## **Indexing tables in SQL Data Warehouse**

## **Index choices:**

SQL Data Warehouse offers several indexing options including clustered columnstore indexes, clustered indexes and nonclustered indexes, and a non-index option also known as heap.

## Clustered columnstore indexes

By default, SQL Data Warehouse creates a clustered columnstore index when no index options are specified on a table. Clustered columnstore tables offer both the highest level of data compression as well as the best overall query performance. Clustered columnstore tables will generally outperform clustered index or heap tables and are usually the best choice for large tables. For these reasons, clustered columnstore is the best place to start when you are unsure of how to index your table.

To create a clustered columnstore table, simply specify CLUSTERED COLUMNSTORE INDEX in the WITH clause, or leave the WITH clause off:

CREATE TABLE myTable

(

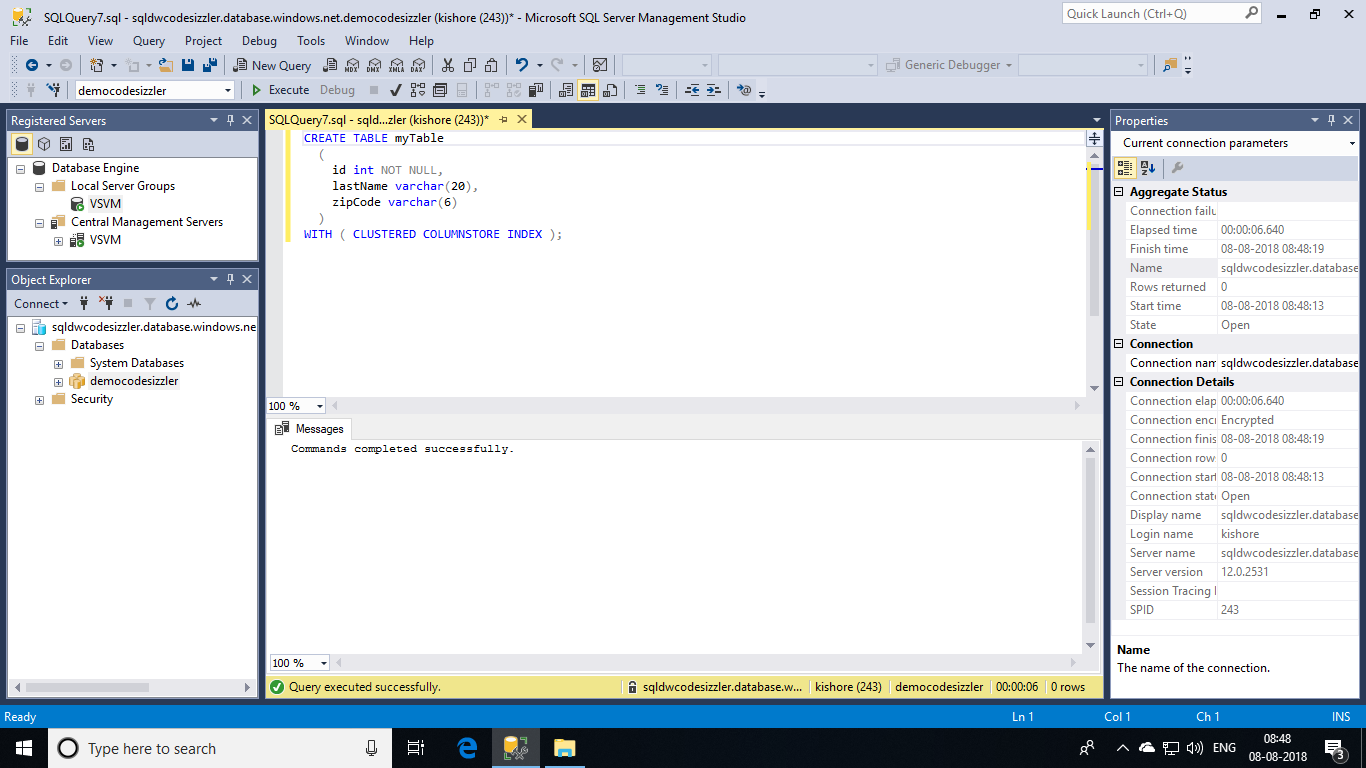
id int NOT NULL,

lastName varchar(20),

zipCode varchar(6)

)

WITH ( CLUSTERED COLUMNSTORE INDEX );

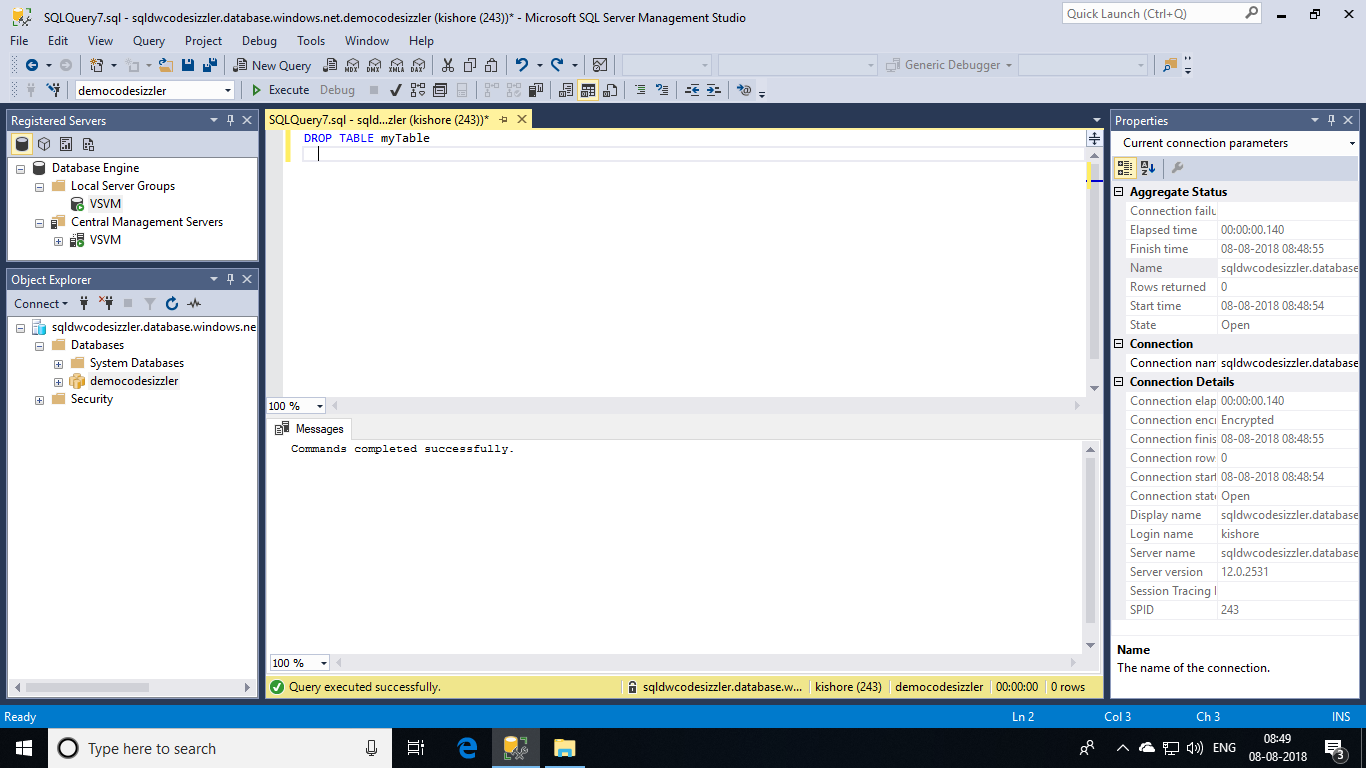


There are a few scenarios where clustered columnstore may not be a good option:

* Columnstore tables do not support varchar(max), nvarchar(max) and varbinary(max). Consider heap or clustered index instead.
* Columnstore tables may be less efficient for transient data. Consider heap and perhaps even temporary tables.
* Small tables with less than 100 million rows. Consider heap tables.

Now let’s delete the table that we created now to do few more indexing operations. For that run the command shown below.

DROP TABLE myTable



## **Heap tables:**

When you are temporarily landing data on SQL Data Warehouse, you may find that using a heap table makes the overall process faster. This is because loads to heaps are faster than to index tables and in some cases the subsequent read can be done from cache. If you are loading data only to stage it before running more transformations, loading the table to heap table is much faster than loading the data to a clustered columnstore table. In addition, loading data to a temporary table loads faster than loading a table to permanent storage.

For small lookup tables, less than 100 million rows, often heap tables make sense. Cluster columnstore tables begin to achieve optimal compression once there are more than 100 million rows.

To create a heap table, simply specify HEAP in the WITH clause:

CREATE TABLE myTable

(

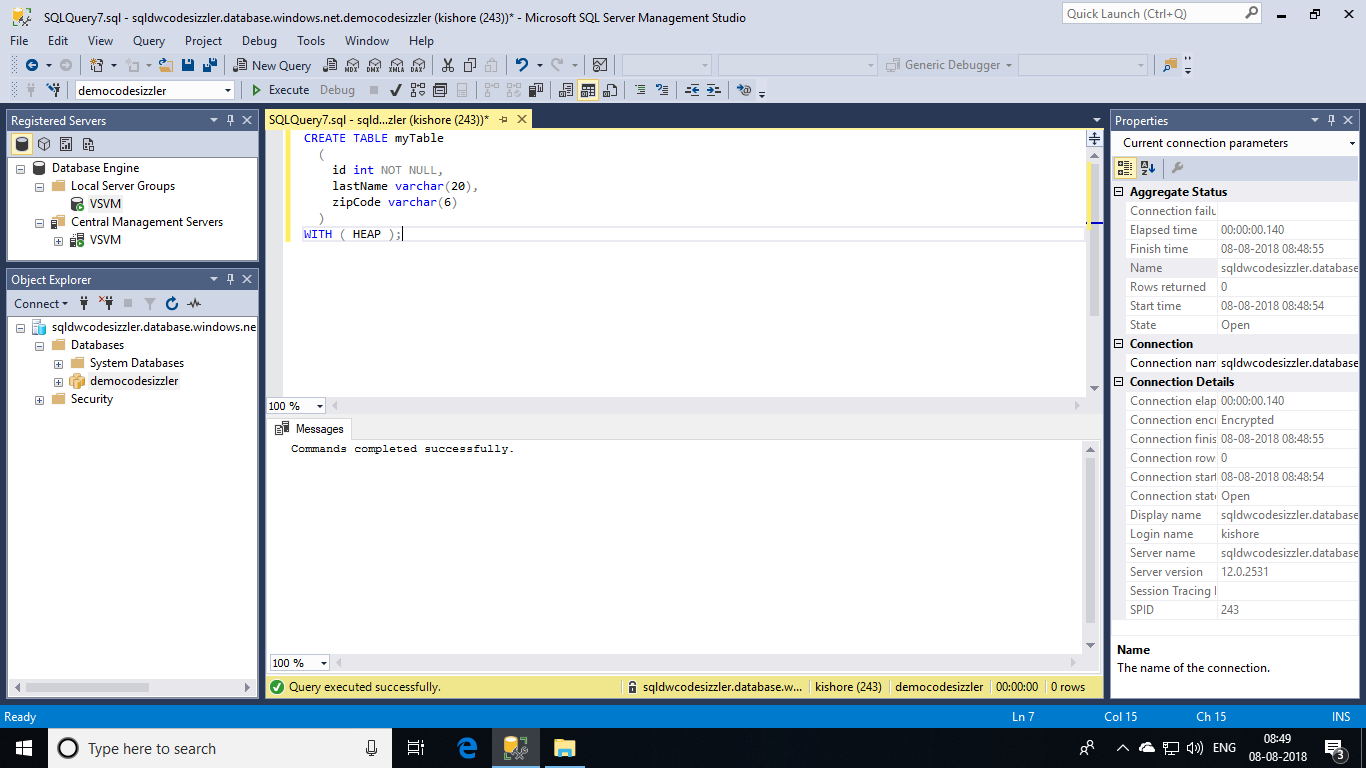
id int NOT NULL,

lastName varchar(20),

zipCode varchar(6)

)

WITH ( HEAP );



## **Clustered and nonclustered indexes:**

Clustered indexes may outperform clustered columnstore tables when a single row needs to be quickly retrieved. For queries where a single or very few row lookup is required to performance with extreme speed, consider a cluster index or nonclustered secondary index. The disadvantage to using a clustered index is that only queries that benefit are the ones that use a highly selective filter on the clustered index column. To improve filter on other columns a nonclustered index can be added to other columns. However, each index which is added to a table adds both space and processing time to loads.

Once again repeat the **DROP TABLE myTable** and delete the table so that you can perform other tasks.

To create a clustered index table, simply specify CLUSTERED INDEX in the WITH clause:

CREATE TABLE myTable

(

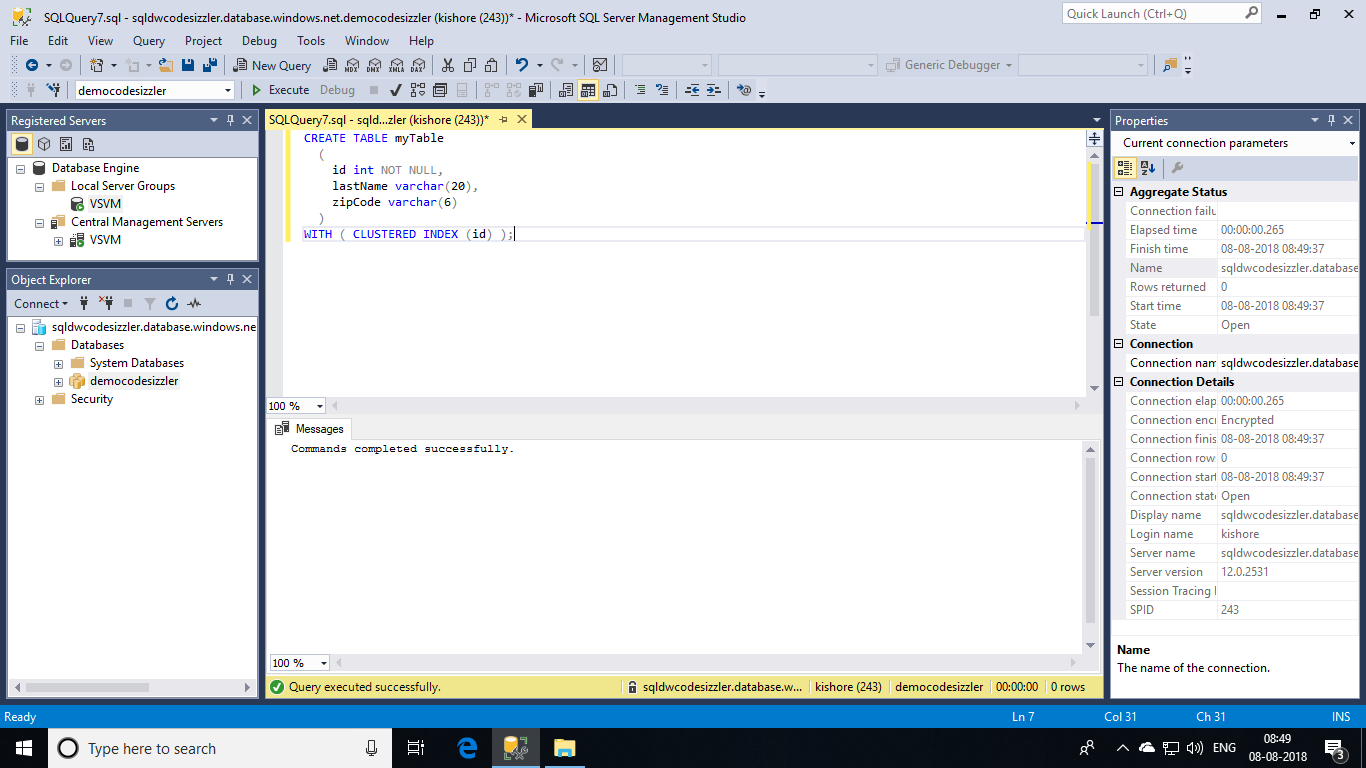
id int NOT NULL,

lastName varchar(20),

zipCode varchar(6)

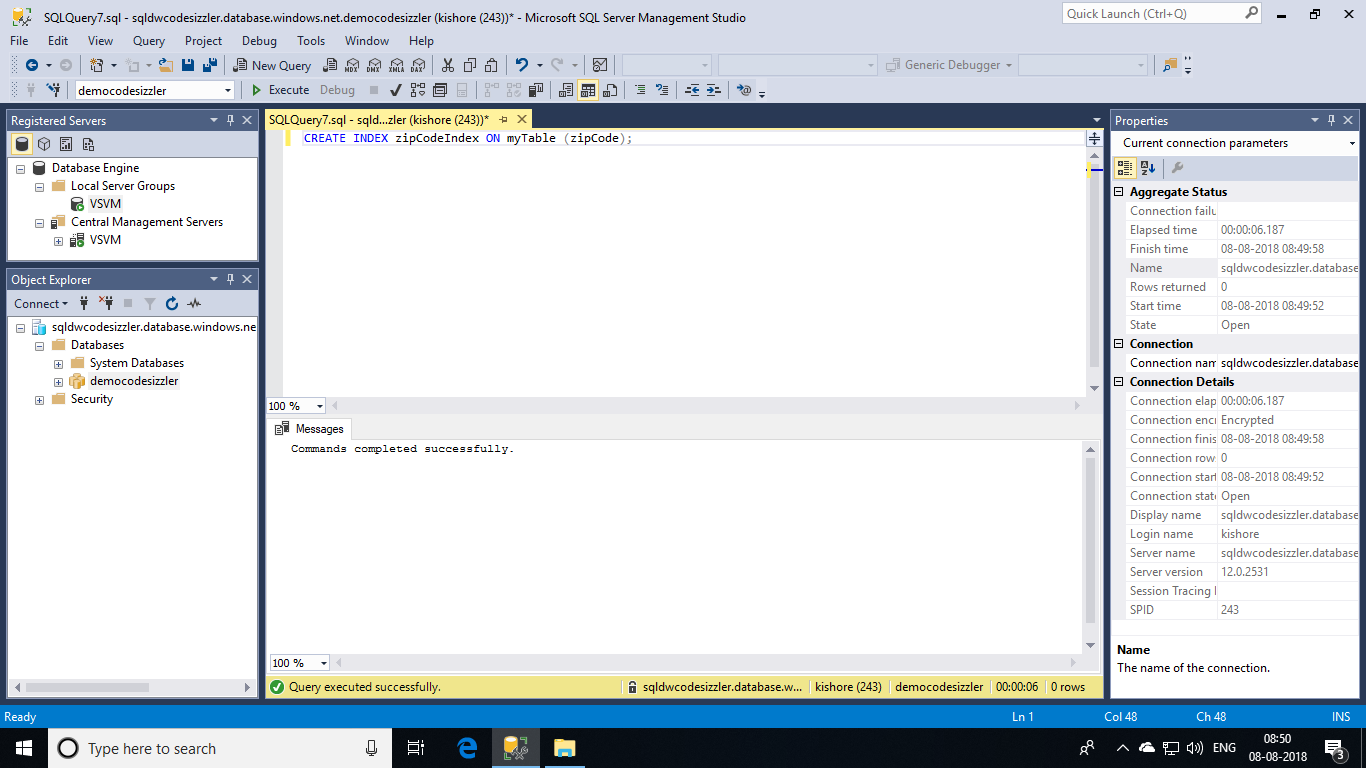
)

WITH ( CLUSTERED INDEX (id) );



To add a non-clustered index on a table, simply use the following syntax:

CREATE INDEX zipCodeIndex ON myTable (zipCode);



## **Optimizing clustered columnstore indexes:**

Clustered columnstore tables are organized in data into segments. Having high segment quality is critical to achieving optimal query performance on a columnstore table. Segment quality can be measured by the number of rows in a compressed row group. Segment quality is most optimal where there are at least 100K rows per compressed row group and gain in performance as the number of rows per row group approach 1,048,576 rows, which is the most rows a row group can contain.

The below view can be created and used on your system to compute the average rows per row group and identify any sub-optimal cluster columnstore indexes. The last column on this view generates a SQL statement which can be used to rebuild your indexes.

CREATE VIEW dbo.vColumnstoreDensity

AS

SELECT

GETDATE() AS [execution\_date]

, DB\_Name() AS [database\_name]

, s.name AS [schema\_name]

, t.name AS [table\_name]

, COUNT(DISTINCT rg.[partition\_number]) AS [table\_partition\_count]

, SUM(rg.[total\_rows]) AS [row\_count\_total]

, SUM(rg.[total\_rows])/COUNT(DISTINCT rg.[distribution\_id]) AS [row\_count\_per\_distribution\_MAX]

, CEILING ((SUM(rg.[total\_rows])\*1.0/COUNT(DISTINCT rg.[distribution\_id]))/1048576) AS [rowgroup\_per\_distribution\_MAX]

, SUM(CASE WHEN rg.[State] = 0 THEN 1 ELSE 0 END) AS [INVISIBLE\_rowgroup\_count]

, SUM(CASE WHEN rg.[State] = 0 THEN rg.[total\_rows] ELSE 0 END) AS [INVISIBLE\_rowgroup\_rows]

, MIN(CASE WHEN rg.[State] = 0 THEN rg.[total\_rows] ELSE NULL END) AS [INVISIBLE\_rowgroup\_rows\_MIN]

, MAX(CASE WHEN rg.[State] = 0 THEN rg.[total\_rows] ELSE NULL END) AS [INVISIBLE\_rowgroup\_rows\_MAX]

, AVG(CASE WHEN rg.[State] = 0 THEN rg.[total\_rows] ELSE NULL END) AS [INVISIBLE\_rowgroup\_rows\_AVG]

, SUM(CASE WHEN rg.[State] = 1 THEN 1 ELSE 0 END) AS [OPEN\_rowgroup\_count]

, SUM(CASE WHEN rg.[State] = 1 THEN rg.[total\_rows] ELSE 0 END) AS [OPEN\_rowgroup\_rows]

, MIN(CASE WHEN rg.[State] = 1 THEN rg.[total\_rows] ELSE NULL END) AS [OPEN\_rowgroup\_rows\_MIN]

, MAX(CASE WHEN rg.[State] = 1 THEN rg.[total\_rows] ELSE NULL END) AS [OPEN\_rowgroup\_rows\_MAX]

, AVG(CASE WHEN rg.[State] = 1 THEN rg.[total\_rows] ELSE NULL END) AS [OPEN\_rowgroup\_rows\_AVG]

, SUM(CASE WHEN rg.[State] = 2 THEN 1 ELSE 0 END) AS [CLOSED\_rowgroup\_count]

, SUM(CASE WHEN rg.[State] = 2 THEN rg.[total\_rows] ELSE 0 END) AS [CLOSED\_rowgroup\_rows]

, MIN(CASE WHEN rg.[State] = 2 THEN rg.[total\_rows] ELSE NULL END) AS [CLOSED\_rowgroup\_rows\_MIN]

, MAX(CASE WHEN rg.[State] = 2 THEN rg.[total\_rows] ELSE NULL END) AS [CLOSED\_rowgroup\_rows\_MAX]

, AVG(CASE WHEN rg.[State] = 2 THEN rg.[total\_rows] ELSE NULL END) AS [CLOSED\_rowgroup\_rows\_AVG]

, SUM(CASE WHEN rg.[State] = 3 THEN 1 ELSE 0 END) AS [COMPRESSED\_rowgroup\_count]

, SUM(CASE WHEN rg.[State] = 3 THEN rg.[total\_rows] ELSE 0 END) AS [COMPRESSED\_rowgroup\_rows]

, SUM(CASE WHEN rg.[State] = 3 THEN rg.[deleted\_rows] ELSE 0 END) AS [COMPRESSED\_rowgroup\_rows\_DELETED]

, MIN(CASE WHEN rg.[State] = 3 THEN rg.[total\_rows] ELSE NULL END) AS [COMPRESSED\_rowgroup\_rows\_MIN]

, MAX(CASE WHEN rg.[State] = 3 THEN rg.[total\_rows] ELSE NULL END) AS [COMPRESSED\_rowgroup\_rows\_MAX]

, AVG(CASE WHEN rg.[State] = 3 THEN rg.[total\_rows] ELSE NULL END) AS [COMPRESSED\_rowgroup\_rows\_AVG]

, 'ALTER INDEX ALL ON ' + s.name + '.' + t.NAME + ' REBUILD;' AS [Rebuild\_Index\_SQL]

FROM sys.[pdw\_nodes\_column\_store\_row\_groups] rg

JOIN sys.[pdw\_nodes\_tables] nt ON rg.[object\_id] = nt.[object\_id]

AND rg.[pdw\_node\_id] = nt.[pdw\_node\_id]

AND rg.[distribution\_id] = nt.[distribution\_id]

JOIN sys.[pdw\_table\_mappings] mp ON nt.[name] = mp.[physical\_name]

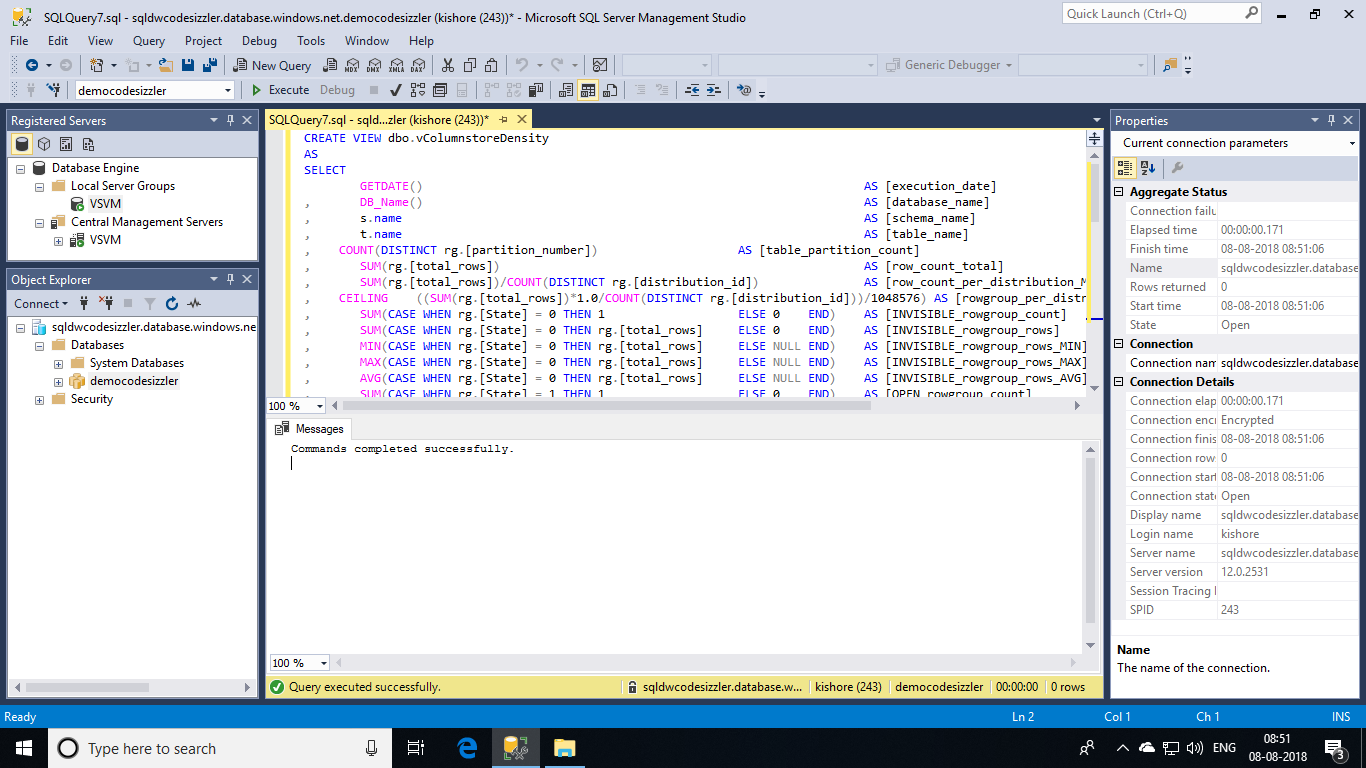
JOIN sys.[tables] t ON mp.[object\_id] = t.[object\_id]

JOIN sys.[schemas] s ON t.[schema\_id] = s.[schema\_id]

GROUP BY

s.[name]

, t.[name];



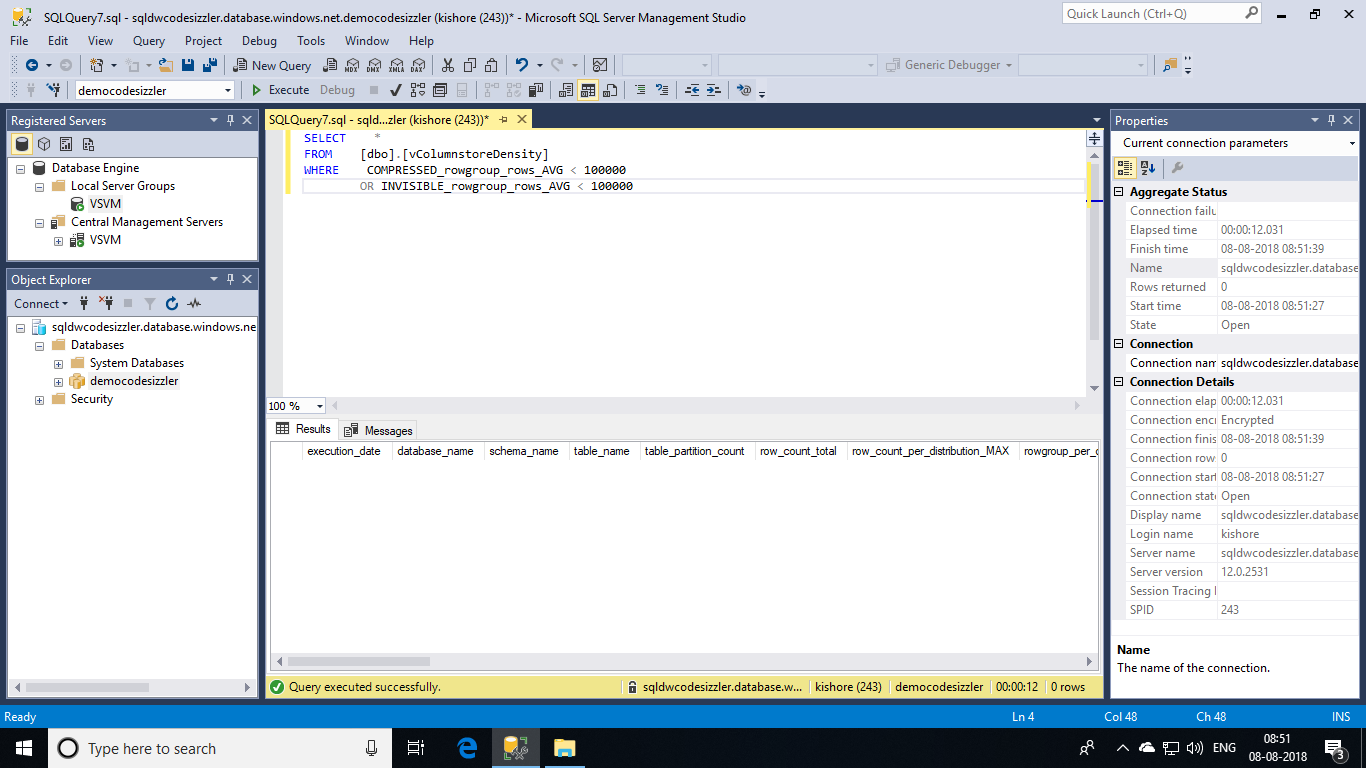
Now that you have created the view, run this query to identify tables with row groups with less than 100K rows. Of course, you may want to increase the threshold of 100K if you are looking for more optimal segment quality.

SELECT \*

FROM [dbo].[vColumnstoreDensity]

WHERE COMPRESSED\_rowgroup\_rows\_AVG < 100000

OR INVISIBLE\_rowgroup\_rows\_AVG < 100000



Once you have run the query you can begin to look at the data and analyse your results. This table explains what to look for in your row group analysis.