Seminar 7

Indexed Views. CASE statement. Columnstore Indexes. Fragmentation - SQL Server -

Indexed Views

Index = a structure associated to a table or a view that optimize the access time to the records of the table or of the view.

Indexed views

- The *first index created* on a view must be a ***unique clustered index***.
- *After* this one can be ***created*** the ***non-clustered indexes***.
- The indexed view can reference only the tables that are in the current database in which is the view

- Creating a unique clustered index on a view ***improves query performance*** (the view is stored in the database in the same way a table with a clustered index is stored).
- The query optimizer may use indexed views to speed up the query execution.

The steps required to create an indexed view (successful implementation of the indexed view):

1. Verify that the SET options are correct for all existing tables that will be referenced in the view.
2. Verify that the SET options for the session are set correctly before you create any tables and the view.
3. Verify that the view definition is deterministic.
4. Create the view by using the WITH SCHEMABINDING option.
5. Create the unique clustered index on the view.

Indexed Views

- To create an ***indexed view***, the view has to be created with the option ***WITH SCHEMABINDING***
- On this view, next, will be created ***a unique clustered index***

→ the obtained view is ***indexed***, or it ***is a view that contains data***

- On this indexed view, can be created ***non-clustered indexes***

- The clause ***WITH SCHEMABINDING*** prevent any modification of the tables that are referenced in the view, thing that could affect the definition of the view
- By using ***WITH SCHEMABINDING***, the SELECT statement from the view definition will contain the name of the tables, views or user-defined functions, having the form **schema_name.object_name**
- To ensure that an indexed view is correctely maintain and returns consistent results, are necessarily the following values that are fixed:

SET options	Required value	Default server value	Default OLE DB and ODBC value	Default DB-Library value
ANSI_NULLS	ON	ON	ON	OFF
ANSI_PADDING	ON	ON	ON	OFF
ANSI_WARNINGS	ON	ON	ON	OFF
ARITHABORT	ON	ON	OFF	OFF
CONCAT_NULL_YIELDS_NULL	ON	ON	ON	OFF
NUMERIC_ROUNDABORT	OFF	OFF	OFF	OFF
QUOTED_IDENTIFIER	ON	ON	ON	OFF

Indexed Views

Restrictions:

- The view cannot reference another view(s)
- The SELECT statement from the view definition cannot contains: subqueries, outer joins, UNION, INTERSECT, EXCEPT, TOP, DISTINCT, ORDER BY, COUNT, MIN, MAX, STDEV, STDEVP, VAR, VARP, AVG, CHECKSUM_AGG, tables from the FROM clause of a SELECT statement
- The definition of the view must be **deterministic**
- A view is ***deterministic*** if all expressions in the select list, including the WHERE and GROUP BY clauses, are deterministic. Deterministic expressions always return the same result independent on the evaluation time with a specific set of input values. Only deterministic functions can participate in deterministic expressions.
  - DATEADD function is *deterministic* because it always returns the same result for any given set of argument values for its three parameters.
  - GETDATE is *not deterministic* because it is always invoked with the same argument, but the value it returns changes each time it is executed.
- The index must be clustered and unique

Indexed Views - example

```
--1. Set the options to support indexed views.
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,  QUOTED_IDENTIFIER, ANSI_NULLS ON;
GO

CREATE TABLE MyTable(Iid INT PRIMARY KEY, Description VARCHAR(50), Quantity INT)
insert into MyTable values (1, 'the first description', 12), (2, 'the second description', 32)

-- 2. Define the view
CREATE VIEW MyView WITH SCHEMABINDING AS
		SELECT Iid, Quantity FROM dbo.MyTable WHERE Quantity > 10
GO

-- 3. Define the unique clustered index on the view
CREATE UNIQUE CLUSTERED INDEX IDX_MyView ON MyView(Quantity)

-- 4 check on Include Live Query Statistics
SELECT Iid  FROM MyTable WHERE Quantity > 30
SELECT Iid FROM MyView WHERE Quantity > 30
-- in both cases, the Unique Clustered Index is used
```

Indexes and Indexed Views - Conclusions

Rules and Good Practices:

- Each table must have a clustered index; it should be small, selective, increasing and static (a table without clustered indexed is called *heap*)
- For the foreign keys can be created non-clustered indexes
- For the fields used in the WHERE, ORDER BY, GROUP BY, JOIN’s clauses can be created non-clustered indexes
- Should not be created simple-column indexes on all the fields of the table because will complicate the maintenance of the table
- In multi-column indexes, the most selective field (“the closer” to be unique) should be placed on the first positions
- The covering non-clustered indexes (all fields from the query) should be created for the most frequently executed queries

CASE

CASE – evaluates a list of conditions and return as a result one of the multiple possible expressions

- Can be specified in the statements like SELECT, UPDATE, DELETE, SET, but also in the clauses WHERE, IN, HAVING, ORDER BY
- Has 2 possible options:
  - **Simple CASE** – compare an expression with a set of simple expressions to get the result
  - **Search CASE** - evaluate a set of boolean expressions to get the result

Simple CASE syntax:

CASE input_expression

WHEN when_expression THEN result_expression [… n]

[ELSE else_result_expression]

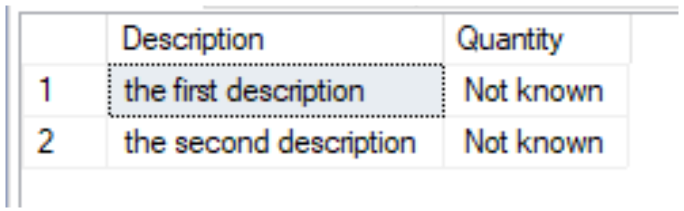
- Functionality
  - The first expression is compared with each expression from the **WHEN** clause (the equivalence is cheked)
  - If the expressions are equivalent, it is going to be returned the expression from the clause **THEN**

CASE

- Only one equality check is allowed
- For each specified **WHEN** clause, is evaluated ***input_expression = when_expression*** (in the specified order)
- It is returned the ***result_expression*** that correspond to the **first comparation *input_expression = when_expression*** that is **TRUE**
- If **none** of the comparations is **TRUE**, it is returned the ***else_result_expression*** (if it is specified in the **ELSE** clause), or **NULL** (if nothing is specified in the **ELSE** clause)

Example: simple CASE in a SELECT statement

```
SELECT Description, Quantity = CASE Quantity
WHEN 1 THEN N'The smallest quantity'
WHEN 2 THEN N'The intermediate quantity'
WHEN 3 THEN N'The highest quantity'
ELSE N'Not known'
END
FROM MyTable
```



	Description	Quantity
1	the first description	Not known
2	the second description	Not known

CASE

Searched CASE syntax:

CASE

WHEN boolean_expression THEN result_expression [… n]

[ELSE else_result_expression]

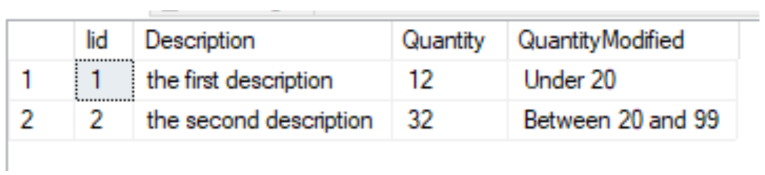
- Functionality
  - It is evaluated in the specified order, the ***boolean_expression*** for each **WHEN** clause
  - It is returned the ***result_expression*** that corresponds to the first ***boolean_expression*** that is evaluated with **TRUE**
  - If there is no ***boolean_expression*** evaluated with **TRUE**, it is returned the ***else_result_expression*** (if it is specified in **ELSE** clause), or **NULL** (if nothing is specified in the **ELSE** clause)

- The CASE statement cannot be used to flow control on the Transact-SQL statements, blocks of statements, used-defined functions, stored procedures

CASE

Example: Searched CASE in a SELECT statement

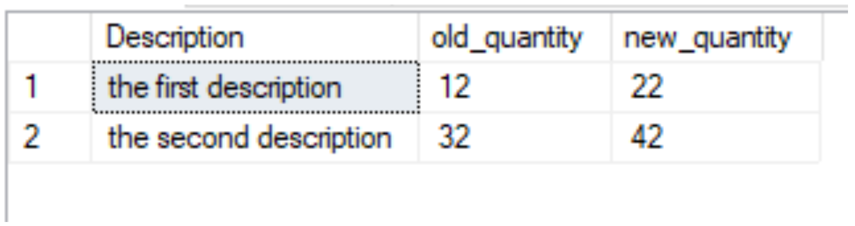
```
SELECT lid, Description, Quantity, QuantityModified = CASE
WHEN Quantity=0 THEN N'No product'
WHEN Quantity<20 THEN N'Under 20'
WHEN Quantity>=20 AND Quantity<100 THEN N'Between 20 and 99'
ELSE N'Greater than 99'
END
FROM MyTable
```



	lid	Description	Quantity	QuantityModified
1	1	the first description	12	Under 20
2	2	the second description	32	Between 20 and 99

Example: Searched CASE in an UPDATE Statement

```
UPDATE MyTable
SET Quantity=(CASE
WHEN Quantity-10<0 THEN Quantity-10
WHEN Quantity-10=0 THEN Quantity
WHEN Quantity-10>0 THEN Quantity +10
ELSE Quantity*2
END)
OUTPUT deleted.Description, deleted.Quantity as old_quantity, inserted.Quantity as
new_quantity
```



	Description	old_quantity	new_quantity
1	the first description	12	22
2	the second description	32	42

Columnstore Indexes

- ***Columnstore index*** goups and stores data relative to the columns, not relative to the rows
- The columnstore indexes can be ***clustered*** or ***non-clustered***
- Are usefull for the data warehouses (read-only queries)
- The queries are 10 times performant, relative to the traditional store that is done through the records
- The data compression is 10 times better than the dimension of the data that is not compressed
- When a columnstore index is created, the ascending / descending order of the columns from the index cannot be specified (because the data is stored in a manner that improves the compression and the performance)

Example: Clustered columnstore index / Non-clustered columnstore index

```
CREATE CLUSTERED COLUMNSTORE INDEX CSIDX_MyTable ON MyTable
CREATE NONCLUSTERED COLUMNSTORE INDEX NCSIDX_MyTable ON MyTable (Description, Quantity)
```

- A table can have ***rowstore indexes*** and also ***columnstore indexes*** (maximum 1 columnstore index per table)

Action	SQL Server 2014	SQL Server 2016
Create a non-clustered columnstore index	The table becomes read-only and cannot be modified	The table doesn't become read-only
Create a clustered columnstore index	Allows the data modification on the table, but not allows the creation of other indexes on that table	Allows the creation of other non-clustered rowstore indexes on the same table

Columnstore Indexes

Columnstore Index

Row 1	Column 1	Column 2	Column 3	Column 4	Column 5
Row 2	Column 1	Column 2	Column 3	Column 4	Column 5
Row 3	Column 1	Column 2	Column 3	Column 4	Column 5
Row 4	Column 1	Column 2	Column 3	Column 4	Column 5
Row 5	Column 1	Column 2	Column 3	Column 4	Column 5
…	Column 1	Column 2	Column 3	Column 4	Column 5
Row n	Column 1	Column 2	Column 3	Column 4	Column 5
	Page 1	Page 2	Page 3	Page 4	Page 5

Columnstore Indexes

Rowstore Index

Row 1	Column 1	Column 2	Column 3	Column 4	Column 5
Row 2	Column 1	Column 2	Column 3	Column 4	Column 5
Row 3	Column 1	Column 2	Column 3	Column 4	Column 5

Page 1

Row 4	Column 1	Column 2	Column 3	Column 4	Column 5
Row 5	Column 1	Column 2	Column 3	Column 4	Column 5
…	Column 1	Column 2	Column 3	Column 4	Column 5
Row n	Column 1	Column 2	Column 3	Column 4	Column 5

Page 2

Fragmentation

- The main unit to store the data in SQL Server is the ***page***
- On the disk, the space of a data file (.mdf or .ndf) from a database is split in contiguous numbered pages from 0 to n
- The input / output operations are performed on the page level (SQL Server read / write data pages)
- A zone (extent) contains 8 contiguous pages (of data), or 64KB

3 types of fragmentation:

- ***internal fragmentation*** – the records are not stored in a contiguous area on the page, or, there is unused space between the records in the page

  - the fullness of the pages can vary over time

  - the unused space can cause inefficient use of the cache and more page transfers between the disk and the main memory, fact that has a negative impact on the query performance
- ***extent fragmentation*** – take place when the physical store of the data and of the zones (extents) from the disk are not contiguous

  - when the extents of a table are not stored contiguous on the disk, the passing from a zone to another, causes higher rotations on the disk

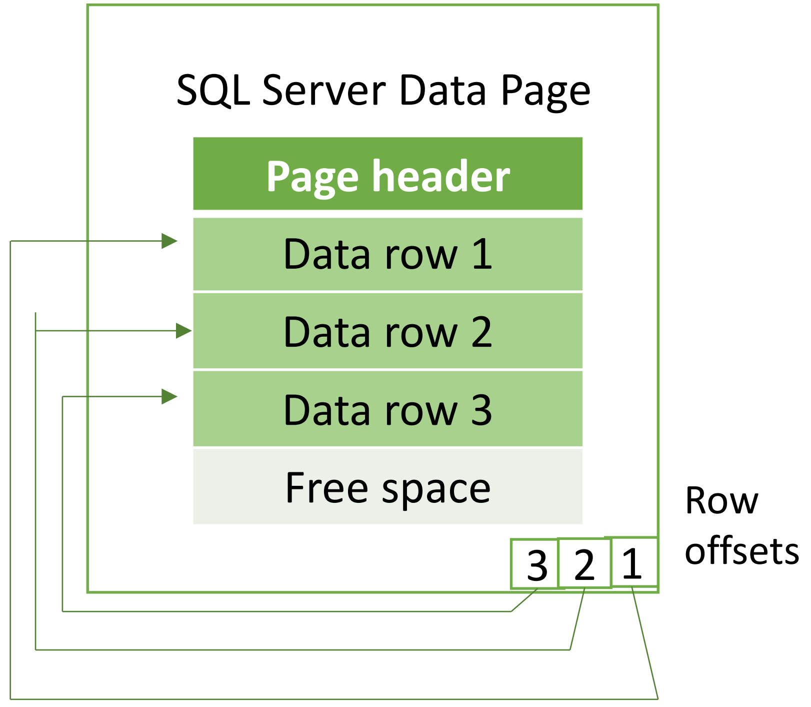
Fragmentation

- ***logical fragmentation*** – each index page is linked with the previous and the next page based on the logical order of the key values

  - when the pages are full and there is no place for new data, the values are redistributed (***Page Split***) and the pages become ***out-of-order***

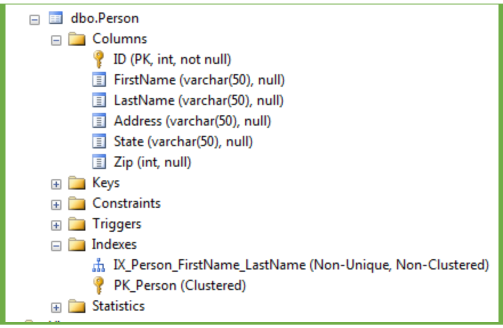
  - a page is out-of-order when the next physical page (in the index) is not the next logical page

- Each data page contains a header, records, free space and an offset-row table that contain an entry for each records on the page
- Each entry from the offset-row table checks for the record how far is the first byte of the record to the beginning of the page
- The entries from the offset-row table are in the opposite order to the records from the page
- A page has 8KB



Fragmentation

Internal fragmentation example:

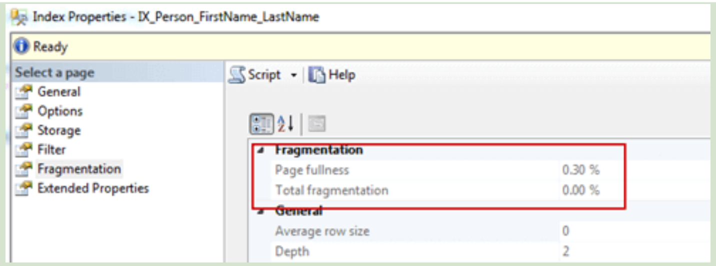


```
dbo.Person
  Columns
    ID (PK, int, not null)
    FirstName (varchar(50), null)
    LastName (varchar(50), null)
    Address (varchar(50), null)
    State (varchar(50), null)
    Zip (int, null)
  Keys
  Constraints
  Triggers
  Indexes
    IX_Person_FirstName_LastName (Non-Unique, Non-Clustered)
    PK_Person (Clustered)
  Statistics
```

create table Person(ID int primary key identity(1,1), FirstName varchar(50), LastName varchar(50), City varchar(50), State varchar(50), Zip int)

create nonclustered index N_idx_Person_FirstName_LastName
on Person(FirstName, LastName)

Right click on the index, click Properties, and Fragmentation to see fragmentation and page fullness. This is a brand new index so it's at 0% fragmentation.



Index Properties - IX_Person_FirstName_LastName

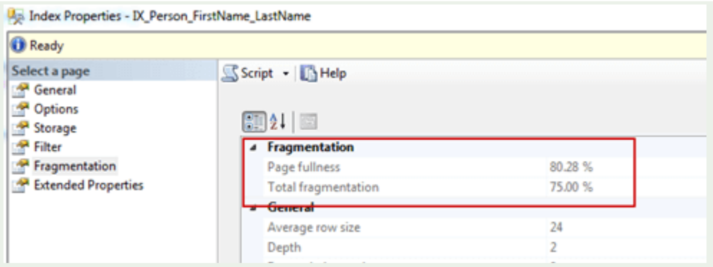
Ready

Select a page: General, Options, Storage, Filter, Fragmentation, Extended Properties

Script | Help

Fragmentation	
Page fullness	0.30 %
Total fragmentation	0.00 %
General	
Average row size	0
Depth	2

INSERT INTO Person VALUES (‘Brady’, ‘Upon’, ‘123 Main Street’, ‘TN’ 55555)
GO 1000



Index Properties - IX_Person_FirstName_LastName

Ready

Select a page: General, Options, Storage, Filter, Fragmentation, Extended Properties

Script | Help

Fragmentation	
Page fullness	80.28 %
Total fragmentation	75.00 %
General	
Average row size	24
Depth	2

The index becomes 75% fragmented and the average percent of full pages (page fullness) increases to 80%.

So, the performance will be affected by the size of the table and by the page counts.

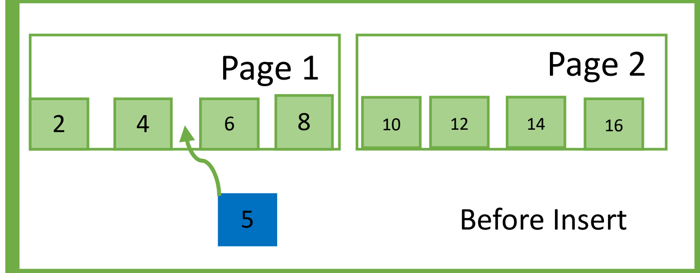
Fragmentation

Extent fragmentation: pages are out of order

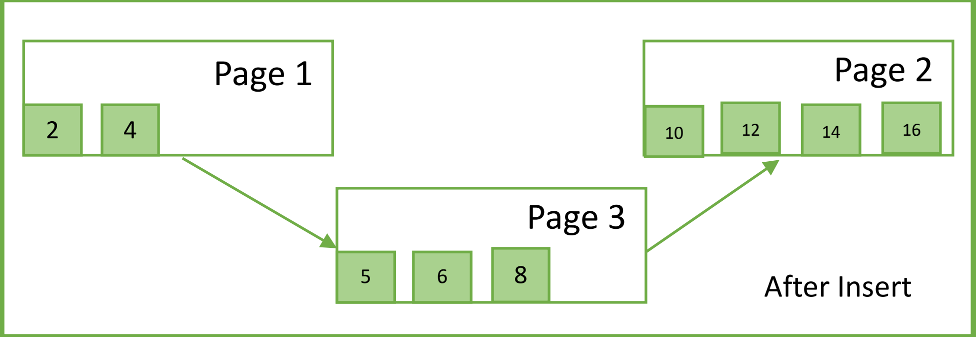
- In the beginning of the day, the table is perfectly ordered
- During the day, appear hundreds of update statements that possibly leave some empty space on one page and try to fit space into other pages.
- This means that the storage has to jump around to obtain the data needed instead of reading in one direction.

Logical fragmentation example: Consider 2 data pages for a table with a clustered index

The data is order and the pages are full



A new row that has the primary key ***5*** needs to be inserted; since it is a clustered index, the new row is inserted in order. Because the target page is full enough that the new row does not fit, SQL Server splits the page roughly in half and inserts the new data on the new page.

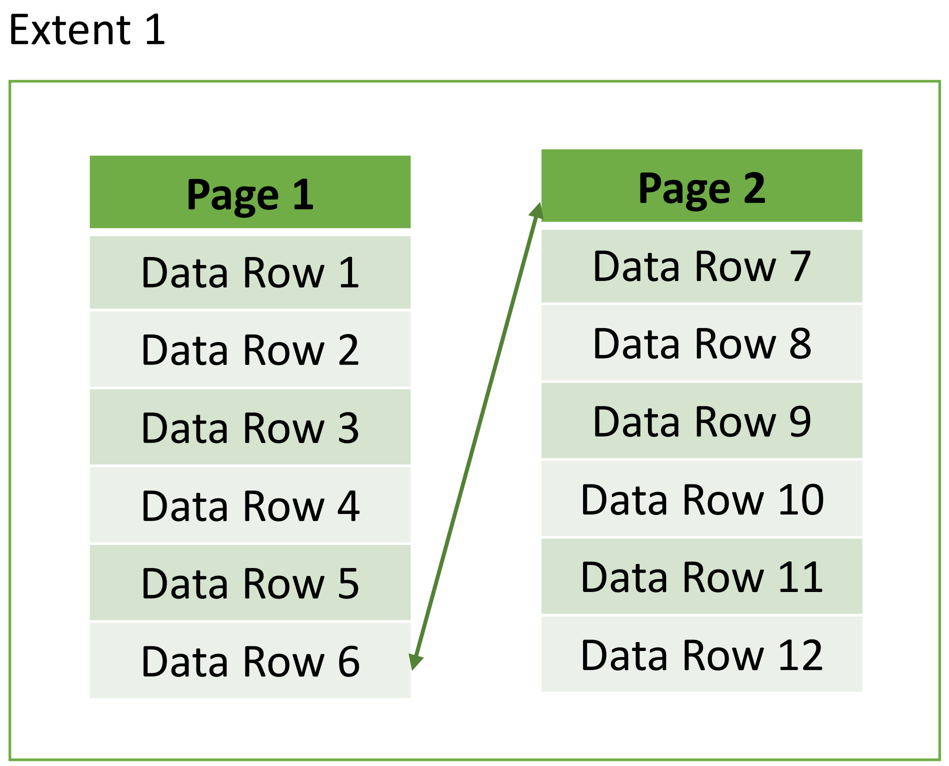


Now, the logical order of the index does not match the physical order, and the index has become fragmented.

Fragmentation

- Read operation – page request – 2
- Extent switches – 0
- Disk space used by the table 16 KB
- avg_fragmentation_in_percent – 0
- avg_page_space_used_in_percent – 100

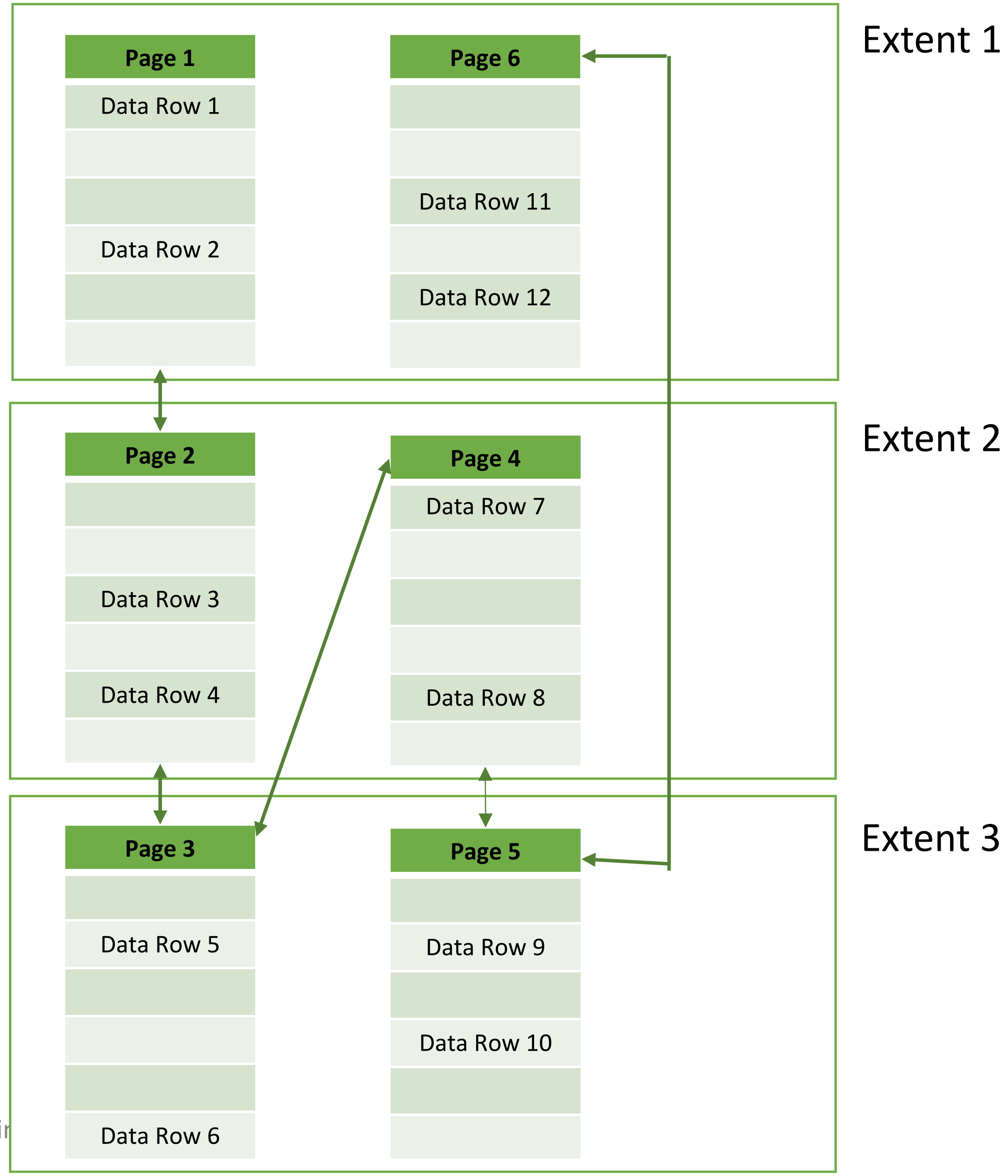
(the pages are ***full*** – no internal fragmentation – because follows one to another without space between data)



Fragmentation

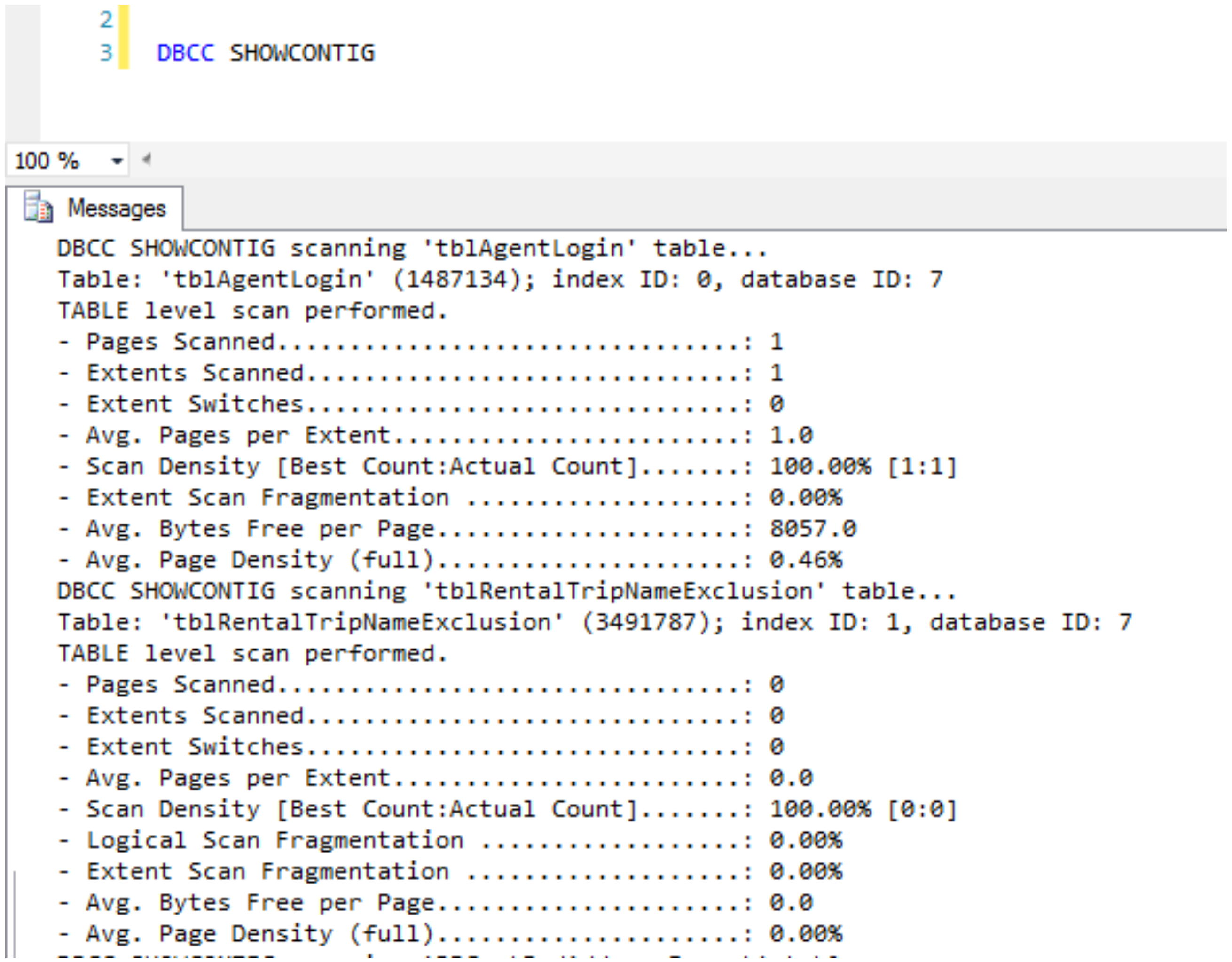
- Read operation – page request – 6
- Extent switches – 5
- Disk space used by the table 48 KB
- avg_fragmentation_in_percent >80
- avg_page_space_used_in_percent – 33

(
- internal fragmentation (because the pages are NOT full)
and
- extent fragmentation (because the pages are not stored contiguous on the disk)
and
- logical fragmentation (because we 'jump' between pages – logical order is not the same as the physical order)
)



Fragmentation

- Return fragmentation: ***DBCC SHOWCONTIG***



```
DBCC SHOWCONTIG
```

100 %

Messages

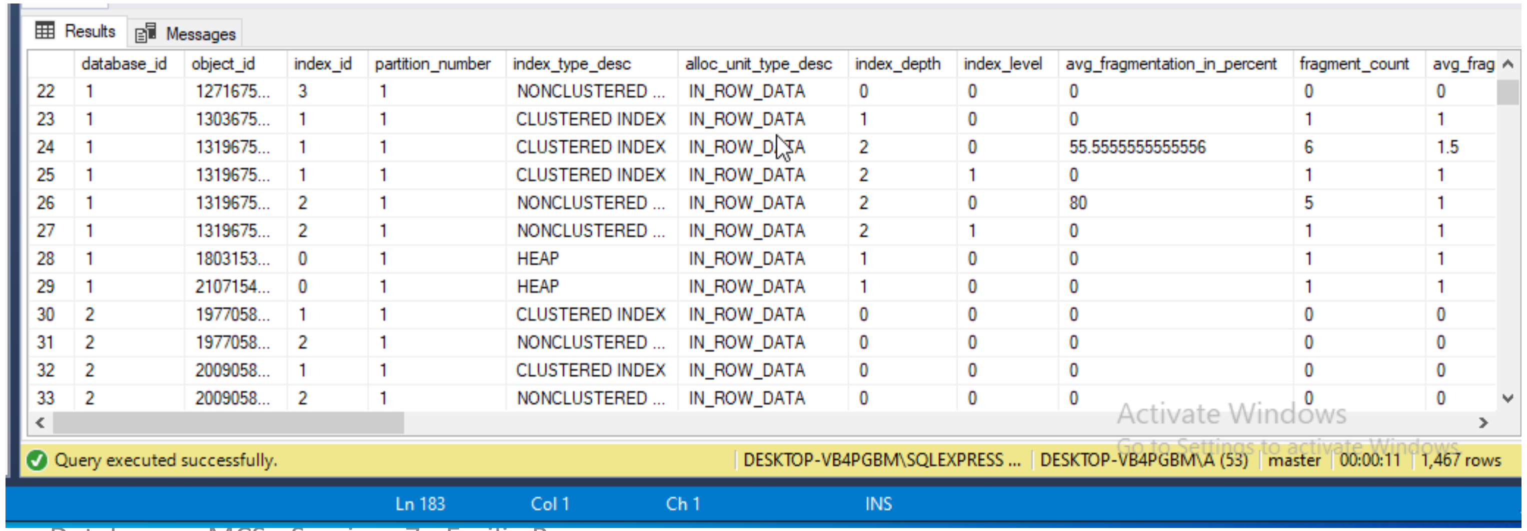
```
DBCC SHOWCONTIG scanning 'tblAgentLogin' table...
Table: 'tblAgentLogin' (1487134); index ID: 0, database ID: 7
TABLE level scan performed.
- Pages Scanned................................: 1
- Extents Scanned..............................: 1
- Extent Switches..............................: 0
- Avg. Pages per Extent........................: 1.0
- Scan Density [Best Count:Actual Count].......: 100.00% [1:1]
- Extent Scan Fragmentation ...................: 0.00%
- Avg. Bytes Free per Page.....................: 8057.0
- Avg. Page Density (full).....................: 0.46%
DBCC SHOWCONTIG scanning 'tblRentalTripNameExclusion' table...
Table: 'tblRentalTripNameExclusion' (3491787); index ID: 1, database ID: 7
TABLE level scan performed.
- Pages Scanned................................: 0
- Extents Scanned..............................: 0
- Extent Switches..............................: 0
- Avg. Pages per Extent........................: 0.0
- Scan Density [Best Count:Actual Count].......: 100.00% [0:0]
- Logical Scan Fragmentation ..................: 0.00%
- Extent Scan Fragmentation ...................: 0.00%
- Avg. Bytes Free per Page.....................: 0.0
- Avg. Page Density (full).....................: 0.00%
```

Fragmentation

- ***sys.dm_db_physical_stats*** – returns information about the dimension and the fragmentation of the data and of the indexes for the specified table / view
- **avg_fragmentation_in_percent** – contains the percentage value of the
  - extent fragmentation – for heaps
  - logical fragmentation – for indexes
- **avg_page_space_used_in_percent** – contains the average percentage of available space in all the pages

Example: SELECT * FROM sys.dm_db_index_physical_stats(DB_ID(N'Seminar7'), OBJECT_ID(N'Seminar7.dbo.Person'), NULL, NULL, 'DETAILED’)

SELECT * FROM sys.dm_db_index_physical_stats (NULL, NULL, NULL, NULL, NULL);



Results | Messages

	database_id	object_id	index_id	partition_number	index_type_desc	alloc_unit_type_desc	index_depth	index_level	avg_fragmentation_in_percent	fragment_count	avg_frag
22	1	1271675...	3	1	NONCLUSTERED ...	IN_ROW_DATA	0	0	0	0	0
23	1	1303675...	1	1	CLUSTERED INDEX	IN_ROW_DATA	1	0	0	1	1
24	1	1319675...	1	1	CLUSTERED INDEX	IN_ROW_DATA	2	0	55.5555555555556	6	1.5
25	1	1319675...	1	1	CLUSTERED INDEX	IN_ROW_DATA	2	1	0	1	1
26	1	1319675...	2	1	NONCLUSTERED ...	IN_ROW_DATA	2	0	80	5	1
27	1	1319675...	2	1	NONCLUSTERED ...	IN_ROW_DATA	2	1	0	1	1
28	1	1803153...	0	1	HEAP	IN_ROW_DATA	1	0	0	1	1
29	1	2107154...	0	1	HEAP	IN_ROW_DATA	1	0	0	1	1
30	2	1977058...	1	1	CLUSTERED INDEX	IN_ROW_DATA	0	0	0	0	0
31	2	1977058...	2	1	NONCLUSTERED ...	IN_ROW_DATA	0	0	0	0	0
32	2	2009058...	1	1	CLUSTERED INDEX	IN_ROW_DATA	0	0	0	0	0
33	2	2009058...	2	1	NONCLUSTERED ...	IN_ROW_DATA	0	0	0	0	0

```
Query executed successfully.   DESKTOP-VB4PGBM\SQLEXPRESS ... | DESKTOP-VB4PGBM\A (53) | master | 00:00:11 | 1,467 rows
Ln 183   Col 1   Ch 1   INS
```

Fragmentation

Reduce the fragmentation

- **In a heap** – should be created a clustered index on the table (the records will be redistributed and the pages will be contiguous) and then drop it
- **In an index**
  - if **avg_fragmentation_in_percent** > 5 % and <= 30 %
    - reorder the leaf pages of the index based on the key
    - ALTER INDEX REORGANIZE
  - if **avg_fragmentation_in_percent** > 30 %
    - the index will be drop and the recreate
    - ALTER INDEX REBUILD
  - if **avg_fragmentation_in_percent** < 5 %
    - Nothing
- A clustered index is drop and recreated
  - when a clustered index is recreated, the data is redistributed and the pages will be full
  - the level of fullness can be specified by using **FILLFACTOR** option on the CREATE INDEX statement

Fragmentation

FILLFACTOR

- When an index is created / recreated, the specified value from **FILLFACTOR** determine the space percentage that should be fulfill with data for each data page from the index on the level of the final nodes, leaving the rest of the space for future inserts
- e.g. a value of 80% for the option **FILLFACTOR** will cause 20% free space on each page from the index at the level of the final nodes; offering space to extend the index when new records will be added

Example: create an index with FILLFACTOR option

```
CREATE INDEX IDX_Person_FN_ASC_LN_DESC ON Person(FirstName ASC, LastName DESC)
WITH (FILLFACTOR=80)
```

Example: Reorganize all indexes from the Person table

```
ALTER INDEX ALL ON Person REORGANIZE ;
```

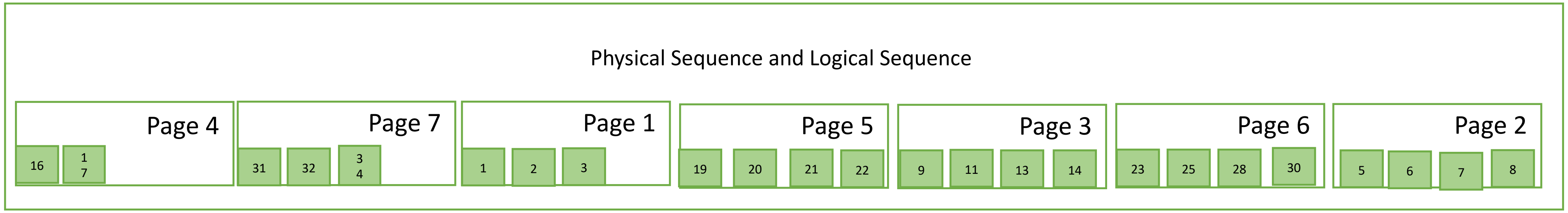
Example: Rebuild index

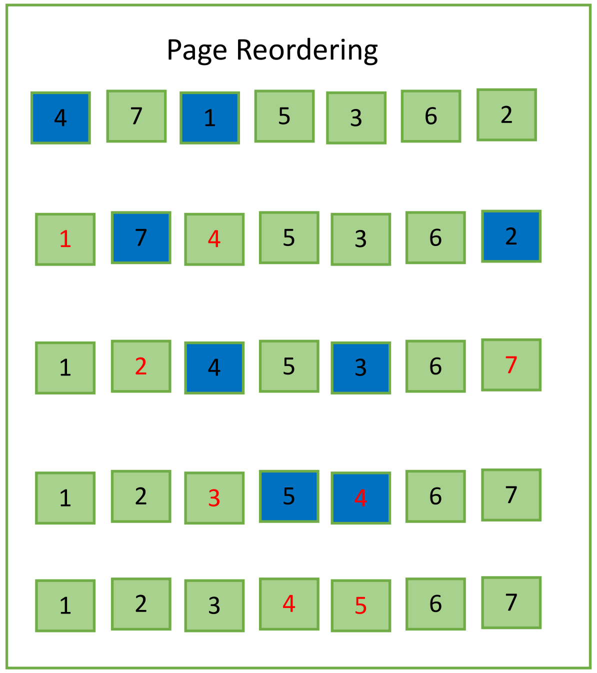
```
ALTER INDEX IDX_Person_FN_ASC_LN_DESC ON Person REBUILD;
```

Fragmentation

Example of defragmentation of an index (with Reorganize): pages after many inserts, updates, deletes

- Page numbering is the logical sequence of pages
- The physical sequence from left to right does not correspond to the logical sequence





SQL Server does the following:

- finds the first physical page (4) and the first logical page (1) and exchanges them
- exchanges the next physical page (7) with next logical page (2)
- exchanges the next physical page (4) with the next logical page (3).
- exchanges the next physical page (5) with the next logical page (4).