Microsoft

SQL Server 2016 CTP2 Columnstore HOL

Contents

Overview and setup

Estimated time to complete lab is 40-45 minutes.

Columnstore indexes are memory-optimized, column-oriented indexes used primarily in data warehousing scenarios, though as we will see, they can also be used for operational analytics. Columnstore indexes can be created as clustered and non-clustered index. Columnstore indexes organize data in columns to achieve high level of data compression to reduce both the storage cost and the IO. Though the compression will depend on the data, but typically you can expect 10x compression over uncompressed data 5x compression if the data was compressed with PAGE compression. Additionally, columnstore provides better query performance using batch-mode execution that processes a set of rows together for execution efficiency. Starting with SQL Server 2016, the Columnstore indexes can be used on memory-optimized tables.

This hands on lab will familiarize you with Columnstore indexes and help you see the positive performance impact they can have. It will also highlight some of the improvements in these indexes with SQL Server 2016 CTP2. In particular, you will learn:

1. A table with clustered Columnstore can now enforce referential integrity through foreign key constraints;
2. A table with clustered Columnstore tables can now have one or more non-clustered btree indexes for efficient execution of queries with equality and short-range predicates.;
3. A table with clustered or non-clustered columnstore indexes can run analytics queries efficiently.;
4. A table with non-clustered columnstore (NCCI) now allows DML operations. In earlier releases of SQL Server, a table with NCCI was only available for querying.
5. A Columnstore index on operational schema can provide efficient execution of analytics with minimal impact on OLTP workload.

At the end of this lab, you will have worked through some of the most common scenarios involved with Columnstore indexes and will have learned about some of the most significant improvements to Columnstore indexes in SQL Server 2016 CTP2.

Setting up your environment

1. Log into virtual machine environment using the following account information.

```
User: labuser
Password: Pass@word12
```

Note, if you have a monitor that supports a larger screen resolution than 1024 x 768, you can change the screen resolution for the lab to go as high as 1920 x 1080. By going to a higher screen resolution, it will be easier to use SQL Server Management Studio.

2. Right click on the desktop and click on **Screen resolution**.

3. Select **1366 x 786** (a good minimum screen size for using SSMS) and click **OK**.

4. Click **Keep Changes**.

5. The script file that contains the script to be used in this lab is located in the **C:\SQL Server 2016 CPT2 HOLs\LQS** folder.

6. The script file that contains the scripts to be used in this lab is located in the **C:\SQL Server 2016 CPT2 HOLs\Columnstore** folder.

7. Open **notepad.exe** to keep track of any information you want to save.

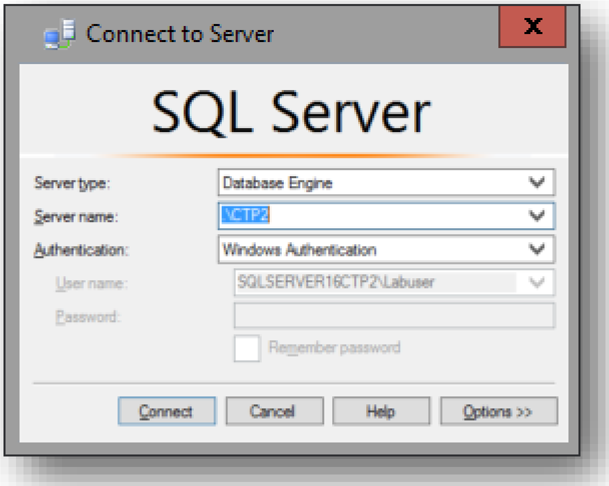
8. This lab will be using the database **AdventureWorks2016DW**. Open up SQL Server Management Studio and set that as your current database before running any code.

Explore Columnstore

In this exercise, you will explore a table with a clustered Columnstore index that has more than 11 million rows, set a foreign key, and run a query on the table. Let's get started!

Verify the Columnstore index and size of the table

1. Open **SQL Server Management Studio** and connect to the **.\CTP2** database engine instance.



Server type: Database Engine

Server name: .\CTP2

Authentication: Windows Authentication

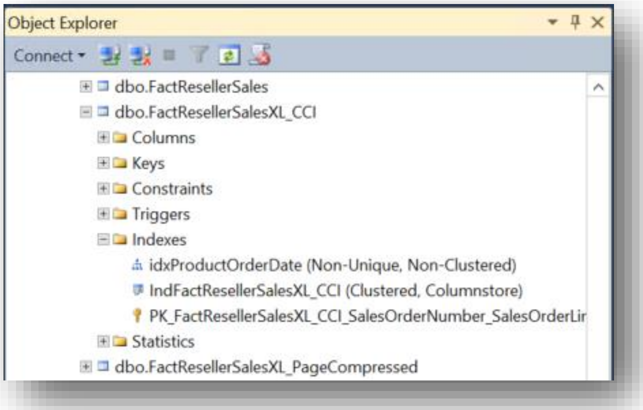
User name: SQLSERVER16CTP2\Labuser

Password:

□ Remember password

2. Expand the **Databases** node and click on **AdventureWorks2016DW**.

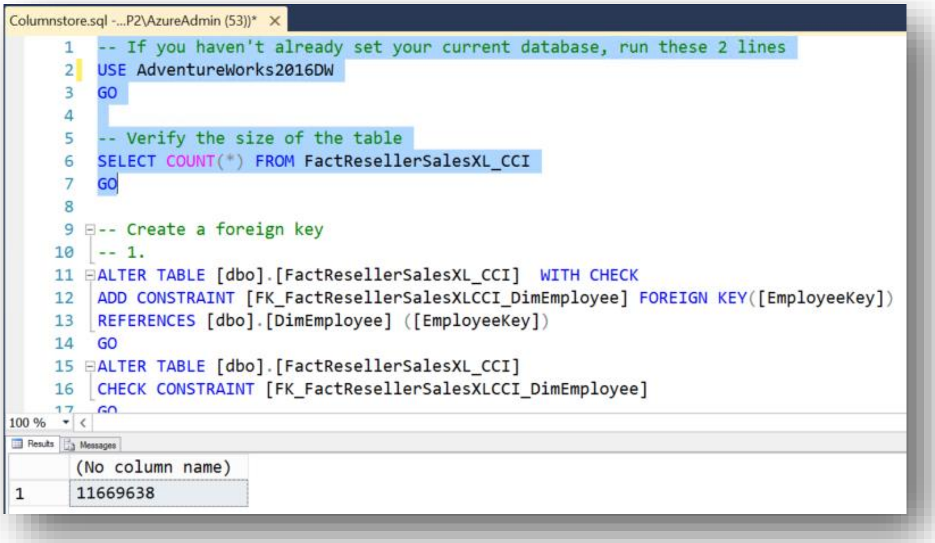
3. Expand the information in **Object Explorer** for the **dbo.FactResellerSalesXL_CCI** table in the **AdventureWorks2016DW** database and expand the **Indexes** folder.



```
⊞ dbo.FactResellerSales
⊟ dbo.FactResellerSalesXL_CCI
    ⊞ Columns
    ⊞ Keys
    ⊞ Constraints
    ⊞ Triggers
    ⊟ Indexes
        idxProductOrderDate (Non-Unique, Non-Clustered)
        IndFactResellerSalesXL_CCI (Clustered, Columnstore)
        PK_FactResellerSalesXL_CCI_SalesOrderNumber_SalesOrderLir
    ⊞ Statistics
⊞ dbo.FactResellerSalesXL_PageCompressed
```

Verify that there is a CCI (clustered columnstore index) on the table.

4. Open the **C:\SQL Server 2016 CPT2 HOLs\Columnstore\Columnstore.sql** in SSMS and run the selected code below.



```
-- If you haven't already set your current database, run these 2 lines
USE AdventureWorks2016DW
GO

-- Verify the size of the table
SELECT COUNT(*) FROM FactResellerSalesXL_CCI
GO

-- Create a foreign key
-- 1.
ALTER TABLE [dbo].[FactResellerSalesXL_CCI]  WITH CHECK
ADD CONSTRAINT [FK_FactResellerSalesXLCCI_DimEmployee] FOREIGN KEY([EmployeeKey])
REFERENCES [dbo].[DimEmployee] ([EmployeeKey])
GO
ALTER TABLE [dbo].[FactResellerSalesXL_CCI]
CHECK CONSTRAINT [FK_FactResellerSalesXLCCI_DimEmployee]
GO
```

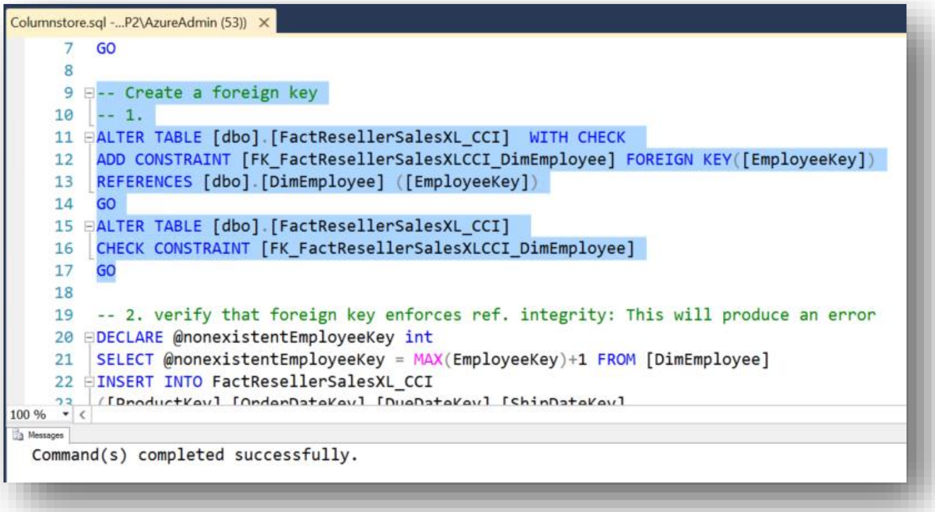
```
  (No column name)
1 11669638
```

Verify that the table has more than 11 million rows.

Create a foreign key

Note that in SQL Server 2014 you could not create a foreign key relationship on a table with a Columnstore index on it.

1. Run the following two statements to create a foreign key relationship with the **DimEmployee** table.

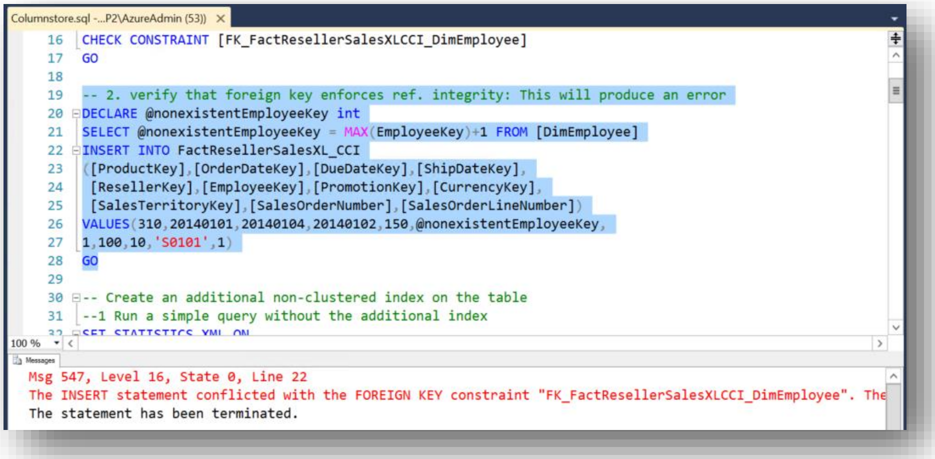


```
GO

-- Create a foreign key
-- 1.
ALTER TABLE [dbo].[FactResellerSalesXL_CCI]  WITH CHECK
ADD CONSTRAINT [FK_FactResellerSalesXLCCI_DimEmployee] FOREIGN KEY([EmployeeKey])
REFERENCES [dbo].[DimEmployee] ([EmployeeKey])
GO
ALTER TABLE [dbo].[FactResellerSalesXL_CCI]
CHECK CONSTRAINT [FK_FactResellerSalesXLCCI_DimEmployee]
GO

-- 2. verify that foreign key enforces ref. integrity: This will produce an error
DECLARE @nonexistentEmployeeKey int
SELECT @nonexistentEmployeeKey = MAX(EmployeeKey)+1 FROM [DimEmployee]
INSERT INTO FactResellerSalesXL_CCI
([ProductKey],[OrderDateKey],[DueDateKey],[ShipDateKey]
```

```
Command(s) completed successfully.
```

2. Verify that the foreign key constraint maintains referential integrity, by not allowing a non-existing key from the

DimEmployee table to be inserted into the **FactResellerSalesXL_CCI** table, with the following script.



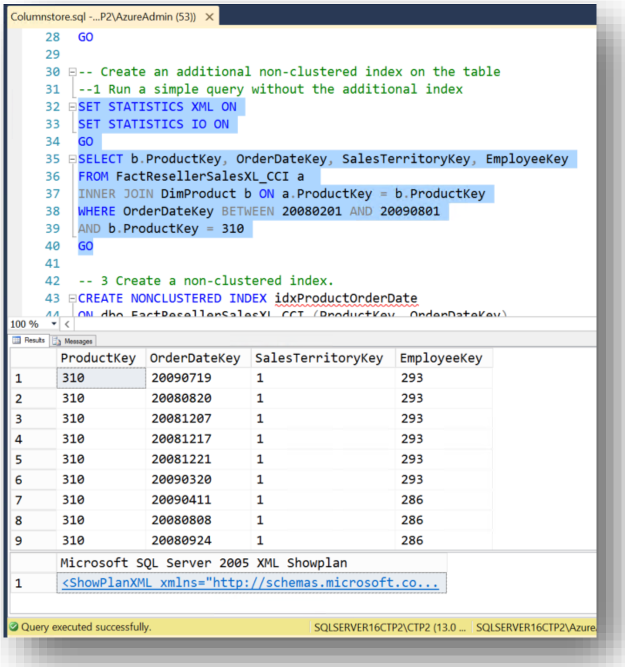
```
CHECK CONSTRAINT [FK_FactResellerSalesXLCCI_DimEmployee]
GO

-- 2. verify that foreign key enforces ref. integrity: This will produce an error
DECLARE @nonexistentEmployeeKey int
SELECT @nonexistentEmployeeKey = MAX(EmployeeKey)+1 FROM [DimEmployee]
INSERT INTO FactResellerSalesXL_CCI
([ProductKey],[OrderDateKey],[DueDateKey],[ShipDateKey],
 [ResellerKey],[EmployeeKey],[PromotionKey],[CurrencyKey],
 [SalesTerritoryKey],[SalesOrderNumber],[SalesOrderLineNumber])
VALUES(310,20140101,20140104,20140102,150,@nonexistentEmployeeKey,
1,100,10,'S0101',1)
GO

-- Create an additional non-clustered index on the table
--1 Run a simple query without the additional index
SET STATISTICS XML ON
```

```
Msg 547, Level 16, State 0, Line 22
The INSERT statement conflicted with the FOREIGN KEY constraint "FK_FactResellerSalesXLCCI_DimEmployee". The
The statement has been terminated.
```

Create an additional non-clustered index on the table

Note that in SQL Server 2014 you could not create an additional index on a table that already had a clustered Columnstore index on it.

3. Run the selected query without the additional index on the table.

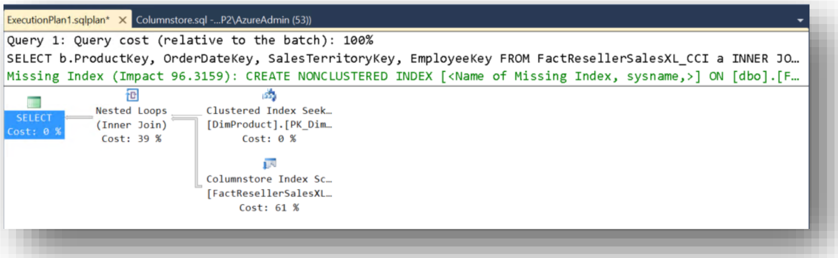

Columnstore.sql -...P2\AzureAdmin (53))
100 %
Results
Messages
Query executed successfully.
SQLSERVER16CTP2\CTP2 (13.0 ...
SQLSERVER16CTP2\Azure

```
GO

-- Create an additional non-clustered index on the table
--1 Run a simple query without the additional index
SET STATISTICS XML ON
SET STATISTICS IO ON
GO
SELECT b.ProductKey, OrderDateKey, SalesTerritoryKey, EmployeeKey
FROM FactResellerSalesXL_CCI a
INNER JOIN DimProduct b ON a.ProductKey = b.ProductKey
WHERE OrderDateKey BETWEEN 20080201 AND 20090801
AND b.ProductKey = 310
GO

-- 3 Create a non-clustered index.
CREATE NONCLUSTERED INDEX idxProductOrderDate
ON dbo.FactResellerSalesXL_CCI (ProductKey, OrderDateKey)
```

	ProductKey	OrderDateKey	SalesTerritoryKey	EmployeeKey
1	310	20090719	1	293
2	310	20080820	1	293
3	310	20081207	1	293
4	310	20081217	1	293
5	310	20081221	1	293
6	310	20090320	1	293
7	310	20090411	1	286
8	310	20080808	1	286
9	310	20080924	1	286

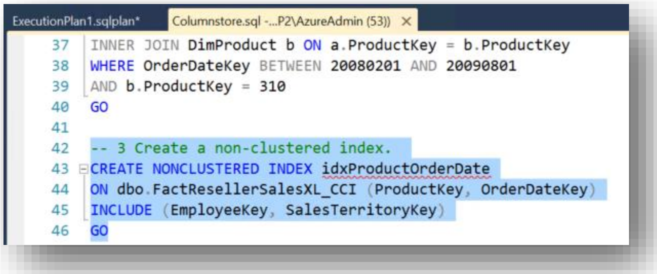
	Microsoft SQL Server 2005 XML Showplan
1	<ShowPlanXML xmlns="http://schemas.microsoft.co...

4. Click on **ShowPlanXML** link to view the query execution plan and notice the **Missing Index** hint. Also note the scanning of the entire index.



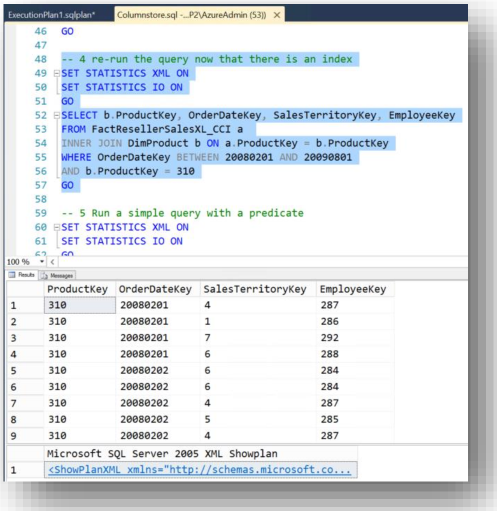
```
Query 1: Query cost (relative to the batch): 100%
SELECT b.ProductKey, OrderDateKey, SalesTerritoryKey, EmployeeKey FROM FactResellerSalesXL_CCI a INNER JO…
Missing Index (Impact 96.3159): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[F…
```

5. Go back to the **Columnstore.sql** query editor window and execute the following statement to add a new, nonclustered index. This will take a few minutes to run.



```
INNER JOIN DimProduct b ON a.ProductKey = b.ProductKey
WHERE OrderDateKey BETWEEN 20080201 AND 20090801
AND b.ProductKey = 310
GO

-- 3 Create a non-clustered index.
CREATE NONCLUSTERED INDEX idxProductOrderDate
ON dbo.FactResellerSalesXL_CCI (ProductKey, OrderDateKey)
INCLUDE (EmployeeKey, SalesTerritoryKey)
GO
```

6. Now, run the above selected query again and look at the execution plan.

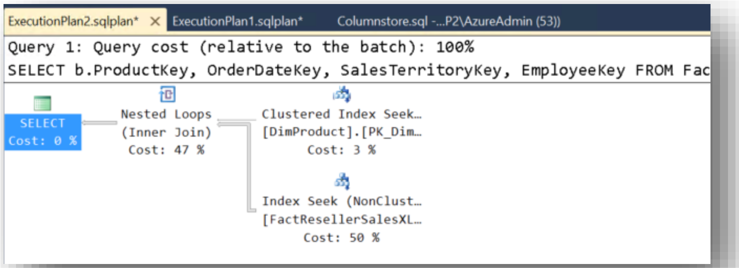


```
GO

-- 4 re-run the query now that there is an index
SET STATISTICS XML ON
SET STATISTICS IO ON
GO
SELECT b.ProductKey, OrderDateKey, SalesTerritoryKey, EmployeeKey
FROM FactResellerSalesXL_CCI a
INNER JOIN DimProduct b ON a.ProductKey = b.ProductKey
WHERE OrderDateKey BETWEEN 20080201 AND 20090801
AND b.ProductKey = 310
GO

-- 5 Run a simple query with a predicate
SET STATISTICS XML ON
SET STATISTICS IO ON
GO
```

	ProductKey	OrderDateKey	SalesTerritoryKey	EmployeeKey
1	310	20080201	4	287
2	310	20080201	1	286
3	310	20080201	7	292
4	310	20080201	6	288
5	310	20080202	6	284
6	310	20080202	6	284
7	310	20080202	4	287
8	310	20080202	5	285
9	310	20080202	4	287

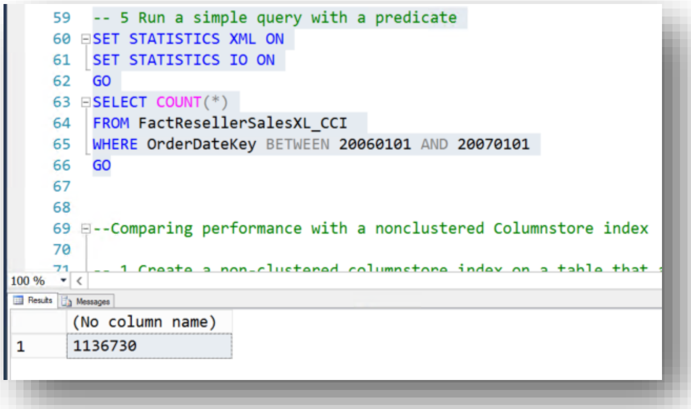
	Microsoft SQL Server 2005 XML Showplan
1	<ShowPlanXML xmlns="http://schemas.microsoft.co...

7. The index scan has now changed to an index seek.



```
Query 1: Query cost (relative to the batch): 100%
SELECT b.ProductKey, OrderDateKey, SalesTerritoryKey, EmployeeKey FROM Fac
```

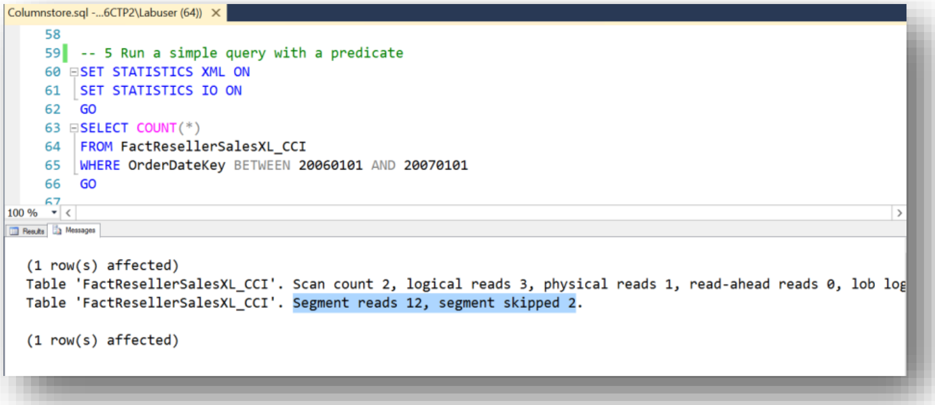
8. Let’s run selected query and look at how many segments were used and how many did not have to be checked because of the index.



```
-- 5 Run a simple query with a predicate
SET STATISTICS XML ON
SET STATISTICS IO ON
GO
SELECT COUNT(*)
FROM FactResellerSalesXL_CCI
WHERE OrderDateKey BETWEEN 20060101 AND 20070101
GO


--Comparing performance with a nonclustered Columnstore index

-- 1 Create a non-clustered columnstore index on a table that
```

	(No column name)
1	1136730

9. Look at the **Messages** tab. Notice that 12 segments were read and 2 did not have to be read. This reduces the amount of I/O that occurs, improving performance.



```

-- 5 Run a simple query with a predicate
SET STATISTICS XML ON
SET STATISTICS IO ON
GO
SELECT COUNT(*)
FROM FactResellerSalesXL_CCI
WHERE OrderDateKey BETWEEN 20060101 AND 20070101
GO
```

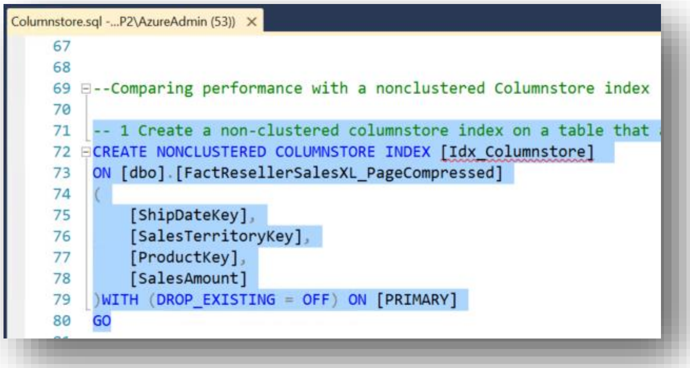
```
(1 row(s) affected)
Table 'FactResellerSalesXL_CCI'. Scan count 2, logical reads 3, physical reads 1, read-ahead reads 0, lob log
Table 'FactResellerSalesXL_CCI'. Segment reads 12, segment skipped 2.

(1 row(s) affected)
```

Add a Columnstore index to an existing table

In this exercise we will start with a large table, with more than 11 million rows, that is currently being used operationally. We will add a nonclustered Columnstore index to it, in order to speed up analytical queries on it. We will observe the performance impact of querying with and without the index. Then we will demonstrate that in SQL Server 2016 CTP2, tables with a nonclustered Columnstore index can now have new data inserted into them.

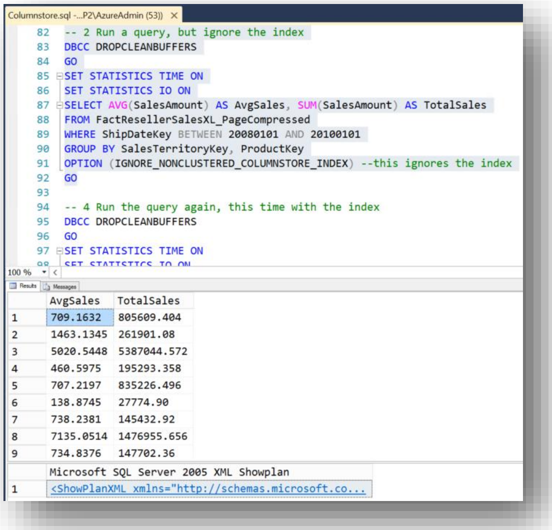
Comparing performance with a nonclustered Columnstore index

1. Create a new nonclustered Columnstore index on the **FactResellerSalesXL_PageCompressed** table.



```
--Comparing performance with a nonclustered Columnstore index

-- 1 Create a non-clustered columnstore index on a table that
CREATE NONCLUSTERED COLUMNSTORE INDEX [Idx_Columnstore]
ON [dbo].[FactResellerSalesXL_PageCompressed]
(
    [ShipDateKey],
    [SalesTerritoryKey],
    [ProductKey],
    [SalesAmount]
)WITH (DROP_EXISTING = OFF) ON [PRIMARY]
GO
```

2. We can first see what the performance would have been like by adding an option to ignore the index.

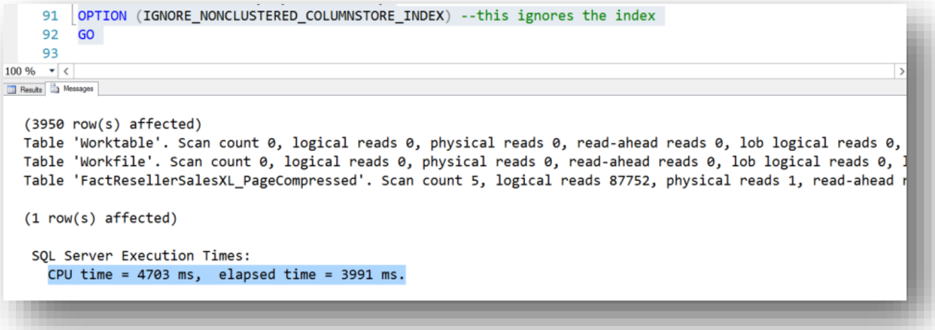


```
-- 2 Run a query, but ignore the index
DBCC DROPCLEANBUFFERS
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
SELECT AVG(SalesAmount) AS AvgSales, SUM(SalesAmount) AS TotalSales
FROM FactResellerSalesXL_PageCompressed
WHERE ShipDateKey BETWEEN 20080101 AND 20100101
GROUP BY SalesTerritoryKey, ProductKey
OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX) --this ignores the index
GO

-- 4 Run the query again, this time with the index
DBCC DROPCLEANBUFFERS
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
```

	AvgSales	TotalSales
1	709.1632	805609.404
2	1463.1345	261901.08
3	5020.5448	5387044.572
4	460.5975	195293.358
5	707.2197	835226.496
6	138.8745	27774.90
7	738.2381	145432.92
8	7135.0514	1476955.656
9	734.8376	147702.36

	Microsoft SQL Server 2005 XML Showplan
1	<ShowPlanXML xmlns="http://schemas.microsoft.co...

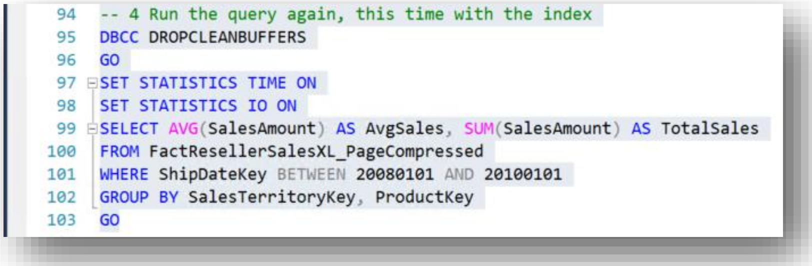
3. Run the query. After the results are returned, scroll to the bottom of the Messages tab.



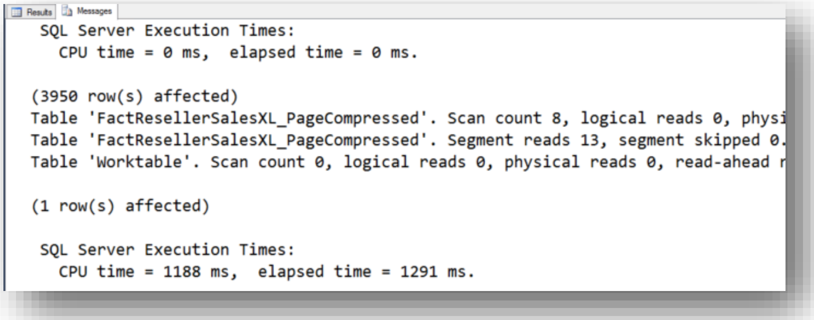
```
OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX) --this ignores the index
GO
```

```
(3950 row(s) affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0,
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, l
Table 'FactResellerSalesXL_PageCompressed'. Scan count 5, logical reads 87752, physical reads 1, read-ahead r

(1 row(s) affected)

 SQL Server Execution Times:
   CPU time = 4703 ms,  elapsed time = 3991 ms.
```

4. Now run again without the OPTION line. Look at the Messages tab to see the performance improvement, especially in the CPU time.

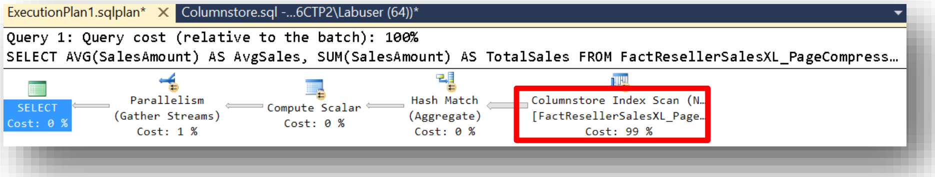


```
-- 4 Run the query again, this time with the index
DBCC DROPCLEANBUFFERS
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
SELECT AVG(SalesAmount) AS AvgSales, SUM(SalesAmount) AS TotalSales
FROM FactResellerSalesXL_PageCompressed
WHERE ShipDateKey BETWEEN 20080101 AND 20100101
GROUP BY SalesTerritoryKey, ProductKey
GO
```



```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(3950 row(s) affected)
Table 'FactResellerSalesXL_PageCompressed'. Scan count 8, logical reads 0, physi
Table 'FactResellerSalesXL_PageCompressed'. Segment reads 13, segment skipped 0.
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead r

(1 row(s) affected)

 SQL Server Execution Times:
   CPU time = 1188 ms,  elapsed time = 1291 ms.
```

5. Now look at the query execution plan. We can see that the Columnstore index is utilized and how it impacts query execution.



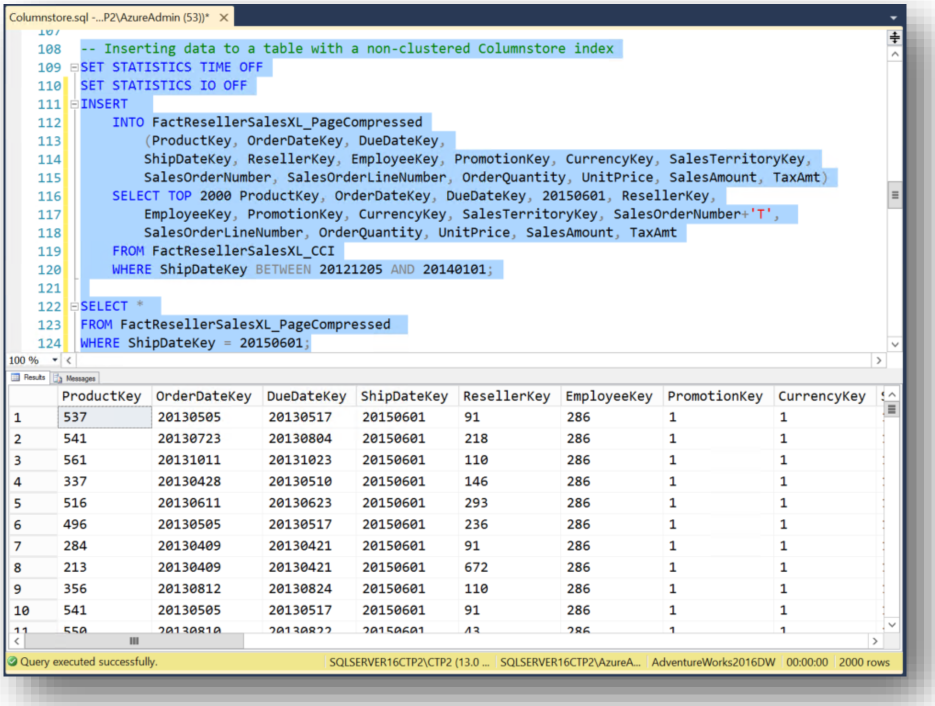
ExecutionPlan1.sqlplan* Columnstore.sql -...6CTP2\Labuser (64))*

```
Query 1: Query cost (relative to the batch): 100%
SELECT AVG(SalesAmount) AS AvgSales, SUM(SalesAmount) AS TotalSales FROM FactResellerSalesXL_PageCompress…
```

Inserting data to a table with a non-clustered Columnstore index

Note that in SQL Server 2014, it was not possible to add data to a table that had a nonclustered Columnstore index.

1. Insert 2,000 new rows by selecting the rows shown for the query below and executing the query. Verify that there is no error.



Columnstore.sql -...P2\AzureAdmin (53))*

```
-- Inserting data to a table with a non-clustered Columnstore index
SET STATISTICS TIME OFF
SET STATISTICS IO OFF
INSERT
    INTO FactResellerSalesXL_PageCompressed
        (ProductKey, OrderDateKey, DueDateKey,
        ShipDateKey, ResellerKey, EmployeeKey, PromotionKey, CurrencyKey, SalesTerritoryKey,
        SalesOrderNumber, SalesOrderLineNumber, OrderQuantity, UnitPrice, SalesAmount, TaxAmt)
    SELECT TOP 2000 ProductKey, OrderDateKey, DueDateKey, 20150601, ResellerKey,
        EmployeeKey, PromotionKey, CurrencyKey, SalesTerritoryKey, SalesOrderNumber+'T',
        SalesOrderLineNumber, OrderQuantity, UnitPrice, SalesAmount, TaxAmt
    FROM FactResellerSalesXL_CCI
    WHERE ShipDateKey BETWEEN 20121205 AND 20140101;

SELECT *
FROM FactResellerSalesXL_PageCompressed
WHERE ShipDateKey = 20150601;
```

	ProductKey	OrderDateKey	DueDateKey	ShipDateKey	ResellerKey	EmployeeKey	PromotionKey	CurrencyKey
1	537	20130505	20130517	20150601	91	286	1	1
2	541	20130723	20130804	20150601	218	286	1	1
3	561	20131011	20131023	20150601	110	286	1	1
4	337	20130428	20130510	20150601	146	286	1	1
5	516	20130611	20130623	20150601	293	286	1	1
6	496	20130505	20130517	20150601	236	286	1	1
7	284	20130409	20130421	20150601	91	286	1	1
8	213	20130409	20130421	20150601	672	286	1	1
9	356	20130812	20130824	20150601	110	286	1	1
10	541	20130505	20130517	20150601	91	286	1	1
11	550	20130810	20130822	20150601	43	286	1	1

Query executed successfully. SQLSERVER16CTP2\CTP2 (13.0 ... SQLSERVER16CTP2\AzureA... AdventureWorks2016DW 00:00:00 2000 rows

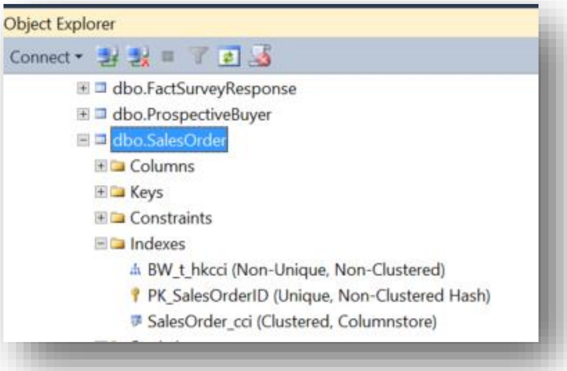
Memory-optimized tables and operational analytics

Investigate memory optimized tables

Memory-optimized tables ("Hekaton") was introduced in SQL Server 2014. We will now look at how these tables interact with Columnstore indexes and how these can be used in operational analytics.

1. Expand the **Indexes** folder in SSMS Object Explorer for the **dbo.SalesOrder** table. Notice that there are 3 indexes, including:

   - Primary key nonclustered hash,
   - Clustered Columnstore, and
   - Nonclustered indexing



Object Explorer

```
⊞ dbo.FactSurveyResponse
⊞ dbo.ProspectiveBuyer
⊟ dbo.SalesOrder
   ⊞ Columns
   ⊞ Keys
   ⊞ Constraints
   ⊟ Indexes
      BW_t_hkcci (Non-Unique, Non-Clustered)
      PK_SalesOrderID (Unique, Non-Clustered Hash)
      SalesOrder_cci (Clustered, Columnstore)
```

2. Also right click on the table to view its **Properties**. Notice that on the **General** page, the table's **Memory optimized** property is set to **True**.

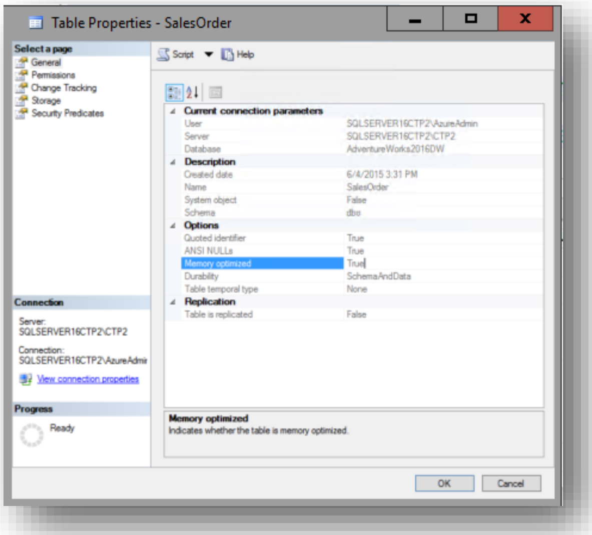


Table Properties - SalesOrder

Current connection parameters	
User	SQLSERVER16CTP2\AzureAdmin
Server	SQLSERVER16CTP2\CTP2
Database	AdventureWorks2016DW
Description	
Created date	6/4/2015 3:31 PM
Name	SalesOrder
System object	False
Schema	dbo
Options	
Quoted identifier	True
ANSI NULLs	True
Memory optimized	True
Durability	SchemaAndData
Table temporal type	None
Replication	
Table is replicated	False

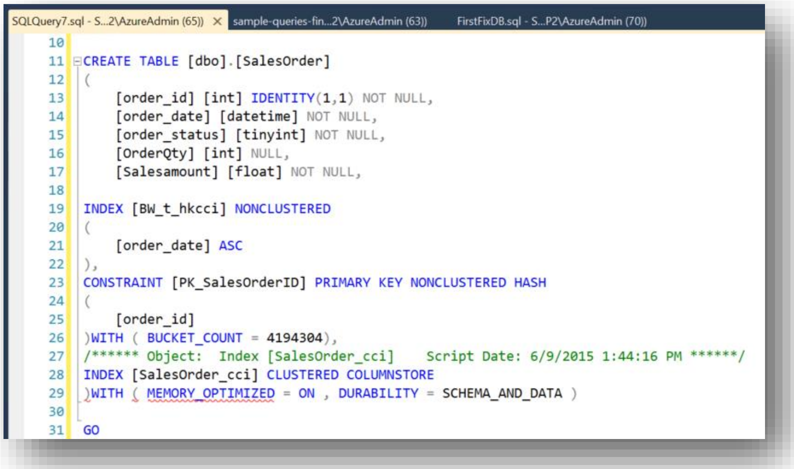
Memory optimized
Indicates whether the table is memory optimized.

Connection

Server:
SQLSERVER16CTP2\CTP2

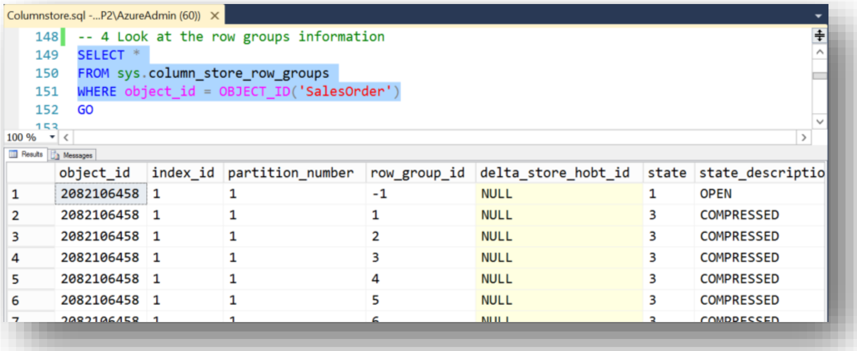
Connection:
SQLSERVER16CTP2\AzureAdmi

3. If you want to see the table creation script, script the table as “CREATE To.”


SQLQuery7.sql - S...2\AzureAdmin (65))
sample-queries-fin...2\AzureAdmin (63))
FirstFixDB.sql - S...P2\AzureAdmin (70))

```
CREATE TABLE [dbo].[SalesOrder]
(
    [order_id] [int] IDENTITY(1,1) NOT NULL,
    [order_date] [datetime] NOT NULL,
    [order_status] [tinyint] NOT NULL,
    [OrderQty] [int] NULL,
    [Salesamount] [float] NOT NULL,

INDEX [BW_t_hkcci] NONCLUSTERED
(
    [order_date] ASC
),
CONSTRAINT [PK_SalesOrderID] PRIMARY KEY NONCLUSTERED HASH
(
    [order_id]
)WITH ( BUCKET_COUNT = 4194304),
/****** Object:  Index [SalesOrder_cci]    Script Date: 6/9/2015 1:44:16 PM ******/
INDEX [SalesOrder_cci] CLUSTERED COLUMNSTORE
)WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA )

GO
```

4. Now, look at the row groups information. This shows how the data is compressed.

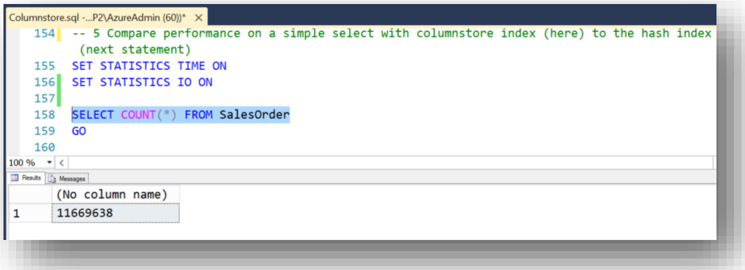


```
-- 4 Look at the row groups information
SELECT *
FROM sys.column_store_row_groups
WHERE object_id = OBJECT_ID('SalesOrder')
GO
```

	object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_descriptio
1	2082106458	1	1	-1	NULL	1	OPEN
2	2082106458	1	1	1	NULL	3	COMPRESSED
3	2082106458	1	1	2	NULL	3	COMPRESSED
4	2082106458	1	1	3	NULL	3	COMPRESSED
5	2082106458	1	1	4	NULL	3	COMPRESSED
6	2082106458	1	1	5	NULL	3	COMPRESSED
7	2082106458	1	1	6	NULL	3	COMPRESSED

Compare performance on memory-optimized tables - count

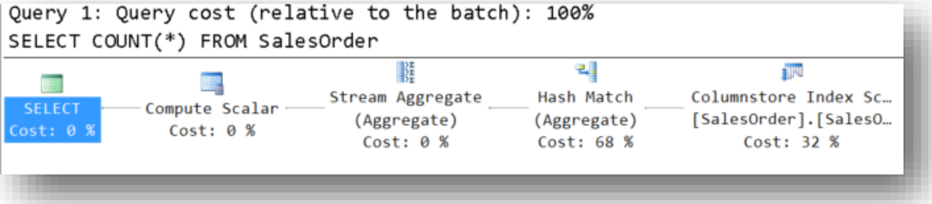
1. Now, we’ll compare performance using the various indexes; first with a simple count query.



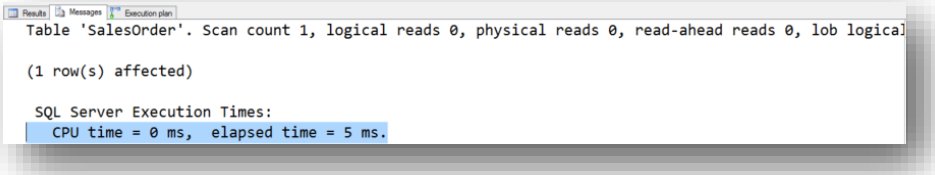
```
-- 5 Compare performance on a simple select with columnstore index (here) to the hash index
 (next statement)
SET STATISTICS TIME ON
SET STATISTICS IO ON

SELECT COUNT(*) FROM SalesOrder
GO

```

	(No column name)
1	11669638

2. Look at the query execution plan. Notice that this query used the clustered Columnstore index. Also look at the time stats on the Messages tab.

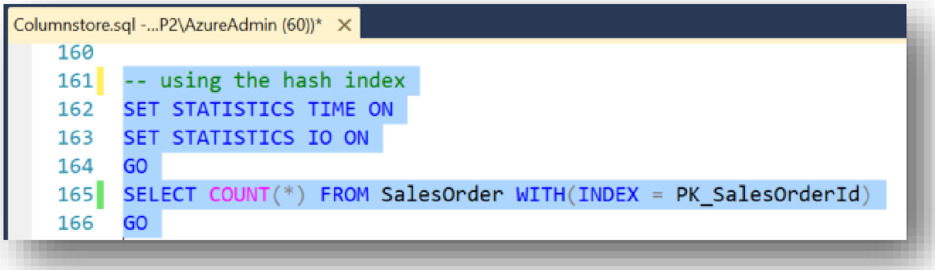


Query 1: Query cost (relative to the batch): 100%
SELECT COUNT(*) FROM SalesOrder



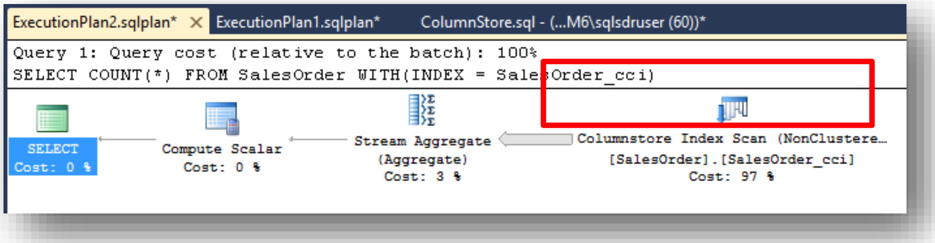
```
 Table 'SalesOrder'. Scan count 1, logical reads 0, physical reads 0, read-ahead reads 0, lob logical

 (1 row(s) affected)

  SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 5 ms.
```

3. Now we can run the query again, but force it to use the nonclustered primary key index.

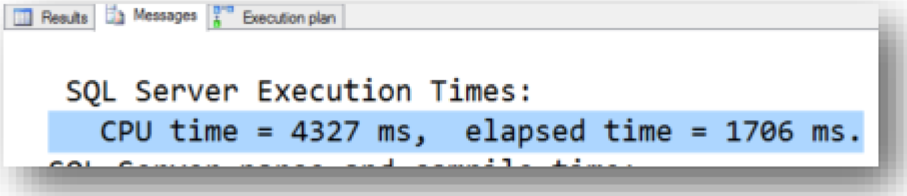


```

-- using the hash index
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT COUNT(*) FROM SalesOrder WITH(INDEX = PK_SalesOrderId)
GO
```

4. Again, look at the query execution plan and the time and you can see that performance is worse.



```
Query 1: Query cost (relative to the batch): 100%
SELECT COUNT(*) FROM SalesOrder WITH(INDEX = SalesOrder_cci)
```

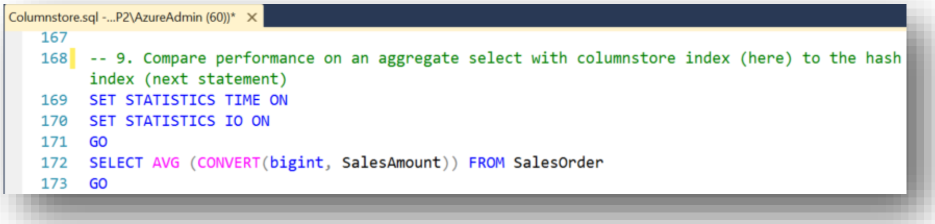


```
 SQL Server Execution Times:
   CPU time = 4327 ms,  elapsed time = 1706 ms.
SQL Server parse and compile time:
```

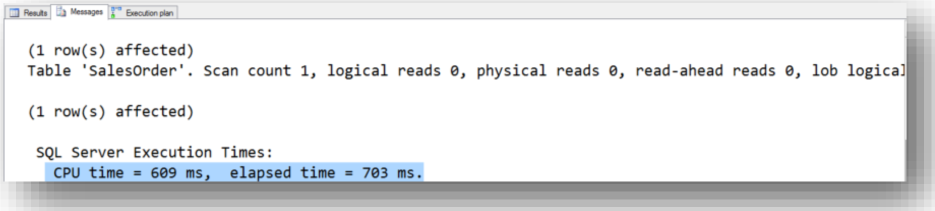
Compare performance on memory-optimized tables - analytics

The performance difference can also be significant when using analytic queries with clustered Columnstore indexes.

1. First run the query normally, so it will use its clustered columnstore index and note its time.

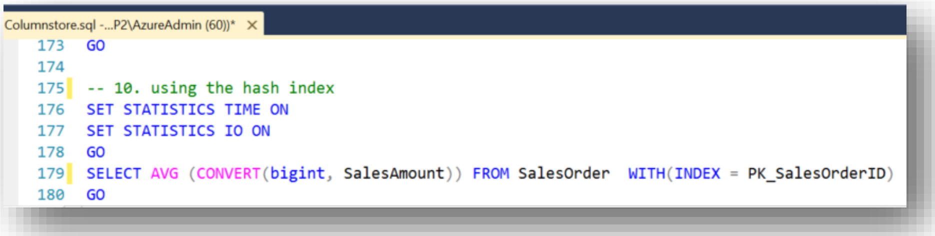


```
-- 9. Compare performance on an aggregate select with columnstore index (here) to the hash
index (next statement)
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT AVG (CONVERT(bigint, SalesAmount)) FROM SalesOrder
GO
```

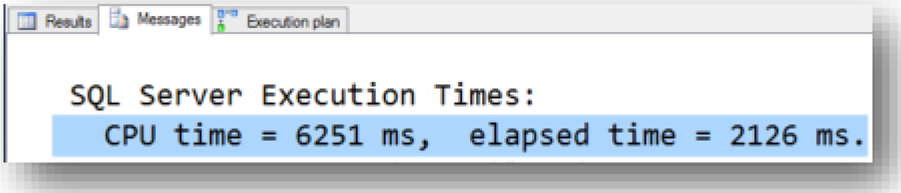


```
(1 row(s) affected)
Table 'SalesOrder'. Scan count 1, logical reads 0, physical reads 0, read-ahead reads 0, lob logical

(1 row(s) affected)

 SQL Server Execution Times:
   CPU time = 609 ms,  elapsed time = 703 ms.
```

2. Now run the same query, but use the clustered Columnstore index and note the time.

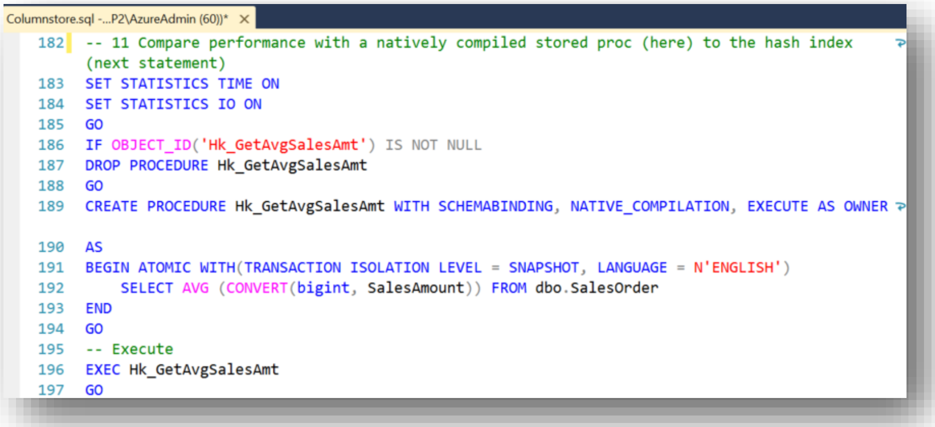


```
GO

-- 10. using the hash index
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT AVG (CONVERT(bigint, SalesAmount)) FROM SalesOrder  WITH(INDEX = PK_SalesOrderID)
GO
```

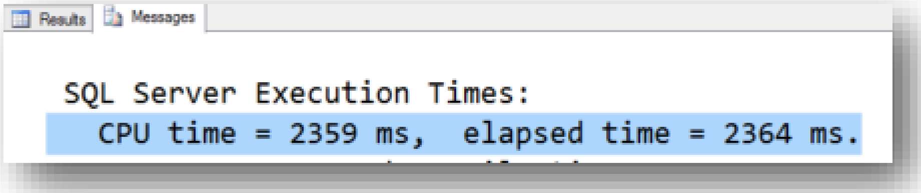


```
SQL Server Execution Times:
   CPU time = 6251 ms,  elapsed time = 2126 ms.
```

3. We can also compare the use of a clustered Columnstore index to the performance of a natively compiled stored procedure. First the stored procedure.

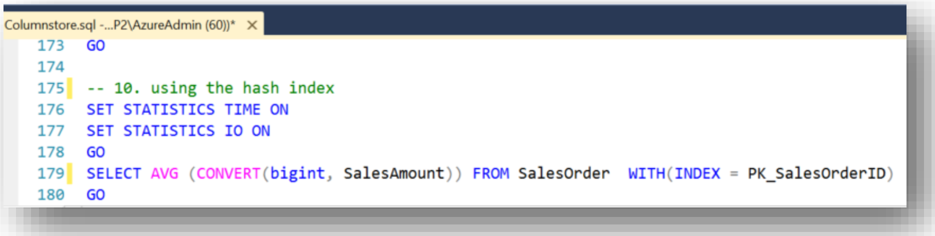


```
-- 11 Compare performance with a natively compiled stored proc (here) to the hash index
(next statement)
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
IF OBJECT_ID('Hk_GetAvgSalesAmt') IS NOT NULL
DROP PROCEDURE Hk_GetAvgSalesAmt
GO
CREATE PROCEDURE Hk_GetAvgSalesAmt WITH SCHEMABINDING, NATIVE_COMPILATION, EXECUTE AS OWNER

AS
BEGIN ATOMIC WITH(TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'ENGLISH')
    SELECT AVG (CONVERT(bigint, SalesAmount)) FROM dbo.SalesOrder
END
GO
-- Execute
EXEC Hk_GetAvgSalesAmt
GO
```

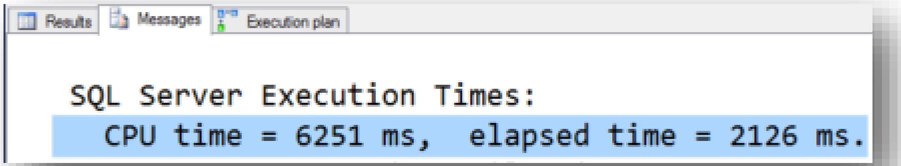


```
SQL Server Execution Times:
   CPU time = 2359 ms,  elapsed time = 2364 ms.
```

4. Compare this to the earlier hash index query, above.



```
GO

-- 10. using the hash index
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT AVG (CONVERT(bigint, SalesAmount)) FROM SalesOrder  WITH(INDEX = PK_SalesOrderID)
GO
```



```
SQL Server Execution Times:
   CPU time = 6251 ms,  elapsed time = 2126 ms.
```

Wrap up

You should now be familiar with Columnstore indexes to see how they have a positive performance impact as described below:

1. A table with clustered Columnstore can now enforce referential integrity through foreign key constraints;
2. A table with clustered Columnstore tables can now have one or more non-clustered btree indexes for efficient execution of queries with equality and short-range predicates;
3. A table with clustered or non-clustered columnstore indexes can run analytics queries efficiently;
4. A table with non-clustered columnstore (NCCI) now allows DML operations. In earlier releases of SQL Server, a table with NCCI was only available for querying;
5. A Columnstore index on operational schema can provide efficient execution of analytics with minimal impact on OLTP workload.

Terms of use

By using this Hands-on Lab, you agree to the following terms:

The technology/functionality described in this Hands-on Lab is provided by Microsoft Corporation in a “sandbox” testing environment for purposes of obtaining your feedback and to provide you with a learning experience. You may only use the Hands-on Lab to evaluate such technology features and functionality and provide feedback to Microsoft. You may not use it for any other purpose. You may not modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this Hands-on Lab or any portion thereof.

COPYING OR REPRODUCTION OF THE HANDS-ON LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS HANDS-ONLAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE. THE TECHNOLOGY/CONCEPTS REPRESENTED IN THIS HANDS-ON LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK. WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS. YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCITONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.

FEEDBACK. If you give feedback about the technology features, functionality and/or concepts described in this Hands-on Lab to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

MICROSOFT CORPORATION HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE HANDS-ON LAB , INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. MICROSOFT DOES NOT MAKE ANY ASSURANCES OR REPRESENTATIONS WITH REGARD TO THE ACCURACY OF THE RESULTS, OUTPUT

THAT DERIVES FROM USE OF THE VIRTUAL LAB, OR SUITABILITY OF THE INFORMATION CONTAINED IN THE VIRTUAL LAB FOR ANY PURPOSE.

DISCLAIMER

This lab contains only a portion of new features and enhancements in Microsoft SQL Server 2016 CTP2. Some of the features might change in future releases of the product. In this exercise, you will learn about some, but not all, new features.