddddddddddddddddd  
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd  
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd

Record Type = PRIMARY\_RECORD

EmpId = 6

EmpName = ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Record Type = PRIMARY\_RECORD

EmpId = 1

EmpName = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

This displays the exact data storage in SQL without any index on table. Now, let’s go and create a Unique Clustered Index on EmpID column.

**Create a Clustered Index**

Now, let us create a unique clustered index on empid column to see how it affects the data that is stored in SQL Server.

CREATE UNIQUE CLUSTERED INDEX DummyTable2\_EmpIndex  
ON DummyTable2 (EmpID)  
GO

Execute:

Select EmpID From DummyTable2

Here are the results:

|  |
| --- |
| **Empid** |
| 1 |
| 3 |
| 4 |
| 6 |

Now, execute the DBCC ind (dbid, tabid, -1)

DBCC TRACEON (3604)  
GO

Declare @DBID Int, @TableID Int

Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable2′)

DBCC ind(@DBID, @TableID, -1)  
GO

Here are the results:

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26411 | 1 | 10 |
| 26410 | 0 | 1 |
| 26412 | 1 | 2 |

MS SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Now, let us view the page 26410 and 26412 and see the page details.

DBCC TRACEON (3604)  
GO

DBCC page(10, 1, 26412, 3)  
GO

Here are the results:

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **ChildPageID** | **EMPID** |
| 1 | 26412 | 26410 | 0 |

The output display many columns, but we are only interested in four of them as shown above.

This will display the output of the index page, which has got link to data page (ChildPageID). EMPID is an index column will contain the starting row of the page.

Now, let us view the page 26410 and see the page details.

DBCC TRACEON (3604)  
GO

DBCC page (10, 1, 26410, 3)  
GO

Here are the results:

Record Type = PRIMARY\_RECORD

EmpId = 1

EmpName = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Record Type = PRIMARY\_RECORD

EmpId = 2

EmpName = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb  
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb  
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb  
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb  
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

Record Type = PRIMARY\_RECORD

EmpId = 3

EmpName = cccccccccccccccccccccccccccccccccccccccccccccccc  
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc  
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc  
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc  
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

Though I have added disorder records, SQL has displayed the data page in sequence because we have got a clustered index on empid. This is absolutely great! Adding a clustered index to the table has physically reordered the data pages, putting them in physical order based on the indexed column.

Now, let’s insert some more rows in our table and view the data and index page storage of our clustered index.

Insert Into DummyTable2 Values (10, Replicate (‘j’,2000))  
GO

Insert Into DummyTable2 Values (2, Replicate (‘b’,2000))  
GO

Insert Into DummyTable2 Values (5, Replicate (‘e’,2000))  
GO

Insert Into DummyTable2 Values (8, Replicate (‘h’,2000))  
GO

Insert Into DummyTable2 Values (9, Replicate (‘i’,2000))  
GO

Insert Into DummyTable2 Values (7, Replicate (‘g’,2000))  
GO

Now, execute the DBCC ind (dbid, tabid, -1)

DBCC TRACEON (3604)  
GO

Declare @DBID Int, @TableID Int

Select @DBID = db\_id(), @TableID = object\_id(‘DummyTable2′)

DBCC ind(@DBID, @TableID, -1)  
GO

Here are the results:

|  |  |  |
| --- | --- | --- |
| **PagePID** | **IndexID** | **PageType** |
| 26411 | 1 | 10 |
| 26410 | 0 | 1 |
| 26412 | 1 | 2 |
| 26255 | 0 | 1 |
| 26408 | 0 | 1 |
| 26409 | 0 | 1 |

Now, we see few more rows than before. Page 26411 displays the page allocation details, and pages 26408, 26409, 26410 and 26255 display the data page details, as before.

In regard to the new pages, page 26411 displays the page allocation details of an index page and 26412 displays the index page details.

MS-SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Let us see what would be the output for page 26411, that is page type = 10.

DBCC TRACEON (3604)  
GO

dbcc page(10, 1, 26411, 3)  
GO

Here are the results:

IAM: Single Page Allocations @0x30A5C08E

—————————————–

Slot 0 = (1:26410)

Slot 1 = (1:26412)

Slot 2 = (1:26255)

Slot 3 = (1:26408)

Slot 4 = (1:26409)

Let us view page 26412 to see the index page details.

DBCC TRACEON (3604)  
GO

DBCC page(10, 1, 26412, 3)  
GO

Here are the results:

|  |  |  |  |
| --- | --- | --- | --- |
| **FileID** | **PageID** | **ChildPageID** | **EMPID** |
| 1 | 26412 | 26410 | 0 |
| 1 | 26412 | 26408 | 4 |
| 1 | 26412 | 26255 | 6 |
| 1 | 26412 | 26409 | 9 |

This helps us to get an idea to decide the need of clustered index. It is really useful to have a clustered index when retrieve many rows of data, ranges of data, and when BETWEEN is used in the WHERE clause. Because, the leaf level of the clustered index is the data. It should be used to save many I/Os. So, it is better to use clustered indexes to solve queries asking for ranges of data, not one row.

For example, to search for an employee ID (empid between 3 and 9) in a table that has a clustered index on the empid column.

Select EmpID, EmpName From DummyTable1 WHEREEMPID Between 3 And 9