SQL ServerCentral.com
SQL Provision NEW Provision databases efficiently and keep data safe TRY FREE
a community service from redgate
Try SQL Clone

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

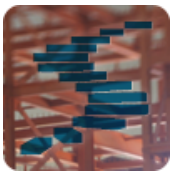
Write For Us
Find out more

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

Improve your SQL Server knowledge daily with more articles by email.
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 7: Optimizing Nonclustered 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.

So far in this series, we have just been creating and using columnstore indexes without giving any consideration to optimizing these indexes. We just trusted that they would work and give us a huge performance benefit. And they did. As with many features in SQL Server, Microsoft has implemented the columnstore index feature in such a way that they will work just fine even if you don't spend any effort on optimizing them.

However, again as with many other features, optimizing the columnstore index right from the start may well gain you even more performance benefits. And lack of proper maintenance to keep the columnstore indexes healthy will probably cause their benefit to deteriorate over time. So in the next levels, we will start to look at what you can do to optimize your columnstore indexes.

In this level, we will focus on optimization techniques to apply while building the nonclustered columnstore index, which is available in all versions of SQL Server from 2012 up. The next level will then show how to apply the same technique to the clustered columnstore index that was introduced in SQL Server 2014, as well as how to do the additional maintenance that is needed for these indexes when data in the table changes over time.

Related Articles

BLOG **Clustered Columnstore Indexes – part 34 ("Deleted Segments Elimination")**
2014/07/20
Continuation from the previous 33 parts, starting from http://www.nikoport.com/2013/07/05/clustered

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

BLOG

ARTICLE

BLOG **Clustered Columnstore Indexes – part 35 ("Trace Flags & Query Optimiser Rules")**
2014/07/24
Continuation from the previous 34 parts, starting from http://www.nikoport.com/2014/07/20/clustered

Tags

columnstore index
nonclustered index
stairway series

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

Write For Us
Find out more

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

In level 2 of this series, I explained the main reasons why columnstore indexes can be so much faster than rowstore indexes: less I/O is needed because column elimination can be applied, rowgroup elimination is sometimes possible, and the data that still has to be read is compressed better. Additionally, batch mode can allow much faster processing. We can try to influence each of these factors, but they require different techniques at different times.

There is no good way to improve the compression effectiveness when building the columnstore index. You might try to ensure that SQL Server processes data in a sequence that is beneficial for compression, but my experiments so far have shown little to no effect. This is because SQL Server always reorders the rows in a rowgroup before starting compression, so whatever ordering I apply before the build process starts is lost. In the next level I will show an option that was introduced in SQL Server 2014 that does give us some control over the compression effectiveness, but this cannot be used while building the index.

There is one way to maximize the benefit of column elimination, and this method is obvious: ensure that all your queries that target a table with a columnstore index on it use only the columns you really need. So you should remove all columns you don't really need from your query, and you definitely should avoid using `SELECT *` at the outermost level. If you have a fact table that has the same data in different formats (e.g. two columns for status code and status description), then don't mix the two in a single query – or better yet, move the description out to its own dimension table. But none of those techniques are new; all of these are relevant for optimizing the performance of rowstore indexes as well, so you probably already apply them.

The last two techniques, rowgroup elimination and batch mode execution, give us the most opportunity for optimization. Optimizing batch mode may require rewriting queries; this will be covered in later levels. Optimizing for rowgroup elimination is the focus of the rest of this level.

As mentioned before, forcing the data to be arranged in a specific order when building the columnstore index will not help for compression, because the data will be reordered as part of the index build process. But that reordering is done after dividing the data up into rowgroups, and that means that we can still use this to increase the effect of rowgroup elimination.

When SQL Server builds a columnstore index, it respects the partitioning scheme. Any rowgroup will only have rows from a single partition. Within each partition (or within the entire table if partitioning is not used), rows are assigned to rowgroups in the order in which they are read. If the table is a heap, then this is the order in which they happen to be allocated on disk – which is for all practical purposes beyond our control. However, if the table has a clustered index, then the rows will be read from that index, so the rows that are logically "first" in the index will all be in the "first" segments, and the rows that are logically "last" will be in the "last" segments.

Let's first examine the nonclustered columnstore index I created back in level 3 on the *FactOnlineSales* table. This table already had a clustered index on the *OnlineSalesKey* column when we built the columnstore index. Let's verify the minimum and maximum values in each rowgroup for a few columns, using a

variation on one of the queries from level 4. (Note that I selected just a few columns, and only those with an integer data type. For other data types, the relationship between the *min_data_id* and *max_data_id* columns and the actual minimum and maximum values is not documented, and beyond the scope of this stairway).

```
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
WHERE       i.name
AND         c.name IN (N'OnlineSalesKey', N'S
GROUP BY    p.partition_number,
            s.segment_id;
```

Listing 7-1: Reviewing some metadata for rowgroup elimination

The results of this query on my system are shown below. The results on your system may be different, depending on processors and memory available when the columnstore index was built, but they should still expose a similar pattern:

	parti...	segm...	row_count	MinOnlineSalesKey	MaxOnlineSalesKey	MinStoreKey	MaxStoreKey	MinProductKey	MaxProductKey
1	1	0	1048576	19560484	23409748	199	307	1	2517
2	1	1	1048576	19709057	23661939	199	307	1	2517
3	1	2	1048576	19658935	23730293	199	307	1	2517
4	1	3	1048576	19609637	23390327	199	307	1	2517
5	1	4	1048576	23405749	28129614	199	307	1	2517
6	1	5	1048576	23390328	27684505	199	307	1	2517
7	1	6	1048576	23661940	28042395	199	307	1	2517
8	1	7	1048576	23730294	28117284	199	307	1	2517
9	1	8	1048576	28129615	31582782	199	307	1	2517
10	1	9	887578	27684506	31866341	199	307	1	2517
11	1	10	948564	28117285	32034748	199	307	1	2517
12	1	11	1048576	28042396	32053451	199	307	1	2517
13	1	12	171066	31582783	31753848	199	307	8	896
14	1	13	134640	32053452	32188091	199	307	1052	1293

Zoom in | Open in new window

Figure 7-1: Rowgroup elimination data

If you look at the data for the *OnlineSalesKey* column above, you will see that the first four rowgroups roughly hold the data for values between 19.5 million and 24 million; the next four rowgroups have data from 23.5 million to 28 million, and all values above 28 million are in the last six rowgroups. If a query has a filter on this column, rowgroup elimination will reduce the number of rowgroups to scan to at most six, and usually just four.

But if you look at the other columns, you will see that filtering on them will not result in any significant reduction – all rowgroups contain the full range of possible *StoreKey* values, and only the two small rowgroups at the end can be eliminated based on *ProductKey*.

In a typical datawarehouse or reporting database, I would not expect to see a lot of queries that filter on the *OnlineSalesKey*. If you want to get more rowgroup elimination on this table, you first have to investigate which columns are most often used as a filter condition in the queries that you want to speed up. Sometimes a single column stands out; sometimes you will have to make a judgement call between several contenders. Unfortunately, you will have to pick a single column that you can optimize for, it is not possible to pick several.

A note on correlated columns

A lot of fact tables have columns with correlated data. The most obvious example is a table for order data, with columns for order date, shipment date, and payment due date. The shipment date and payment due date will never be before the order date, and for most businesses they will hardly ever be more than a month after the order date. In such cases, a "free" side effect of this correlation is that when you optimize for rowgroup elimination on one of these columns, the rowgroup elimination benefit on the correlated columns will also improve. You will most likely not get as much benefit for these related columns because the correlation will probably not be 100%, but it can still help.

Since a lot of datawarehouse queries on order tables actually do filter on the various dates in those tables, optimizing for rowgroup elimination on the order table is in fact a very common strategy, which also nicely coincides with the partition switching strategies for working around the read-only limitation of nonclustered columnstore indexes.

Once you have decided for which column you want to optimize the columnstore index, you can drop the existing columnstore index, change the table to be clustered on the selected column, and then recreate the columnstore index. (Note that changing the choice of clustered index will also impact the performance of queries that use the rowstore indexes on the same table, so you will have to take that into account as well). For example, the code below will replace the existing nonclustered columnstore index on this table with a new one that is optimized for rowgroup elimination based on the *ProductKey* column. (Note that this script takes some time, because it has to create a new clustered index, create a nonclustered index, and then recreate the columnstore index for a 12.6 million row table. On my laptop, these operations took a total of 4:30 minutes).

```
USE ContosoRetailDW;
GO

-- First drop the existing nonclustered colu
```

```
DROP INDEX NCI_FactOnlineSales ON dbo.FactOn
GO

-- If the table has a lot of nonclustered rc
-- Recreate them at the end of the script if
-- Dropping and recreating is faster than ke

-- Clustered index is tied to primary key, s
ALTER TABLE dbo.FactOnlineSales
DROP CONSTRAINT PK_FactOnlineSales_SalesKey

-- Now create a clustered index on the desir
CREATE CLUSTERED INDEX ix_FactOnlineSales_Pr
ON dbo.FactOnlineSales(ProductKey);

-- Recreate the PRIMARY KEY constraint, now
ALTER TABLE dbo.FactOnlineSales
ADD CONSTRAINT PK_FactOnlineSales_SalesKey
    PRIMARY KEY (OnlineSalesKey);

-- If the table has a lot of nonclustered rc
-- Do not forget to script the indexes (if r

-- Finally, recreate the nonclustered colum
CREATE NONCLUSTERED COLUMNSTORE INDEX NCI_Fa
ON dbo.FactOnlineSales
   (OnlineSalesKey, DateKey, StoreKey, Produ
    CustomerKey, SalesOrderNumber, SalesOrde
    ReturnQuantity, ReturnAmount, DiscountQu
    UnitCost, UnitPrice, ETLLoadID, LoadDate
GO
```

Listing 7-2: Optimizing rowgroup elimination for a specific column

Once the code from listing 7-2 has finished, you can rerun the query in listing 7-1 to see how the selected columns will now fare in rowgroup elimination. The results on my system are shown in figure 7-2 below.

	parti...	segm...	row_count	MinOnlineSalesKey	MaxOnlineSalesKey	MinStoreKey	MaxStoreKey	MinProductKey	MaxProductKey
1	1	0	1048576	19560526	31868521	199	307	66	900
2	1	1	1048576	19560484	31541783	199	307	81	1052
3	1	2	1048576	19560809	31868520	199	307	1	991
4	1	3	1048576	19560666	31963596	199	307	13	952
5	1	4	1048576	19560489	32188091	199	307	1052	1865
6	1	5	1048576	19560518	32175757	199	307	900	1820
7	1	6	1048576	19560615	31003367	199	307	991	1805
8	1	7	1048576	19560509	31003399	199	307	952	1864
9	1	8	1048576	19560534	31003351	199	307	1865	2506
10	1	9	1048576	19560671	31003287	199	307	1820	2511
11	1	10	61429	19561391	30915534	199	307	2506	2507
12	1	11	7147	20342898	29828626	199	307	2511	2511
13	1	12	1008628	19560661	31003359	199	307	1864	2517
14	1	13	1048576	19560513	31003003	199	307	1805	2517
15	1	14	16068	19689829	30552770	199	307	2517	2517

Zoom in | Open in new window

Figure 7-2: Optimized rowgroup elimination data

As you can see, the *ProductKey* values now have a much better distribution, allowing for a lot of rowgroups to be eliminated if a query filters on that column. However, almost every rowgroup (except the small ones for the leftover rows) now spans almost the entire range of *OnlineSalesKey*, so any queries that filter on this column will now be slower. That is why it is so important to optimize for rowgroup elimination on the column that is most often used as a filter in the queries that are the most important to optimize – because whatever choice you make, you will always pay a price for it.

In level 3 of this stairway, I explained that limiting the degree of parallelism can help to reduce the memory required to build the columnstore index, at the expense of a longer duration for the process. But that is not the only effect. A lower degree of parallelism will also usually reduce the number of small rowgroups for the leftover rows at the end of the build process for a partition, and you can get an ever better optimization for rowgroup elimination by using fewer cores when creating the index.

As you can see in figure 7-2, there are four rowgroups that all have data for *ProductKey* values up to approximately 1000; four rowgroups for *ProductKey* values between 1000 and 1800; and seven for the values above 1800. This is because a single scan over the entire table, in order of the clustered index on *ProductKey*, was feeding four threads at the same time when the index was being built. A query that filters on a single *ProductKey* value will usually still have to scan between four and seven rowgroups. A lot of rowgroups can be eliminated, but we can optimize this even further by building the index on a single thread.

The code in listing 7-3 once again drops the columnstore index and then recreates it using a `MAXDOP` hint that limits the process to a single processor.

```
USE ContosoRetailDW;
GO

-- First drop the existing nonclustered colu
DROP INDEX NCI_FactOnlineSales ON dbo.FactOn
GO

-- Then recreate the nonclustered columnstor
CREATE NONCLUSTERED COLUMNSTORE INDEX NCI_Fa
ON dbo.FactOnlineSales
   (OnlineSalesKey, DateKey, StoreKey, Produ
    CustomerKey, SalesOrderNumber, SalesOrde
    ReturnQuantity, ReturnAmount, DiscountQu
    UnitCost, UnitPrice, ETLLoadID, LoadDate
WITH (MAXDOP = 1);
GO
```

Listing 7-3: Rebuilding the columnstore index without parallelism

Figure 7-3 below shows the results on my system of rerunning the code from listing 7-1 after rebuilding the nonclustered columnstore index on a single thread. As you can see, SQL Server now never has to scan more than at most two rowgroups when a query filters on a single *ProductKey* value.

	parti...	segm...	row_count	MinOnlineSalesKey	MaxOnlineSalesKey	MinStoreKey	MaxStoreKey	MinProductKey	MaxProductKey
1	1	0	1048576	19560717	31868521	199	307	1	176
2	1	1	1048576	19560526	31541783	199	307	176	429
3	1	2	1048576	19560773	31003167	199	307	429	711
4	1	3	1048576	19560484	31963606	199	307	711	957
5	1	4	1048576	19560801	32188090	199	307	957	1204
6	1	5	1048576	19560509	32188091	199	307	1204	1457
7	1	6	1048576	19560518	31003399	199	307	1457	1683
8	1	7	1048576	19560489	31002875	199	307	1683	1845
9	1	8	1048576	19560730	31003359	199	307	1845	2151
10	1	9	1048576	19560513	31003351	199	307	2151	2445
11	1	10	1048576	19561359	31003183	199	307	2445	2502
12	1	11	1048576	19560534	31000556	199	307	2502	2517
13	1	12	44696	19623547	30981647	199	307	2517	2517

Zoom in | Open in new window

Figure 7-3: Fully optimized rowgroup elimination data

If your maintenance window is long enough, this is the best possible scenario for rowgroup elimination if *ProductKey* is the column used as a filter in most of your time-critical queries. An additional benefit is that there are now fewer "small" rowgroups: all leftover rows are stored in a single rowgroup.

In this level we showed how building a carefully chosen clustered index before building a nonclustered columnstore index can result in rowgroups that can benefit far more from rowgroup elimination. It will take more time to build the index that way, and it is important to remember that the table becomes read-only after building the nonclustered columnstore index, so it is not possible to change the clustered index again after building the columnstore index. This can be relevant if you also have workloads that use the rowstore indexes, as they might lose performance due to the changed clustered index. In such a case you may end up having to make a trade off.

The same preordering mechanism can be applied to clustered columnstore indexes as well, but the syntax required is slightly different. This, along with the additional tasks needed to keep the clustered columnstore index in good shape as data changes, will be covered in the next level.

This article is part of the Stairway to Columnstore Indexes Stairway

Sign up to our RSS feed and get notified as soon as we publish a new level in the Stairway!

Keep up to date with SQL Server - new articles every day.

your@email.com

Sign up

Thank this author by sharing:

0

Rate this | Join the discussion | Add to briefcase

Total article views: 2088 | Views in the last 30 days: 15