## **Partitioning tables in SQL Data Warehouse**

## **Partitions:**

Table partitions enable you to divide your data into smaller groups of data. In most cases, table partitions are created on a date column. Partitioning is supported on all SQL Data Warehouse table types; including clustered columnstore, clustered index, and heap. Partitioning is also supported on all distribution types, including both hash or round robin distributed.

Partitioning can benefit data maintenance and query performance. Whether it benefits both or just one is dependent on how data is loaded and whether the same column can be used for both purposes, since partitioning can only be done on one column.

### Benefits to loads

The primary benefit of partitioning in SQL Data Warehouse is to improve the efficiency and performance of loading data by use of partition deletion, switching and merging. In most cases data is partitioned on a date column that is closely tied to the order in which the data is loaded into the database. One of the greatest benefits of using partitions to maintain data it the avoidance of transaction logging. While simply inserting, updating, or deleting data can be the most straightforward approach, with a little thought and effort, using partitioning during your load process can substantially improve performance.

Partition switching can be used to quickly remove or replace a section of a table. For example, a sales fact table might contain just data for the past 36 months. At the end of every month, the oldest month of sales data is deleted from the table. This data could be deleted by using a delete statement to delete the data for the oldest month. However, deleting a large amount of data row-by-row with a delete statement can take too much time, as well as create the risk of large transactions that take a long time to rollback if something goes wrong. A more optimal approach is to drop the oldest partition of data. Where deleting the individual rows could take hours, deleting an entire partition could take seconds.

### Benefits to queries

Partitioning can also be used to improve query performance. A query that applies a filter to partitioned data can limit the scan to only the qualifying partitions. This method of filtering can avoid a full table scan and only scan a smaller subset of data. With the introduction of clustered columnstore indexes, the predicate elimination performance benefits are less beneficial, but in some cases there can be a benefit to queries. For example, if the sales fact table is partitioned into 36 months using the sales date field, then queries that filter on the sale date can skip searching in partitions that don’t match the filter.

## Sizing partitions

While partitioning can be used to improve performance some scenarios, creating a table with **too many** partitions can hurt performance under some circumstances. These concerns are especially true for clustered columnstore tables. For partitioning to be helpful, it is important to understand when to use partitioning and the number of partitions to create. There is no hard fast rule as to how many partitions are too many, it depends on your data and how many partitions you loading simultaneously. A successful partitioning scheme usually has tens to hundreds of partitions, not thousands.

When creating partitions on **clustered columnstore** tables, it is important to consider how many rows belong to each partition. For optimal compression and performance of clustered columnstore tables, a minimum of 1 million rows per distribution and partition is needed. Before partitions are created, SQL Data Warehouse already divides each table into 60 distributed databases. Any partitioning added to a table is in addition to the distributions created behind the scenes. Using this example, if the sales fact table contained 36 monthly partitions, and given that SQL Data Warehouse has 60 distributions, then the sales fact table should contain 60 million rows per month, or 2.1 billion rows when all months are populated. If a table contains fewer than the recommended minimum number of rows per partition, consider using fewer partitions in order to increase the number of rows per partition.

## **Syntax differences from SQL Server:**

SQL Data Warehouse introduces a way to define partitions that is simpler than SQL Server. Partitioning functions and schemes are not used in SQL Data Warehouse as they are in SQL Server. Instead, all you need to do is identify partitioned column and the boundary points. While the syntax of partitioning may be slightly different from SQL Server, the basic concepts are the same. SQL Server and SQL Data Warehouse support one partition column per table, which can be ranged partition.

The following example uses the **CREATE TABLE** statement to partition the FactInternetSales table on the OrderDateKey column. Run the below given query in your SQL Warehouse using SSM.

CREATE TABLE [dbo].[FactInternetSales]

(

[ProductKey] int NOT NULL

, [OrderDateKey] int NOT NULL

, [CustomerKey] int NOT NULL

, [PromotionKey] int NOT NULL

, [SalesOrderNumber] nvarchar(20) NOT NULL

, [OrderQuantity] smallint NOT NULL

, [UnitPrice] money NOT NULL

, [SalesAmount] money NOT NULL

)

WITH

( CLUSTERED COLUMNSTORE INDEX

, DISTRIBUTION = HASH([ProductKey])

, PARTITION ( [OrderDateKey] RANGE RIGHT FOR VALUES

(20000101,20010101,20020101

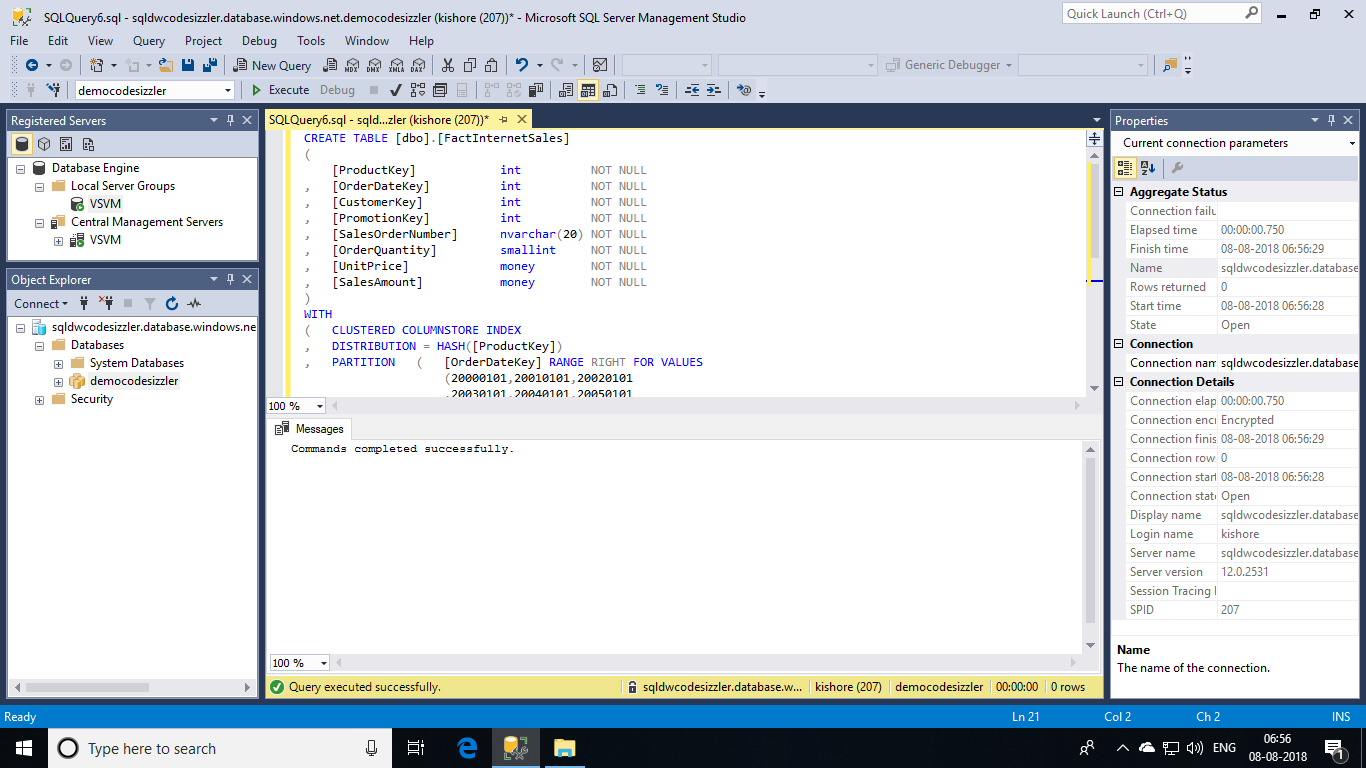
,20030101,20040101,20050101

)

)

)

;



## **Workload management:**

One final consideration to factor in to your table partition decision is workload management. Workload management in SQL Data Warehouse is primarily the management of memory and concurrency. In SQL Data Warehouse, the maximum memory allocated to each distribution during query execution is governed by resource classes. Ideally your partitions are sized in consideration of other factors like the memory needs of building clustered columnstore indexes. Clustered columnstore indexes benefit greatly when they are allocated more memory. Therefore, you want to ensure that a partition index rebuild is not starved of memory. Increasing the amount of memory available to your query can be achieved by switching from the default role, smallrc, to one of the other roles such as largerc.

Information on the allocation of memory per distribution is available by querying the Resource Governor dynamic management views. In reality, your memory grant is less than the results of the following query. However, this query provides a level of guidance that you can use when sizing your partitions for data management operations. Try to avoid sizing your partitions beyond the memory grant provided by the extra large resource class. If your partitions grow beyond this figure, you run the risk of memory pressure, which in turn leads to less optimal compression.

Now, give a try for the below given script.

SELECT rp.[name] AS [pool\_name]

, rp.[max\_memory\_kb] AS [max\_memory\_kb]

, rp.[max\_memory\_kb]/1024 AS [max\_memory\_mb]

, rp.[max\_memory\_kb]/1048576 AS [mex\_memory\_gb]

, rp.[max\_memory\_percent] AS [max\_memory\_percent]

, wg.[name] AS [group\_name]

, wg.[importance] AS [group\_importance]

, wg.[request\_max\_memory\_grant\_percent] AS [request\_max\_memory\_grant\_percent]

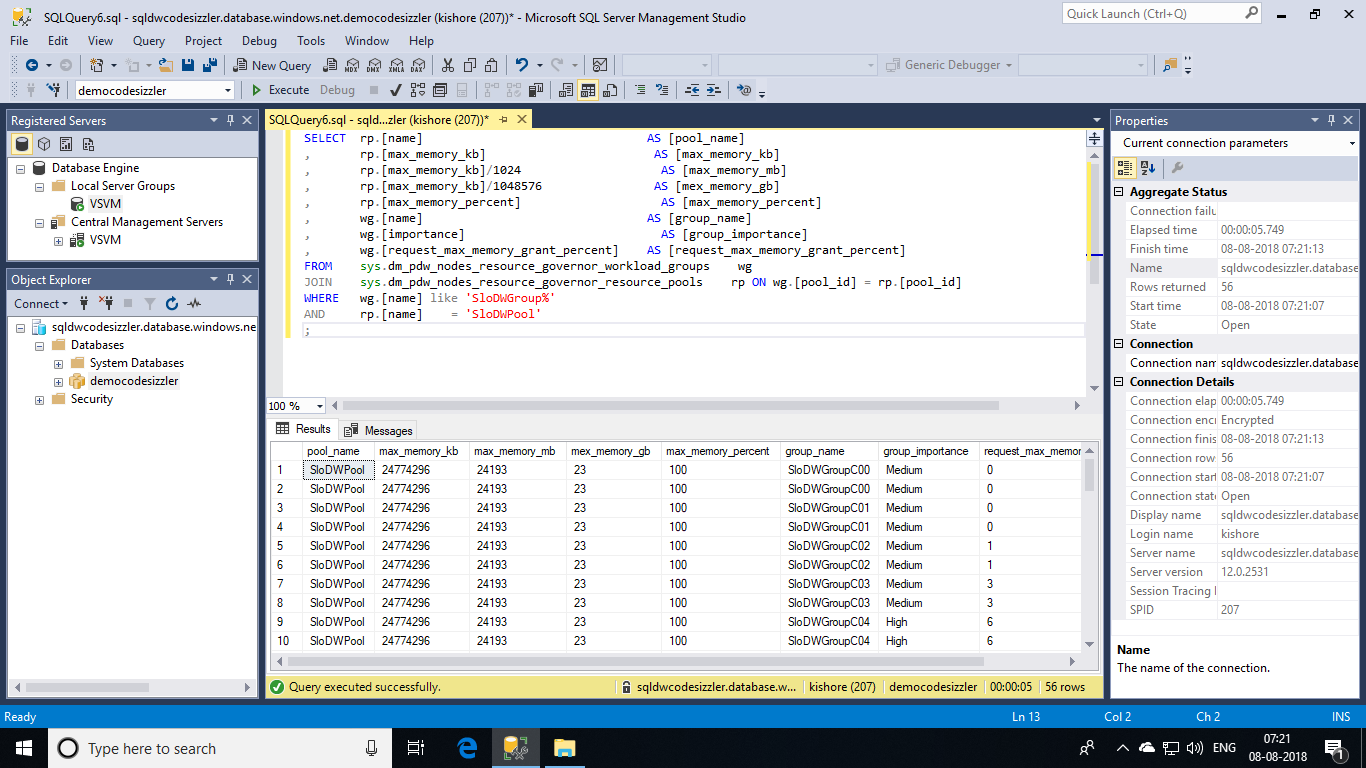
FROM sys.dm\_pdw\_nodes\_resource\_governor\_workload\_groups wg

JOIN sys.dm\_pdw\_nodes\_resource\_governor\_resource\_pools rp ON wg.[pool\_id] = rp.[pool\_id]

WHERE wg.[name] like 'SloDWGroup%'

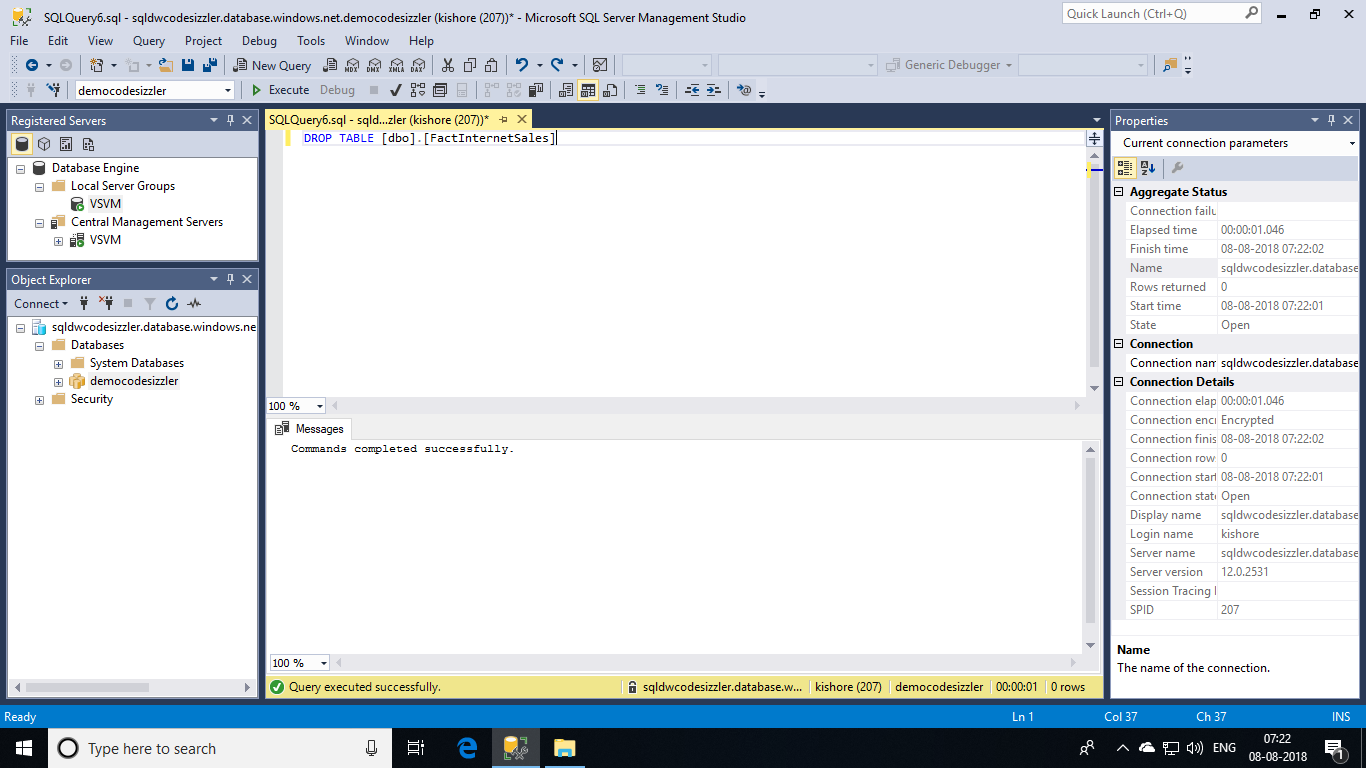
AND rp.[name] = 'SloDWPool'

;



Now, remove the table **[dbo].[FactInternetSales]** from the database to continue further Demos.

DROP TABLE [dbo].[FactInternetSales]



### **Splitting a partition that contains data:**

The most efficient method to split a partition that already contains data is to use a CTAS statement. If the partitioned table is a clustered columnstore, then the table partition must be empty before it can be split.

The following example creates a partitioned columnstore table. It inserts one row into each partition:

CREATE TABLE [dbo].[FactInternetSales]

(

[ProductKey] int NOT NULL

, [OrderDateKey] int NOT NULL

, [CustomerKey] int NOT NULL

, [PromotionKey] int NOT NULL

, [SalesOrderNumber] nvarchar(20) NOT NULL

, [OrderQuantity] smallint NOT NULL

, [UnitPrice] money NOT NULL

, [SalesAmount] money NOT NULL

)

WITH

( CLUSTERED COLUMNSTORE INDEX

, DISTRIBUTION = HASH([ProductKey])

, PARTITION ( [OrderDateKey] RANGE RIGHT FOR VALUES

(20000101

)

)

)

;

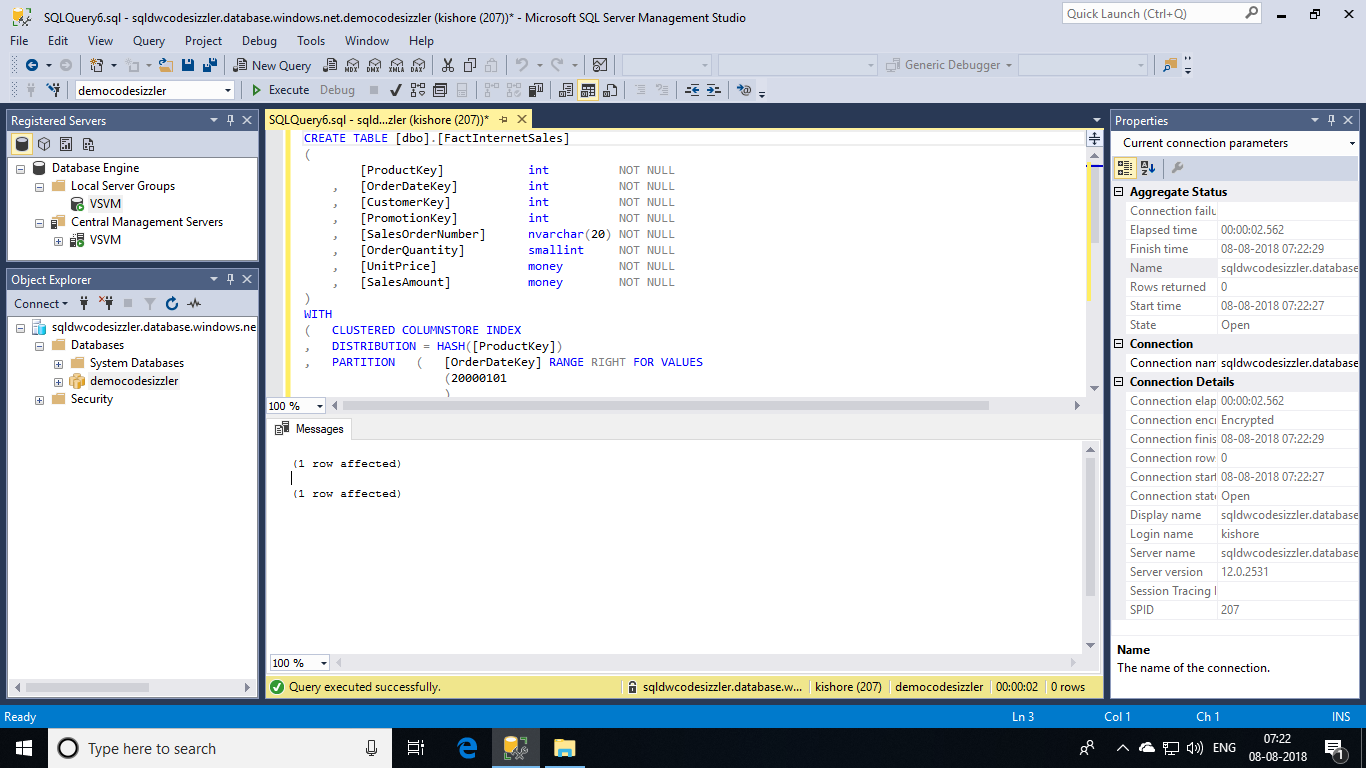
INSERT INTO dbo.FactInternetSales

VALUES (1,19990101,1,1,1,1,1,1);

INSERT INTO dbo.FactInternetSales

VALUES (1,20000101,1,1,1,1,1,1);

CREATE STATISTICS Stat\_dbo\_FactInternetSales\_OrderDateKey ON dbo.FactInternetSales(OrderDateKey);



The following query finds the row count by using the **sys.partitions** catalogue view:

SELECT QUOTENAME(s.[name])+'.'+QUOTENAME(t.[name]) as Table\_name

, i.[name] as Index\_name

, p.partition\_number as Partition\_nmbr

, p.[rows] as Row\_count

, p.[data\_compression\_desc] as Data\_Compression\_desc

FROM sys.partitions p

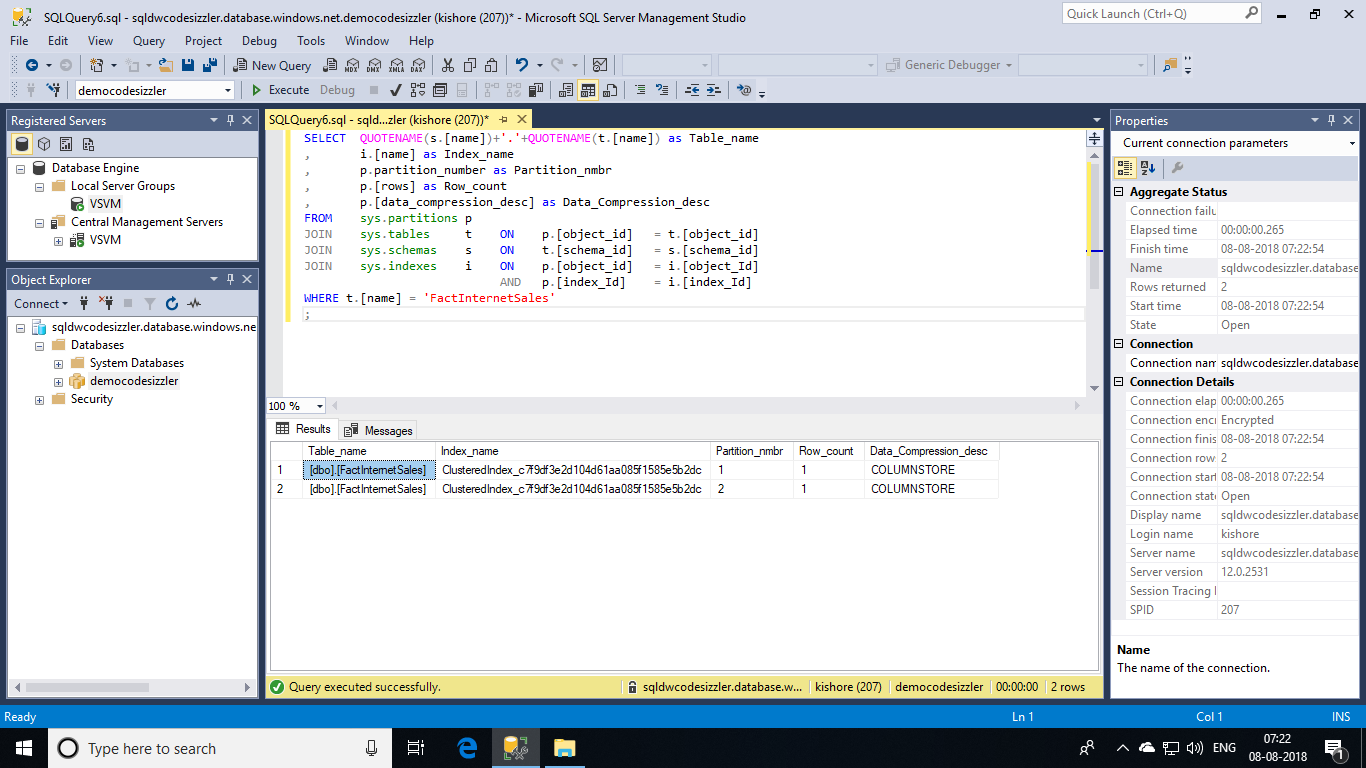
JOIN sys.tables t ON p.[object\_id] = t.[object\_id]

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

JOIN sys.indexes i ON p.[object\_id] = i.[object\_Id]

AND p.[index\_Id] = i.[index\_Id]

WHERE t.[name] = 'FactInternetSales';



The following split command receives an error message:

ALTER TABLE FactInternetSales SPLIT RANGE (20010101);

**Msg 35346, Level 15, State 1, Line 44 SPLIT clause of ALTER PARTITION statement failed because the partition is not empty**. You will get an error like this. Only empty partitions can be split in when a columnstore index exists on the table. Consider disabling the columnstore index before issuing the ALTER PARTITION statement, then rebuilding the columnstore index after ALTER PARTITION is complete.

However, you can use CTAS to create a new table to hold the data.

CREATE TABLE dbo.FactInternetSales\_20000101

WITH ( DISTRIBUTION = HASH(ProductKey)

, CLUSTERED COLUMNSTORE INDEX

, PARTITION ( [OrderDateKey] RANGE RIGHT FOR VALUES

(20000101

)

)

)

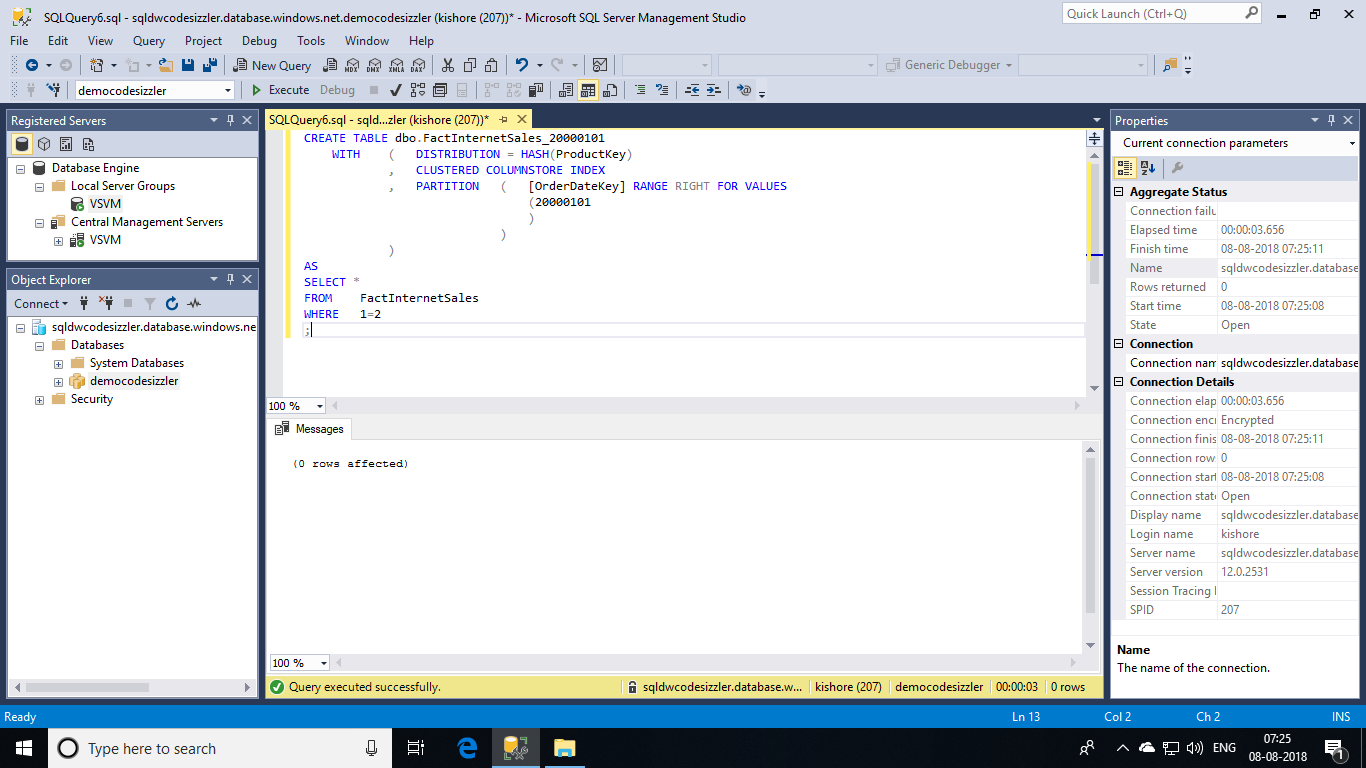
AS

SELECT \*

FROM FactInternetSales

WHERE 1=2

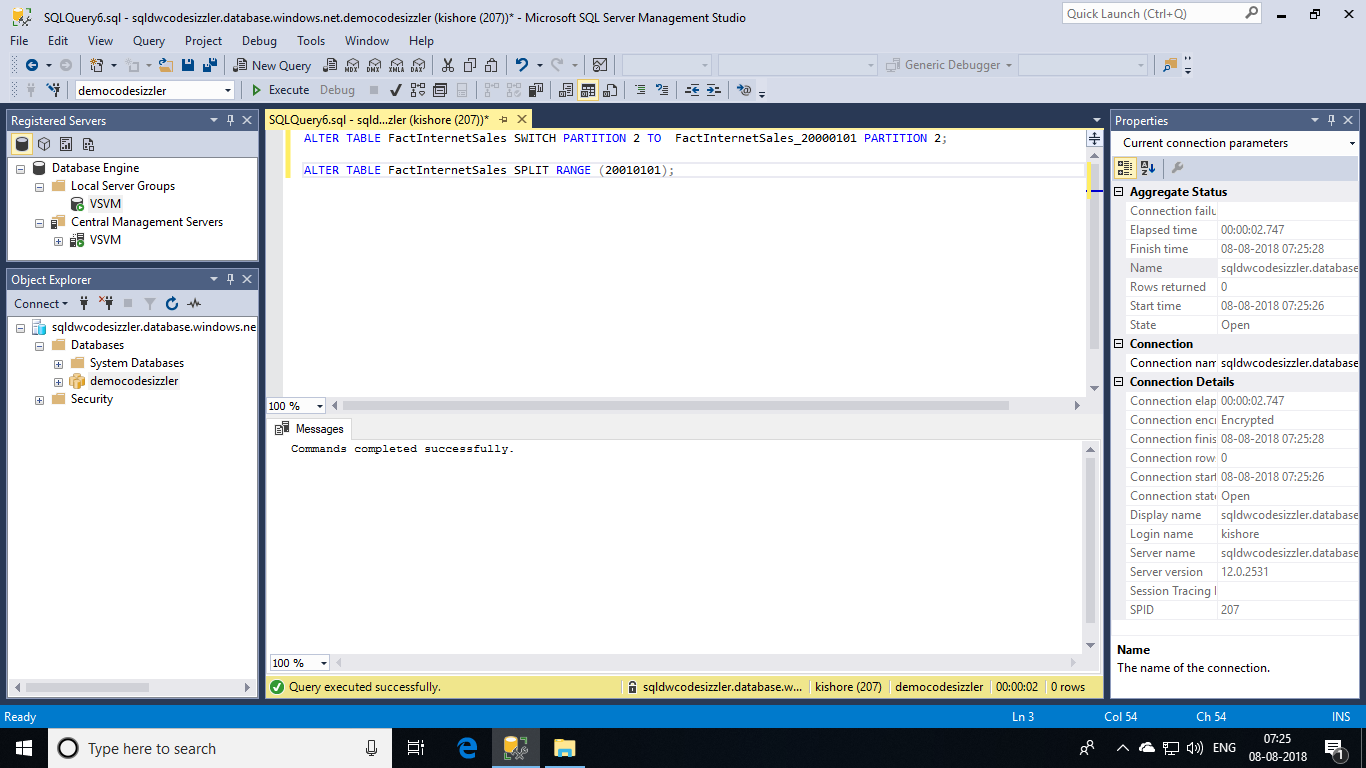
;



As the partition boundaries are aligned, a switch is permitted. This will leave the source table with an empty partition that you can subsequently split.

ALTER TABLE FactInternetSales SWITCH PARTITION 2 TO FactInternetSales\_20000101 PARTITION 2;

ALTER TABLE FactInternetSales SPLIT RANGE (20010101);



All that is left is to align the data to the new partition boundaries using CTAS, and then switch the data back into the main table.

CREATE TABLE [dbo].[FactInternetSales\_20000101\_20010101]

WITH ( DISTRIBUTION = HASH([ProductKey])

, CLUSTERED COLUMNSTORE INDEX

, PARTITION ( [OrderDateKey] RANGE RIGHT FOR VALUES

(20000101,20010101

)

)

)

AS

SELECT \*

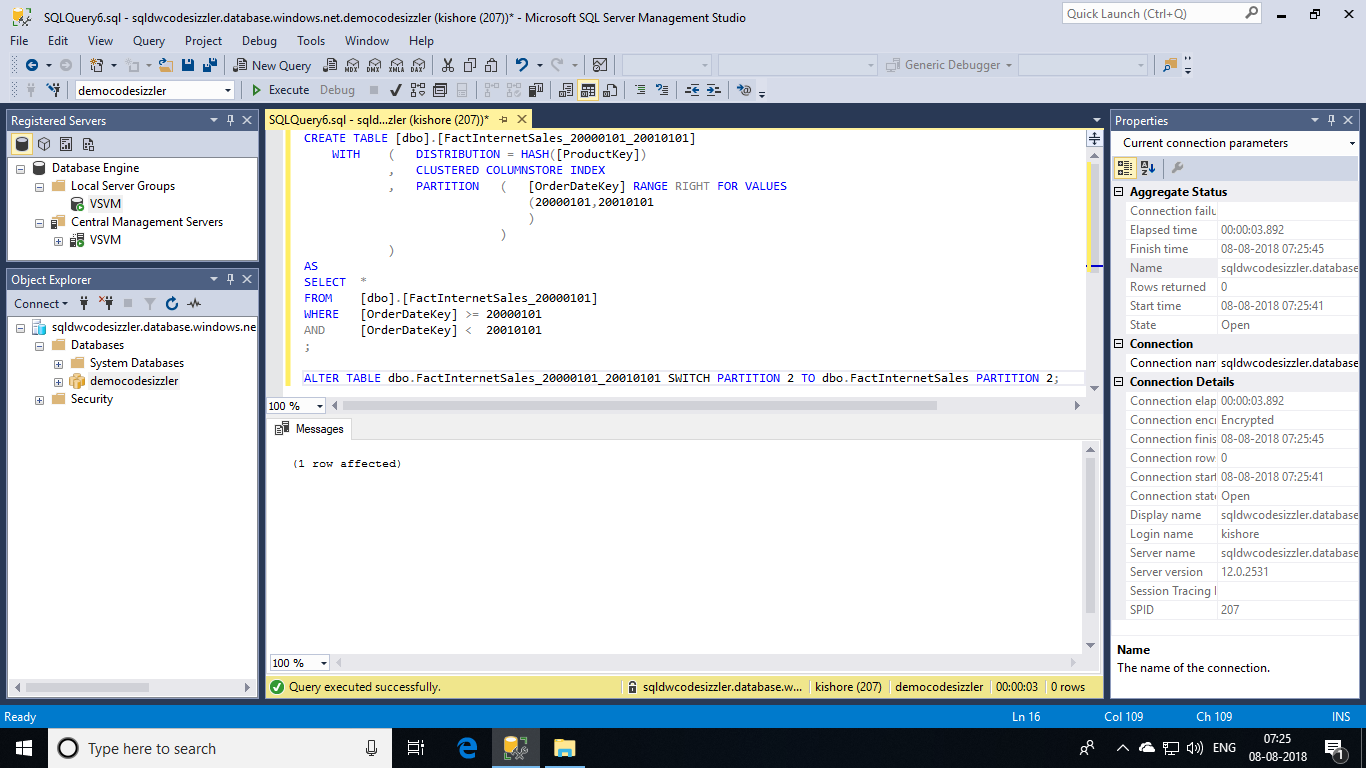
FROM [dbo].[FactInternetSales\_20000101]

WHERE [OrderDateKey] >= 20000101

AND [OrderDateKey] < 20010101

;

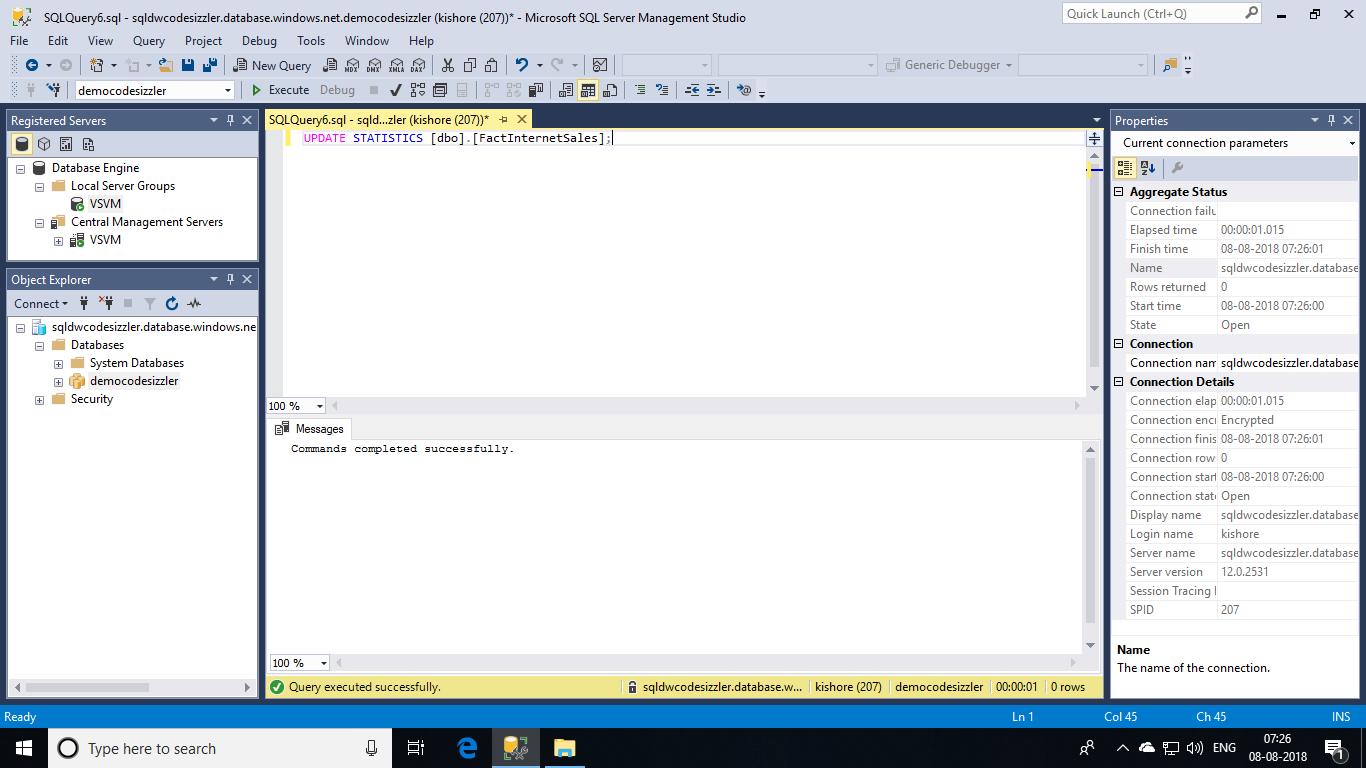
ALTER TABLE dbo.FactInternetSales\_20000101\_20010101 SWITCH PARTITION 2 TO dbo.FactInternetSales PARTITION 2;



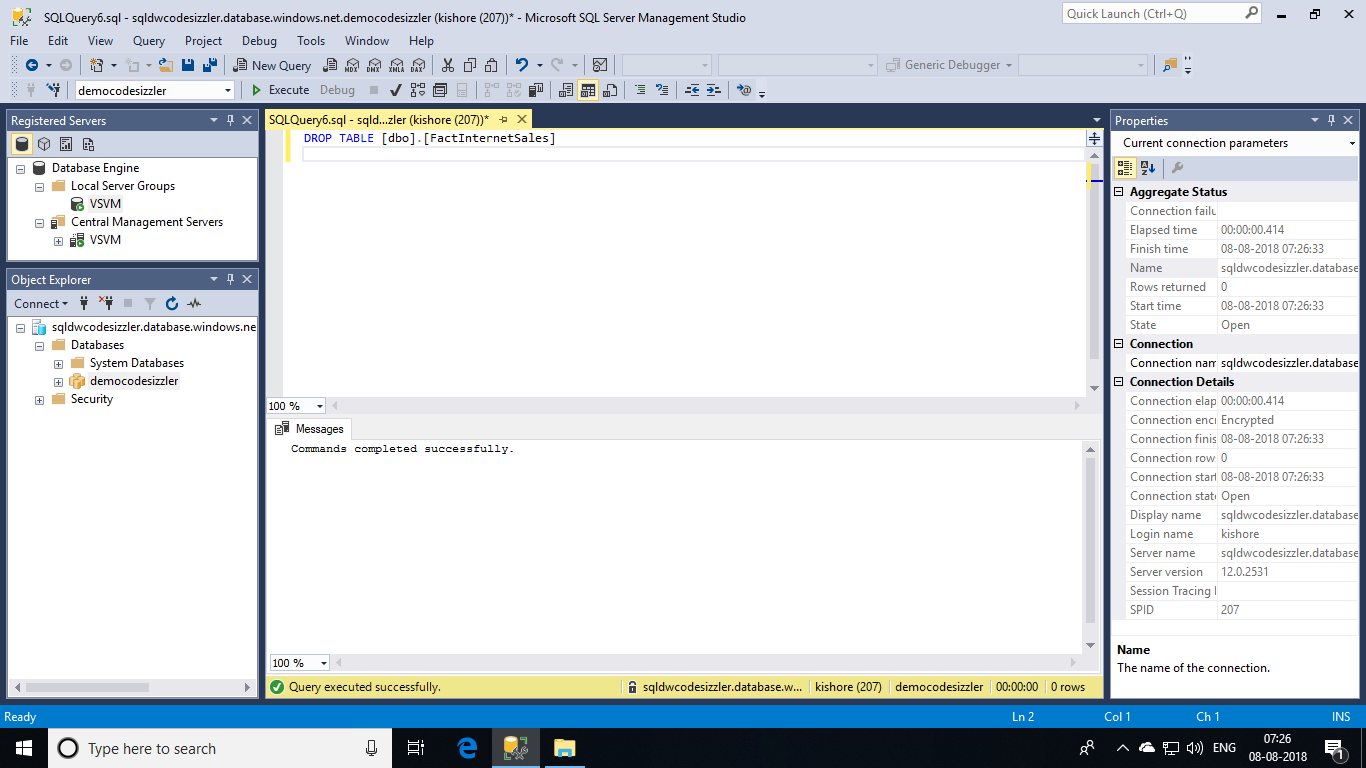
Once you have completed the movement of the data, it is a good idea to refresh the statistics on the target table. Updating statistics ensures the statistics accurately reflect the new distribution of the data in their respective partitions.

SQLCopy

UPDATE STATISTICS [dbo].[FactInternetSales];



Again **DROP** the table **[dbo].[FactInternetSales]** to continue further in demos.



### **Table partitioning source control:**

To avoid your table definition from **rusting** in your source control system, you may want to consider the following approach:

1. Create the table as a partitioned table but with no partition values

CREATE TABLE [dbo].[FactInternetSales]

(

[ProductKey] int NOT NULL

, [OrderDateKey] int NOT NULL

, [CustomerKey] int NOT NULL

, [PromotionKey] int NOT NULL

, [SalesOrderNumber] nvarchar(20) NOT NULL

, [OrderQuantity] smallint NOT NULL

, [UnitPrice] money NOT NULL

, [SalesAmount] money NOT NULL

)

WITH

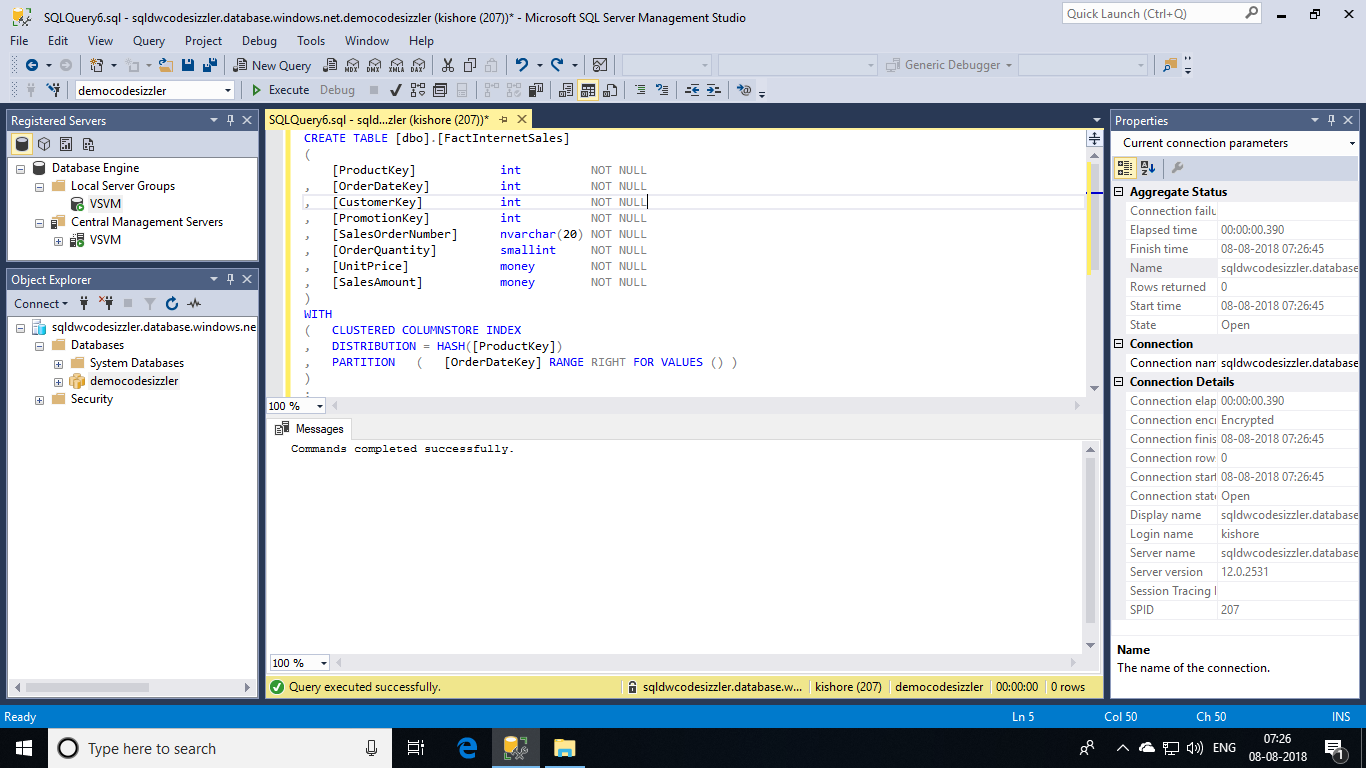
( CLUSTERED COLUMNSTORE INDEX

, DISTRIBUTION = HASH([ProductKey])

, PARTITION ( [OrderDateKey] RANGE RIGHT FOR VALUES () )

)

;



1. **SPLIT** the table as part of the deployment process:

-- Create a table containing the partition boundaries

CREATE TABLE #partitions

WITH

(

LOCATION = USER\_DB

, DISTRIBUTION = HASH(ptn\_no)

)

AS

SELECT ptn\_no

, ROW\_NUMBER() OVER (ORDER BY (ptn\_no)) as seq\_no

FROM (

SELECT CAST(20000101 AS INT) ptn\_no

UNION ALL

SELECT CAST(20010101 AS INT)

UNION ALL

SELECT CAST(20020101 AS INT)

UNION ALL

SELECT CAST(20030101 AS INT)

UNION ALL

SELECT CAST(20040101 AS INT)

) a

;

-- Iterate over the partition boundaries and split the table

DECLARE @c INT = (SELECT COUNT(\*) FROM #partitions)

, @i INT = 1 --iterator for while loop

, @q NVARCHAR(4000) --query

, @p NVARCHAR(20) = N'' --partition\_number

, @s NVARCHAR(128) = N'dbo' --schema

, @t NVARCHAR(128) = N'FactInternetSales' --table

;

WHILE @i <= @c

BEGIN

SET @p = (SELECT ptn\_no FROM #partitions WHERE seq\_no = @i);

SET @q = (SELECT N'ALTER TABLE '+@s+N'.'+@t+N' SPLIT RANGE ('+@p+N');');

-- PRINT @q;

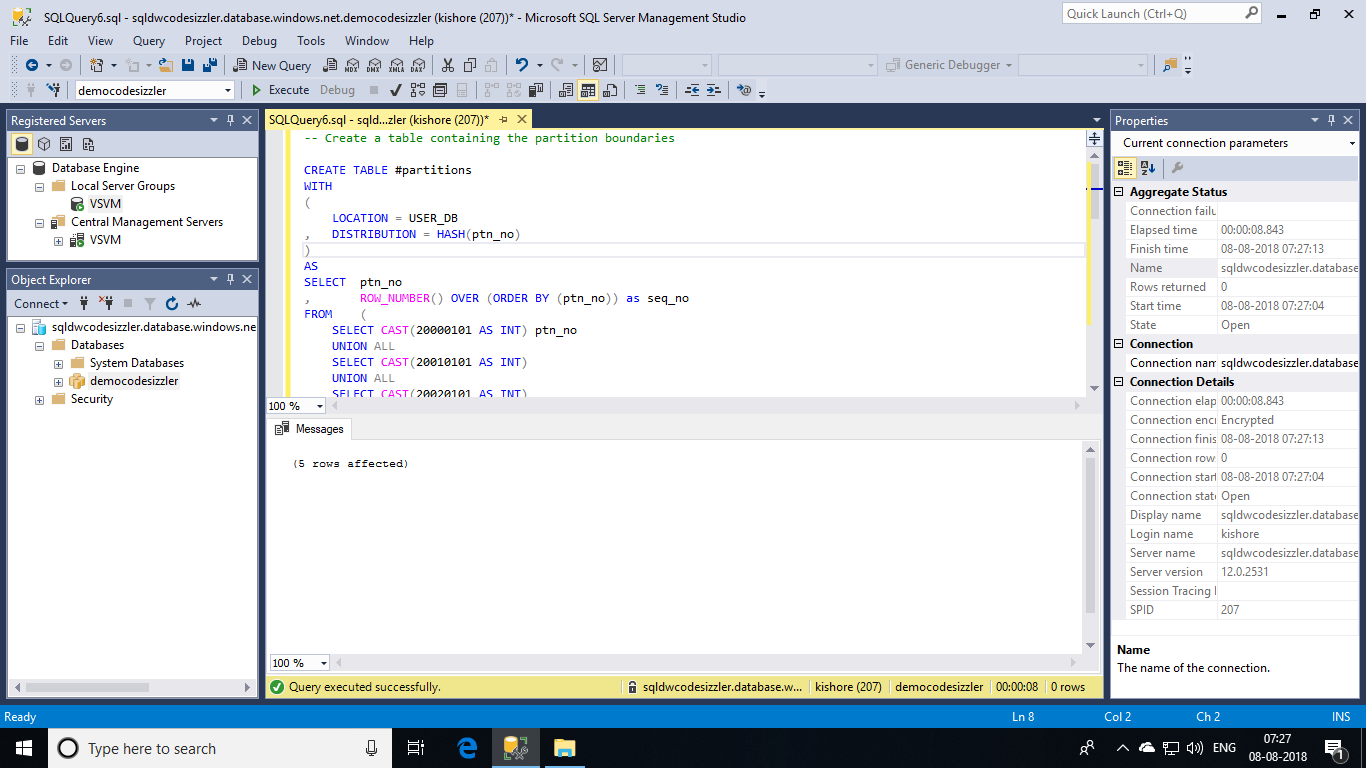
EXECUTE sp\_executesql @q;

SET @i+=1;

END

-- Code clean-up

DROP TABLE #partitions;



With this approach the code in source control remains static and the partitioning boundary values are allowed to be dynamic; evolving with the warehouse over time.