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

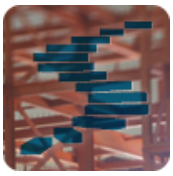
Write For Us
Find out more

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: 3
Rate this | Join the discussion | Add to briefcase

Stairway to Columnstore Indexes Level 13: Improvements in SQL Server 2016



By Hugo Kornelis, 2017/06/07

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.

This stairway series was started in 2015. As such, the focus was on SQL Server 2012 and SQL Server 2014 only. When SQL Server 2016 was released, with lots of improvements in the columnstore technology, I decided to finish the planned levels with the original focus on SQL Server 2012 and 2014, and add one extra level with a brief overview of the improvements available in SQL Server 2016.

The amount of change is huge: more than enough for a full stairway series of its own. So in this level I can only touch on the many changes, to give you a glimpse of what to expect. If you currently work with columnstore indexes on SQL Server 2016, or if you are considering or preparing for an upgrade to SQL Server 2016, then I absolutely recommend reading the excellent series of blog posts that Niko Neugebauer has written, covering columnstore indexes in far more detail and far more depth than in this stairway. It is a long series: currently 104 (!!) posts and still growing, but definitely worth reading. The first 53 posts focus on columnstore indexes in SQL Server 2014; posts 54 and up are dedicated to SQL Server 2016.

For the demos in this level, I will once more use Microsoft's ContosoRetailDW demo database. You can download the .BAK file from http://www.microsoft.com/en-us/download/details.aspx?id=18279. This is a database backup file that you then can restore in your SQL Server 2016 instance if you want to follow along.

Related Articles

ARTICLE **Stairway to ColumnStore Indexes Level 7: Optimizing Nonclustered Columnstore Indexes**
2016/03/11
In this level, we will focus on optimization techniques to apply while building the nonclustered col...

ARTICLE **Stairway to Columnstore Indexes Level 12: Clustered or Nonclustered?**
2017/05/17
T
d
o

BLOG **C**
(
U
C
2
C
th
h

BLOG **x**
i
2
With the release of SQL Server 2012, Microsoft introduced a new type of nonclustered index called xV...

BLOG **SQL Server 2014: Columnstore Index improvements**
2013/07/30
In SQL Server 2012, a new feature was added called Columnstore Indexes that resulted in huge query p...

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.

To prevent backwards compatibility issues, most of the improvements are only available when the database compatibility level is set to 120. Since we want to look at all those new features in this level, we will start by setting the database compatibility level:

```
USE [master];
go

ALTER DATABASE ContosoRetailDW
SET COMPATIBILITY_LEVEL = 130;
```

Listing 13-1: Increase the compatibility level to activate new features

Note that not all improvements depend on the compatibility level; if you do not run the code above then some of the demos in this level will still work, others will fail.

As discussed in the previous level, both clustered and nonclustered columnstore indexes have their pros and cons. But in SQL Server 2014, the nonclustered columnstore index still made the table read-only, and that limitation often is a showstopper for the nonclustered version. In SQL Server 2016, this restriction is now finally lifted. Both the clustered as well as the nonclustered version of the columnstore index now fully support updates to the underlying table.

Another improvement in SQL Server 2016 that may be relevant to some specific workloads is that nonclustered columnstore indexes can now be created with a filter. The ability to define a filter for a traditional rowstore index was introduced a long time ago, in SQL Server 2008; the effect is that only rows matching the filter are included in the index. This reduces the size of the index (saving disk space and speeding up index scans), but of course limits its use to only queries that need only rows included in the index. Typically, filtered indexes are used for tables that contain lots of data that is rarely ever needed in queries but cannot be permanently archived or deleted.

The code below shows these two new features. Assuming that almost all queries are for data in 2008 or newer, and since that subset of the data is large enough to warrant using a columnstore index, we create a nonclustered columnstore that includes data from 2008 only. We then update some rows in this set, to demonstrate that the nonclustered columnstore index no longer makes the data read only.

```
USE ContosoRetailDW;
go

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)
WHERE  DateKey >= '20080101';

UPDATE dbo.FactOnlineSales
SET    ReturnQuantity = 1,
       ReturnAmount = 268.5
WHERE  DateKey = '20090408';
```

Listing 13-2: Nonclustered columnstore index is now updatable and can be filtered

With these improvements, it is now fair to say that the nonclustered columnstore index is finally ready for prime time. It was a great feature since its introduction in SQL Server 2012 but the read only limitation always required workarounds; now you can finally choose this type of index without having to work around the limitations. But be aware: all the caveats that apply to updating the clustered columnstore index since SQL Server 2014 apply to the now updatable nonclustered columnstore index as well.

The ability to filter nonclustered columnstore indexes enables some use cases that were previously impossible to implement. For instance, if you know that recent data (e.g. orders placed in the last month) is subject to lots of changes while older data tends to be stable, you can choose to create a filtered nonclustered columnstore index on just the older data, while using filtered rowstore indexes on the recent data. This does require manual work to periodically rebuild the indexes with a more recent threshold in the filter.

Where, before SQL Server 2016, the nonclustered columnstore index was crippled by not being updatable, the clustered columnstore had other problems. Its lack of support for constraints meant that every update done to the data came with the risk of entering inconsistent data into the database. Not being able to create additional nonclustered rowstore indexes meant that the increased performance for data warehouse style queries came at the price of decreased performance for lookups and other OLTP-style work.

As you can see when you run the code below, these limitations have now been lifted. The clustered columnstore index now can easily be combined with constraints, and additional nonclustered (rowstore) indexes can be created to get better performance for very selective queries.

```
USE ContosoRetailDW;
go

-- Drop nonclustered columnstore created in
DROP INDEX NCI_FactOnlineSales
ON dbo.FactOnlineSales;

-- Drop existing clustered index (and the co
ALTER TABLE dbo.FactOnlineSales
DROP CONSTRAINT PK_FactOnlineSales_SalesKey

-- Create a clustered columnstore index
CREATE CLUSTERED COLUMNSTORE INDEX CCI_FactO
ON dbo.FactOnlineSales;

-- Recreate primary key (now using a nonclus
ALTER TABLE dbo.FactOnlineSales
ADD CONSTRAINT PK_FactOnlineSales_SalesKey
        PRIMARY KEY NONCLUSTERED (OnlineSale

-- Create an extra nonclustered index
CREATE NONCLUSTERED INDEX ix_FactOnlineSales
ON dbo.FactOnlineSales(ProductKey);

-- Show that foreign keys are still there, a
SELECT [name],
       OBJECT_NAME(referenced_object_id) AS
       is_disabled
FROM   sys.foreign_keys
WHERE  parent_object_id = OBJECT_ID('dbo.Fac
```

Listing 13-3: Clustered columnstore indexes with constraints and other indexes

As you can see, I had to drop the nonclustered columnstore index before I could create the clustered columnstore index. It is not permitted on any version of SQL Server to have more than one columnstore index on the same table. It is also not permitted to have multiple clustered indexes, regardless of whether they are rowstore or columnstore, so I had to drop the primary key constraint and its supporting clustered index as well. However, after creating the clustered columnstore index I was able to recreate the constraint (supported by a nonclustered rowstore index) and then create an additional nonclustered index. And the final query shows that I did all this without ever having to drop or disable the existing foreign key constraints, as can be seen in the output below:

Results | Messages

	name	Referenced table	is_disabled
1	FK_FactOnlineSales_DimCurrency	DimCurrency	0
2	FK_FactOnlineSales_DimCustomer	DimCustomer	0
3	FK_FactOnlineSales_DimDate	DimDate	0
4	FK_FactOnlineSales_DimProduct	DimProduct	0
5	FK_FactOnlineSales_DimPromotion	DimPromotion	0
6	FK_FactOnlineSales_DimStore	DimStore	0

Zoom in | Open in new window

Figure 13-1: Foreign key constraints on a table with a clustered columnstore index

In SQL Server 2014, the updatability of the clustered columnstore came at a price: no other indexes and no constraints were

possible. Now that SQL Server 2016 removes these limitations, I can say without hesitation that the columnstore feature has now finally reached maturity.

Memory optimized tables (also known by the feature's code name, Hekaton) were first introduced in SQL Server 2014 and, also, hugely improved in SQL Server 2016. The Hekaton feature was introduced in order to give high volume OLTP applications a performance boost comparable to what columnstore indexes did for data warehousing and reporting. All the table data is stored in memory, the amount of logging (which is still on disk) is minimized, all data structures are optimized to benefit from being stored in memory, and all data access uses a fully lock- and latch-free method so that concurrent connections never block each other. (If you want to know more about this feature, I recommend Kalen Delaney's whitepaper).

This feature might seem to be as far removed from columnstore indexes as it gets, but in SQL Server 2016, Microsoft enabled us to create a clustered columnstore index on a memory-optimized table. This columnstore index itself will be memory-optimized as well: it is completely stored in memory, and the data structure is very similar to that of a disk-based columnstore index, with some small modifications to minimize the impact on OLTP performance.

The code in listing 13-4 shows how to create a columnstore index on a memory-optimized table. The first part of the code sets up a special filegroup for memory-optimized tables in the sample database. This is a one-time task only; once the filegroup exists you can create as many memory-optimized tables, with or without a columnstore index, as you wish.

```
USE ContosoRetailDW;
go

-- Create a filegroup for memory-optimized t
ALTER DATABASE ContosoRetailDW
ADD FILEGROUP Contoso_MO
    CONTAINS MEMORY_OPTIMIZED_DATA;

ALTER DATABASE ContosoRetailDW
ADD FILE (NAME = 'Contoso_MemOpt',
          FILENAME = 'F:\DATA\Contoso_MemOpt
    TO FILEGROUP Contoso_MO;
go

CREATE TABLE dbo.MemOptDemo
   (KeyCol int NOT NULL,
    DataCol int NOT NULL,
    CONSTRAINT PK_MemOptDemo
        PRIMARY KEY NONCLUSTERED HASH (KeyCo
        WITH (BUCKET_COUNT = 65536))
WITH (MEMORY_OPTIMIZED = ON);

ALTER TABLE dbo.MemOptDemo
ADD INDEX csix_MemOptDemo CLUSTERED COLUMNST
```

Listing 13-4: Creating a memory-optimized table with a columnstore index

In the code above, I have chosen to create the clustered columnstore index in a separate ALTER TABLE statement. It is also possible to define the index as part of the CREATE TABLE statement, as was required with all indexes on memory-optimized tables in SQL Server 2014.

Note, however, that a memory-optimized table with a clustered columnstore index does not allow any further schema modifications. So if you plan to also create multiple regular memory-optimized indexes on the table, make sure to create the clustered columnstore index last!

All three of the improvements for columnstore indexes mentioned above are targeted towards the same goal, dubbed "real-time operational analytics" by Microsoft marketing. A very short summary of what this term encompasses is a shift away from the classic distinction of OLTP and data warehouse. Traditionally we set up two databases: a purely transactional database optimized for the concurrency and performance needs of an OLTP workload, and a data warehouse optimized for the speed of reporting and analytics. ETL jobs run at set intervals to bring the data in the data warehouse up to date, but immediately after that the data warehouse starts to fall behind until the ETL job runs again.

With real-time operational analytics, you use a new architecture. Just a single database, designed to handle both highly concurrent OLTP workloads and large scale analytics queries without paying the penalty of having two separate database servers, with separate storage, separate license costs, ETL jobs to copy the data, and lag in the reports. The various options we have in SQL Server 2016 to combine columnstore indexes (for the analytical queries) with traditional rowstore and even memory-optimized indexes (for the OLTP workload) facilitate such an architecture.

The assumption here is that the traditional data warehouse got its data from a single SQL Server OLTP database. In more complex scenarios where the data warehouse is fed from a variety of sources, and the ETL jobs include business logic to handle inconsistent or even conflicting versions of the truth from those versions, you will still need the separate data warehouse if only to serve as a single version of the truth for the business reports. But even in that case, you can benefit from some of the improvements to change your ETL feeds from a periodic big job to a continuous near-realtime processing of changes.

There is, of course, no free lunch. Running this type of mixed workload on a single database will always give you lower performance for both OLTP and analytics as compared to having two separate databases. But when you are prepared to pay that price, you are rewarded by some nice benefits as well: reports and dashboards created by analytics always draw from the most current real-time data. Plus, you also save a lot in server, storage, and licensing cost.

Specifically, for nonclustered columnstore indexes, you can exercise control over how much the columnstore index affects OLTP performance by creating a filtered nonclustered columnstore index. Let's say, for example, that you have an order

entry system where each order passes through multiple stages. In the first stages the order data is "hot": it changes frequently and you like to avoid the overhead associated with modifying data in a columnstore index (as described in level 6 of this series). But once the order is final, you do want to have the benefit of a columnstore index. In that case, you could create a nonclustered columnstore index with a filter on the Status column.

There are also scenarios where you do not have a simple filter to separate the hot data out from the rest. In that case, you can achieve a similar benefit by using the new "compression delay" feature. This option forces SQL Server to wait at least the specified time before compressing data in closed rowgroups. So if you know that most of the changes to an order row will take place within an hour after order entry, you could set the compression delay to 60 minutes. Changes made within that period will always operate on an open or closed delta store. If changes are made after that time, then the data might already be in compressed format, resulting in some extra overhead to process the change. But if you chose the compression delay wisely, then this should only happen rarely. The code in listing 13-5 shows how you can set up such a compression delay.

```
USE ContosoRetailDW;
go

-- Drop clustered columnstore created in a p
DROP INDEX CCI_FactOnlineSales
ON dbo.FactOnlineSales;

-- Create a nomclustered columnstore index w
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 (COMPRESSION_DELAY = 60);
```

Listing 13-5: Creating a nonclustered columnstore index with compression delay