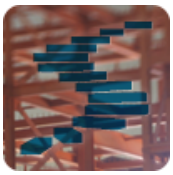
SQL ServerCentral.com
SQL Prompt Code analysis as you type
SQLQuery1.sql
Try it now redgate
a community service from redgate
Try SQL Clone

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 6: Updating and Deleting Data in a Columnstore Index



By Hugo Kornelis, 2016/01/20

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 the previous level I discussed the difficulties of adding data to columnstore indexes, and the workarounds implemented in SQL Server 2014 to enable inserts into tables with columnstore indexes – even if the inserted data is not always immediately converted into column-oriented format.

But even in data warehouses, inserts are not the only modification that has to be supported. We must also be able to update and delete data. This level looks in detail at what happens when we update or delete data from a clustered columnstore index, the impact it has on concurrent data access, and how without careful maintenance the efficiency of columnstore indexes can degrade over time.

Challenges of modifying data in Columnstore indexes

Deleting or updating data in compressed column-oriented storage comes with another new set of challenges. Many compression algorithms replace a value with a shorter value that is derived from all preceding data. Changing or removing a single value at the start of a block of compressed data would mean that the compressed value for all successive values has to be recomputed. And even in a relatively simple compression method such as row-length encoding, a single update can result in having to rewrite many pages of data.

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

Related Articles

ARTICLE Stairway to Columnstore Indexes Level 5: Adding New Data To Columnstore Indexes
2015/11/05
Earlier levels have shown how Columnstore Indexes work effectively with static data. In most tables ...

BLOG COLUMNSTORE INDEX... DEMO
2015/07/11
COLUMNSTORE INDEX How data is

BLOG

ARTICLE

ARTICLE
2014/07/21
SQL Server 2012 introduced columnstore indexes, which can immensely improve the performance of OLAP ...

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 © 2002-2016 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

Cost of in-place update: example

One of the compression methods used for columnstore indexes, and one of the simpler compression methods to understand, is row-length encoding. This method relies heavily on sorting data efficiently before compressing it. If, for instance, the sorted data has 17 consecutive rows with the value `Niko` in the *Name* column, those 17 values would be replaced by a single *{Value, Occurrences}* pair: `{Niko, 17}` (which can be read as "repeat the value `Niko` 17 times).

If the 6th of those rows would be updated to `Kalen`, the most efficient choice would be to move it to the end of the block of 17. But because rows are tied together by the relative position of values in each segment that would necessitate moving values in the corresponding segments for other columns as well, so this is not possible. Hence, the only way to reflect this change in a row-length encoded column would be to replace the single pair `{Niko, 17}` with a sequence of three pairs: `{Niko, 5} / {Kalen, 1} / {Niko, 11}`. This obviously takes more space, so all the data following it now has to be moved down to make room for this expansion. If this happens at the start of a one-million row segment, you can imagine that a lot of pages will have to be updated for this one modification.

It is, in theory, possible to delete data directly from a columnstore index. But it's not easy. The corresponding values need to be found in, and deleted from each of the rowgroup's segments. Depending on the compression type, this may require de- and recompressing parts or even all of the data in the segment. The end effect would be a `DELETE` statement that runs at a speed that most people would not find acceptable. It is for that reason that Microsoft has picked a different solution. To see how this works, let's first run the code in listing 5-1. This code will delete one percent of the data, spread across all rowgroups.

```
USE ContosoRetailDW;
GO

-- Delete one percent of the rows, throughou
DELETE FROM dbo.FactOnlineSales2
WHERE       OnlineSalesKey % 100 = 33;

-- Check the rowgroups metadata
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
i.name                      AS IndexName,
i.type_desc                 AS IndexType,
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.objec
AND   i.index_id                     = rg.index
WHERE       i.name = N'CCI_FactOnlineSales2'
ORDER BY    TableName, IndexName,
rg.partition_number, rg.row_group_id;
```

Listing 6-1: Deleting data from a columnstore

Results | Messages | Execution plan

	TableNa...	IndexName	IndexType	part...	row_g...	total_ro...	size_in_...	deleted_rows	state	state_description
1	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	0	1048576	10677496	10221	3	COMPRESSED
2	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	1	1048576	10458680	10199	3	COMPRESSED
3	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	2	1048576	11041440	10668	3	COMPRESSED
4	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	3	1048576	11529976	10826	3	COMPRESSED
5	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	4	1048576	11710424	10446	3	COMPRESSED
6	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	5	1048576	11445048	10601	3	COMPRESSED
7	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	6	1048576	11923744	10430	3	COMPRESSED
8	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	7	1048576	12285032	10524	3	COMPRESSED
9	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	8	1048576	12521896	10441	3	COMPRESSED
10	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	9	1048576	12838888	10442	3	COMPRESSED
11	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	10	37656	519584	368	3	COMPRESSED
12	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	11	2048	28920	17	3	COMPRESSED
13	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	12	1048576	12929024	10549	3	COMPRESSED
14	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	13	1048576	13034408	10492	3	COMPRESSED
15	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	14	1920	25680	20	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	15	3072	38536	32	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	17	150000	1643536	1500	3	COMPRESSED
18	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	18	66255	NULL	NULL	1	OPEN
19	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	19	1048576	8166568	10486	3	COMPRESSED

Zoom in | Open in new window

Figure 6-1: Metadata after deleting data

In comparison to the queries in the previous level, I added one additional column to the resultset of the DMV query: *deleted_rows* . This column is not used for deltastore rowgroups: as these are stored in traditional row-oriented format, it is easy enough for SQL Server to simply delete these rows. So that is what SQL Server will do, for all rows to be deleted from both open and closed deltastores. If you compare *total_rows* of rowgroup 18 to what it was before deleting data, you will see that 669 rows were deleted from this rowgroup.

But for rowgroups that are already in columnstore format (state 3, `COMPRESSED`), a different method was used. These rows were not actually deleted from the columnstore blobs. Instead, SQL Server simply marks the row as deleted without removing any data. The data itself is still present in the index, but the "deleted" mark tells SQL Server to ignore this data and pretend it isn't there. The data returned by the DMV query reflects this method of logically deleting the data without physically removing it – the *total_rows* of the rowgroups has not been changed, because all the rows are still there; the *deleted_rows* column tells us how many of these rows are no longer valid. If you have a clustered columnstore index that is frequently modified, you should monitor these numbers to decide when it is time to free up wasted space (and regain performance) by physically removing the deleted data – I'll cover more about that in the next level of this series.

The bitmap that is not a bitmap

When rows in a columnstore index have to be marked as deleted, an extra storage structure is added to the index. This storage structure is often referred to as the " *Deleted Bitmap* ", but that is actually a misleading name. The delete marks are not stored as a bitmap, but in an internal, row-compressed B-tree, as a virtual two-column table storing only rowgroup number and row number of the row within that rowgroup.

Luckily, we do not have to concern us with these implementation details. I do want to point it out, because bitmaps tend to have a fixed size, whereas the actual storage of delete marks grows as more rows in the rowgroup are deleted.

Logically speaking, updating data can be considered as equivalent to deleting the old version of the data and inserting the new data. In the case of SQL Server, this is exactly what happens when rows in a columnstore are updated. Not the rows that are still in a deltastore: these are still in a B-tree, so they can easily be processed by the regular update mechanism. But for the rows that are already compressed in columnstore rowgroups, every update is processed by marking the old row as deleted and trickle inserting the new data in a deltastore.

You can see this in action by running the code in listing 6-2. This code updates about 20% of the data in the table, well over two million rows. That takes time (12 minutes on my test system, a VM on my laptop), but I chose this high number because I wanted to demonstrate that, even for extremely large updates, the new rows are always added as trickle inserts, not using the bulk load mechanism that loads the data directly in column-oriented format.

```
USE ContosoRetailDW;
GO

-- Update twenty percent of the rows, throug
UPDATE     dbo.FactOnlineSales2
SET        UpdateDate = CURRENT_TIMESTAMP
WHERE      OnlineSalesKey % 5 = 2;

-- Check the rowgroups metadata
SELECT     OBJECT_NAME(rg.object_id)   AS Ta
i.name                          AS IndexName,
i.type_desc                     AS IndexType,
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.objec
AND  i.index_id                   = rg.index
WHERE      i.name = N'CCI_FactOnlineSales2'
ORDER BY   TableName, IndexName,
rg.partition_number, rg.row_group_id;
```

Listing 6-2: Updating data in a columnstore

Results | Messages

	TableNa...	IndexName	IndexType	part...	row_g...	total_ro...	size_in_...	deleted_rows	state	state_description
1	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	0	1048576	10677496	219934	3	COMPRESSED
2	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	1	1048576	10458680	219881	3	COMPRESSED
3	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	2	1048576	11041440	220385	3	COMPRESSED
4	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	3	1048576	11529976	220553	3	COMPRESSED
5	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	4	1048576	11710424	220105	3	COMPRESSED
6	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	5	1048576	11445048	220395	3	COMPRESSED
7	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	6	1048576	11923744	220152	3	COMPRESSED
8	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	7	1048576	12285032	220377	3	COMPRESSED
9	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	8	1048576	12521896	220008	3	COMPRESSED
10	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	9	1048576	12838888	220221	3	COMPRESSED
11	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	10	37656	519584	7873	3	COMPRESSED
12	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	11	2048	28920	433	3	COMPRESSED
13	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	12	1048576	12929024	220390	3	COMPRESSED
14	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	13	1048576	13034408	220042	3	COMPRESSED
15	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	14	1920	25680	403	3	COMPRESSED
16	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	15	3072	38536	645	3	COMPRESSED
17	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	17	150000	1643536	31500	3	COMPRESSED
18	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	18	1048576	NULL	NULL	2	CLOSED
19	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	19	1048576	8166568	220202	3	COMPRESSED
20	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	20	1048576	NULL	NULL	2	CLOSED
21	FactOnli...	CCI_Fac...	CLUSTERED CO...	1	21	734340	NULL	NULL	1	OPEN

Zoom in | Open in new window

Figure 6-2: Metadata after updating data

If you look at the result after running the code in listing 6-2, you will see that there are now many rows marked as deleted in the compressed rowgroups. You also see two closed and one open rowgroup, the result of adding over two million "new" (updated) rows as trickle inserts into the deltastores. After at most five minutes, the tuple mover will kick in and start to convert these rowgroups to columnstore format. As explained before, this will be done for one rowgroup at a time, with intervals of 10-15 seconds between the rowgroups.

If we had updated the entire table, all originally compressed rowgroups would now be empty. They would still exist and contain all the original data, but all rows in them would be marked as deleted. At the same time, all 13.5 million rows in the table would have been inserted in 14 deltastores, that would then, over time, be converted to columnstore format by the tuple mover. As mentioned in the previous level, the tuple mover is a slow worker so it will take time until all rows are converted to columnstore format; during this time, queries that use the columnstore index will be slow. And after that, we still have all the old rowgroups taking up disk space because the deleted data is still there, until we take explicit action to rebuild the index. The mechanism used for updating data in a columnstore index has not been designed for this type of mass update. It works very well and with only minimal performance impact for occasional updates of single rows or small subsets of the population, but not for mass updates of the entire table or large subsets.

The read-only limitation of the nonclustered columnstore index automatically limited the options for data modification to just two strategies: either you would disable the index, make the modifications, then rebuild the index, or you would swap in and out entire partitions of the table – a very common strategy for the incremental load of a data warehouse. Both of these strategies will normally be done when there is no concurrent activity. But the option of modifying data in a clustered columnstore index in SQL Server 2014 means that we will now see modifications during business hours – so obviously we can also expect to see locking and blocking. To check how these modifications affect concurrency, let's investigate the locks taken for a simple update of a single row. First, run the code from listing 6-3. Note that the `ROLLBACK` statement has deliberately been put in a comment block. We will later undo the change by selecting just the line with the `ROLLBACK` statement and then executing that selection, but we need the transaction to remain open for now.

```
USE ContosoRetailDW;
GO

-- Start a transaction, so we can investigat
BEGIN TRAN;

-- Update just a single row
UPDATE dbo.FactOnlineSales2
SET    UpdateDate = CURRENT_TIMESTAMP
WHERE  OnlineSalesKey = 30000002;

-- Do not rollback the transaction yet!
/*
```

```
ROLLBACK TRAN;
*/
```

Listing 6-3: Updating a single row

After running this update, open another tab in Management Studio and use it to run the code in listing 6-4 to get an overview of all locks currently held in the database.

```
USE ContosoRetailDW;
GO

-- Check the locks that are taken
-- (Query adapted from http://www.nikoport.c
SELECT     l.request_session_id,
DB_NAME(l.resource_database_id) AS database_
CASE
WHEN l.resource_type = 'OBJECT'
THEN OBJECT_NAME(l.resource_associated_entit
ELSE OBJECT_NAME(p.[object_id])
END AS [object_name],
i.name AS index_name,
l.resource_type,
l.resource_description,
l.request_mode,
l.request_status
FROM       sys.dm_tran_locks AS l
INNER JOIN sys.partitions AS p
ON   p.hobt_id = l.resource_associated_entit
INNER JOIN sys.indexes AS i
ON   i.[object_id] = p.[object_id]
AND  i.index_id = p.index_id
WHERE      l.resource_associated_entity_id >
AND        l.resource_database_id = DB_ID()
ORDER BY   l.request_session_id, resource_as
```

Listing 6-4: Checking the locks

Results | Messages

	reque...	database_name	object_name	index_name	resource_type	resource_description	reque...	request_s...
1	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:21	X	GRANT
2	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:23	X	GRANT
3	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:1	U	GRANT
4	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:0	U	GRANT
5	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:3	U	GRANT
6	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a000001:2	U	GRANT

Zoom in | Open in new window

Figure 6-3: Locks for a single update

As you see, updating a single row results in two exclusive locks, both on a full rowgroup. To see which rowgroup is locked, check the last part of the resource_description column. On my system, rowgroups 21 and 23 are locked. Rowgroup 23 is where the updated row used to be stored; the lock is needed to mark the row as deleted. Rowgroup 21 is the open deltastore, where the contents of the row after the update are added. When you run this code, you may see different rowgroup numbers. That's because how rows are distributed among rowgroups, and even the number of rowgroups created, depends on factors like processors and memory available when the columnstore index is created. Even when dropping and recreating the same columnstore index on the same instance, the rowgroup distribution can be different. You may also see only a lock on the open delta store; in that case the row you updated was moved there as a result of a previous update.

In figure 6-3 above, you also see U locks on rowgroups 0 through 3. SQL Server takes a U lock on all rowgroups it scans for an `UPDATE` statement. Normally that U lock is released when the rowgroups is scanned with no rows found to be updated, but due to internals of the processing mechanism that are out of scope for this article, these U locks are sometimes held. Do not worry if these locks do not show up when you run the code.

Looking full rowgroups is not very granular. However, it gets worse when you investigate the actual result of these locks. To see why, open a third query window and execute the code in listing 6-5, to retrieve data about a single row. On my system, this row resides in rowgroup 2. (I verified this before running the code in listing 6-3, by running that same code with *OnlineSalesKey* 31000000, using the code in listing 6-4 to check the locked rowgroups, and then rolling back the change). This rowgroup has only a U lock, which does not block queries that read data, so you might expect that this query will return. That is not the case. It will wait indefinitely, because it is blocked by the update from our first query. We can easily verify this by rerunning listing 6-4 to check the locks, which now returns eight rows as shown in figure 6-4.

```
USE ContosoRetailDW;
GO

-- Run a simple query
SELECT OnlineSalesKey, StoreKey, CustomerKey
FROM   dbo.FactOnlineSales2
WHERE  OnlineSalesKey = 31000000;
```

Listing 6-5: Trying to read a single row

Results | Messages

	reque...	database_name	object_name	index_name	resource_type	resource_description	reque...	request_...
1	52	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:23	S	WAIT
2	52	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:21	S	WAIT
3	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:23	X	GRANT
4	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:21	X	GRANT
5	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:1	U	GRANT
6	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:0	U	GRANT
7	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:3	U	GRANT
8	56	ContosoRetailDW	FactOnlineSales2	CCI_FactOnlineSales2	ROWGROUP	ROWGROUP: 7:100000000a00000:2	U	GRANT

Zoom in | Open in new window

Figure 6-4: Select blocked by the update

These blocks occur because the optimizer doesn't know that values in the *OnlineSalesKey* column are unique. We were unable to tell SQL Server, because one of the many restrictions for clustered columnstore indexes is that they don't support `PRIMARY KEY` or `UNIQUE` constraints on the table. (We'll see more details on these restrictions in a later level). So even though SQL Server does find a row in rowgroup 2, it still wants to search all other rowgroups to see if there are more matching rows; hence it has to wait until the locks on rowgroups 21 and 23 are lifted. If you go back to listing 6-3 and execute only the `ROLLBACK` statement, you will see that the query running listing 6-5 now finishes, and rerunning listing 6-4 will not return any results. Committing the update would of course have had the same effect.

Effectively, the locks taken at rowgroup level block **almost** all concurrent access to the table. There are only a few very specific scenarios where this is not the case. To see one of them, rerun listing 6-3 to acquire the same locks again, then run listing 6-6. This is the same as listing 6-5, but with a `TOP(1)` clause added. This query will actually return results immediately. After finding the matching row in rowgroup 2, SQL Server no longer needs to check for matches in other rowgroups. It does not need