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

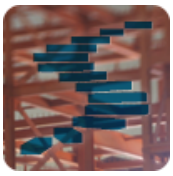
Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

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

Improve your SQL Server knowledge daily with more articles by email.
your@email.com
Sign up

Stairway to Columnstore Indexes Level 3: Building The Columnstore



By Hugo Kornelis, 2018/02/28 (first published: 2015/06/10)

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 levels of this Stairway we have looked at the performance benefits that can be achieved when reading data from a columnstore index. It is a well-known fact that every benefit comes at a cost; in the case of columnstore indexes, the benefit is the increased performance when reading data from the index; the cost is the expensive process required to build the index.

A second down-side to columnstore indexes is that their structure does not lend itself to updates. For this reason, a nonclustered columnstore index, introduced in SQL Server 2012, is read-only – and by extension has the effect of making the underlying table read-only. The clustered columnstore index that was introduced in SQL Server 2014 does allow updates to the underlying data, but this feature should be handled with care. In one of the later levels, we will cover how to use this feature without shooting yourself in the foot.

Because nonclustered columnstore indexes are read only, you will have to disable or remove them before you can load new data, and then recreate or rebuild afterwards. (For partitioned tables, it is usually possible to do this for a single partition only, using partition switching). For clustered columnstore indexes, the process of (re)building the index is often not directly visible, but it happens too. In order to understand the impact this may have on your server, you need to understand the steps SQL Server takes when building (or rebuilding) a columnstore index.

Related Articles

BLOG Columnstore indexes and memory
2012/05/23
Columnstore indexes are built and processed completely in memory. You will receive an out-of-memory...

BLOG Columnstore Indexs
2012/03/16
Columnstore Indexes for Fast DW The SQL Server 11.0 release (2012) introduces a new data warehouse...

ARTICLE

BLOG

BLOG What is a Non-Clustered Columnstore Index?
2014/04/29
First introduced in SQL Server 2012, the Columnstore index is a new in-memory feature that allows fo...

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
columnstore index
stairway series

Write For Us
Find out more

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

The most obvious way to tell SQL Server to build a columnstore index is to use the `CREATE INDEX` statement. There are also a few other statements and conditions that can cause SQL Server to build or rebuild a columnstore for all or some of the data in a table; these statements will be covered in a later level. The internal process of building the columnstore is basically the same regardless of how the process is initiated.

The syntax for the `CREATE INDEX` statement depends on whether you want to create a clustered or a nonclustered columnstore index, but is pretty simple and straightforward in either case. Figures 3-1 and 3-2 show the syntax for creating nonclustered and clustered columnstore indexes.

```
CREATE [ NONCLUSTERED ] COLUMNSTORE INDEX In
ON [ [ DatabaseName. ] SchemaName. ] TableNa
 ( ColumnName
        , ColumnName
 , ...)
[ WITH ([ DROP_EXISTING = { ON | OFF } ], [
[ ON { FileGroup | PartitionScheme(ColumnNar
```

Figure 3-1: Simplified syntax for creating a nonclustered columnstore index

```
CREATE CLUSTERED COLUMNSTORE INDEX IndexName
ON schema.TableName
[ WITH ([ DROP_EXISTING = { ON | OFF } ], [
[ ON { FileGroup | PartitionScheme(ColumnNar
```

Figure 3-2: Simplified syntax for creating a clustered columnstore index

As you can see from the syntax diagrams, the syntax for the two is very similar. The main difference, apart from the (optional) keyword `NONCLUSTERED` versus the mandatory keyword `CLUSTERED`, is that no column list can be specified for a clustered columnstore index, whereas a nonclustered columnstore index does require the columns to be listed. But a sort order (`ASC` or `DESC`) cannot be specified, since columnstore indexes do not store the data in ordered fashion. The `ON` clause is only required if you want to allocate the columnstore index on a non-default filegroup or if you want it to be partitioned.

In SQL Server 2012, supported data types in a nonclustered columnstore index are limited to `char`, `nchar`, `varchar`, and `nvarchar` (with a maximum length of 8000 bytes); `decimal` and `numeric` (with a maximum precision of 18 digits); `bit`, `tinyint`, `smallint`, `int`, and `bigint`; `float`, `real`, `money`, and `smallmoney`; and all date/time datatypes except `datetimeoffset` with a scale of 3 or more. Columns may be nullable, but they may be neither computed, nor `SPARSE`. These restrictions may appear to be very limiting, but in reality, most databases that are designed for the type of BI workload that columnstore indexes are intended for will already only include the supported data types in their large fact tables. As a rule of thumb, when creating a nonclustered columnstore index, best practice is to include all the columns in the table except those that are not supported. Remember that unlike for rowstore indexes, additional columns in a columnstore index will not reduce query

performance. There is additional cost when building the columnstore index, but you end up with an index that can be used by more queries.

In SQL Server 2014, the supported data types for both clustered and nonclustered columnstore indexes has been extended to include all data types except `varchar(max)`, `nvarchar(max)`, `varbinary(max)`, `text`, `ntext`, `image`, `rowversion` (aka `timestamp`), `sql_variant`, `hierarchyid`, `geometry`, `geography`, `xml`, `cursor`, and all user-defined CLR data types. If a table includes any columns that are not supported for a columnstore index, the index cannot be created. You can still create a nonclustered columnstore index on only the columns that are supported, but a clustered columnstore index is not possible because this index type always includes all columns in the table.

When SQL Server creates a columnstore index (either clustered or nonclustered), it will kick off one or more threads. Each of these threads will read an entire rowgroup (up to a million rows) into memory, then sort those rows in order to maximize compression benefits. After that, the rowgroup is further divided into segments: one segment per column, so each segments now contains up to a million values from a single column. SQL Server then determines the best compression method for each segment, performs the actual compression, and then stores the compressed data in the database file, on special LOB pages (the same type of storage that is also used to store `varchar(max)`, `nvarchar(max)`, and `varbinary(max)` data).

During this process, SQL Server will also track, for each column, the minimum and maximum value in each segment. These will be stored in the metadata for the columnstore index, where they can be used for segment elimination. We will take a closer look at this and other metadata in a later level.

In SQL Server 2012, the global (primary) dictionary is formed while the index is built. While that makes the build process relatively efficient, it also leads to a global dictionary that is optimal for the first segments, but may be less useful for the complete table, especially if the data encountered in the first segments is not representative. For example, when building a columnstore index for the sales table of a clothing shop, the first rowgroups processed might all be from spring and summer. SQL Server will then fill the global dictionary with typical summer clothing, because they are common in these rowgroups. When then later the process progresses to the autumn and winter data, the global dictionary might already be full, and all winter items can only be placed in the local dictionaries.

To help the global dictionary be a better reflection of the data in the entire table, the build process has been changed in SQL Server 2014. It now first kicks off on a single thread that reads a sampled selection of data pages from the entire table in order to form a global dictionary for each column; after that the second phase starts to use all available threads to build the actual columnstore index. The execution plans in figures 2-3 and 2-4 clearly show these differences. Both execution plans were generated by running the code in listing 3-1 below, on different versions of SQL Server. (Note that these plans require generating an 'actual' execution by by running the query. An 'estimated' execution plan will not give this level of detail.) The plan for SQL

Server 2012 has only a single branch that simply builds the index; the plan for SQL Server 2014 has two branches, a serial section on top for building the global dictionary and below that a parallel section for actually building all the segments.

Zoom in | Open in new window

Figure 3-3: Building a columnstore index in SQL Server 2012

Zoom in | Open in new window

Figure 3-4: Building a columnstore index in SQL Server 2014

The number of rows that SQL Server 2014 will sample when building the global dictionary depends on the total number of rows in the table ("cardinality"), as shown in table 3-1. Note that the number of rows sampled is an approximation; the exact number of rows sampled might vary slightly. Also note that for table sizes up to a million rows, where the full table is sampled, the execution plan will still have the same two branches, to first build the dictionary and then build the actual index. The second branch will in this case not use parallelism. In effect, the entire table is scanned twice for these small tables.

Cardinality	Rows Sampled
< 1 million rows	All rows
1 million – 100 million rows	1 million rows
> 100 million rows	1% of the rows in the table

Table 3-1: Number of rows sampled to build the global dictionary

The process to build a columnstore index requires a hefty amount of memory. As a ballpark figure, the amount of required memory in megabytes can be estimated as *[(4.2 * IC) + 68] * T + (ISC * 34)*, where *IC* is the number of indexed columns, *T* is the number of threads, and *ISC* is the number of indexed string columns. The number of rows in the table is not a factor in this formula, because each thread processes one rowgroup at a time, and then reuses the same memory to process the next rowgroup. The only tools we have for reducing the memory footprint of this process is to reduce the number of columns in a nonclustered columnstore index, or to reduce the number of threads by forcing SQL Server to reduce the degree of parallelism. The latter is the recommended option; this can be done by adding a `MAXDOP` hint to the `CREATE INDEX` statement, or by using Resource Governor to ensure that the index creation runs in a workload group with a reduced `MAX_DOP` setting. You can also use Resource Governor to directly limit the available memory. Before you do so, make sure you understand how SQL Server will respond to memory shortage while creating a columnstore index – and be aware that this response has changed dramatically from SQL Server 2012 to SQL Server 2014, as explained below.

Using the Resource Governor

The easiest way to cap degree of parallelism and memory usage through Resource Governor is to change the settings

of the default workload group. You can do this through the graphical interface of SQL Server Management Studio (found under the "Management" tab in the Object Explorer) and then use the Script button on the Properties window to save a T-SQL script to make the changes when needed. Note that this will typically affect all activity on a server, so make sure to change the settings only just before you need it, and revert back to the original settings directly after.

There are also alternative ways of allowing additional resource for the creation of columnstore indexes . Memory can also be capped at the resource pool level, and you could even create a dedicated resource pool and/or workload group for this activity. An extensive treatment of Resource Governer is beyond the scope of this Starway. An excellent article can be found here: https://www.simple-talk.com/sql/learn-sql-server/resource-governor/.

As an example, let's return to the nonclustered columnstore we created in level 1. The number of columns included in the index is 21. One of these columns (SalesOrderNumber) uses a string datatype (`nvarchar(20)`); all others are `int`, `money`, or `datetime`. My laptop has two quad-code CPUs, so a maximum of 8 threads was available when this executed. When I put these numbers in the formula above, I calculate an estimated memory requirement of [(4.2 * 21) + 68] * 8 + (1 * 34) = 1283.6 MB. If I do not want the index creation process to consume that much memory, I can reduce it by almost 50% by adding a `MAXDOP` hint that limits SQL Server to use only four cores, as shown in listing 3-1. This does not affect the 34 MB for the string column, but all other memory usage will now be cut in half; the effective total now is [(4.2 * 21) + 68] * 4 + (1 * 34) = 658.8 MB. The downside is that the index creation process now will take more time -- so I use less memory, but use it for a longer time. I can tweak the `MAXDOP` value further up and down to find the perfect balance for my system between memory use and execution time.

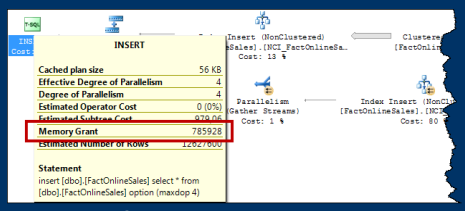
```
USE ContosoRetailDW;

CREATE NONCLUSTERED COLUMNSTORE INDEX NCI_Fa
ON dbo.FactOnlineSales
   (OnlineSalesKey,
    DateKey,
    StoreKey,
    ProductKey,
    PromotionKey,
    CurrencyKey,
    CustomerKey,
    SalesOrderNumber,
    SalesOrderLineNumber,
    SalesQuantity,
    SalesAmount,
    ReturnQuantity,
    ReturnAmount,
    DiscountQuantity,
    DiscountAmount,
    TotalCost,
    UnitCost,
    UnitPrice,
    ETLLoadID,
```

```
    LoadDate,
    UpdateDate)
WITH (MAXDOP = 4);
```

Listing 3-1: Creating the nonclustered columnstore index

Note that the memory usage given by the formula above is an approximation. The exact memory is determined at run-time. The easiest way to check the memory used is by looking at the *Memory Grant* property in the actual execution plan after running the `CREATE INDEX` statement, as shown in figure 3-5. As you can see, the actual memory reserved **for the entire plan** was 785,928 KB.



🔍 **Zoom in | Open in new window**

Figure 3-5: Checking the memory grant

It is also possible to look at the memory grant while the query is running by querying the *sys.dm_exec_query_memory_grants* DMV. It is unfortunately not possible to see the exact memory grant before the query starts. Most queries expose their *Memory Grant* in the estimated execution plan as well, but `CREATE INDEX` queries are an exception.

If during the build process the system runs out of memory, the build process will fail in SQL Server 2012. In SQL Server 2014 the build process will automatically reduce the size of the rowgroups until sufficient memory is available for all threads. The result is that the columnstore index will end up having more rowgroups and segments, but each will be smaller in size.

There may be some cases where this can actually be beneficial. Having more and smaller segments can in some cases increase the granularity of segment elimination, reducing the overall amount of data read further than would have been the case with larger segments. But these would be the exception, not the rule – the architecture of columnstore indexes is optimized for a segment size of one million rows, so while the improved segment elimination may reduce the amount of data read, you run the risk of finding that the processing speed of the data is still slower than it would have been on full-sized rowgroups.

If you think that your data might actually benefit from having smaller rowgroups, you can force this scaling down of rowgroups by using the Resource Governor, as described above. It is not possible to predict the exact rowgroup size that will result, and you will find that the size changes between executions. My recommendation is to only do this in a production system after running extensive tests that show beyond doubt a significant overall performance increase for all processes in the application, and repeat these tests after every service pack or major version upgrade and after every hardware upgrade.

Creating a columnstore index requires a significant amount of resources. It requires sufficient memory to store all the data for an entire rowgroup for each thread used in the process, plus additional memory for the global dictionaries for each string column. Reducing the degree of parallelism is the recommended way to reduce the memory footprint.

On SQL Server 2012, insufficient memory will cause the index creation process to fail. On SQL Server 2014, the process will continue, but using smaller rowgroups. As a rule of thumb, it is best to avoid this condition.

The discussion here has focused on manually creating a columnstore index, but the same process is used (and the same caveats apply) to various other situations and statements that cause a columnstore index to be fully or partially (re)built.

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!

Improve your SQL Server knowledge daily with more articles by email.

your@email.com

Sign up

Thank this author by sharing: 0

Rate this | Join the discussion | Add to briefcase

Total article views: 5108 | Views in the last 30 days: 147