**A script to automatically align columnstore indexes to enhance segment elimination (and hence performances)**

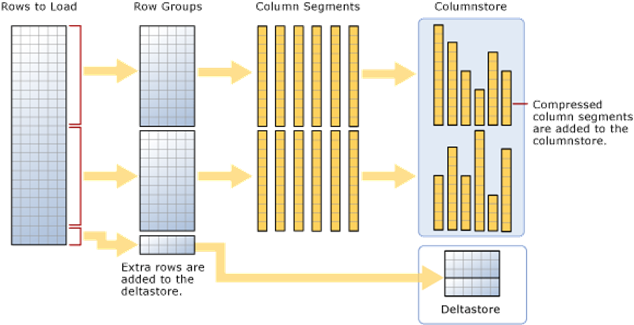
By [Emanuele Meazzo](https://tsql.tech/author/menny/)

June 4, 2018

5 Min read

FROM: <https://tsql.tech/a-script-to-automatically-align-columnstore-indexes-to-enhance-segment-elimination-and-hence-performances/>

Columstore indexes are a “new” neat data structure that I like, even if technically they’ve been around for years, only recently they’ve become usable by most customers.  
Let me recap a little bit what we’re talking about, so the point of this will be clearer:

Obligatory Microsoft Image Example

The table is divided in rowgroups of about one million rows max, then each column is stored by itself in a unit called segment (i.e. a segment is a single column of a rowgroup), then segments are compressed and compression ratio depends on the type and contents of the data, etc, etc, etc…

The idea here is that segments have metadata, and in this metadata there is for example the information of what’s the biggest and what’s the smallest value of this particular segment, you see where I’m going?  
Segments are basically like partitions on a partitioned table, and as in partitioning SQL Server is able to just skip segments in which it’s sure not to find the data that we’re looking for; I’m already imagining what’s going on in your mind: “Amazing! So you’re telling me that for Columnstore not only Batch Mode operators are enabled, IO is reduced due to compression and only the columns requested are fetched saving even more IO, but segments can be eliminated too in order to fetch the smallest dataset possible! ”  
Well, if it was that easy, I wouldn’t be writing this, you should have expected it  ¯\\_(ツ)\_/¯

Let’s build a test case and let’s keep things simple, I have only a Clustered Columnstore Index in my table that I’ll be hammering most of the time using Column 1:

DROP TABLE IF EXISTS SegmentDemo\_CC

CREATE TABLE SegmentDemo\_CC

( [int] INT,

[Decimal] DECIMAL(18, 3),

[String] VARCHAR(20),

[UString] NVARCHAR(20)

);

--Insert 10 Million Records

WITH Tally (N) AS

(

SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))

FROM sys.all\_columns a CROSS JOIN sys.all\_columns b

)

INSERT INTO SegmentDemo\_CC WITH (TABLOCK) ([int],[Decimal],[String],[UString])

SELECT TOP 10000000 N, N, N, N

FROM Tally;

--Create Clustered Columnstore Index

CREATE CLUSTERED COLUMNSTORE INDEX cc\_SegmentDemo ON SegmentDemo\_CC

1. DROP TABLE IF EXISTS SegmentDemo\_CC
2. CREATE TABLE SegmentDemo\_CC
3. ( [int] INT,
4. [Decimal] DECIMAL(18, 3),
5. [String] VARCHAR(20),
6. [UString] NVARCHAR(20)
7. );
8. --Insert 10 Million Records
9. WITH Tally (N) AS
10. (
11. SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))
12. FROM sys.all\_columns a CROSS JOIN sys.all\_columns b
13. )
14. INSERT INTO SegmentDemo\_CC WITH (TABLOCK) ([int],[Decimal],[String],[UString])
15. SELECT TOP 10000000 N, N, N, N
16. FROM Tally;
17. --Create Clustered Columnstore Index
18. CREATE CLUSTERED COLUMNSTORE INDEX cc\_SegmentDemo ON SegmentDemo\_CC

DROP TABLE IF EXISTS SegmentDemo\_CC

CREATE TABLE SegmentDemo\_CC

( [int] INT,

[Decimal] DECIMAL(18, 3),

[String] VARCHAR(20),

[UString] NVARCHAR(20)

);

--Insert 10 Million Records

WITH Tally (N) AS

(

SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))

FROM sys.all\_columns a CROSS JOIN sys.all\_columns b

)

INSERT INTO SegmentDemo\_CC WITH (TABLOCK) ([int],[Decimal],[String],[UString])

SELECT TOP 10000000 N, N, N, N

FROM Tally;

--Create Clustered Columnstore Index

CREATE CLUSTERED COLUMNSTORE INDEX cc\_SegmentDemo ON SegmentDemo\_CC

The issue here is that in Columnstore Indexes the Segments are not aligned to anything by default (and there is no explicit way to align it to anything), take a look at this query:

SELECT segment\_id, s.row\_count, s.column\_id, s.min\_data\_id, s.max\_data\_id

FROM sys.column\_store\_segments s

JOIN sys.partitions p ON p.hobt\_id = s.hobt\_id

WHERE p.object\_id = OBJECT\_ID('yourtable')

AND column\_id = 1 --Column 1 Just Because

ORDER BY Segment\_id,column\_id,s.min\_data\_id;

1. SELECT segment\_id, s.row\_count, s.column\_id, s.min\_data\_id, s.max\_data\_id
2. FROM sys.column\_store\_segments s
3. JOIN sys.partitions p ON p.hobt\_id = s.hobt\_id
4. WHERE p.object\_id = OBJECT\_ID('yourtable')
5. AND column\_id = 1 --Column 1 Just Because
6. ORDER BY Segment\_id,column\_id,s.min\_data\_id;

SELECT segment\_id, s.row\_count, s.column\_id, s.min\_data\_id, s.max\_data\_id

FROM sys.column\_store\_segments s

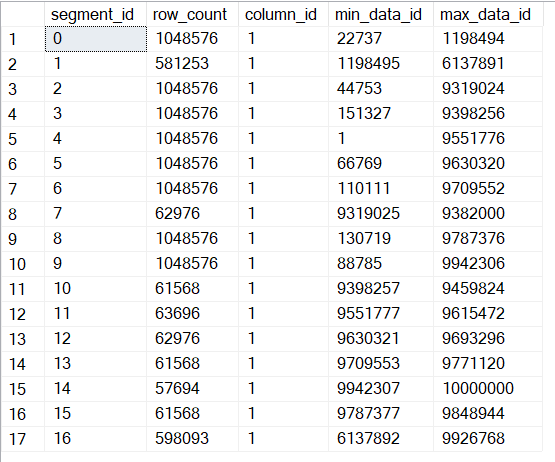
JOIN sys.partitions p ON p.hobt\_id = s.hobt\_id

WHERE p.object\_id = OBJECT\_ID('yourtable')

AND column\_id = 1 --Column 1 Just Because

ORDER BY Segment\_id,column\_id,s.min\_data\_id;

The query above is is a simple way to extract segment info for a specific Table/Column, showing how many rows each segment has and what’s the minimum and maximum value contained in each segment:



As you can see, values are scattered around segments randomly, that’s because there is nothing enforcing ordering and that ColumnStore Index Builds are multithreaded, so rows gets shuffled around then building the index, getting into whatever segment is available.

Let’s see how queries behave with this index, we’ll be focusing only on segment elimination by using simple queries.  
Let’s say that for mysterious reporting reason I need to make an average of 400K records from my table, like so:

SET STATISTICS IO, TIME ON;

SELECT AVG(INT)

FROM SegmentDemo\_CC

WHERE int BETWEEN 9319000 AND 9719000

1. SET STATISTICS IO, TIME ON;
2. SELECT AVG(INT)
3. FROM SegmentDemo\_CC
4. WHERE int BETWEEN 9319000 AND 9719000

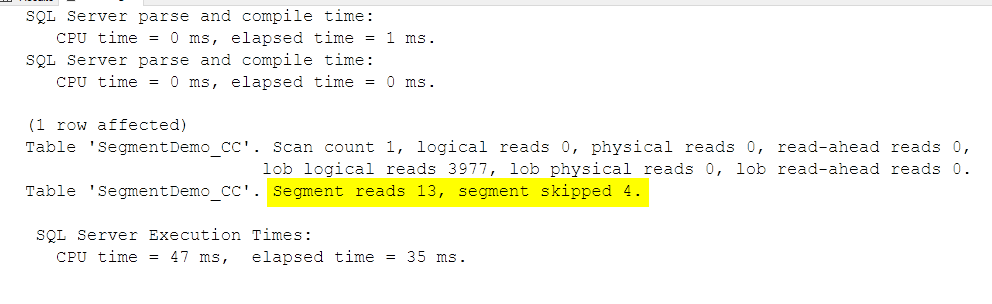
SET STATISTICS IO, TIME ON;

SELECT AVG(INT)

FROM SegmentDemo\_CC

WHERE int BETWEEN 9319000 AND 9719000

We get one row, of which we really don’t care, and this messages in the message window:



Of all the 17 Segments, we had to read 13, since the interval that we were analyzing was included in 13 Segments; Since we’re only looking at 400k rows, and RowGroups are of about 1 Million rows (maybe less, if there is a reason for SQL Server to close the rowgroup earlier) we expect to read 1 or at most 2 segments, as opposed as 13.

Here it comes the actual point of this article, how we can align the Columnstore index in order to maximize the Segment Elimination for this predicate?  
The standard practice is to drop che Columnstore Index, create a Clustered index on the key that you wish to align, then recreate the Columnstore Index with DROP\_EXISTING = ON and with MAXDOP = 1 in order to have SQL Server rebuild the Columnstore index with a single thread and hence maintaining the order first enforced by the Clustered Index.

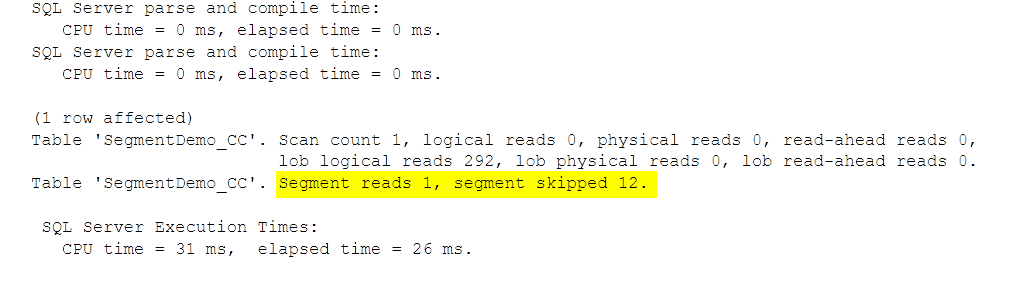
But Why bothering with that when I’ve created a script that does that for you for both Clustered and NonClustered Columnstore Indexes?  
You can get the code [here in my public GitHub repo](https://github.com/EmanueleMeazzo/SQL-Code-snippets/blob/master/Maintenance/Align%20Columnstore%20Index.sql), and install it in in the DB where you need to align stuff, and then the alignment it’s just a few keywords away, like so:

AlignColumnstore @SchemaName = 'dbo', @TableName = 'SegmentDemo\_CC', @AlignToColumn = 'INT', @IndexToAlign = 'cc\_SegmentDemo', @PrintOnly = 0

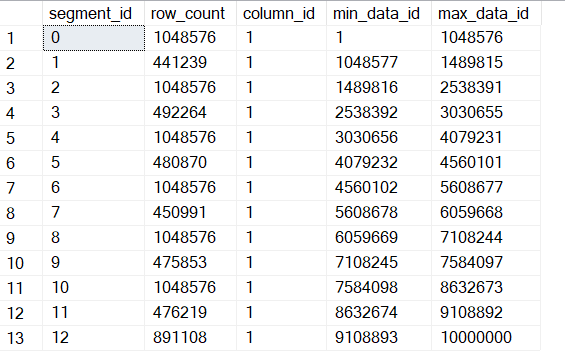
1. AlignColumnstore @SchemaName = 'dbo', @TableName = 'SegmentDemo\_CC', @AlignToColumn = 'INT', @IndexToAlign = 'cc\_SegmentDemo', @PrintOnly = 0

AlignColumnstore @SchemaName = 'dbo', @TableName = 'SegmentDemo\_CC', @AlignToColumn = 'INT', @IndexToAlign = 'cc\_SegmentDemo', @PrintOnly = 0

Now, running again the same query as above, we get a different result:



Hey, the number of segments went down too! This is just dumb luck and not to be expected by this kind of process, it just means that SQL Server was able to create bigger RowGroups on average (on why rowgroups can be closed before reaching the max number, you can read [here](http://www.nikoport.com/2015/08/12/columnstore-indexes-part-62-row-groups-trimming/) from Niko’s the Columnstore Guru)  
Again, this is a very small example so time savings are negligible, but imagine having a bigger table and a more complicate logic that needs to fetch and aggregate data from multiple columns and maybe be able to skip not 12 segments, but 100 or 1000, a whole another life.  
We’re now able to skip 12 segments of 13 because as you now see from the same DMV query as above, now the ranges for my predicate column are aligned and not overlapping:



Remember that segment elimination works only for numeric data types, so don’t even try to align a columnstore index to a string column, as segment elimination doesn’t work at all.

