SQL ServerCentral.com
Proactively monitor your growing SQL Server environment.
Try free SQL Monitor
a community service from redgate
Try SQL Clone

Keep up to date - daily newsletter:
your@email.com
Sign up

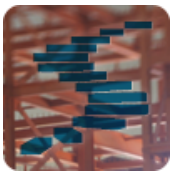
Write For Us
Find out more

SQL Prompt
Code analysis as you type
SQLQuery1.sql
Try it now
redgate

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: G+ 0
Rate this | Join the discussion | Add to briefcase

Stairway to Columnstore Indexes Level 5: Adding New Data To Columnstore Indexes



By Hugo Kornelis, 2015/11/04

The Series

This article is part of the Stairway Series: Stairway to Columnstore Indexes

SQL Server 2012 and later offer a very different type of index from the traditional b-tree, the in-memory columnstore index. These indexes use a column-based storage model, as well as a new 'batch mode' of query execution and can offer huge performance increases for certain workloads. But how are they built, how do they work, and why do they manage to have such a dramatic impact on performance? In this stairway, Hugo Kornelis explains all, with his usual mix of concise description and detailed demonstration.

The procedure for converting data into columnstore format as described in level 3 of this series is optimal for static data. SQL Server expends considerable resources spent to try to find the best possible compression for the data. But in most tables, data is hardly ever static. We are constantly inserting new rows, and updating or deleting existing rows. If you think about what this means for a columnstore index, you will realize that this comes with some unique challenges.

In a data warehouse, adding new data is the most common type of modification. Every day, new sales are added in the database, along with the corresponding inventory movement records, invoices, payment data, and general ledger entries. All this new data is added to the warehouse, either continuously or during a periodic load process. In this level we will look at the mechanisms in SQL Server 2014 that support adding new rows to columnstore indexes, and what consequences this should have for your load processes.

When thinking about adding data to an existing columnstore, a naïve approach could be to simply add the new rows at the end of the last rowgroup. But this sounds easier than it is. For any column that is dictionary encoded, the value to be inserted has to be looked up in the global and local dictionary, and possibly added to the latter. Depending on the compression algorithms used, it might also be necessary to first process all prior values in a segment before the new value can be added. These actions would make the insert very slow. Also, once the last rowgroup

Related Articles

BLOG COLUMNSTORE INDEX... DEMO
2015/07/11
COLUMNSTORE INDEX How data is stored in traditional way For physical storage of a table, its rows...

ARTICLE Stairway to Columnstore Indexes Level 6: Updating and Deleting Data in a Columnstore Index
2016/01/04

ARTICLE

BLOG

ARTICLE Learning Columnstore Indexes
2015/06/25
References and links to learn more about columnstore indexes.

Fresh articles daily:
Get the SQL Server Central newsletter and get a new SQL Server article each day. Get Database Weekly for a roundup of all the biggest SQL news from around the web.
your@email.com
Sign up
No thanks

Tags
columnstore index
stairway series

Write For Us
Find out more

Copyright © 2002-2018 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

reaches its maximum size, a new rowgroup would have to be opened, and the compression methods to be used for its segments would have to be decided based on possibly just a single row to be inserted -- this will never result in optimal choices!

This challenge, and other challenges associated with updating and deleting data (to be covered in the next level) are probably the reason why Microsoft released the nonclustered columnstore index in SQL Server 2012 as a read-only index -- after creating the index, no inserts, updates, or deletes are allowed to the table, so that none of the problems above will ever occur. But because a business expects their databases to reflect the current situation, including any changes that just happened, this read-only restriction hampered adoption of the columnstore index feature, and Microsoft luckily found ways to deal with the issues and make an updatable version of the columnstore index in SQL Server 2014. Note, though, that the architecture changes that allow updating the data have been applied only to the new clustered columnstore index; a nonclustered columnstore index still makes the table read-only even on SQL Server 2014. (Based on CTP releases so far, it appears that this limitation will finally be lifted in SQL Server 2015).

As shown above, just adding new data to existing columnstore rowgroups would be both inefficient and impractical. So when Microsoft created the columnstore index without a read-only restriction, they had to find a way around that problem. Their solution is called the deltastore, an extra storage area that you can think of as a temporary holding area where new data is stored until sufficient rows have been collected to justify the effort of converting them to the columnstore format.

The data in the deltastore is not stored in column oriented fashion, but in a B-tree structure, the same format used for traditional row oriented indexes. The deltastore does use page compression, but this is a far cry from the compression ratios achieved in the normal columnstore rowgroups. Additionally, the deltastore does not support segment elimination either. In short, none of the typical columnstore performance benefits apply to data in the deltastore. However, in normal usage scenarios the amount of rows in a deltastore will not be big enough that this relative slowness causes noticeable problems.

If you have SQL Server 2014 available and you followed the demos in the previous steps, you will now have a table *FactOnlineSales* that has a nonclustered columnstore index and is hence read-only, and also a table *FactOnlineSales2* with a clustered columnstore index. This table can be modified, so we will now see what happens if we add some rows to it. The code in listing 5-1 adds 500 rows, then queries the *sys.column_store_row_groups* view using the same query we already saw in listing 4-8 of the previous level, but with two extra columns added. The results after running this code on my system are shown in figure 5-1.

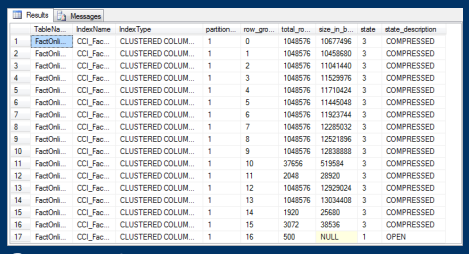
```
-- SQL Server 2014 only!!

USE ContosoRetailDW;
GO
```

```
-- Add 500 rows
INSERT      dbo.FactOnlineSales2
           (DateKey, StoreKey, ProductKey, Pr
            CurrencyKey, CustomerKey, SalesOr
            SalesOrderLineNumber, SalesQuanti
            ReturnQuantity, ReturnAmount, Dis
            DiscountAmount, TotalCost, UnitCo
            ETLLoadID, LoadDate, UpdateDate)
SELECT TOP(500)
            DateKey, StoreKey, ProductKey, Pr
            CurrencyKey, CustomerKey, SalesOr
            SalesOrderLineNumber, SalesQuanti
            ReturnQuantity, ReturnAmount, Dis
            DiscountAmount, TotalCost, UnitCo
            ETLLoadID, LoadDate, UpdateDate
FROM        dbo.FactOnlineSales;

-- Check the rowgroups metadata
SELECT      OBJECT_NAME(rg.object_id)   AS Ta
            i.name                      AS In
            i.type_desc                 AS In
            rg.partition_number,
            rg.row_group_id,
            rg.total_rows,
            rg.size_in_bytes,
            rg.[state],
            rg.state_description
FROM        sys.column_store_row_groups AS rg
INNER JOIN  sys.indexes                 AS i
      ON    i.object_id                  = rg
      AND   i.index_id                   = rg
WHERE       i.name = N'CCI_FactOnlineSales2'
ORDER BY    TableName, IndexName
```

Listing 5-1: Observing the effect of adding rows



Results | Messages

	TableNa...	IndexName	IndexType	partition...	row_gro...	total_ro...	size_in_b...	state	state_description
1	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	0	1048576	10677496	3	COMPRESSED
2	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	1	1048576	10458680	3	COMPRESSED
3	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	2	1048576	11041440	3	COMPRESSED
4	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	3	1048576	11529976	3	COMPRESSED
5	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	4	1048576	11710424	3	COMPRESSED
6	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	5	1048576	11445048	3	COMPRESSED
7	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	6	1048576	11923744	3	COMPRESSED
8	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	7	1048576	12285032	3	COMPRESSED
9	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	8	1048576	12521896	3	COMPRESSED
10	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	9	1048576	12838888	3	COMPRESSED
11	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	10	37656	519584	3	COMPRESSED
12	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	11	2048	28920	3	COMPRESSED
13	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	12	1048576	12929024	3	COMPRESSED
14	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	13	1048576	13034408	3	COMPRESSED
15	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	14	1920	25680	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	15	3072	38536	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	16	500	NULL	1	OPEN

Zoom in | Open in new window

Figure 5-1: Rowgroup metadata after adding rows

If you compare the results to what we saw when we first ran this query (back in level 4), you will notice that as a result of inserting new rows, one extra row has been added to the *sys.column_store_row_groups* DMV. If you look at the two additional columns, state and state_description, you will see that all original rowgroups have state 3 (`COMPRESSED`). The new rowgroup is in state 1 (`OPEN`). This indicates that this is not a "normal" columnstore rowgroup as we saw in the previous levels, but a deltastore rowgroup. This deltastore was automatically created when I added the new rows.

It is important to note that, even though a deltastore has now been added to the columnstore index, all queries will still work as before. The execution plans of your query will not change, and

you will still get correct results. This is because the Clustered Columnstore Index Scan operator that reads data from the columnstore index understands the deltastore format as well, and seamlessly integrates this data with the columnstore data into a single data stream for the other operators to process.

To examine the effect of inserting different numbers of rows in a single `INSERT` statement, let's now run the code in listing 5-2. I first insert a large number of rows (150,000) at once, followed by a smaller number of rows (15,000). The output of the metadata query at the end of the listing (shown in figure 5-2) reveals some interesting results.

```
-- SQL Server 2014 only!!

USE ContosoRetailDW;
GO

-- Add another 150,000 rows
INSERT     dbo.FactOnlineSales2
          (DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate)
SELECT TOP(150000)
           DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate
FROM       dbo.FactOnlineSales;

-- And yet another 15,000 rows
INSERT     dbo.FactOnlineSales2
          (DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate)
SELECT TOP(15000)
           DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate
FROM       dbo.FactOnlineSales;
```

Listing 5-2: Adding rows in bigger chunks

Results | Messages

	TableNa...	IndexName	IndexType	partition...	row_gro...	total_ro...	size_in_b...	state	state_description
1	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	0	1048576	10677496	3	COMPRESSED
2	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	1	1048576	10458680	3	COMPRESSED
3	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	2	1048576	11041440	3	COMPRESSED
4	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	3	1048576	11529976	3	COMPRESSED
5	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	4	1048576	11710424	3	COMPRESSED
6	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	5	1048576	11445048	3	COMPRESSED
7	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	6	1048576	11923744	3	COMPRESSED
8	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	7	1048576	12285032	3	COMPRESSED
9	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	8	1048576	12521896	3	COMPRESSED
10	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	9	1048576	12838888	3	COMPRESSED
11	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	10	37656	519584	3	COMPRESSED
12	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	11	2048	28920	3	COMPRESSED
13	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	12	1048576	12929024	3	COMPRESSED
14	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	13	1048576	13034408	3	COMPRESSED
15	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	14	1920	25680	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	15	3072	38536	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	16	15500	NULL	1	OPEN
18	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	17	150000	1643536	3	COMPRESSED

Zoom in | Open in new window

Figure 5-2: Rowgroup metadata after adding more rows

As you can see, the 150,000 rows that were added in the first query of listing 5-2 were inserted into a new rowgroup, and this rowgroup has state 3 (COMPRESSED), indicating that this rowgroup is not a deltastore but a regular columnstore rowgroup. The 15,500 rows of the second insert were then added to the open deltastore rowgroup that remained after running the code in listing 5-1. The state of this rowgroup has not changed, but its *total_rows* value has increased to 15,500.

What we saw from running listing 5-2 are the two basic modes of inserting rows into a clustered columnstore index. Any insert of 102,399 or fewer rows is considered a "trickle insert". These rows are added to an open deltastore if one is available (and not locked), or else a new deltastore rowgroup is created for them. However, if 102,400 rows or more are inserted at once through any means that supports bulk loading (see below), then it's considered a "bulk load". This means that a new rowgroup is formed out of all inserted rows, using the columnstore creation process outlined in level 3 (except that no new global dictionary is built). The upper limit of 1,048,576 rows per rowgroup still applies. If the number of rows inserted exceeds that limit, SQL Server will first create a new rowgroup for the first 1,048,576 rows, then compare the remaining number of rows with the 102,400 threshold to determine whether they should be bulk loaded or trickle inserted.

> **Methods for bulk loading**
>
> An insert of 102,400 rows or more will only be bulk-loaded if the "Bulk Insert API" is used. When using standard tools, this means that you would have to use tools like bcp.exe, the 'fast-load' SSIS OleDB destination, or the T-SQL `BULK INSERT` statement. The batch size used in these tools needs to exceed the 102,400 rows threshold in order to enable bulk loading. For bcp.exe, the batch size is set with the **-b** option. For SSIS, you should set the *DefaultBufferMaxRows* and *DefaultBufferSize* properties of the data flow component to the maximum values, but you will also need to set the *Maximum insert commit size* of the OLE DB Destination component to 0, as described here. And for `BULK INSERT`, you simply use the `BATCHSIZE` parameter.
>
> The standard T-SQL `INSERT ... SELECT` statement will also use the bulk load method if the total number of rows inserted is large enough, as shown by the results of listing 5-2. However, the T-SQL `MERGE` statement will always use the trickle insert method, regardless of the number of rows to be inserted.

In most cases, columnstore indexes perform best with fewer and bigger rowgroups. This means that controlling batch size is crucial when designing your data loads – both the batch size as set in the options, as well as the actual number of rows. For

instance, if you have between 100,000 and 120,000 new rows per day, a single daily load would result in either lots of rows in the delta store, or lots of small rowgroups. Switch to a weekly import, and now you add one rowgroup per week with about 750,000 rows – close enough to the maximum size that you can expect good performance. But beware that any queries you run now report from older data, unless you add an extra table to temporarily hold this week's data and modify the queries to add that extra table.

In typical usage scenarios, a columnstore index will have hundreds of millions of rows, and the amount of rows added through trickle insert will be just a small fraction of that. But given enough time, this would still result in deltastores that, eventually, become big enough to have a noticeable impact on performance. This scenario is prevented by a process called the *tuple mover*. To see this process in action, run the code in listing 5-3.

```
-- SQL Server 2014 only!!

USE ContosoRetailDW;
GO

-- Add another 1,100,000 rows, in 11 batches
INSERT     dbo.FactOnlineSales2
          (DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate)
SELECT TOP(100000)
           DateKey, StoreKey, ProductKey, Pr
           CurrencyKey, CustomerKey, SalesOr
           SalesOrderLineNumber, SalesQuanti
           ReturnQuantity, ReturnAmount, Dis
           DiscountAmount, TotalCost, UnitCo
           ETLLoadID, LoadDate, UpdateDate
FROM       dbo.FactOnlineSales;
go 11

-- Check the rowgroups metadata
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
           i.name                      AS In
           i.type_desc                 AS In
           rg.partition_number,
           rg.row_group_id,
           rg.total_rows,
           rg.size_in_bytes,
           rg.[state],
           rg.state_description
FROM       sys.column_store_row_groups AS rg
INNER JOIN sys.indexes                 AS i
      ON   i.object_id                  = rg
      AND  i.index_id                   = rg
WHERE      i.name = N'CCI_FactOnlineSales2'
```

Listing 5-3: Overflowing a deltastore rowgroup

Results | Messages

	TableNa...	IndexName	IndexType	partition...	row_gro...	total_ro...	size_in_b...	state	state_description
15	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	14	1920	25680	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	15	3072	38536	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	16	1048576	NULL	2	CLOSED
18	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	17	150000	1643536	3	COMPRESSED
19	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	18	66924	NULL	1	OPEN

	TableNa...	IndexName	IndexType	partition...	row_gro...	total_ro...	size_in_b...	state	state_description
15	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	14	1920	25680	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	15	3072	38536	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	17	150000	1643536	3	COMPRESSED
18	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	18	66924	NULL	1	OPEN
19	FactOnli...	CCI_Fac...	CLUSTERED COLUM...	1	19	1048576	8166568	3	COMPRESSED

Zoom in | Open in new window

Figure 5-3: The tuple mover in action

The code in listing 5-3 first adds 1.1 million rows, using 11 batches of 100,000 rows each -- just below the threshold for bulk loading, so these rows are all trickle inserted into the deltastore. (Note that this is NOT an example of a best practice for loading data!). Since there already were 15,500 rows in the deltastore, you might expect to see 1,115,500 rows in it now. But the results of the DMV query show differently: once the deltastore contains 1,048,575 rows, it is considered full. The state changes to 2 (`CLOSED`), and a new deltastore rowgroup is created for the remaining 66,924 rows. Note that the data in a closed rowgroup is still in the same row-oriented format as when it was open; the only difference is that the rowgroup is now marked so it can no longer accept new rows, and it is ready to be converted to the compressed column-oriented format we expect in a columnstore index.

After waiting five minutes, the DMV query is repeated. Now the closed rowgroup has gone, and is replaced by a new rowgroup with state 3 (`COMPRESSED`). This is caused by the tuple mover -- a background process that wakes up every five minutes, checks to see if there are any rowgroups with state 2, and if there are, converts the data in these rowgroups to columnstore format.

The tuple mover is a slow worker. If it finds multiple closed rowgroups, it will convert just one of them and then sleep for a short time (about 10-15 seconds) before moving on to the next. An import process can easily produce closed deltastores at a much higher rate; in such cases, the tuple mover will fall behind, and it may take a long time before it catches up. During that period, query performance on your columnstore will suffer: reading a single deltastore when processing a columnstore index will not significantly affect performance, but having to read dozens of them will. This is not a bug. The tuple mover is deliberately designed to be slow, in order to prevent this background process from overburdening your system. A properly designed data load process ensures that the majority of data is bulk loaded, and only small amounts of rows are trickle inserted in the deltastore. With a well-designed import process, the tuple mover should never fall behind.

One of the major blockers for adoption of the columnstore index in SQL Server 2012 was the read-only limitation. Removing that limitation in SQL Server 2014 is a major step forward. It is now possible to achieve the performance benefits of columnstore indexes without having to jump through hoops whenever data needs to change.

In this level we focused on adding data. When adding large amounts of data, the "bulk load" method will be used. This method immediately converts the new data to columnstore format, so that later queries can immediately benefit from the better performance. For smaller inserts, the new data is