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

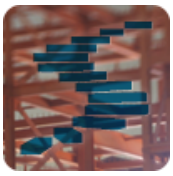
Keep up to date - daily newsletter:
your@email.com
Sign up

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 12: Clustered or Nonclustered?



By Hugo Kornelis, 2017/05/24

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.

The previous levels of this stairway describe details, features, and limitations of columnstore indexes in SQL Server. But they do not answer what should be the first question for every database professional: should columnstore indexes be used in my databases; on what tables should they be used; and should they be clustered or nonclustered columnstore indexes?

With SQL Server 2012, the latter question is not relevant; that version has only nonclustered columnstore indexes, so you can choose to use them or not. The limitations of this type of columnstore index in SQL Server 2012 have made adoption slow: many companies have chosen to forfeit the possible performance and storage benefits because they feel it is not worth the struggle to work around the limitations. In particular, the read-only limitation on tables with columnstore indexes was often seen as a huge problem. However, there are ways to alleviate this pain somewhat, as we will show later in this level.

If you look at the marketing slides surrounding the release of SQL Server 2014, or at the majority of blog posts and conference presentations, you might get the impression that the clustered columnstore index is so much better that it is the only choice, and that the only reason for not removing the nonclustered version from the product is backwards compatibility. That is not the case. There are definitely some cases where, even on SQL Server 2014, I would prefer a nonclustered columnstore index over a clustered one.

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...

FORUM how to create unique nonclustered index on partitioned table without including partition column
2
h
o
p

FORUM h
n
p
i
2
h
o
p

BLOG C
("Loading Data into Non-Updatable Nonclustered Columnstore")
2016/03/07
Continuation from the previous 78 parts, the whole series can be found at http://www.nikoport.com/co...

BLOG Columnstore Indexes – part 117 ("Clustered vs Nonclustered")
2017/12/29
Continuation from the previous 116 parts, the whole series can be found at http://www.nikoport.com/c...

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.

In this level, we will help you choose whether or not to use columnstore indexes, what tables to use them on, and what type to choose, by presenting a short summary overview of the two types of columnstore indexes, focusing on benefits, limitations, and workarounds.

The sample database

All sample code in this level uses Microsoft's *ContosoRetailDW* sample database and builds upon the code samples from the previous levels. If you didn't follow this stairway from the start, or if you did other tests in that database and are now concerned that this might impact the code in this level, you can easily rebuild the sample database. First, download the Contoso BI Demo Database from https://www.microsoft.com/en-us/download/details.aspx?id=18279, choosing the *ContosoBIdemoBAK.exe* option that contains a backup file. After that, download the scripts attached to this article and execute the one appropriate for your system (either SQL Server 2012 or SQL Server 2014. If you are running SQL Server 2016, I suggest using the 2014 version of this script; keep in mind however that there have been significant changes in this newer version so many of the demo scripts in this stairway series will not work the same on SQL Server 2016. We will cover the changes in SQL Server 2016 in a later level). Do not forget to change the RESTORE DATABASE statement at the start, set the correct location of the downloaded backup file, and set the location of the database files to locations that are appropriate for your system.

Once the script has finished, you will have a *ContosoRetailDW* database in exactly the same state as when you had executed all scripts from all previous levels. (The only exceptions might be due to small variations in the index creation process that are impossible to avoid).

A nonclustered columnstore index in SQL Server 2012 makes the underlying table effectively read-only. In SQL Server 2014, the clustered columnstore index does not have this effect, but the nonclustered columnstore index still does. This can be a major blocker in many situations. But let's not forget that the primary use case for columnstore indexes is for large tables in data warehouses, and most data warehouses use partitioning on their large fact tables in order to facilitate an easier data load process. Exactly this partitioning makes it a lot easier to use nonclustered columnstore indexes in data warehouses. And, although with more effort, partitioning can also be used to work around the read-only limitation in other situations.

The basic idea of this strategy is to order the fact table by, for instance, the order entry date, or another generally increasing column (e.g. an IDENTITY key column). New data is collected in a staging table that has the same structure and the same rowstore indexes as the fact table, but not the columnstore index – so this table can still be changed. Once all data is collected, the nonclustered columnstore index is added to the staging table; this will take some time to complete, but far less then rebuilding the columnstore index on the entire fact table would. After that, a technique called "partition switching" (explained in more detail below) is used to swap the data in the staging table with an

empty partition in the fact table. This operation, which takes just a fraction of a second, effectively adds all data to the fact table and empties the staging table – which is now ready for the next iteration of the loading process.

There are basically two strategies to employ table partitioning as a workaround for the read-only limitation of nonclustered columnstore indexes:

1. Align the partition ranges to the data load frequency. This means that if the ETL adds data to the data warehouse daily, you need a partition for each day; if you want just one partition per month you can load only once per month.The drawback of this is that you need to choose between infrequent data loads, or lots of partitions. This can make the individual partitions so small that they cannot fully benefit from columnstore indexes.
2. Do not align the data load frequency to the partition ranges. So you can, for instance, load data every night, but still use monthly partitions.In order to implement this option, the load process has to be more complex. You start by swapping the most recent partition with an empty staging table, so that the data that was loaded before is in the staging table; then drop the nonclustered columnstore index on that staging table to make it updateable. After that you run the ETL process (adding new rows, and optionally even deleting or updating rows); then once more create the nonclustered columnstore index and do another partition swap to move the now updated data back in. You will also need an additional conditional branch in the ETL jobs to handle the less often occurring creation of a new partition.

In the description above, I mention a staging table. This is how you would typically use this technique in a pure data warehouse scenario. However, if you want to employ a nonclustered columnstore index on a very large table in an OLTP system, you can replace all mentions of the work "staging table" with "working table for current data", and still use these techniques. A lot of operations in an OLTP system only need to access the most recent data, which would always be in the "working table for current data". Reports and searches across all data have to combine the data in the working table with the data in the larger table that holds the older information and has a columnstore index. You might be tempted to create a single view that combines the two tables with a UNION ALL expression and then use that view in your reports and search queries; but beware that this prevents batch mode execution on SQL Server 2012. In level 10 you will find the (unfortunately not so simple) workaround to this issue.

Let's first set up a version of our sample table that is partitioned. The sample database used in this stairway is actually much too small for this, so the end result will be very unrealistic. The partitions span three months each, way more than I would want for a real-world situation; and the number of rows per partition is still so small that we will get limited performance benefits. But it will at least work to illustrate the technique of incremental data loading by partition swapping. Running the code in listing 12-1 will create a partitioned version of the table, load it with a copy of the existing data, and create a nonclustered columnstore index on it. It also creates the staging table that we will use to show the data loading process.

```
USE ContosoRetailDW;
GO

-- Create the partition function for the lar
-- For this demo, partitions are three month
CREATE PARTITION FUNCTION pfDateKey (datetir
AS RANGE RIGHT
   FOR VALUES('20070101', '20070401', '2007(
              '20080101', '20080401', '2008(
              '20090101', '20090401', '2009(
              '20100101');

-- Create the partition scheme
-- (Partition swapping requires all partitic
CREATE PARTITION SCHEME psDateKey
AS PARTITION pfDateKey ALL TO ([PRIMARY]);

-- Create a new version of the FactOnlineSal
CREATE TABLE dbo.FactOnlineSales_Partitioned
   (OnlineSalesKey       int          NOT NU
    DateKey              datetime     NOT NU
    StoreKey             int          NOT NU
    ProductKey           int          NOT NU
    PromotionKey         int          NOT NU
    CurrencyKey          int          NOT NU
    CustomerKey          int          NOT NU
    SalesOrderNumber     nvarchar(20) NOT NU
    SalesOrderLineNumber int          NULL,
    SalesQuantity        int          NOT NU
    SalesAmount          money        NOT NU
    ReturnQuantity       int          NOT NU
    ReturnAmount         money        NULL,
    DiscountQuantity     int          NULL,
    DiscountAmount       money        NULL,
    TotalCost            money        NOT NU
    UnitCost             money        NULL,
    UnitPrice            money        NULL,
    ETLLoadID            int          NULL,
```

Listing 12-1: Create and fill a partitioned version of the sample table

Note that this script takes a long time to run, depending on your hardware it could be from five minutes to fifteen minutes or even more. It will also cause the demo database to grow to approximately 4 GB for the data file and 5.5 GB for the log file. You then have a partitioned table with the same data as the original FactOnlineSales table. It has a total of fourteen partitions; both the first and last are empty. Keeping at least one empty partition at each side of a partitioned table is a good idea, because this allows you to quickly create new partitions by splitting an empty partition.

The indexes have been modified to include the partitioning column, which is a requirement for partition switching. Because of this, the original primary key on the IDENTITY column can no longer be enforced and has been left out. (Frankly, if I had designed the table I would not have included an IDENTITY

column at all; in a data warehouse with a star schema they add no value to the fact tables.)

With the partitioned table and the staging table in place, we can now run through a demo script to see how a typical ETL process might load new data into the staging table and then swap it into the full table. This is demonstrated in listing 12.2:

```
USE ContosoRetailDW;
GO

-- First, disable the columnstore index on t
ALTER INDEX NCI_FactOnlineSales_Staging
    ON dbo.FactOnlineSales_Staging DISABLE;
go

-- ===> START of simulation of the ETL job
-- (Simulated ETL: by using copy from Q1 of
INSERT INTO dbo.FactOnlineSales_Staging
SELECT      DATEADD(year, 1, DateKey), Store
            CurrencyKey, CustomerKey, SalesO
            SalesQuantity, SalesAmount, Retu
            DiscountQuantity, DiscountAmount
            ETLLoadID, LoadDate, UpdateDate
FROM        dbo.FactOnlineSales_Partitioned
WHERE       DateKey >= '20090101'
AND         DateKey < '20090401';
-- ===> END of simulation of the ETL job
GO

-- Rebuild the columnstore index on the now
ALTER INDEX NCI_FactOnlineSales_Staging
    ON dbo.FactOnlineSales_Staging REBUILD;
GO

-- Add a constraint to ensure we only have d
ALTER TABLE dbo.FactOnlineSales_Staging
ADD CONSTRAINT CK_Correct_Partition
        CHECK (DateKey >= '20100101' AND Dat

-- Split the empty tail-end partition into a
-- (receives data), and a new empty tail-end
ALTER PARTITION SCHEME psDateKey
    NEXT USED [PRIMARY];
ALTER PARTITION FUNCTION pfDateKey()
    SPLIT RANGE ('20100401');
```

Listing 12.2: Using partition swapping to work around the read-only limitation

Only two steps in listing 12.2 take time. The simulated ETL process takes the same time it would take in a regular rowstore scenario, and rebuilding the columnstore index on the staging table takes less time that it would to build or rebuild it on the entire table. The actual partition swapping is instantaneous, because SQL Server does this by merely swapping some pointers instead of moving all the data around.

Note that in this example the `CHECK` constraint that is needed to ensure that all data in the staging table is in the correct date

range was added after the data was loaded and the columnstore index was already built. For a data warehouse with data load from a trusted source, this is the fastest way to handle this. However, if you use the staging table as a working table for current data in an OLTP scenario, the CHECK constraint should always be in place to prevent incorrect data from being stored in this table.

If you have a scenario that allows you to do your ETL as illustrated above, then you might find that a nonclustered columnstore index can be interesting on SQL Server 2012 despite the read-only limitation. And even on SQL Server 2014, this ETL trick might convince you to choose the nonclustered columnstore index over the clustered version.

Another important consideration to keep in mind when choosing between a clustered or a nonclustered columnstore index on SQL Server 2014 is the long list of restrictions on what columns can be allowed in columnstore indexes. For instance, computed columns or columns defined with the SPARSE attribute are not permitted in a columnstore index, and several data types are not allowed either.

Since a clustered columnstore index always includes all columns in the table, any column that is not permitted makes it impossible to create a clustered columnstore index on that table; however, you can still create a nonclustered columnstore index as long as you restrict the list of columns in the index to only those with a permitted data type. The code in listing 12-3 illustrates this difference:

```
USE ContosoRetailDW;
GO

CREATE TABLE dbo.TestDatatypes
   (KeyCol int NOT NULL,
    DataCol_Supported varchar(500) NOT NULL,
    DataCol_Unsupported varchar(max) NOT NUL
GO

-- Fails because of unsupported data type
CREATE CLUSTERED COLUMNSTORE INDEX ccix_Test
    ON dbo.TestDataTypes;
GO

-- Fails because of unsupported data type
CREATE NONCLUSTERED COLUMNSTORE INDEX ncix_T
    ON dbo.TestDataTypes(KeyCol, DataCol_Sup
GO

-- Succeeeds because unsupported column is n
CREATE NONCLUSTERED COLUMNSTORE INDEX ncix_T
    ON dbo.TestDataTypes(KeyCol, DataCol_Sup
GO

DROP TABLE dbo.TestDataTypes;
GO
```

Listing 12-3: Including only supported columns in a nonclustered columnstore index

If you run the code in listing 12-3 statement by statement, you will see that you get an error message when trying to create the clustered columnstore index. The first attempt to create a nonclustered columnstore index also fails, because it includes a column that uses a data type that is not supported for columnstore indexes. However, for the nonclustered columnstore index we at least have the option to omit that column from the column list and successfully create a nonclustered columnstore index on the other columns in the table. With a clustered columnstore index, we do not have that option; so if we really want to create a columnstore index on this table without losing the ability to update data, we will have to change the database design (e.g. by changing the data type, or by moving the offending column out to a separate table).

If you followed along with the examples in level 5 and level 6, you will have realized that changing the data in a clustered columnstore index comes at a price. New data is added into an open deltastore, which is compressed when "full"; the data in it is probably not optimal for rowgroup elimination. Depending on your data load pattern you also run the risk of getting too many rowgroups with too few rows each. Deleting data, but also changing data, logically removes the rows from the compressed rowgroups, but the data is not physically removed. Remember that one of the reasons for the speed of columnstore indexes is the reduced I/O because of the compression – but if a large enough percentage of the compressed data is logically removed, we are actually doing more I/O instead of less to get the same amount of data. For example, let's say you have 100 GB of data, and the columnstore index reduces the storage size to just 15 GB. But if you then delete 90% of your data, the columnstore index will still take up 15 GB, even though the uncompressed data size would now be 10 GB. And because updates are implemented as DELETE followed by INSERT, you would see similar patterns on tables that have lots of UPDATE operations.

In level 8 you saw that a reorganize operation does not address these issues. A rebuild of the index does physically remove the logically deleted data, at the price of using more time and resources. If you also want to optimize for rowgroup elimination, you will have to invest even deeper: you need to replace the clustered columnstore index with a clustered rowstore index, and then replace that once more with the original clustered columnstore index. This effectively means two full index rebuilds in succession (and the second rebuild must be forced to avoid parallelism if you want the best possible optimization).

If you really need the ability to continuously update data in a table and still want the benefits of a columnstore index on that table, then these issues are the price you have to pay. But if that price is too high, you might want to consider using a nonclustered columnstore index as the alternative. If you are able to set up a partitioning scheme that allows you to use the partition switching technique described earlier, you'll never have to reorganize or rebuild the columnstore index; you only create the columnstore index once for each partition. But nothing comes for free, in this case the price is a much clumsier update process, and having to write a more complex query if you also want to include the most recent data without losing batch mode execution.