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

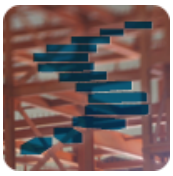
Write For Us
Find out more

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 4: Catalog Views



By Hugo Kornelis, 2015/08/05

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.

If you need to find information about columnstore indexes in your database, you can use the tools and options in SQL Server Management Studio. But it is generally easier (once you've climbed the learning curve) to get the information you need by directly querying the catalog views – and as a bonus, this metadata exposes additional information that cannot be accessed from the Management Studio GUI.

In this level, we will first take a look at how to recognize columnstore indexes in some of the generic catalog views. After that we will investigate the new catalog views that have been added just for columnstore indexes.

Preparation

In the previous levels, I have used only a nonclustered columnstore index in the sample code, so that readers who have not yet upgraded to SQL Server 2014 can still follow along. But in this level I want to show how both types of columnstore index are shown in the catalog views, so I will now create a clustered columnstore index as well. Readers using SQL Server 2012 can still follow along, but they will have to skip the first listing, and they will obviously not get the output related to the clustered columnstore index.

The code in this level is all based on the ContosoRetailDW database as it was after the first level, so after restoring the original database and creating the nonclustered columnstore index. The code in listing 4.1 creates a copy of the FactOnlineSales table, called FactOnlineSales2, and creates a

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 Clustered Columnstore Indexes – part 48 ("Improving Dictionary Pressure")
2015/02/18

BLOG

BLOG

BLOG Columnstore Indexes – part 84 ("Practical Dictionary Cases")
2016/06/20
Continuation from the previous 83 parts, the whole series can be found at http://www.nikoport.com/co...

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
catalog views
columnstore index
stairway series

Write For Us
Find out more

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

clustered columnstore index on it. Note that creating the index this way will probably not result in the best possible performance; we will look at better ways in a later level.

```
USE ContosoRetailDW;
GO

SELECT *
INTO   dbo.FactOnlineSales2
FROM   dbo.FactOnlineSales;

CREATE CLUSTERED COLUMNSTORE INDEX CCI_Fact(
ON dbo.FactOnlineSales2;
```

Listing 4-1: Creating a table with a clustered columnstore index

After running this code, you will now have two tables: *FactOnlineSales*, with a nonclustered columnstore index *NCI_FactOnlineSales*; and *FactOnlineSales2*, with a clustered columnstore index *CCI_FactOnlineSales2*. Readers who are using SQL Server 2012 will have only the first of these two tables, and hence only the nonclustered columnstore index.

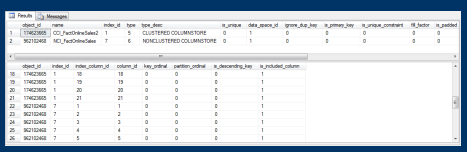
You can find information about indexes in two generic catalog views: *sys.indexes* and *sys.index_columns*. The code in listing 4-2 queries these two views for just the two existing columnstore indexes, and figure 4-1 shows partial results on my system. The full output of the second query is 42 rows: 2 tables, 21 columns in each, and all of them are included in the indexes.

```
USE ContosoRetailDW;
GO

SELECT     i.*
FROM       sys.indexes       AS i
WHERE      i.name            IN (N'NCI_Fact(
                                 N'CCI_Fact(
--WHERE      i.object_id       IN (OBJECT_ID
--                                 OBJECT_ID
--AND        i.type            IN (5, 6);

SELECT     ic.*
FROM       sys.index_columns AS ic
INNER JOIN sys.indexes       AS i
      ON   i.object_id        = ic.object_i(
      AND  i.index_id         = ic.index_id
WHERE      i.name            IN (N'NCI_Fact(
                                 N'CCI_Fact(
```

Listing 4-2: Querying index and index columns information



Zoom in | Open in new window

Figure 4-1: Index and index columns information

Both these views existed before columnstore indexes were added in SQL Server 2012, and no new columns have been added to them. The range of possible values for the columns *type* and *type_desc* in *sys.indexes* has been expanded, however: type 6 with description `NONCLUSTERED_COLUMNSTORE` was introduced in SQL Server 2012; in SQL Server 2014, type 5 with description `CLUSTERED_COLUMNSTORE` was added. Many of the columns in this table are not used for columnstore indexes and are therefore always 0 when the index is a columnstore.

The data returned from *sys.index_columns* is pretty useless by itself; you will usually want to join it to other catalog views such as *sys.objects*, *sys.indexes*, and *sys.columns* to translate the *object_id*, *index_id*, and *column_id* values in the names of the table, index, and column. But even then, this data is still rather uninteresting for columnstore indexes: There will always be one row for each column in the index, and each column will be marked as an included column. This is strictly speaking not entirely true, as the concept of "included" columns is very much tied to the design of rowstore indexes. But within the design of this catalog view, this was probably the best Microsoft could have done without changing the view and potentially breaking existing applications that rely on this view.

Filter on index name or table name?

The queries in listing 4-2, as well as the queries in the rest of this article, all filter based on the name of the columnstore index. Unless your company has a very strict naming policy, chances are that you do know the table name, but do not know the index name. In that case, you can use the alternative form of the filter shown as a comment in the first query in listing 4-2. Keep in mind that this second form of the query will return information on all indexes, not just the columnstore indexes; if you remove the additional filter on the *type* column if you use the same alternative for the *sys.index_columns* table, you can use the *object_id* column from that table instead of the *object_id* from *sys.indexes* – and if you also decide not to filter on *type*, you can then even remove the join to *sys.indexes*.

When Microsoft implemented columnstore indexes, they did not just add new values for some columns in existing catalog views; they implemented several completely new catalog views as well, designed to give insight in the specific columnstore features. Two of these were released in SQL Server 2012; one more was added in SQL Server 2014. One of the two views available since the first release of columnstore indexes is *sys.column_store_segments*, which (as the name implies) gives detailed information about the individual segments within the columnstore index.

As with most catalog views, you need to join this view to others to get useful information in a readable format. This is actually quite tricky for this view. As you can see in listing 4-3, a lot of additional catalog views need to be joined in to get useful results. This listing also includes a filter on the specific indexes that we are interested in. In this case that WHERE clause doesn't have much effect because we have no other columnstore indexes in the demo database. But I wanted to show how you can easily focus on specific tables even in a database that has more tables with columnstore indexes.

```
USE ContosoRetailDW;
GO

SELECT      OBJECT_NAME(i.object_id)
            i.name
            i.type_desc
            COALESCE(c.name, '* Internal *')
            p.partition_number,
            s.segment_id,
            s.row_count,
            s.on_disk_size,
            s.min_data_id,
            s.max_data_id
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

ORDER BY    TableName, IndexName,
            s.column_id, p.partition_number,
```

Listing 4-3: Querying segment information

Results | Messages

	TableName	IndexName	IndexType	ColumnName	partition_number	segment_id	row_count	on_disk_size	min_data_id	max_data_id
374	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	10	1048576	1237904	1	2517
375	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	11	1048576	1256792	1	2517
376	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	12	38680	51832	1	2517
377	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	13	896	1312	299	2491
378	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	14	2048	3328	120	2494
379	FactOnlineS...	CCI_FactOnli...	CLUSTER...	ProductKey	1	15	3072	4480	311	2517
380	FactOnlineS...	CCI_FactOnli...	CLUSTER...	PromotionKey	1	0	1048576	8032	1	28
381	FactOnlineS...	CCI_FactOnli...	CLUSTER...	PromotionKey	1	1	1048576	1872	1	28
382	FactOnlineS...	CCI_FactOnli...	CLUSTER...	PromotionKey	1	2	1048576	1720	1	28
383	FactOnlineS...	CCI_FactOnli...	CLUSTER...	PromotionKey	1	3	1048576	1096	1	28

Zoom in | Open in new window

Figure 4-2: Segment information

Remember that when the index was built, the data was first divided into rowgroups of about a million rows each. On my SQL Server 2014 system, the nonclustered columnstore index uses 15 rowgroups and the clustered columnstore index is 16 rowgroups (results may vary on your system). These rowgroups were then further divided into segments for each of the 21 columns. Which is why on my system, the full results of listing 4-3 is 651 rows: 15 * 21 for the nonclustered columnstore index, plus 16*21 for the clustered columnstore.

To identify a single segment, you need to know the rowgroup and the column name. Rowgroups are identified by a zero-based counter, which is stored in this view in the badly named column *segment_id*. The *row_count* column should not actually have been in this view, because it stores the number of rows in the rowgroup. So it will be the same for each segment of the rowgroup. However, for monitoring and troubleshooting it can be very convenient to have this number available directly in this view.

The *on_disk_size* column, when used in conjunction with *row_count*, can be used to quickly see how well (or how bad) the data in a segment was compressed. This obviously also depends on the data type, so if you don't know the design of the table you might want to add a few more columns from the *sys.columns* catalog view to the query.

The values in *min_data_id* and *max_data_id* are, if the data type of the column supports it, used for segment elimination (as described in level 2 of this series). If you want to get the best possible performance, you should definitely monitor these values whenever one of your columnstore indexes has been (re)built. The data returned from the query can be hard to interpret, because non-integer columns are represented in an internal format. But you can look at the values for different segments of the same column. For instance, the data in figure 4-2 shows that minimum and maximum data id for the *ProductKey* column are the same for rowgroups 10, 11, and 12 (and, not visible in the screenshot, for rowgroups 0 – 9 as well). In rowgroups 13, 14, and 15, the minimum is higher and the maximum is lower. Regardless of what actual values are represented, it is clear that the interval in these last rowgroups is much smaller. That means that these segments have some potential for segment elimination based on the *ProductKey* column. But these are very small rowgroups so there will be no gain; *all* of the full rowgroups include the full range of *ProductKey* values and will not be able to use segment elimination. In a later level I will show how you can remedy this situation.

There are more columns in the *sys.column_store_segments* view, but they are not really useful for monitoring or troubleshooting. They do expose some of the internal details of how SQL Server encodes the data in the columnstore index, so you can add them to the query if you want to look at these values. A deep coverage of these columns is beyond the scope of this article, so I will only briefly describe them. The *has_nulls* columns is 1 if at least one actual NULL value was encountered in the segment when building the index. If that is the case, then *null_value* represents the magic value that SQL Server has chosen to use to represent NULL values in this segment.

The *encoding_type* is, as the name implies, the type of encoding used in the segment. If you add this column to the query, you will see that different encoding types can be used for different segments of the same column. The possible values in this column and their meaning are as follows:

Value	Type of encoding
1	Value based
2	Dictionary encoding of non-strings
3	Dictionary encoding of strings
4	No encoding

The *base_id* and *magnitude* columns are only relevant when value based encoding is used. They are used to scale down numbers, so that fewer bits are needed to store the individual values. For instance, when all values are multiples of 100, *magnitude* is set to 100 and all values are divided by 100 before being stored. And when all values are between 12345 and 13579, *base_id* is set to 12345 (the actual value might be slightly

less or more) and that number is subtracted from the value, so that values between 0 and 1234 are stored.

The *primary_dictionary_id* and *secondary_dictionary_id* columns apply when dictionary encoding is used. They are references to the global and local dictionaries (originally called primary and secondary dictionaries, the column names have not been changed for backwards compatibility reasons).

Finally, the *version* column is always 1 on both SQL Server 2012 and SQL Server 2014. The description in Books Online suggests that this is included for future use, to enable Microsoft to add alternative storage formats for columnstore segments in a future release.

Some background to the query

When looking at the query in listing 4-3, you might wonder why the *sys.index_columns* and *sys.columns* views are joined with an outer join. The reason for this is that a nonclustered columnstore index will sometimes contain additional columns that were not included in the `CREATE INDEX` statement. If you have not explicitly included all the columns of the table's clustered (row-store) index, SQL Server will still add all of them to the nonclustered columnstore index. Additionally, if the table's clustered index is a non-unique index, SQL Server will have created a hidden uniquefier that you cannot explicitly specify, but will still be added to the columnstore index. And if the table has no clustered index at all, SQL Server will add a hidden column for the RID (or Row ID, which is the location of the row in the heap). Data for each of these columns is included in *sys.column_store_segments*, but not in *sys.index_columns*. The outer join to *sys.index_columns* ensures that these "hidden columns" in *sys.column_store_segments* are still exposed; the join to *sys.indexes* must then also be an outer join because otherwise this data would still be lost.

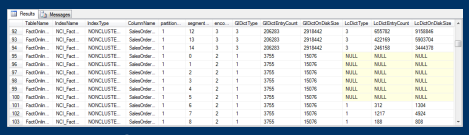
An *encoding_type* of 2 or 3 in *sys.column_store_segments* means that dictionary encoding is used. SQL Server uses a global dictionary or a local dictionary (or both) to store a list of values that can then be replaced by a reference to the dictionary entry in the actual data. You can query the *sys.column_store_dictionaries* view, using the *primary_dictionary_id* and *secondary_dictionary_id* entries found in *sys.column_store_segments*, to retrieve additional information about these dictionaries. One possible way to do so is by using the query shown in listing 4-4.

```
USE ContosoRetailDW;
GO

SELECT      OBJECT_NAME(i.object_id)
            i.name
            i.type_desc
            COALESCE(c.name, '* Internal *')
            p.partition_number,
            s.segment_id,
            s.encoding_type,
            dG.type
```

```
            dG.entry_count
            dG.on_disk_size
            dL.type
            dL.entry_count
            dL.on_disk_size
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
LEFT  JOIN  sys.column_store_dictionaries
      ON    dG.hobt_id
      AND   dG.column_id
      AND   dG.dictionary_id
LEFT  JOIN  sys.column_store_dictionaries
      ON    dL.hobt_id
      AND   dL.column_id
      AND   dL.dictionary_id
WHERE       i.name
```

Listing 4-4: Querying dictionary information



	TableName	IndexName	IndexType	ColumnName	partition_...	segment_...	enco...	GDictType	GDictEntryCount	GDictOnDiskSize	LcDictType	LcDictEntryCount	LcDictOnDiskSize
92	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	12	3	3	206283	2918442	3	655752	9155846
93	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	13	3	3	206283	2918442	3	422169	5903704
94	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	14	3	3	206283	2918442	3	248158	3444378
95	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	0	2	1	3755	15076	NULL	NULL	NULL
96	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	1	2	1	3755	15076	NULL	NULL	NULL
87	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	2	2	1	3755	15076	NULL	NULL	NULL
88	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	3	2	1	3755	15076	NULL	NULL	NULL
89	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	4	2	1	3755	15076	NULL	NULL	NULL
100	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	5	2	1	3755	15076	NULL	NULL	NULL
101	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	6	2	1	3755	15076	1	312	1304
102	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	7	2	1	3755	15076	1	1217	4824
103	FastOrder...	NCI_Fact...	NONCLUSTE...	SalesOrder...	1	8	2	1	3755	15076	1	188	808

Zoom in | Open in new window

Figure 4-3: Dictionary information

The query in listing 4-4 adds two joins to the logic of listing 4-3, to add information about both the global dictionary and the local dictionary. Both joins are outer joins, because segments that use dictionary encoding may use only the global dictionary, only a local dictionary, or both. The number of rows returned by this query will never exceed the number of rows returned by listing 4-3, because the additional filter on the *encoding_type* column restricts this query to only segments that use a dictionary. On my system, 567 rows were returned.

Three of the columns in the *sys.column_store_dictionaries* view are exposed in this query. Two of them, *entry_count* and *on_disc_size*, have self-descriptive names. You can divide them to find out the length of the average entry in that particular dictionary. The last one, *type*, tells you what type of values are stored in the dictionary: this column shows 1 for a dictionary of integer values, 3 for a dictionary of string values, or 4 for a dictionary of floating point values.

There are a few other columns in *sys.column_store_dictionaries* as well, that I did not include in the query. The *version* column is documented as the "version of the dictionary format": the valid values and their meaning are not documented but I have never observed a value other than 1 so I guess that this column is included for future plans. The *last_id* column contains the last