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

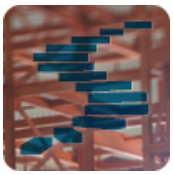
Write For Us
Find out more

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: G+ 0
Rate this | Join the discussion | Add to briefcase

Stairway to Columnstore Indexes Level 1: A First Look At Columnstore Indexes



By Hugo Kornelis, 2016/05/30 (first published: 2015/02/25)

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.

One of the most exciting new features in SQL Server 2012 was the introduction of columnstore indexes, or, as they are officially called, “in-memory columnstore indexes”. In SQL Server 2014, Microsoft made this great feature even better. A lot better!

So what is it all about? Most of the time, columnstore indexes are described as a feature for data warehousing and reporting. And that is indeed the environment where you are most likely to find a good use for this new type of index. But even in OLTP databases, you might have a few reports that are slow, because they draw from very large tables. With proper planning and care, even those reports can benefit from columnstore indexes.

An important keyword in the previous paragraph is “very large”. Columnstore indexes are intended to be used with large tables. There is no hard minimum size, but as a rule of thumb, I would say that you need at least tens of millions of rows in a single table to get real benefit from columnstore indexes.

When talking about columnstore indexes, I am actually talking about a combination of two closely related new features. One is a different architecture for index storage, storing data by column instead of by row. In the next level, we will look at this architecture in more detail, and we will see how this reduces the storage size for large tables. The second new feature is "batch mode", a new mode of query execution which is optimized for modern hardware. Batch mode will be covered later in the series.

Related Articles

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

ARTICLE Columnstore Index Changes in SQL Server 2014
2014/07/21
SQL Server 2012 introduced columnstore indexes, which can immensely improve th

BLOG S In 2 Ir a re

ARTICLE In w 2 S c c

ARTICLE S In Nonclustered Columnstore Indexes
2016/03/11
In this level, we will focus on optimization techniques to apply while building the nonclustered col...

Fresh articles daily:
Get the SQL Server Central newsletter and get a new SQL Server article each day. Get Database Weekly for a roundup of all the biggest SQL news from around the web.
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.

For the demos in this series, I will use the `ContosoRetailDW` demo database, created by Microsoft, that can be downloaded from http://www.microsoft.com/en-us/download/details.aspx?id=18279. This is a 625MB download of a database backup, which restores to a 1.2GB database. This is a bit small for real columnstore use, but large enough to demonstrate the features.

The `ContosoRetailDW` database does not include any columnstore indexes. This is actually not a bad thing. To see the benefit of columnstore indexes, I want to compare performance of the same query with and without a columnstore index. So the first thing to do, before creating a columnstore index, is to test performance of the database as-is. I wrote a sample query (see listing 1) that can be considered typical for a query from an information worker in a BI role. It shows an overview of sales, broken down by quarter, for all products in the "Contoso" brand.

```
WITH ContosoProducts
AS (SELECT *
    FROM   dbo.DimProduct
    WHERE  BrandName                    = 'C
SELECT     cp.ProductName,
           dd.CalendarQuarter,
           COUNT(fos.SalesOrderNumber) AS Nu
           SUM(fos.SalesQuantity)      AS Qu
FROM       dbo.FactOnlineSales         AS fo
INNER JOIN dbo.DimDate                 AS do
      ON   dd.Datekey                   = fo
INNER JOIN ContosoProducts             AS cp
      ON   cp.ProductKey                = fo
GROUP BY   cp.ProductName,
           dd.CalendarQuarter
ORDER BY   cp.ProductName,
           dd.CalendarQuarter;
```

Listing 1: A typical BI query

On my laptop, the query in listing 1 takes, on average, about 6.27 seconds to complete when the required data is already in cache. If the data has to be read from disk, the execution time goes up 8.11 seconds. Since the `FactOnlineSales` table has over 12.5 million rows and this query has to scan the entire clustered index, which is actually not bad at all. But if you spend all day doing these queries, the sluggish response will quickly become a nuisance. And you can probably easily imagine how the performance would be for databases that are 10 times or even 100 times bigger.

Note that these execution times will vary based on the hardware used. If you repeat these tests on a high end server, all queries will run much faster. On a laptop you bought as a bargain three years ago, they will take more time. But you should see big savings after creating a columnstore index on all of those systems.

Columnstore indexes come in two flavors: clustered and nonclustered. There are many similarities between the two, but

also a lot of differences. One such difference is that in SQL Server 2012, only nonclustered columnstore indexes are available. The clustered version has been added in SQL Server 2014. For the first example, we will create a nonclustered columnstore index, so that readers without access to SQL Server 2014 can follow along.

```
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);
```

Listing 2: Creating the nonclustered columnstore index

We will take a closer look at the syntax for creating nonclustered indexes later. Those who are used to minimizing the amount of columns in an index may be shocked to see all the table's columns included in the list above. This, too, will be explained later. For now, please just accept that columnstore indexes have their own rules and guidelines.

Executing this code will take some time (I had to wait approximately 43 seconds for it to complete on my laptop). But this is a one-time action that in a real datawarehouse would typically be done during the nightly load. Once the index is created, it can be used by SQL Server to increase the speed of many queries.

When I now run the code from listing 1 again, I get the same results as before, but the results are returned almost instantaneously. The query now finishes in about 0.34 seconds, more than 18 times faster than before I added the columnstore index. Of course, even with columnstore indexes the query will run more slowly if data has to be read from disk. With an empty cache, the query takes 1.54 seconds, still more than five times faster than the 8.11 seconds we saw before.

> **Tip**
>
> The only way to evaluate how useful a columnstore index is for a specific database is to compare performance with and

without the columnstore index. But you have seen that it takes a lot of time to build a columnstore index, so you may be hesitant to drop the index and then have to recreate it again, just for a test. Luckily, there is a better option: adding the query hint `OPTION(IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX)` to a query will instruct the optimizer to execute the query as if the nonclustered columnstore index does not exist. This hint works for nonclustered columnstore indexes in both SQL Server 2012 and SQL Server 2014. Unfortunately, there is no similar option for clustered columnstore indexes in SQL Server 2014. If you want to do performance tests for the clustered version, dropping and rebuilding the index in between tests is the only way to go unless you want to maintain two tables, one with the clustered columnstore index and one without.

The performance gains from nonclustered columnstore indexes in SQL Server 2012 can be impressive, but you need to be very careful when writing your queries. Almost every query against a table with a columnstore index would find some benefit, but you may have to code around many limitations in order to get the full potential. One such limitation is related to outer joins.

If the information worker who wrote the query in listing 1 decides that all products of the Contoso brand need to be included in the results, even if they were never sold, the obvious change would be to change the `INNER JOIN` to a `RIGHT OUTER JOIN`, as shown in listing 3.

```
WITH ContosoProducts
AS (SELECT *
    FROM   dbo.DimProduct
    WHERE  BrandName                    = '(
SELECT     cp.ProductName,
           dd.CalendarQuarter,
           COUNT(fos.SalesOrderNumber) AS Nu
           SUM(fos.SalesQuantity)      AS Qu
FROM       dbo.FactOnlineSales         AS fo
INNER JOIN dbo.DimDate                 AS do
      ON   dd.Datekey                   = fo
RIGHT JOIN ContosoProducts             AS cp
      ON   cp.ProductKey                = fo
GROUP BY   cp.ProductName,
           dd.CalendarQuarter
ORDER BY   cp.ProductName,
           dd.CalendarQuarter;
```

Listing 3: Introducing an outer join

Without a columnstore index (or with the `IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX` hint to simulate that situation), the query runs in 6.71 seconds when the data is already in cache. Changing an `INNER JOIN` to a `RIGHT [OUTER] JOIN` introduces some extra overhead in the execution plan, causing the query execution to take another 0.44 seconds, an increase of approximately 7%.

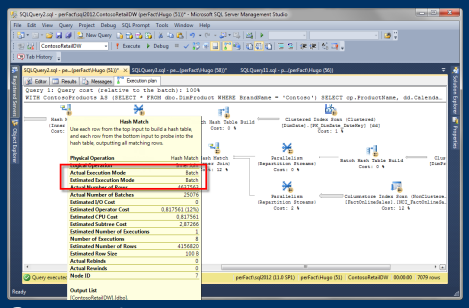
When running this query without a hint the optimizer will immediately pick a plan that uses the nonclustered columnstore

index on my SQL Server 2012 instance, and this results (as expected) in a faster execution – approximately 4.24 seconds. This is of course still better than the 6.71 seconds without the columnstore index, but it pales in comparison with the 0.34 seconds I measured for the original query with an INNER JOIN. So why is the performance impact of changing an inner join to an outer join so much bigger when the query is using a columnstore index?

The reason for the big performance difference between inner and outer joins is that the speed gain of columnstore indexes builds on two components. The obvious one is the index itself. In subsequent levels of this stairway you will see how the structures used to implement columnstore indexes result in a massive reduction of I/O.

But much of the secret sauce of the better performance with columnstore indexes is achieved by using a new mode of query execution. This so-called "batch mode execution" represents a completely different way of executing queries, designed to remove a lot of overhead and optimized to take optimal advantage of the characteristics of modern hardware. But the SQL Server 2012 implementation of batch mode was very limited. Only a handful of operators are available in batch mode in SQL Server 2012. As soon as a query requires the use of one of the unavailable operators, all or most of the query will fall back into the traditional row mode execution. An outer join is but one of many language elements that will cause this fallback to row mode, where you still get the performance benefit that results from the I/O reduction of a columnstore index, but not the additional performance benefit of batch mode.

If the performance benefit you experience is less than you were hoping for, inability to use batch mode may be the cause. The quickest way to verify the mode being used is to bring up the graphical execution plan for the query in SQL Server Management Studio, then check the "Estimated Execution Mode" and "Actual Execution Mode" properties of the operators that process the most data. See Figure 1-1 for an example of a query running in batch mode.



Zoom in | Open in new window

Figure 1-1: Execution plan showing batch mode

Later in this series, we will cover batch mode in more detail, and will discuss all the limitations. We will look at ways to rewrite queries to work around the limitations and maximize the benefit of batch mode execution.

If you have followed along on your own test server with the queries I used for comparing performance, you should have seen similar performance differences – if you are running on SQL Server 2012. But if you try the same on SQL Server 2014, you

will find very different performance effects. A very big improvement that Microsoft made after SQL Server 2012 was released was to add batch mode support for many more operators, so that queries that read data from a columnstore index are far more likely to run in batch mode. The query with the outer join in listing 3 is a great example where you can see this difference – on SQL Server 2012 the outer join will cause it to fall back to row mode execution, but in SQL Server 2014 it will run in batch mode.

What benefit can you expect?

There is no simple way to predict the performance benefit when you create a columnstore index. The only reliable way is to compare the performance of the queries that are typically used on your real database, or on a test database that mimics size, specification, and data distribution of your real database as close as possible.

For queries that are able to run in batch mode, I have seen performance benefits ranging from just five times faster to over seventy (70!) times faster after adding a columnstore index. For queries that run in row mode, the performance benefit is always less, typically ranging from a 50% increase in performance to up to twenty times faster.

The performance benefit available by using columnstore indexes is caused by two major factors. One is the I/O savings caused by the new index structure, the other is batch mode execution. With SQL Server 2012, batch mode execution was unfortunately limited to only a handful of operators, causing many common query constructs to force a query back into row mode execution, forfeiting the performance benefit of batch mode. Most of these limitations have been lifted in SQL Server 2014.

In the rest of this stairway series, I will explain more about column-oriented storage and batch execution mode, I will demonstrate how to create and use columnstore indexes, what the limitations are, and how to work around them; I will discuss the improvements made in SQL Server 2014, including the new clustered columnstore index; and give tips and best practices for getting the most out of columnstore indexes in both versions.

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: 12905 | Views in the last 30 days: 34