About Indexes just for Fun and Understanding

In this chapter, we use DBCC IND and DBCC PAGE to provide a better understanding about indexes. We don't learn much new, but hopefully better at understanding index structures and therefore better at creating the best indexes in a given situation. Perhaps we also understand why some of the claims are wrong and must therefore be forgotten and not repeated to others.

In the examples, we will use rows that take up about 850 bytes. We can have 9 to 10 rows on each side. We use the ID, Firstname, Lastname and Address columns and add the Txt column to have wide rows so we can see what happens when we insert new data, delete a row, ….

The table used for the first example is created as follows.

```
CREATE TABLE dbo.t
(
        ID              INT             NOT NULL
                        CONSTRAINT PK_t PRIMARY KEY,
        Firstname       VARCHAR (20)    NOT NULL,
        Lastname        VARCHAR (20)    NOT NULL,
        Address         VARCHAR (30)    NULL,
        Txt             CHAR (800)      NOT NULL
                        CONSTRAINT DF_t_Filler DEFAULT (REPLICATE ('A', 800))
);
```

We insert the following 10 rows and create a NonClustered index on the Address column. The table has the ID column as PRIMARY KEY. This constraint is created as a Clustered Index.

```
INSERT INTO dbo.t (ID, Firstname, Lastname, Address) VALUES
        (1, 'Ida', 'Hansen', 'Vestergade'),  (2, 'Hans Ole', 'Olsen', NULL),      (3, 'Ane', 'Petersen', 'Strandvejen'),      (4, 'Per', 'Karlsen', NULL),
        (5, 'Susan', 'Larsen', 'Nygade'),    (6, 'Mia', 'Poulsen', 'Pouls Plads'), (7, 'Lars', 'Andersen', 'Anegade'),    (8, 'Tina', 'Olsen', 'Torvet'),
        (9, 'Carl', 'Iversen', 'Vestergade'), (10, 'Bent', 'Olsen', NULL);

CREATE NONCLUSTERED INDEX nc_t_Address ON dbo.t (Address);
```

First we look at the command DBCC IND, The first parameter are the database name, the second the name of the table and the last parameter are the index_id from the system table sys.indexes. To send output from the commands to the client/SSMS we must set trace 3604. Index_id = 1 is the Clustered Index and index_id = 2 is the created NonClustered Index nc_t_Address. The 4 file_id columns are removed from the result tables below because we only have one file, object_id is removed because it is always data about the table dbo.t that is shown. Information about the partition is removed because we only have a table with one partition. Unnecessary columns are removed to provide a better overview.

```
DBCC TRACEON (3604);

DBCC IND ('TestDB', 'dbo.t', 1);
DBCC IND ('TestDB', 'dbo.t', 2);
```

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
334	NULL	1	In-row data	10	NULL	0	0
352	334	1	In-row data	1	0	354	0
353	334	1	In-row data	2	1	0	0
354	334	1	In-row data	1	0	0	352

The following information can be read from the above table, which is the result from selecting data about the Clustered Index.

- The table is a Clustered Index, but still have an IAM page - page 334, PageType = 10.
- Page 353 is the root level page- IndexLevel = 1, PageType = 2.
- Pages 352 and 354 are leaf level pages and are linked together - NextPagePID and PrevPagePID.

It shows that data from the Clustered Index can be selected by using the ClusterKey or by using the IAM if the optimizer choose a Scan operation because of the hint TABLOCK/TABLOCKX. When using the Clustered Index, the data is selected in ClusterKey order if the operation is not performed in parallel.

Next we look at data about the NonClustered Index nc_t_Address.

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
377	NULL	2	In-row data	10	NULL	0	0
384	377	2	In-row data	2	0	0	0

From the table of data about the NonClustered Index we can see that

- The index has an IAM page - PageType= 10.
- This index has only the root page which is also the leaf level - we have only 10 rows in the table.

We can look at data from page 384 , the root/leaf level page for the NonClustered Index. We use the command DBCC PAGE. With the last parameter set to 3, we see the details about the index. Again, columns are removed.

```
DBCC PAGE ('TestDB', 1, 384, 3);
```

PageId	Row	Level	Address (key)	ID (key)	Row Size
384	0	0	NULL	2	8
384	1	0	NULL	4	8
384	2	0	NULL	10	8
384	3	0	Anegade	7	19
384	4	0	Nygade	5	18
384	5	0	Pouls Plads	6	23
384	6	0	Strandvejen	3	23
384	7	0	Torvet	8	18
384	8	0	Vestergade	1	22
384	9	0	Vestergade	9	22

We can see that data in the column Address is ordered and NULL comes first in the sort order. We can also see that the ClusterKey - the ID column - is added as an IndexKey.

We change the table to a HEAP. We create the index on the column Address but do not have a PRIMARY KEY defined for the table.

PageId	Row	Level	Address (key)	HEAP RID (key)	Row Size
360	0	0	NULL	0x6001000001000100	12
360	1	0	NULL	0x6001000001000300	12
360	2	0	NULL	0x6101000001000000	12
360	3	0	Anegade	0x6001000001000600	23
360	4	0	Nygade	0x6001000001000400	22
360	5	0	Pouls Plads	0x6001000001000500	27
360	6	0	Strandvejen	0x6001000001000200	27
360	7	0	Torvet	0x6001000001000700	22
360	8	0	Vestergade	0x6001000001000000	26
360	9	0	Vestergade	0x6001000001000800	26

Now we have a RID value that points to the data in the HEAP. The RID can be used to join data from this index with data from another index if the optimizer choose this to solve a problem using more than one index.

If we create a NonClustered Index on the LastName column, we get the following result. We look at the first row in the table below and the fourth row in the previous table. Both have the RID value

0x6001000001000600

so both point to the row 7/Lars/Andersen/Anegade.

PageId	Row	Level	Lastname (key)	HEAP RID (key)	Row Size
392	0	0	Andersen	0x6001000001000600	24
392	1	0	Hansen	0x6001000001000000	22
392	2	0	Iversen	0x6001000001000800	23
392	3	0	Karlsen	0x6001000001000300	23

392	4	0	Larsen	0x6001000001000400	25
392	5	0	Olsen	0x6001000001000100	21
392	6	0	Olsen	0x6001000001000700	21
392	7	0	Olsen	0x6101000001000000	21
392	8	0	Petersen	0x6001000001000200	24
392	9	0	Poulsen	0x6001000001000500	23

The following data is from the HEAP when executing DBCC IND.

PagePID	IAMPID	ObjectID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
334	NULL	901578250	0	In-row data	10	NULL	0	0
352	334	901578250	0	In-row data	1	0	0	0
353	334	901578250	0	In-row data	1	0	0	0

We can see that the pages in the HEAP are not linked together, as the two columns with references to the data pages - pages with PageType = 1 - do not have a value in the NextPagePID and PrevPagePID columns. Only the IAM is specified and can be used for a scan.

For being able to look more at the output from DBCC IND we create a table and insert the data from the DBCC IND command. In this example we create a very badly defined table, but not a problem because the table is only used for evaluation and understanding. Of course, if data is to be used in production, the table should be defined more carefully. Since DBCC IND is one of the undocumented commands, it should never be used in production, so this example shows that it is possible to spend very little time on the uninteresting and instead concentrate on learning from the result. The used table looks like this.

```
CREATE TABLE dbo.Ind
(
        PageFID         BIGINT,
        PagePID         BIGINT,
        IAMFID          BIGINT,
        IAMPID          BIGINT,
        ObjectID        BIGINT,
        IndexID         BIGINT,
        PartitionNumber BIGINT,
        PartitionID     BIGINT,
        iam_chain_type  VARCHAR (30),
        PageType        BIGINT,
        IndexLevel      BIGINT,
        NextPageFID     BIGINT,
        NextPagePID     BIGINT,
        PrevPageFID     BIGINT,
        PrevPagePID     BIGINT
);
```

The system function sys.dm_db_database_page_allocations give the same information and returns a table.

We will look at the table dbo.Person with about 15,000,000 rows. We store about 170,000 rows in the table dbo.Ind if the following statement is executed.

```
INSERT INTO dbo.Ind
        EXEC ('DBCC IND (IndexDB, Person, 1)');
```

With the following statements we can see the overall structure of the table.

```
SELECT  ix.name,
        ps.object_id,
        ps.index_type_desc,
        ps.index_depth,
        ps.index_level,
        ps.page_count,
        ps.record_count
        FROM sys.dm_db_index_physical_stats (DB_ID (), OBJECT_ID ('Person'), NULL, NULL, 'DETAILED')    AS ps
                INNER JOIN sys.indexes AS Ix            ON      ps.object_id = ix.object_id     AND
                                                                ps.index_id = ix.index_id;
```

name	object_id	index_type_desc	index_depth	index_level	page_count	record_count
PK_Person	1109578991	CLUSTERED INDEX	3	0	170813	15034218
PK_Person	1109578991	CLUSTERED INDEX	3	1	279	170813
PK_Person	1109578991	CLUSTERED INDEX	3	2	1	279

Let's try to look at some of the data from the table dbo.Ind. If we select all the IAM pages - PageType = 10 - we get the following result where four rows are returned.

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
16	NULL	1	In-row data	10	NULL	13	0
13	NULL	1	In-row data	10	NULL	8	16
8	NULL	1	In-row data	10	NULL	12	13
12	NULL	1	In-row data	10	NULL	0	8

We can see that with this size of a table, there are four IAM pages and that they are linked together. PagePID 16 is the first page, because we do not have any PrevPagePID.

If we look at the intermediate levels we get the following result. Not all of the 279 rows are shown.

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
8672	12	1	In-row data	2	1	8673	122527
8673	12	1	In-row data	2	1	8674	8672
8674	12	1	In-row data	2	1	8675	8673
8675	12	1	In-row data	2	1	117488	8674
10512	12	1	In-row data	2	1	10513	117495
…	…	…	…	…	…	…	…

We can see that all pages at the intermediate level are linked together. If we look at all levels we can see that there are no references between the levels. There is an indirect reference by looking at the first row at a level. This is a reference to the first row at the next level. There is no column containing this information.

In the next example we look at what happens if we do not have a unique column in the table. The table is created the following way.

```
CREATE TABLE dbo.t
(
        ID              INT             NOT NULL,
        Firstname       VARCHAR (20)    NOT NULL,
        Lastname        VARCHAR (20)    NOT NULL,
        Address         VARCHAR (30)    NULL,
        Filler          CHAR (800)      NOT NULL
                        CONSTRAINT DF_t_Filler DEFAULT (REPLICATE ('A', 8))
);
```

We insert the following data into the table.

```
INSERT INTO dbo.t (ID, Firstname, Lastname, Address) VALUES
        (10, 'Ida', 'Hansen', 'Vestergade'),    (20, 'Hans Ole', 'Olsen', NULL),    (30, 'Ane', 'Petersen', 'Strandvejen'),
        (40, 'Per', 'Karlsen', NULL),           (50, 'Susan', 'Larsen', 'Nygade'),  (60, 'Mia', 'Poulsen', 'Pouls Plads'),
        (70, 'Lars', 'Andersen', 'Anegade'),    (80, 'Tina', 'Olsen', 'Torvet'),    (99, 'Carl', 'Iversen', 'Vestergade'),
        (100, 'Bent', 'Olsen', NULL);
```

We create two indexes on the table. A Clustered Index on the Firstname column and a NonClustered on the column Address. It is important to note that the ClusterKey is not unique.

```
CREATE CLUSTERED INDEX nc_t_Firstname ON dbo.t (Firstname);
CREATE NONCLUSTERED INDEX nc_t_Address ON dbo.t (Address);
```

When we use DBCC IND to see which pages are in use, the last parameter tells us which index to return data about. This table has a Clustered Index, but both 0 and 1 are accepted as the index_id. A table has index_id = 0 if the table is stored as a HEAP and index_id = 1 if the table is stored as a Clustered Index. But never both. If we execute the command and specify index_id = 10

```
DBCC IND ('TestDB', 'dbo.t', 10);
```

we get the following error message because this index does not exist. So for NonClustered Index only existing index_id values are allowed.

```
Msg 7999, Level 16, State 7, Line 111
Could not find any index named '10' for table 't'.
```

One of the following two commands should give a similar error, but this does not happen. We know that a table is either stored as a HEAP or as a Clustered Index, but both commands execute without error.

```
DBCC IND ('TestDB', 'dbo.t', 1);
DBCC IND ('TestDB', 'dbo.t', 0);
```

It appears that the first command executes using the Clustered Index and includes the IAM as a row in the result. There is also the root page for the Clustered Index - PageType = 2/IndexLevel = 1

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
335	NULL	1	In-row data	10	NULL	0	0
360	335	1	In-row data	1	0	368	0
368	335	1	In-row data	1	0	0	360
400	335	1	In-row data	2	1	0	0

For the second command, the output shows that it is executed as if the table is stored as a HEAP. We saw earlier that a table with a Clustered Index also has an IAM page and that this can be used to select data. The root side of the clustered index is ignored and does not appear in the result. IAM only refers to the leaf level pages.

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
335	NULL	1	In-row data	10	NULL	0	0
360	335	1	In-row data	1	0	368	0
368	335	1	In-row data	1	0	0	360

We look at the data using DBCC PAGE. Remember that the ClusterKey is not unique. We look at details from page 400, the root page of the Clustered Index.

```
DBCC PAGE ('TestDB', 1, 400, 3);
```

As shown in the result, we have two rows because there are two leaf-level pages and no intermediate pages. We can also see that a column named UNIQUIFIER has been added to make each row in an index unique. This is a numeric column. The first leaf-level page - page 360 - starts with FirstName NULL and the second page - page 368 - starts with Tina. We can also see that Tina is the first Tina because the value of UNIQUIFIER is 0.

PageId	Row	Level	ChildFileId	ChildPageId	Firstname (key)	UNIQUIFIER (key)	Row Size
400	0	1	1	360	NULL	NULL	14
400	1	1	1	368	Tina	0	15

Let's add two rows with a Firstname values less than Tina.

```
INSERT INTO dbo.t (ID, Firstname, Lastname, Address) VALUES
            (12, 'Ida', 'Hansen', 'Vestergade'),
            (13, 'Hans Ole', 'Olsen', NULL);
```

We can see that it results in a page split. NextPagePID for page 360 is now page 401 and not page 368 as before. The new page 401 has the 'old' page 368 as the NextPagePID. So, we can see, that page 401 is inserted in logical order between the 2 original pages and the logical order is now 360 -> 401 -> 368. This is a way to prove that data is not stored in physical order. These three pages are even stored in three different extents. So, REBUILD/REORGANIZE should be performed. The new page is a page in the same extent as the root page.

PagePID	IAMPID	IndexID	iam_chain_type	PageType	IndexLevel	NextPagePID	PrevPagePID
335	NULL	1	In-row data	10	NULL	0	0
360	335	1	In-row data	1	0	401	0
368	335	1	In-row data	1	0	0	401
400	335	1	In-row data	2	1	0	0
401	335	1	In-row data	1	0	368	360

We look at the index nc_t_Address. The next two examples show how SQLServer will optimize how data is stored. Not information that is of much use to the developer, but just to show that sometimes something happens behind the scenes and that not everything is common knowledge, but still helps for better performance.

```
CREATE TABLE dbo.t
(
        ID              INT             NOT NULL,
        Firstname       VARCHAR (20)    NOT NULL,
        Lastname        VARCHAR (20)    NOT NULL,
        Address         VARCHAR (30)    NULL,
        Filler          CHAR (800)      NOT NULL
                        CONSTRAINT DF_t_Filler DEFAULT (REPLICATE ('A', 8))
);
```

On the table we will create two indexes, a Clustered Index and a NonClustered. None of the index are unique. Then we insert 320 rows, where the same name and address is repeated – the rows are created by 'GO 5'.

```
CREATE CLUSTERED INDEX nc_t_Firstname ON dbo.t (Firstname);

CREATE NONCLUSTERED INDEX nc_t_Address ON dbo.t (Address, ID);
GO
INSERT INTO dbo.t (ID, Firstname, Lastname, Address) VALUES
        (10, 'Ida', 'Hansen', 'Vestergade'),        (20, 'Hans Ole', 'Olsen', NULL),    (30, 'Ane', 'Petersen', 'Strandvejen'),
        (40, 'Per', 'Karlsen', NULL),               (50, 'Susan', 'Larsen', 'Nygade'),  (60, 'Mia', 'Poulsen', 'Pouls Plads'),
        (70, 'Lars', 'Andersen', 'Anegade'),        (80, 'Tina', 'Olsen', 'Torvet'),    (90, 'Carl', 'Iversen', 'Vestergade'),
        (100, 'Bent', 'Olsen', NULL);
GO
INSERT INTO dbo.t (ID, Firstname, Lastname, Address)
        SELECT  ID + (SELECT            MAX (ID)
                        FROM dbo.t),
                Firstname,
                Lastname,
                Address
                FROM dbo.t;
GO 5
```

We look at the data about the NonClustered Index nc_t_Address.

```
DBCC IND ('TestDB', 'dbo.t', 2);
```

PagePID	IAMPID	IndexID	PageType	IndexLevel	NextPagePID	PrevPagePID
378	NULL	2	10	NULL	0	0
352	378	2	2	0	354	0
353	378	2	2	1	0	0
354	378	2	2	0	0	352

First we look at the root page. Seen from the above, the root page is page 353 - PageType = 2 and PageLevel = 1. The first leaf level page is 352. For this row all the Key Columns are NULL. The second of the leaf level pages are page 354. On this page the first row is 50/Susan/Nygade. Because we do not have a unique Clustered Index we also have the UNIQUIFIER column. The ClusterKey is the column Firstname.

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
353	0	1	352	NULL	NULL	NULL	NULL
353	1	1	354	Nygade	50	Susan	0

PageId	Row	Level	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
352	0	0	NULL	100	Bent	0
…	…	…	…	…	…	…

PageId	Row	Level	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
354	0	0	Nygade	50	Susan	0
354	1	0	Nygade	150	Susan	1
354	2	0	Nygade	250	Susan	2
354	3	0	Nygade	350	Susan	3
354	4	0	Nygade	450	Susan	4
354	5	0	Nygade	550	Susan	5
…	…	…	…	…	…	…

If we look at the first rows from page 354 we can see, that the first row is 50/Susan/Nygade.

If we delete the row 50/Susan/Nygade, the row is removed from the leaf level, but the link from the root page/page 353 is not updated. This update is 'saved away' because it doesn't matter to the quality of data and for the navigation in the index. After the delete the reference at the root page is to a none existing row, but with a value lower than all the values on the page and higher than the values on the previous logical page if such one exists. Therefore, the reference on the root page is ID = 50, but the first row on page 354 is ID 150.

```
DELETE
        FROM dbo.t
        WHERE ID = 50;
```

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
353	0	1	352	NULL	NULL	NULL	NULL
353	1	1	354	Nygade	50	Susan	0

PageId	Row	Level	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
354	0	0	Nygade	150	Susan	1
354	1	0	Nygade	250	Susan	2
354	2	0	Nygade	350	Susan	3
354	3	0	Nygade	450	Susan	4
354	4	0	Nygade	550	Susan	5
…	…	…	…	…	…	…

The same principle is used for the first row on the root page. All key values are NULL and therefore no rows can have lower values. If we want to insert a row with values between 50/Sanne/Nygade and 150/Sanne/Nygade, a Seek in the index tells us that the index row should be inserted on page 354. Then the 'normal' operation for inserting a row is performed. Maybe there is room on the page or maybe a page split needs to be made. If we insert 1080 more rows, we can look at the root page and see that there are now 6 pages at leaf level. We can also see that there has been a page split, as page 356 has been inserted between the two pages 352 and 354, which were the original leaf level pages in the index. This shows that page 352 has been split. It can also be ascertained that the values from root level to the leaf level for page 354 are still to the missing data.

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
353	0	1	352	NULL	NULL	NULL	NULL
353	1	1	356	NULL	2200	Bent	21
353	2	1	354	Nygade	50	Susan	0
353	3	1	358	Pouls Plads	2761	Mia	59
353	4	1	355	Strandvejen	2630	Ane	26
353	5	1	357	Vestergade	690	Carl	6

If we REBUILD the index all data is moved to new pages. The root page is now page 424.

```
ALTER INDEX nc_t_Address ON dbo.t REBUILD;
```

If we look at the root page, we can now see, that we still have 6 pages, but other pages than before REBUILD. We can see that all pages are new pages. Resources are used to move data.

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
424	0	1	384	NULL	NULL	NULL	NULL
424	1	1	392	NULL	2520	Hans Ole	25
424	2	1	393	Nygade	1350	Susan	13
424	3	1	394	Strandvejen	634	Ane	102
424	4	1	395	Vestergade	113	Ida	65
424	5	1	396	Vestergade	2993	Carl	93

Instead of REBUILD we can execute the following statement where data is reorganized.

```
ALTER INDEX nc_t_Address ON dbo.t REORGANIZE;
```

Before the REORGANIZE statement is executed, the root page - PageId = 353 - contains the following rows. We can see, that if data is selected from more than one page in logical order, the next page is often a page from another extent. The example shows swap between two extents, the blue extent with the pages 352 to 359 and the green extent with the pages 560 to 567. Consider what happens if a table has 100,000 rows.

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
353	0	1	352	NULL	NULL	NULL	NULL
353	1	1	560	NULL	1104	Bent	106
353	2	1	356	NULL	2200	Bent	21
353	3	1	561	Anegade	171	Lars	33
353	4	1	564	Anegade	1674	Lars	112
353	5	1	354	Nygade	50	Susan	0
353	6	1	565	Nygade	3153	Susan	93
353	7	1	358	Pouls Plads	2761	Mia	59
353	8	1	359	Strandvejen	1138	Ane	203
353	9	1	355	Strandvejen	2630	Ane	26
353	10	1	563	Torvet	1881	Tina	50
353	11	1	357	Vestergade	690	Carl	6
353	12	1	562	Vestergade	1910	Ida	19

After REORGANIZATION, the pages are arranged in logical order. If several pages are to be accessed, they are often pages from the same extent. The most important thing about reorganization is that the external fragmentation is removed. The benefit of reorganizing data depends on how the data is used. The data is in logical order on each page and the problem arises if we need data from previous or next page in logical order. The pointer to the previous and next page can be to a page in a different extent, but better for performance if it is a page in the same extent.

PageId	Row	Level	ChildPageId	Address (key)	ID (key)	Firstname (key)	UNIQUIFIER (key)
353	0	1	352	NULL	NULL	NULL	NULL
353	1	1	354	NULL	1244	Per	108
353	2	1	355	NULL	2509	Bent	248
353	3	1	356	Anegade	1374	Lars	109
353	4	1	357	Nygade	1350	Susan	13
353	5	1	358	Pouls Plads	1066	Mia	170
353	6	1	359	Strandvejen	638	Ane	198
353	7	1	560	Strandvejen	2630	Ane	26
353	8	1	561	Torvet	1881	Tina	50
353	9	1	562	Vestergade	690	Carl	6
353	10	1	563	Vestergade	1910	Ida	19

As the last example in this chapter, we look at indexes with INCLUDE columns. We have a Clustered Index on the table for the PRIMARY KEY. We create three NonClustered Index, where one is without INCLUDE of columns and two with INCLUDE of columns. We look at the physical structures using DBCC IND and DBCC PAGE, but we also look at the result from DBCC SHOW_STATISTICS. There is a discrepancy between what the different commands shows.

```
CREATE TABLE dbo.t
(
        ID              INT             NOT NULL
                        CONSTRAINT PK_t PRIMARY KEY,
        Firstname       VARCHAR (20)    NOT NULL,
        Lastname        VARCHAR (20)    NOT NULL,
        Address         VARCHAR (30)    NULL,
        Zipcode         SMALLINT        NULL,
        Filler          CHAR (800)      NOT NULL
                        CONSTRAINT DF_t_Filler DEFAULT (REPLICATE ('A', 8))
);
GO
INSERT INTO dbo.t (ID, Firstname, Lastname, Address, Zipcode) VALUES
        (10, 'Ida', 'Hansen', 'Vestergade', 2000),          (20, 'Hans Ole', 'Olsen', NULL, NULL),
        (30, 'Ane', 'Petersen', 'Strandvejen', 2000),       (40, 'Per', 'Karlsen', NULL, NULL),
        (50, 'Susan', 'Larsen', 'Nygade', 3000),            (60, 'Mia', 'Poulsen', 'Pouls Plads', 4000),
        (70, 'Lars', 'Andersen', 'Anegade', 3000),          (80, 'Tina', 'Olsen', 'Torvet', 2000),
        (90, 'Carl', 'Iversen', 'Vestergade', 4000),        (100, 'Bent', 'Olsen', NULL, NULL),
        (110, 'Hans Ole', 'Olsen', NULL, NULL),             (120, 'Ane', 'Petersen', 'Torvet', 2000),
        (130, 'Per', 'Karlsen', NULL, NULL);
GO
CREATE NONCLUSTERED INDEX nc_t_Address_Zipcode ON dbo.t (Address, Zipcode);
CREATE NONCLUSTERED INDEX nc_t_Address_5 ON dbo.t (Address) INCLUDE (Zipcode);
CREATE NONCLUSTERED INDEX nc_t_Address_5_1 ON dbo.t (Address) INCLUDE (Zipcode, ID);
```

First we look at the index nc_t_Address_Zipcode which do not have INCLUDE columns.

If we look at the result table from the following command, we can see that the column ID, which is the ClusterKey, is added as the last IndexKey column. The word (key) is by the system added to the column names for each of the Key columns. Only a few rows from the page are shown because it is the column names that are of interest in this example. Page 360 is both the root and leaf level of the index.

```
DBCC PAGE ('TestDB', 1, 360, 3);
```

PageId	Row	Level	Address (key)	Zipcode (key)	ID (key)
360	0	0	NULL	NULL	20
360	1	0	NULL	NULL	40
360	2	0	…	…	…

If we look at the result from the following command

```
DBCC SHOW_STATISTICS ('dbo.t', 'nc_t_Address_Zipcode') WITH DENSITY_VECTOR;
```

All density	Average Length	Columns
0,1428571	5,153846	Address
0,125	6,384615	Address, Zipcode
0,07692308	10,38461	Address, Zipcode, ID

We can see that density is calculated for all three IndexKey columns. Hopefully as expected.

Next we look at the following index which contains the same columns but the column Zipcode is now an INCLUDE column.

```
CREATE NONCLUSTERED INDEX nc_t_Address_5 ON dbo.t (Address) INCLUDE (Zipcode);
```

The result from DBCC PAGE and DBCC SHOW_STATISTICS are

PageId	Row	Level	Address (key)	ID (key)	Zipcode
392	0	0	NULL	20	NULL
392	1	0	NULL	40	NULL
392	2	…	…	…	…

All density	Average Length	Columns
0,1428571	5,153846	Address
0,07692308	9,153847	Address, ID

The ClusterKey column ID is added but as the second IndexKey Column. The column Zipcode are the third column in the index because it is not an IndexKey Column but an INCLUDE column. Both results above shows that Address and ID are IndexKey Columns.

The third index contains the same columns but now both Zipcode and ID are Include Columns. Remember, that ID is the Cluster Key.

```
CREATE NONCLUSTERED INDEX nc_t_Address_5_1 ON dbo.t (Address) INCLUDE (Zipcode, ID);
```

Now we have a difference in the results from the two commands.

PageId	Row	Level	Address (key)	ID (key)	Zipcode
400	0	0	NULL	20	NULL
400	1	0	NULL	40	NULL
400	2	0	…	…	…

We again have the column ID as an IndexKey Column together with the column Address even if the column is specified as an INCLUDE column in the CREATE INDEX statement.

The result from DBCC SHOW_STATISTICS shows that there is only one IndexKey Column in the table. The result above shows two IndexKey columns.

All density	Average Length	Columns
0,1428571	5,153846	Address

In the index statistics, density is normally calculated for all IndexKey columns. But not in this situation. By specifying the ID as an Include Column, we save resources when calculating statistics. Only statistics for the Address column are calculated as seen from DBCC SHOW_STATISTICS.

Why is the ID column still listed as a key column when we look at the output from the DBCC PAGE? It's just a guess - but a good guess - that it's optimization. Storing data in a NonClustered Index in order of ClusterKey means that if we select many rows with the same value in the IndexKey column, the following lookup can look up all the desired rows from one page/extent before jumping to the next page.

If this order is not implemented, we must jump around between all the pages of the Clustered Index and perhaps return to a previously used page or extent. Better performance requires that the Clustered Index have low external fragmented. The columns to the left in the following table shows ordered ID values and the columns to the right have unordered values. Imagine that the example returns at least 1000 rows in the result.

Key Column Value
…
Vvvvvvvvvvv
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Xxxxx
Yyyyyyyyy
…

ClusterKey	Cluster Page	Extent
…	…	
…	…	
23	1:234	29
26	1:234	29
32	1:234	29
33	1:234	29
36	1:234	29
51	1:238	29
53	1:238	29
54	1:238	29
57	1:238	29
59	1:238	29
118	1:499	62
120	1:499	62
125	1:499	62
167	1:588	73
171	1:589	73
…	…	
…	…	

ClusterKey	Cluster Page	Extent
…	…	
…	…	
32	1:234	29
53	1:238	29
54	1:238	29
125	1:499	62
171	1:589	73
23	1:234	29
118	1:499	62
57	1:238	29
33	1:234	29
167	1:588	73
36	1:234	29
51	1:238	29
59	1:238	29
26	1:234	29
120	1:499	62
…	…	
…	…	

Some of the above examples show that SQL Server optimizes in ways that are not immediately known to us. We must go into the details to investigate what is happening. Knowledge that is probably reserved for the few, but included to show that more unknown optimizations are taking place.

The conclusion of this optimization tells us that jumping between pages is expensive, so it is 'necessary' for the system to add this optimization behind the scenes. We must assume that all optimization is done for a researched and evaluated reason and not just for fun. This knowledge can be used when we create tables and need to figure out which column should be the ClusterKey column. As an example, we can look at an OrderLine table. If the table has two candidate keys, where one is a surrogate/autonumber column and the other is a composite key consisting of (OrderID, ProductID), the composite key containing (OrderID, ProductID) must be the ClusterKey. The surrogate/autonumber column must be defined as UNIQUE. It must be assumed that it is often all the rows for one Order that are read. The surrogate key as PRIMARY KEY will spread OrderLines for an Order across the extents/pages used for the table, if not all OrderLines for an Order are inserted at the same time or inserted mixed with inserting OrderLines from other Orders in a multi-user system. If the PRIMARY KEY is (OrderID, ProductID), fragmentation can occur when we have mixed insert of OrderLines from several Orders at the same time. Therefore, the assessment must include how the data is used afterwards.

The next version may have new and better ways to fix a problem to improve performance. Therefore, be careful not to 'play' optimizer. SQL Server usually handles it well. If and when we try to optimize, we need to both have a deeper knowledge of how SQL Server works and also be able to assess whether it improves performance. Different solutions can give a difference in performance without having an impact on returning a correct result.

Column Order in a MultiColumn Index

In this example, we will look at the order of columns in a MultiColumn Index. Unfortunately, it is widely believed that the column with the greatest selectivity should be the first column in an index, etc. This is not correct. In the following examples, we look at how the order should be chosen.

We look at an example that we have used earlier in the book. For the table dbo.Person, we will create indexes so that we can search on the Firstname column, only search on the Lastname column or search on both columns at the same time. We can choose to create 3 indexes, an index that is optimal for each of the three options. The choice may be to create only two indexes to minimize the number of indexes and then have fewer indexes to maintain.

Therefore, we can choose between one of the following options

- An index on Firstname and a MultiColumn Index on Lastname, Firstname
- An index on Lastname and a MultiColumn Index on Firstname, Lastname

The choice between the two possibilities can be decided based on whether it is the Firstname column that is often used in a Seek alone, or whether it is the Lastname column. The MultiColumn Index is then determined by this choice and not by the selectivity of the two columns. However, there is no problem when seeking using both columns at the same time if the conditions is with =/equal to. We can sketch an example where we have to search for Firstname = 'Lisbet' and Lastname = 'Jensen'.

First we look at an index where Firstname is the first column and Lastname is the second column. Via root and the intermediate levels, we will point to the page with the first row for Firstname = 'Lisbet' and Lastname = 'Jensen'. All IndexKey columns are included in the index from the root level, so we look for a combination of both values and not just Firstname = 'Lisbet'. It can be the first row on the page, the last row or one row in the middle, that we are looking for.

5 rows are selected from leaf level. The 5th row, which is the green row in the sketch, is Lisbet/Karlsen and we stop searching, as no more rows exists with the name Lisbet/Jensen.

Firstname	Lastname
Anders	Andersen
…	…
Anders	Thomsen
Bo	Andersen
…	…
Karl	Carlsen
Karl	Didriksen
Lisbet	Frandsen
Lisbet	Jensen
Lisbet	Jensen
Lisbet	Jensen
Lisbet	Jensen
Lisbet	Karlsen
…	…

When we change the order of the columns in the definition of the index, still only 5 rows are read and 4 rows are returned.

Lastname	Firstname
Andersen	Anders
…	…
Ibsen	Tom
Jensen	Adam
Jensen	Karin
Jensen	Lisbet
Jensen	Lisbet
Jensen	Lisbet
Jensen	Lisbet
Jensen	Mie
…	…

To evaluate this further, the following indexes are being created.

```
CREATE INDEX nc_Person_Firstname_Lastname ON dbo.Person (Firstname, Lastname);
CREATE INDEX nc_Person_Lastname_Firstname ON dbo.Person (Lastname, Firstname);
```

To evaluate, we execute the following statement with forced use of each of the two indexes. We will use index hints to be sure which index is used in the evaluation.

```
SELECT    PersonID,
          Firstname,
          Lastname
          FROM dbo.Person     WITH (INDEX = nc_Person_Firstname_Lastname)
                    --        WITH (INDEX = nc_Person_Lastname_Firstname)
          WHERE     Firstname = 'Lisbet'    AND
                    Lastname = 'Jensen';
```

If we look at the execution plans to determine the estimate for the number of rows, we can see that both statements estimate to return 3288.31 rows. The execution plans in XML show this and have the same value for both plans.

```
<Statements>
  <StmtSimple StatementCompId="2" StatementEstRows="3288.31" StatementId="1" StatementOptmLevel="TRIVIAL"
```

When we execute both statements and look at the result from SET STATISTICS IO ON, we get the following results. This shows that it is the same number of pages - 31/32 - which is accessed. The difference can be one page depending on, if the first Lisbet/Jensen is one of the first rows on the first page used or one of the last rows on the first page. It is shown with read of 8 rows where each page have 5 rows. If data is not fragmented, the difference in the number of pages read is 0 or 1.

Page	Row	Row
1		
1	**1**	
1	**2**	
1	**3**	
1	**4**	**1**
2	**5**	**2**
2	**6**	**3**
2	**7**	**4**
2	**8**	**5**
2		**6**
3		**7**
3		**8**
3		
3		
3		

7695 rows affected)
Table 'Person'. Scan count 1, logical reads 32, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

(7695 rows affected)
Table 'Person'. Scan count 1, logical reads 31, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

Since both indexes are MultiColumn, we can look at the statistics for the different combinations of the columns. We can see for the combination of Firstname, Lastname and the combination Lastname, Firstname, that All density has the same value. We can also see that there is a big difference between the All density for Firstname and the All density for Lastname. However, as shown above, it does not affect the result for estimates of rows and for the execution plan. The statistics is used only by the optimizer to evaluate if the index should be use and if the index should be used to a Seek or to a Scan. The number of pages read are the same.

```
DBCC SHOW_STATISTICS (Person, nc_Person_Firstname_Lastname) WITH DENSITY_VECTOR;
DBCC SHOW_STATISTICS (Person, nc_Person_Lastname_Firstname) WITH DENSITY_VECTOR;
```

All density	Average Length	Columns
0,002906977	4,856819	Firstname
0,0001322402	11,66913	Firstname, Lastname
6,176464E-08	15,66913	Firstname, Lastname, PersonID

All density	Average Length	Columns
0,02	6,812311	Lastname
0,0001322402	11,66913	Lastname, Firstname
6,176464E-08	15,66913	Lastname, Firstname, PersonID

SQL Server suggests missing index and we can learn from that. In the example we are looking at queries that have at least two conditions combined with AND. The conditions can either be equal to (=) or an interval (<, <=, >, >=, BETWEEN). We are using the dbo.Person table with approximately 16,000,000 rows. There are 1916 different values in the column Created and 4 different values in the column Persontype. The values in the Created column are evenly distributed, but for the Persontype column, approximately 99% of the rows are A, 1% of the rows are B, 0.1% have the value C, and 0.01% of the rows have the value D.

For the proposed missing indexes, the names of the indexes are changed, but not the content and order of columns. With the first two statements, we can see that SQL Server suggests the two columns in different order in the index.

```
SELECT    Created,
          Persontype,
          Gender
          FROM dbo.Person
          WHERE     Persontype = 'B'        AND
                    Created BETWEEN '2020-09-10' AND '2020-09-26';

CREATE NONCLUSTERED INDEX ix_1
          ON dbo.Person (Persontype, Created) INCLUDE (Gender);
```

The index proposal is with the columns in reverse order if we perform the following statement.

```
SELECT    Created,
          Persontype,
          Gender
          FROM dbo.Person
          WHERE     Persontype BETWEEN 'B' AND 'D'              AND
                    Created = '2020-09-10';

CREATE NONCLUSTERED INDEX ix_2
          ON dbo.Person (Created, Persontype) INCLUDE (Gender);
```

We create the first index ix_1 with the columns (Persontype, Created). The first statement will perform a Seek operation. Only Seek Predicates appear in the execution plan. The Seek operation starts with the first row that meets both conditions B/2020-09-01 and stops when the end date is exceeded or if the value of the Persontype column change.

If we use the same index for the second statement, there is both a Seek Predicate and a Predicate. Seek Predicates are the conditions used when looking up/seeking in the index. Predicates are the subsequent filtering of the rows found in the index, either by testing on other IndexKey columns or on INCLUDE columns.

The start position in the index is where Created = '2020-09-10' and Persontype = 'B', which is the Seek Predicate. We need to read past all dates that are greater than the specified date, all dates for Persontype 'C' and all dates from start to and including '2020-09-10' for Persontype = 'D'. There is also a Predicate, which filter/de-selects all the rows of dates that should not be included in the result. As seen from the output below from SET STATISTICS IO ON, the first statement returns more rows, but by accessing fewer pages - 2114 rows from 7 pages compared to 105 rows from 41 pages. It is important to remember that for both statements, the index is a Covered Index, which contains the three columns used by the statement but also that the conditions of the statements are different.

(2114 rows affected)
Table 'Person'. Scan count 1, logical reads 7, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

(105 rows affected)
Table 'Person'. Scan count 1, logical reads 41, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

A sketch of the index shows the problem. The first statement just read the rows from the red box and all data must be used/returned as they all meet the condition. Just a Seek Predicate. The second statement start at the correct position but must read past a lot of rows that are not included in the result, e.g., C/2020-09-11 and D/2019-03-10. Multiple rows must be read, but a filter Predicate is used to deselects the rows that should not be included in the result.

Persontype	Created	First statement	Second statement
…	…		
A	2020-01-07		
…	…		
A	2020-11-28		
B	2019-02-04		
…	…		
B	2020-09-10		
…	…		
B	2020-09-26		
B	2020-09-27		
…	…		
C	2019-02-03		
C	2019-02-04		
…	…		
C	2020-09-10		
C	2020-09-10		
C	2020-09-11		
…	…	.	
D	2019-03-10		
…	…		
D	2020-09-10		
D	2020-09-11		
…	…		

The first conclusion we can come up with is, that the columns to be searched with ‘equal to’/= must come first and the columns to be searched with an interval must come after ‘equal to’ columns as the last IndexKey columns in the index. Never the density or selectivity!

If we have multiple range search conditions as in the following statement, in the missing index proposal, SQL Server chooses to list the columns in the order in which they are created in the table. The order can be seen by selecting the columns from the table sys.columns.

```
SELECT    Created,
          Persontype,
          Gender
          FROM dbo.Person
          WHERE     Persontype BETWEEN 'B' AND 'D'                    AND
                    Zipcode BETWEEN 2000 AND 6000                     AND
                    Created BETWEEN '2020-09-10' AND '2020-09-26';
```

Suggestions for the missing index are

```
CREATE NONCLUSTERED INDEX IX_10
          ON dbo.Person (Zipcode, Persontype, Created) INCLUDE (Gender);
```

The following statement shows the order in which the columns are created/appears in the table.

```
SELECT    STRING_AGG (CONCAT (column_id, ':', name), ', ') WITHIN GROUP (ORDER BY column_id)
          FROM sys.columns
          WHERE object_id = OBJECT_ID ('Person');
```

1:PersonID, 2:Firstname, 3:Lastname, 4:Address, 5:Zipcode, 6:Gender, 7:CountryCode, 8:PhoneNumber,
9:Persontype, 10:CPR, 11:Created, 12: CountryCodePhoneNumber, 13:Name

In the last example, there are two equal expressions and two interval/range expressions. In the proposed index, the columns specified first are the columns included in an 'equal to' expression. Then come the 'interval' columns. Both groups are in column definition order.

```
SELECT    Created,
          Persontype,
          Gender
          FROM dbo.Person
          WHERE     Created BETWEEN '2020-09-10' AND '2020-09-26'      AND
                    Persontype          = 'B'                          AND
                    Zipcode BETWEEN 2000 AND 6000                      AND
                    Gender     = 'F';
```

Suggestions for the missing index are

```
CREATE NONCLUSTERED INDEX IX_30
          ON dbo.Person (Gender, Persontype, Zipcode, Created);
```

It is not the selectivity or the density that is decisive for the order of columns in a MultiColumn Index, but how the columns are used in the condition. If the missing index is used as a suggestion for new indexes, the order in which the IndexKey columns are specified must be considered. Maybe the index can be used for other statements or there can be other indexes already created for the table that must be a part of the consideration. The order of INCLUDE columns does not matter.

If we have the suggestion (Firstname, Lastname) for the Firstname and Lastname columns, we should swap the columns if we already have an index on Firstname. If we do not swap the columns, we do not have a useful index when we only search on Lastname because no index has Lastname as the first column. We have seen that the order of the columns in a MultiColumn Index does not matter and by swapping, we get one more option with only two indexes to be able to perform a Seek. Without swap we can Seek when we search on (Firstname) and (Firstname, Lastname). If we swap we can Seek on (Firstname), seek on (Lastname) and seek on (Lastname, Firstname). Perhaps we are currently not searching on the Lastname column, but it does not mean anything for performance that we also fulfil this option. Perhaps the need will arise later, and thus we will not have a problem that needs to be solved. Alternatively, we can drop the index on the FirstName column because in most cases we don't need this index because we can use the index on (FirstName, LastName).

Many tells that Missing Index should not be used. It is true that it is bad just to create all the indexes that is suggested. Missing Index can be a way of learning which index SQL Server perceives as the best indexes for a statement. The myth that it is the selectivity that determines the order of columns in a MultiColumn Index can be killed. The Missing Index recommendations will also add the ClusterKey columns in the most optimal position and not just as the last columns in the index Key Columns, which is the default position of the ClusterKey columns in a NonClustered Index.

First, the columns where the condition is 'equal to' must be specified, but in the correct order depending on other already defined indexes and not necessarily in the order of definition of the columns in the table. Then the columns with the 'not equal'/range/inequality condition. Therefore, it can make good sense to define two indexes that consist of the same columns, but with the IndexKey columns listed in different orders. It should therefore be considered to remove 'duplicate'/redundant indexes by simply looking at which columns are included. Suggestions for this can be viewed online. Different order of the columns gives different indexes and maybe better performance by having both indexes.

Finally, consideration should be given whether to INCLUDE columns to make the index a Covered Index. If Missing Index recommends INCLUDE columns, they can be ‘changed’ to IndexKey Columns in the index definition to make the index useful for other statements.

We can look at the defined indexes but we can also collect data about the Missing Indexes over a period and then use this information for an analysis. Finding out if some indexes are problematic or maybe one index needs to be dropped and another changed is a manual process that includes, among other things, knowledge of how the table is used. Even what time of day a given index is used can be important information if the load on the server changes throughout the day.

Remember that not all statement should perform as fast as possible, maybe only ‘good enough’. There can be a lot to learn from looking at Missing Indexes and maybe learn, that changing of the indexes will be an advantage. It can be easier to use Missing Index than looking for all possible statements against a table. Data about Missing Index is one of our tools.