SQL ServerCentral.com
redgate | Hub Insights Learning Forums Ideas Discover the new Redgate Hub
a community service from redgate
Try SQL Clone

Daily SQL Articles by email:
your@email.com
Sign up

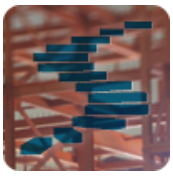
Write For Us
Find out more

Proactively monitor your growing SQL Server environment.
Try free
SQL Monitor

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: 0
Rate this | Join the discussion | Add to briefcase

Stairway to ColumnStore Indexes Level 8: Optimizing Clustered Columnstore Indexes



By Hugo Kornelis, 2016/03/16

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.

In Level 7, we looked at optimizing rowgroup elimination for a nonclustered columnstore index. For a clustered columnstore index, the same technique can be used but the steps and syntax change a bit. This will be covered later – but first, let's take a look at another significant difference between nonclustered and clustered columnstore indexes, which is that the former are read-only (in SQL Server 2012 and 2014), whereas the latter allow updates. This makes the index far more flexible and usable, but it has a down side as well: no matter how much effort you put into optimizing the index structure when building the index, it will not remain that way. When you start to use clustered columnstore indexes, you will also have to plan for index maintenance.

Reorganizing the index

For rowstore index maintenance, there are two options: reorganize and rebuild, with reorganize being the more lightweight option. For columnstore indexes, those same two options are available, though they do not have the exact same effect.

Reorganizing a clustered columnstore index can be useful after doing a large data load that did not use bulk load, or after doing a large update – in short, after any operation that causes a lot of trickle inserts. Trickle inserts go into an open deltastore, which will be closed when it reaches 1,048,576 rows. At that point, without further action, it will wait for the background tuple mover process to pick it up and compress it into columnstore format. But

Related Articles

BLOG Rebuilding and Reorganizing Clustered Columnstore Indexes
2014/12/12
Clustered Columnstore Indexes, as well as "regular" indexes, support the Rebuild and Reorganize oper...

ARTICLE Stairway to Columnstore Indexes Level 12: Clustered or Nonclustered?
2017/05/17
The previous levels of this stairway

ARTICLE

ARTICLE

index is offset by the...

ARTICLE Stairway to Columnstore Indexes Level 6: Updating and Deleting Data in a Columnstore Index
2016/01/04
This level looks in detail at what happens when we update or delete data from a clustered columnstor...

Stay up to date:
Daily newsletters with brand new articles, scripts, editorials and a Question of the Day help you keep on top of SQL Server.
your@email.com
Sign up
No thanks

Tags
clustered index
columnstore index
stairway series

Write For Us
Find out more

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

that process is designed to use minimal resources, so it can be slow – so if a lot of closed rowgroups are created in a short time, it can fall behind. Until it catches up, you can experience suboptimal performance from your columnstore index.

When you reorganize the index, you tell SQL Server to not wait for the tuple mover, but immediately compress all closed deltastore rowgroups for the specified index. This will have the same effect as waiting for the tuple mover to kick in, but much faster – and at the expense of using more resources. Reorganizing a columnstore index is an online operation, which means that the index can be used normally during the process.

```
ALTER INDEX IndexName ON schema.TableName
REORGANIZE [ PARTITION = [ PartitionNumber
```

Figure 8-1: Syntax for reorganizing a clustered columnstore index

The most typical use case for reorganizing a columnstore index is when you perform a data load process that you know will generate lots of closed deltastore rowgroups. If you find that the tuple mover sometimes does not catch up before the end of the maintenance window, then you can use the syntax as shown in figure 8-1 to force SQL Server to immediately compress all of them. Beware of the impact that the additional resource usage can have on other tasks running at the same time, though – test to ensure that you are not causing issues for other jobs that need to finish before the end of the maintenance window.

If your columnstore index is partitioned, you can choose to reorganize only specific partitions by specifying the partition number in the `ALTER INDEX` statement, or you can either specify `PARTITION = ALL` or just leave out the partition specification (both of which will result in all partitions being processed).

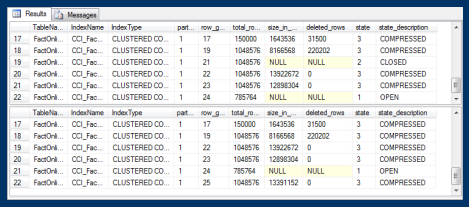
To see this process in action, run the code in listing 8-1, which is almost identical to the code we used in level 5 (listing 5-2) to demonstrate the tuple mover. The only difference is that I replaced the `WAITFOR` command (used to wait for the tuple mover to kick in) with an `ALTER INDEX REORGANIZE` statement. The code first adds 1.1 million rows in 11 batches of 100,000 each, to ensure that at least one closed deltastore rowgroup will be created. It then shows the rowgroup metadata, reorganizes the index, and then queries the metadata again to show the effect of reorganizing the index.

```
USE ContosoRetailDW;
GO

-- Add 1,100,000 rows, in 11 batches of 100,
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.deleted_rows,
            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 8-1: Reorganizing a clustered columnstore index

As you can see from the partial results on my system (figure 8-2), there is a new compressed rowgroup to replace the closed rowgroup generated by the inserts, but all other rowgroups are completely unaffected. (If you run this on your system, then the rowgroup numbers and some other details may be different – as in previous levels, this is because there is no way to force the index build process to produce the exact same result when conditions are different).



Results | Messages

	TableNa...	IndexName	IndexType	part...	row_g...	total_ro...	size_in_...	deleted_rows	state	state_description
17	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	17	150000	1643536	31500	3	COMPRESSED
18	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	19	1048576	8166568	220202	3	COMPRESSED
19	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	21	1048576	NULL	NULL	2	CLOSED
20	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	22	1048576	13922672	0	3	COMPRESSED
21	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	23	1048576	12898304	0	3	COMPRESSED
22	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	24	785764	NULL	NULL	1	OPEN

	TableNa...	IndexName	IndexType	part...	row_g...	total_ro...	size_in_...	deleted_rows	state	state_description
17	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	17	150000	1643536	31500	3	COMPRESSED
18	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	19	1048576	8166568	220202	3	COMPRESSED
19	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	22	1048576	13922672	0	3	COMPRESSED
20	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	23	1048576	12898304	0	3	COMPRESSED
21	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	24	785764	NULL	NULL	1	OPEN
22	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	25	1048576	13391152	0	3	COMPRESSED

Zoom in | Open in new window

Figure 8-2: Effect of reorganizing a clustered columnstore index

It is important to realize that this code to reorganize the index is does not need to be run. If I had just waited a few minutes, the tuple mover would have kicked in and done the same work in the background. You only have to reorganize a clustered columnstore index if there is a reason why you need the closed rowgroups to be compressed immediately.

As we saw in the level 5 and level 6, modifying data causes a columnstore index to deteriorate in many ways: Bulk loading results in rowgroups that can be much smaller than the maximum size, deleted rows (which includes old versions of updated rows) are marked as deleted but never actually deleted from the rowgroups, and trickle inserts usually leave at least one rowgroup open. Reorganizing the index does not help here, and in fact can result in even more rowgroups at sub-optimal size. The only way

to remove all of this and clean up the entire index is to rebuild it. Which, unfortunately, is an offline operation – meaning that SQL Server will take an exclusive lock on the index, making the index and hence the table inaccessible for the duration of the process.

To see the rebuild of a clustered columnstore index in action, run the code in listing 8-2 to rebuild the clustered columnstore index using only the default options. Before and after the rebuild, it queries the rowgroup metadata; we have not seen this particular query before but it is the same basic logic as the metadata query used in listing 8-1 as well as several previous levels. The difference is that it now presents aggregated numbers instead of showing the individual rowgroups.

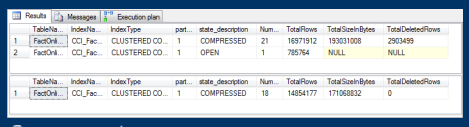
```
USE ContosoRetailDW;
GO

-- Check the rowgroups metadata (aggregated
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
           i.name                      AS In
           i.type_desc                 AS In
           rg.partition_number,
           rg.state_description,
           COUNT(*)                    AS Nu
           SUM(rg.total_rows)          AS To
           SUM(rg.size_in_bytes)       AS To
           SUM(rg.deleted_rows)        AS To
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'
GROUP BY   rg.object_id, i.name, i.type_desc
           rg.partition_number, rg.state_des
ORDER BY   TableName, IndexName, rg.partitio
GO

-- Rebuild the index
ALTER INDEX CCI_FactOnlineSales2 ON dbo.Fact
GO

-- Check the rowgroups metadata again
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
           i.name                      AS In
           i.type_desc                 AS In
           rg.partition_number,
           rg.state_description,
           COUNT(*)                    AS Nu
           SUM(rg.total_rows)          AS To
           SUM(rg.size_in_bytes)       AS To
           SUM(rg.deleted_rows)        AS To
FROM       sys.column_store_row_groups AS rg
```

Listing 8-2: Rebuilding a clustered columnstore index

Rebuilding a columnstore is conceptually exactly the same as rebuilding a rowstore index. SQL Server reads all data from the existing columnstore index, builds a new columnstore index on that data, then drops the old index. After that process, the index

will be as good as new, having all rowgroups (except the last) completely full, all deleted rows really deleted, and all open and closed rowgroups replaced by compressed rowgroups.

As you can see in figure 8-3, running this on my system reduced the number of rowgroups from 22 to 18, and saved almost 21 MB of storage (actually more, because the metadata query doesn't show the storage used for the open rowgroup). While it may appear as if the number of rows has gone down, this is not actually the case – just remember that the reported number of rows in compressed rowgroups includes the rows that are marked as deleted.



Results | Messages | Execution plan

	TableNa...	IndexNa...	IndexType	part...	state_description	Num...	TotalRows	TotalSizeInBytes	TotalDeletedRows
1	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	COMPRESSED	21	16971912	193031008	2903499
2	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	OPEN	1	785764	NULL	NULL

	TableNa...	IndexNa...	IndexType	part...	state_description	Num...	TotalRows	TotalSizeInBytes	TotalDeletedRows
1	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	COMPRESSED	18	14854177	171068832	0

Zoom in | Open in new window

Figure 8-3: Effect of rebuilding a clustered columnstore index

Now that we have seen an index rebuild with default options, let's take a look at the syntax as shown in figures 8-4 and 8-5, to see what other options we have.

```
ALTER INDEX IndexName ON schema.TableName
REBUILD [ PARTITION = ALL ]
        [ WITH ([ DATA_COMPRESSION = { COLUM
                              [ ON PARTITIONS (
                [ , ... ]
                [ , MAXDOP = number ]
                ) ];
```

Figure 8-4: Syntax for rebuilding a complete clustered columnstore index

```
ALTER INDEX IndexName ON schema.TableName
REBUILD PARTITION = number
        [ WITH ([ DATA_COMPRESSION = { COLUM
                [ , MAXDOP = number ]
               ) ];
```

Figure 8-5: Syntax for rebuilding one partition in a clustered columnstore index

As you can see, there are quite a few different options, which is why I split the syntax description over two diagrams. The options in figure 8-4 are used to rebuild the entire index. You can make this explicit by including the `PARTITION = ALL` specification, or you can omit this part. The optional `WITH` clause allows you to choose between either the normal columnstore compression, or the archive columnstore compression that saves even more space by using a more aggressive compression algorithm. For a partitioned table, you can use the `ON PARTITIONS` clause to force different compression types for individual partitions, by specifying the partition number or a range of partition numbers. If you rebuild without specifying a `WITH` clause, each partition will retain its current compression type.

The second set of options, in figure 8-5, rebuilds only a single partition of a clustered columnstore index and optionally changes the compression type.

Both versions allow you to specify the degree of parallelism. When not specified, the rebuild process, just like the initial build, will use all the nodes it can get.

The `COLUMNSTORE_ARCHIVE` keyword seen in the syntax for rebuilding a clustered columnstore index is used to invoke archival expression. As already mentioned briefly in level 2, this is a more aggressive compression algorithm that reduces the on-disk size of the data even more than regular columnstore expression, but at the expense of using more CPU, both when rebuilding the columnstore index and when retrieving data from it. You would normally use this compression type only on partitioned tables, and only on the partitions that contain historical data – data that is so old that it is queried very infrequently. For that type of data, the overhead of performing the extra decompression when retrieving the data will be outweighed by the disk space saving. But for data that is still queried on a regular basis, standard columnstore compression is the recommended level.

The code in listing 8-3 once more rebuilds the entire clustered columnstore index, this time choosing archival compression, and then looks at the space taken. Since the sample sales table used for this series is not partitioned, I can only choose a single compression type for the whole table.

```
USE ContosoRetailDW;
GO

-- Rebuild the index, applying archival comp
ALTER INDEX CCI_FactOnlineSales2 ON dbo.Fact
REBUILD WITH (DATA_COMPRESSION = COLUMNSTORE
GO

-- Check the rowgroups metadata again
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
           i.name                      AS In
           i.type_desc                 AS In
           rg.partition_number,
           rg.state_description,
           COUNT(*)                    AS Nu
           SUM(rg.total_rows)          AS To
           SUM(rg.size_in_bytes)       AS To
           SUM(rg.deleted_rows)        AS To
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'
GROUP BY   rg.object_id, i.name, i.type_desc
           rg.partition_number, rg.state_des
ORDER BY   TableName, IndexName, rg.partitio
GO
```

Listing 8-3: Using archival compression

As can be seen in figure 8-6, the total on-disk space has been reduced from over 163 MB (see figure 8-3) to just over 82 MB – a 50% reduction over the already impressive regular columnstore compression.

Results | Messages | Execution plan

	TableNa...	IndexNa...	IndexType	part...	state_description	Num...	TotalRows	TotalSizeInBytes	TotalDeletedRows
1	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	COMPRESSED	18	14854177	86022418	0

Zoom in | Open in new window

Figure 8-5: Effect of archival compression

I want to stress that the above is for the sake of demonstration only, and not a recommended practice. If you want to use archival compression, choose a partitioning scheme that separates the rarely used data from the frequently used data, and use archival compression only for the partitions with little usage. There are only very few scenarios where archival compression for an entire table is the correct choice.

So far, we have not looked at the effectiveness of rowgroup elimination for the clustered columnstore index. But chances are that this is pretty abysmal. Looking back over the previous levels, we have created the *FactOnlineSales2* table as a heap, which is always completely unordered. We then built a clustered columnstore index directly on top of that, so even that first version of the index was unlikely to have any correlation between the data values and the rowgroups. And then we have added rows, deleted rows, updated rows, and rebuilt the index twice without ever considering order at all. With all that, I would be very surprised if there is even a single column that will ever qualify for rowgroup elimination.

```
USE ContosoRetailDW;
GO

SELECT      p.partition_number,
            s.segment_id,
            MAX(s.row_count) AS row_count,
            MAX(CASE WHEN c.name = N'OnlineSa
                     THEN s.min_data_id END)
            MAX(CASE WHEN c.name = N'OnlineSa
                     THEN s.max_data_id END)
            MAX(CASE WHEN c.name = N'StoreKey
                     THEN s.min_data_id END)
            MAX(CASE WHEN c.name = N'StoreKey
                     THEN s.max_data_id END)
            MAX(CASE WHEN c.name = N'Productk
                     THEN s.min_data_id END)
            MAX(CASE WHEN c.name = N'Productk
                     THEN s.max_data_id END)
FROM        sys.column_store_segments
INNER JOIN  sys.partitions
      ON    p.hobt_id
INNER JOIN  sys.indexes
      ON    i.object_id
      AND   i.index_id
LEFT  JOIN  sys.index_columns
      ON    ic.object_id
      AND   ic.index_id
      AND   ic.index_column_id
LEFT  JOIN  sys.columns
      ON    c.object_id
      AND   c.column_id
```