SQL ServerCentral.com
SQL Prompt Code analysis as you type
SQLQuery1.sql
Try it now redgate
a community service from redgate
Try SQL Clone

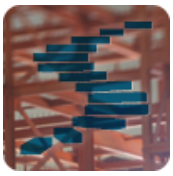
Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

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

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Stairway to Columnstore Indexes Level 10: Optimizing Queries For Batch Mode (Part 1)



By Hugo Kornelis, 2016/11/23

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 Level 9 of this Stairway, I covered batch mode. I explained why batch mode grants an additional speed boost in addition to the already considerable performance improvements achieved by reading from a columnstore index. I also showed a "combined mode" execution plan, where batch mode is used for some operators, and row mode for the rest.

The example I used displayed "good" combined mode. This means that all operators that process large amounts (millions) of rows run in batch mode, and only operators that work on far less data (usually after aggregation) use row mode.

In the next two levels, I will look at batch mode limitations. You will see queries that use only row mode, or that use "bad" combined mode (using row mode operators on huge volumes of data). In each case, I will point out the root cause of the batch mode failure, but I will also show how a rewrite of the query can help SQL Server find a "good" combined mode plan instead.

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-

Related Articles

FORUM Problem while executing Query batch
2009/07/14
Problem while executing Query batch using c# code

ARTICLE Stairway to Columnstore Indexes Level 9: Batch Mode Execution
2016/09/06
In this level, Hugo explains what batch mode execution is, how it differs from row m

FORUM E a 2 d e

FORUM A 2 C

FORUM H 2 H

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

Write For Us
Find out more

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

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 latest 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. (Except for small variations in the index creation process that are impossible to avoid).

Also note that the examples in this level are all intended to demonstrate issues specific to SQL Server 2012. All code will run without error on later versions, but you will not see the same problems and you will not benefit from using the suggested workarounds.

In SQL Server 2012, only a small number of execution plan operators support batch mode; a few more were added in SQL Server 2014 but there are still omissions. As soon as any other operator is needed, execution falls back to row mode. This was a one-way transition in SQL Server 2012. In SQL Server 2014, support was added for "mixed mode plans", where execution can transition between batch mode and row mode multiple times; however SQL Server tries to minimize these transitions because of the overhead involved.

In all these cases, queries will still run faster than they would when using a rowstore index, but they will not give you the full performance benefit. However, it is often possible to work around these limitations by rewriting the query. This usually requires some extra thought and the resulting queries can be harder to maintain, but they will give you better performance.

Which operators support batch mode?

SQL Server 2012 only supports batch mode in a few operators. Apart from (obviously) the Columnstore Index Scan, these are Filter, Compute Scalar, and Hash Match – and the latter only for the logical operations Aggregate and Inner Join. In addition, a special new operator, Batch Hash Table Build, was introduced that is in some cases required to prepare the data before it can be processed by a Hash Match operator.

In SQL Server 2014, the Batch Hash Table Build operator is no longer needed and has been removed. The Hash Match operator now supports batch mode in all its logical join variations (outer joins, semi-joins, and anti-semi joins), as well as in the new "Global Aggregate" logical operation. Also, support for batch mode has been added to the Concatenation operator.

Now, while the list of batch-enabled operators seems rather short, these few operators (especially after the SQL Server 2014 improvements) can actually already make a lot of queries run in batch mode.

For people who live and dream in execution plans, the above lists of operators will be very useful. But if you spend more time in the query editor than in the execution plan display, then you will need to know how these limitations translate to queries. So in the next two levels, I will focus on query patterns that are known to cause issues with batch mode (or other limitations related to columnstore indexes). As mentioned, the patterns demonstrated in this level all apply to SQL Server 2012 only; if you run the same code on newer versions you will not see the same problems. In the next level we will also highlight some patterns that cause issues on SQL Server 2014 as well. (I will describe the improvements in SQL Server 2016 in a later level). I will also show how to work around these limitations.

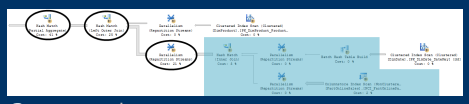
As already mentioned in level 1, outer joins cause a fallback to row mode (in SQL Server 2012 only). Depending on the rest of the query, the result will either use only row mode, or it will use "bad" combined mode. An example of the latter can be seen by running the code in listing 10-1:

```
USE ContosoRetailDW;
go

WITH NonContosoProducts
AS (SELECT *
    FROM   dbo.DimProduct
    WHERE  BrandName                    <> '(
SELECT     ncp.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 NonContosoProducts          AS no
      ON   ncp.ProductKey               = fo
GROUP BY   ncp.ProductName,
           dd.CalendarQuarter
ORDER BY   ncp.ProductName,
           dd.CalendarQuarter;
```

Listing 10-1: Using an outer join

The relevant part of the execution plan for this query is shown in figure 10-1. As in the previous level, the shaded area shows which operators run in batch mode.



Zoom in | Open in new window

Figure 10-1: Outer join causes row mode fallback

The scan of the FactOnlineSales table and the inner join to the DimDate table are executed in batch mode, but then execution

falls back into row mode for the outer join to the non-Contoso products from the *DimProduct* table. The three marked operators, Parallelism (Repartition Streams), Hash Match (Left Outer Join), and Hash Match (Partial Aggregate), each process a huge number of rows – over 12 million for the first two, and almost 8 million for the third. The rest of the plan (not shown in the screenshot) processes fewer than 20.000 rows – also in row mode, but for such a low number of rows that's not an issue.

The easiest way to speed up this query would be to change the outer join back to an inner join, to enable batch mode for the marked operators. But that can change the results of the query, which is of course not acceptable. However, we can change the join type if we also make other changes to compensate, as shown in listing 10-2. The query is more complex, but it runs a lot faster because we now get a "good" combined mode plan:

```
USE ContosoRetailDW;
go

WITH NonContosoProducts
AS (SELECT      *
    FROM        dbo.DimProduct
    WHERE       BrandName                    <
, JoinedAndAggregated
AS (SELECT      ncp.ProductName,
                dd.CalendarQuarter,
                COUNT(fos.SalesOrderNumber) A
                SUM(fos.SalesQuantity)      A
    FROM        dbo.FactOnlineSales         A
    INNER JOIN  dbo.DimDate                 A
          ON    dd.Datekey
    INNER JOIN  NonContosoProducts          A
          ON    ncp.ProductKey
    GROUP BY    ncp.ProductName,
                dd.CalendarQuarter)
SELECT          ncp.ProductName,
                jaa.CalendarQuarter,
                COALESCE(jaa.NumOrders, 0)  A
                jaa.QuantitySold            A
FROM            JoinedAndAggregated         A
RIGHT  JOIN     NonContosoProducts          A
       ON       ncp.ProductName
ORDER BY        ncp.ProductName,
                jaa.CalendarQuarter;
```

Listing 10-2: Rewrite of an outer join to use more batch mode

The trick used here is to consider the similarities and differences between the inner and the outer join. Both joins combine sales data with product data for non-Contoso products; both joins will exclude sales of products that are not in the *NonContosoProducts* list. The only difference is what happens with non-Contoso products that have no sales. The inner join removes these products; the outer join includes row for each of them with `NULL` values for all columns from the other table. In the aggregation that follows, each of these extra rows is a separate product that will become a separate "group" (consisting of that single row) in the aggregation; the *CalendarQuarter* column will

be NULL, a COUNT aggregation will return 0 for COUNT(*columnname*), or 1 for COUNT(*) or COUNT(*constant*).

In listing 10-2, I start with an inner join, so I lose these extra rows. Then, after aggregation, I add another join to the non-Contoso products, this time an outer join, for the sole purpose of reintroducing those lost rows. So instead of first adding one extra row for each unmatched product and then aggregating per product, I now first aggregate per product and then add an extra row for unmatched products. The final results are equal (as long as I take care to do a proper COALESCE for the COUNT results), but now all the joins and most of the aggregation are done in batch mode. The execution plan has become more complicated, but all operators that run in row mode now process a small number of rows.

The explanation above proves that the two queries are logically equivalent; they will always, under any circumstances, produce the same results. That is a key consideration for all the rewrites presented in this and the next level. A rewrite is only acceptable if you can prove without any doubt that it will never change the results. Just testing is not enough; sometimes an erroneous rewrite will only cause differences in some edge cases that might not exist in your test data. However, the proof should not replace testing, it is still possible that you made an oversight in your logic, or that you introduced a bug by a simple typo in the query.

On my laptop, performance improves from 5.8 seconds for the original query to 0.7 seconds after the rewrite. After proving that the rewrite will not change the results and testing that I didn't make an error, I am happy to give this performance improvement to my users. Of course with a long comment in the code to explain to my coworkers why I replaced a simple outer join query with a more complex query, and the proof that these are in fact equivalent.

Another annoying limitation for batch mode execution on SQL Server 2012 occurs when combining result sets using UNION ALL. This may not be a common SQL construct in data warehouse and reporting applications, but it might be used even without the data analyst being aware of it, especially on a SQL Server 2012 installation.

A common method to work around the read-only limitation for columnstore indexes on SQL Server 2012 is to partition the large fact tables by date. Data that is old enough to be stale is stored in the fact table (with a columnstore index). More recent data (that is still changing), is stored in a smaller table with only rowstore indexes. When a time period closes, a columnstore index is built on the smaller table after which partition switching can be used to add this data to the larger table. (See https://social.technet.microsoft.com/wiki/contents/articles/5069.add-data-to-a-table-with-a-columnstore-index-using-partition-switching.aspx for an example of this technique).

The benefit of this technique is that you get the performance benefit of the columnstore index for most of the data, and still have read/write possibilities on the much smaller "recent" fraction of the data. But the down-side is that now the data is spread out over two tables. In order to facilitate easy querying, many DBAs will create a view that combines the data from the two tables.

Data analysts can then query that view instead of having to query both tables and combined the results.

In the Contoso sample database, there is no table that can be easily used to set up a very realistic example of this technique, so I have to resort to the rather contrived example shown in listing 10-3.

```
USE ContosoRetailDW;
go

-- Quick and dirty way to create an empty co
SELECT *
INTO   dbo.FactOnlineSales_RowStore
FROM   dbo.FactOnlineSales
WHERE  1 = 0;
go

-- Add a few rows of sample data
INSERT INTO     dbo.FactOnlineSales_RowStore
SELECT TOP (10) DateKey, StoreKey, ProductKe
                SalesOrderNumber, SalesOrder
                ReturnQuantity, ReturnAmount
                TotalCost, UnitCost, UnitPri
FROM            dbo.FactOnlineSales
ORDER BY        OnlineSalesKey;

-- Create the same indexes as the fact table
ALTER TABLE dbo.FactOnlineSales_RowStore
ADD CONSTRAINT PK_FactOnlineSales_RowStore_S
    PRIMARY KEY NONCLUSTERED (OnlineSalesKey

CREATE CLUSTERED INDEX ix_FactOnlineSales_Ro
ON dbo.FactOnlineSales_RowStore (ProductKey)
go

-- Create a view that combines the two table
CREATE VIEW dbo.FactOnlineSales_Combined
AS
SELECT  *
FROM    dbo.FactOnlineSales
UNION ALL
SELECT  *
FROM    dbo.FactOnlineSales_RowStore;
```

Listing 10-3: Creating a view to combine columnstore and rowstore data

The code in listing 10-3 creates an additional table (*FactOnlineSales_RowStore*) to mimic the table for recent data, and a view (*FactOnlineSales_Combined*) that combines both tables for easy querying. Let's see how this view performs in a typical data warehousing query:

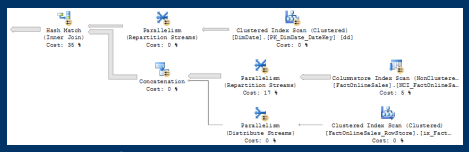
```
USE ContosoRetailDW;
go

SELECT     dp.ProductName,
           dd.CalendarYear,
```

```
            SUM(fos.SalesQuantity)       AS (
            AVG(SalesAmount)             AS A
FROM        dbo.FactOnlineSales_Combined AS f
INNER JOIN  dbo.DimDate                  AS d
      ON    dd.Datekey                     = f
INNER JOIN  dbo.DimProduct               AS d
      ON    dp.ProductKey                  = f
GROUP BY    dp.ProductName,
            dd.CalendarYear
ORDER BY    dp.ProductName,
            dd.CalendarYear;
```

Listing 10-4: Data warehousing query on a view that uses UNION ALL

Because of the view, there is a `UNION ALL` "hidden" in this query. But if a DBA created the view, then the data analyst who writes this query might not be aware of this. They will find out, though, when they run this query. On my laptop, it takes almost five seconds to run, much longer than other queries on the same table. This is because the entire query runs in row mode.



Zoom in | Open in new window

Figure 10-2: Row mode is used for a UNION ALL

Figure 10-2 shows a part of the execution plan. This plan fragment is rather straightforward: the two *FactOnlineSales* tables are read and the data is combined using the *Concatenation* operator (the execution plan's equivalent of a `UNION ALL`). This operator doesn't support batch mode in SQL Server 2012, so the only part of the plan that could in theory be executed in batch mode is the *Columnstore Index Scan* operator. In this case, even that operator is executed in row mode — probably because the optimizer expects the overhead of the mode transition to cost more than the benefit of scanning in batch mode.

In order to work around the batch mode limitation for `UNION ALL`, we have to give up the convenience of using the single view. The same results can be returned much faster on SQL Server 2012 by using this query:

```
USE ContosoRetailDW;
go

WITH ResultsForColumnsStore
AS (SELECT     dp.ProductName,
               dd.CalendarYear,
               SUM(fos.SalesQuantity)
               SUM(fos.SalesAmount)
               COUNT(fos.SalesAmount)
    FROM       dbo.FactOnlineSales
    INNER JOIN dbo.DimDate
          ON   dd.Datekey
    INNER JOIN dbo.DimProduct
          ON   dp.ProductKey
    GROUP BY   dp.ProductName,
               dd.CalendarYear)
```