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

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

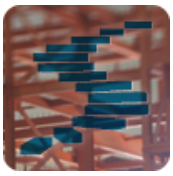
Write For Us
Find out more

redgate | Hub
Insights
Learning
Forums
Ideas
Discover the new Redgate Hub

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

Stairway to Columnstore Indexes Level 11: Optimizing Queries For Batch Mode (Part 2)



By Hugo Kornelis, 2017/01/04

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.

We started our exploration of query constructions that inhibit batch mode execution in Level 10 of this series (which you can read here: http://www.sqlservercentral.com/articles/Stairway+Series/148676) with two examples that affect SQL Server 2012 only: outer joins and `UNION ALL`.

In this level we will look at many other similar issues and their workaround. Some of them affect SQL Server 2012 only, others affect SQL Server 2014 as well – I will point out the affected versions in each case. Note that for all the rewrites presented in this level, just as with those from the last level, you should only use them if you understand why they are guaranteed to never change the results, after proper testing, and with sufficient comments in the code to ensure that anyone looking at the code in the future will understand what's going on.

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.

Related Articles

ARTICLE Aggregate Queries
2007/10/02
They are a basic type of query that every DBA and developer should be able to write, but aggregates ...

FORUM Aggregation
2018/02/22
Confused about aggregation

FORUM select query
2013/12/04

FORUM

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
Stairway Series

Write For Us
Find out more

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

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 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).

If a query filter allows multiple values, then those values can be stored in another table. For instance, the values could be available in a table-valued parameter that was passed into the stored procedure, or they could be inserted into a temporary table that was filled by splitting a comma-separated list in a `varchar(max)` parameter. The actual query then uses either an `IN` predicate or the equivalent `EXISTS` predicate to filter on the requested values. When using this method to filter a table with a columnstore index, this introduces two problems: the filter cannot be used for rowgroup elimination so that the entire table is read, and in many cases the query execution will also fall back into row mode. Both these problems apply to SQL Server 2012 only.

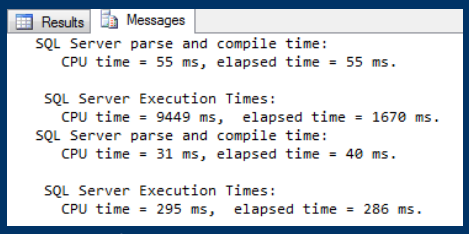
For this problem, there is a very easy workaround. Instead of writing this query with an `IN` predicate or an `EXISTS` predicate, we could write it using a JOIN. The JOIN query returns the same results as long as you can be sure that there are no duplicates in the table that holds the values to be selected. But with the join, you will always get both rowgroup elimination and batch mode execution.

```
USE ContosoRetailDW;
GO

-- Temporary table for promotion selection
CREATE TABLE #Selection
   (PromotionKey int NOT NULL);
INSERT INTO #Selection (PromotionKey)
VALUES (2), (40), (5), (22), (18), (5);

SET STATISTICS TIME ON;

-- Standard query using IN - row mode and no
SELECT     dp.ProductName,
           dd.CalendarQuarter,
           SUM(fos.SalesQuantity) AS Quantit
FROM       dbo.FactOnlineSales    AS fos
INNER JOIN dbo.DimDate            AS dd
      ON   dd.Datekey              = fos.Dat
INNER JOIN dbo.DimProduct         AS dp
```

```
      ON    dp.ProductKey           = fos.Pro
WHERE       fos.PromotionKey       IN (SELECT
                                       FROM
GROUP BY    dp.ProductName,
            dd.CalendarQuarter
ORDER BY    dp.ProductName,
            dd.CalendarQuarter;

-- Using a join instead returns the same res
WITH DistinctSelection
AS (SELECT DISTINCT PromotionKey
    FROM    #Selection)
SELECT      dp.ProductName,
            dd.CalendarQuarter,
            SUM(fos.SalesQuantity) AS Quantit
FROM        dbo.FactOnlineSales    AS fos
INNER JOIN  dbo.DimDate            AS dd
      ON    dd.Datekey              = fos.Dat
INNER JOIN  dbo.DimProduct         AS dp
```

Listing 11-1: Replacing IN (or EXISTS) with a join

The code in listing 11-1 allows you to see the problem with IN and how one might work around it by using a JOIN. Because there are duplicates in the *#Selection* table, I had to put in a CTE (common table expression) to prevent duplicates in the results. If you can be sure that there are no duplicates (for instance because there is a PRIMARY KEY or UNIQUE constraint on the selection table), then you can join directly to the selection table without the need for any CTE at all.



Zoom in | Open in new window

Figure 11-1: Using a join is much faster than using IN

The second version of the query is only slightly more complex than the first, but it performs much better. As you can see above, the first runs in 1.7 seconds while burning 9.4 seconds of CPU time on my system; the second version finishes in 0.3 seconds and uses just 0.3 seconds CPU.

This problem is similar to the one before. The only difference is that this time the values in the selection table should be excluded instead of included. This problem is typically handled similarly to the one before, this time using NOT IN or NOT EXISTS. And again, that query construction will prevent both predicate pushdown and batch mode execution on SQL Server 2012.

One way to work around this is to first get a list of all distinct values that exist in the column you are selecting on, remove the values in the selection list, then use the remaining values in the WHERE clause of the query. This changes the query construction from NOT IN to IN, for which we have already seen the workaround. However, this requires an extra scan of the fact table

to find all possible values, and if you are excluding only a few special customers, then the produced list of customers to include can become very long. Finally, if the fact table allows NULL values in the selection column, then this rewrite causes incorrect results.

There is however an alternative method. This method uses the same basic idea as the workarounds you saw in the previous level: find a way to do as much work as possible in batch mode, then aggregate down as far as possible and accept that the rest of the work on a much smaller data set will be done in row mode. The code in listing 11-2 demonstrates both the performance issue with NOT EXISTS and this rewrite method.

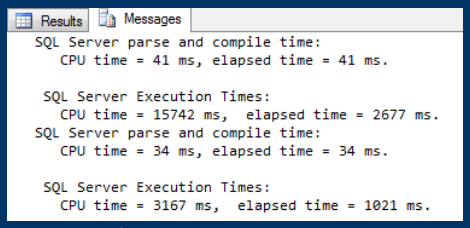
```
USE ContosoRetailDW;
GO

-- Temporary table for promotion selection
CREATE TABLE #Selection
   (PromotionKey int NOT NULL);
INSERT INTO #Selection (PromotionKey)
VALUES (1), (40), (5), (22), (18), (5), (8),

SET STATISTICS TIME ON;

-- Standard query using NOT EXISTS
SELECT     dp.ProductName,
           dd.CalendarQuarter,
           SUM(fos.SalesQuantity) AS Quantit
FROM       dbo.FactOnlineSales    AS fos
INNER JOIN dbo.DimDate            AS dd
      ON   dd.Datekey              = fos.Dat
INNER JOIN dbo.DimProduct         AS dp
      ON   dp.ProductKey           = fos.Pro
WHERE NOT EXISTS
   (SELECT *
    FROM   #Selection             AS s
    WHERE  s.PromotionKey          = fos.Pro

GROUP BY   dp.ProductName,
           dd.CalendarQuarter
ORDER BY   dp.ProductName,
           dd.CalendarQuarter;

-- Using NOT EXISTS after aggregation is muc
WITH AllAggregated
AS (SELECT     dp.ProductName,
               dd.CalendarQuarter,
               fos.PromotionKey,
               SUM(fos.SalesQuantity) AS Qua
    FROM       dbo.FactOnlineSales    AS fos
    INNER JOIN dbo.DimDate            AS dd
```

Listing 11-2: Speeding up NOT IN (or NOT EXISTS)

The trick used here is that, instead of filtering individual rows from the fact table, we now filter entire groups after aggregation. The original query groups by *ProductName* and *CalendarQuarter*; at that level it is impossible to filter on *PromotionKey*. That's why the

first phase has to add the PromotionKey to the GROUP BY, after this first stage of aggregation the result is then filtered using NOT IN, and the remaining data is then further aggregated to produce the final result. Note that for this is the same local-global aggregation pattern used in Level 10, so the same guidelines for handling various aggregates apply.

Obviously, because we force SQL Server to first aggregate all data before applying the filter, there is no way that rowgroup elimination can ever be used in this workaround. Also because we had to add an extra level to the first GROUP BY, the number of rows left when falling back to row mode is higher than in previous examples – just under 100,000 in this case. Because of this, you should not use this rewrite on SQL Server 2014 or newer, where the original query runs (slightly) faster than the rewrite. However, on SQL Server 2012 this rewrite does result in a significant speed increase because it allows the joins for the 12.6 million rows and the local aggregation to all operate in batch mode; only the filtering and the global aggregation of the remaining less than 100,000 rows requires row mode execution.



Zoom in | Open in new window

Figure 11-2: Using local-global aggregation makes NOT EXISTS faster

As shown in figure 11-2, the original query took 2.7 seconds and used 15.7 seconds CPU time on my system; after the rewrite the query finished in 1.0 seconds and used 3.2 seconds of CPU.

For any type of table and any type of index, a WHERE clause that uses OR can cause problems. Columnstore indexes are no exception. In this case, the problem is not related to execution mode (you will see batch mode if the OR condition is the only problematic element in your query), but you do lose rowgroup elimination. When each of the individual filters is very selective, then this can still affect query performance, and this is a problem that affects performance on every version of SQL Server.

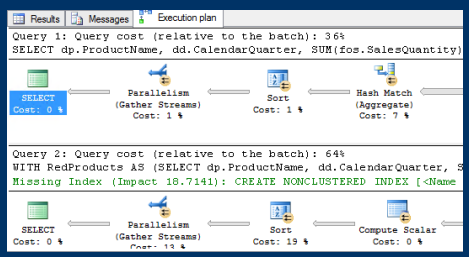
An alternative approach would be to once more use local-global aggregation. For example, if you have a query that reports on products that are red or expensive, then you can instead use two queries, one for all red products and another for all expensive products, then combine the results. But that would count some products twice: items that are both red and expensive are included in both results. To prevent that, you have to remove those from one of the two. This is shown in listing 11-3, where you first see a query with OR, and then the same query rewritten to use local-global aggregation on two sets: red products and expensive non-red products. Note the handling of NULL values in the second part of this query; if a column does not allow NULL values then this can be removed (making the query even faster).

```
USE ContosoRetailDW;
GO
```

```
SET STATISTICS TIME ON;

-- A simple query using OR
SELECT     dp.ProductName,
           dd.CalendarQuarter,
           SUM(fos.SalesQuantity) AS Quantit
FROM       dbo.FactOnlineSales    AS fos
INNER JOIN dbo.DimDate            AS dd
      ON   dd.Datekey              = fos.Dat
INNER JOIN dbo.DimProduct         AS dp
      ON   dp.ProductKey           = fos.Pro
WHERE      dp.ColorName            = 'Red'
OR         fos.UnitCost            > 500
GROUP BY   dp.ProductName,
           dd.CalendarQuarter
ORDER BY   dp.ProductName,
           dd.CalendarQuarter;

-- A much longer but in this case also faste
WITH RedProducts
AS (SELECT     dp.ProductName,
               dd.CalendarQuarter,
               SUM(fos.SalesQuantity) AS Qua
    FROM       dbo.FactOnlineSales    AS fos
    INNER JOIN dbo.DimDate            AS dd
          ON   dd.Datekey              = fos
    INNER JOIN dbo.DimProduct         AS dp
          ON   dp.ProductKey           = fos
    WHERE      dp.ColorName            = 'Re
    GROUP BY   dp.ProductName,
               dd.CalendarQuarter)
, ExpensiveProducts
AS (SELECT     dp.ProductName,
               dd.CalendarQuarter,
```

Listing 11-3: Avoiding OR in a query

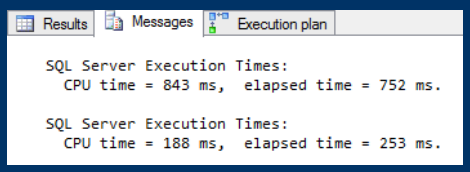
The execution plans shown in figure 11-3 were generated on SQL Server 2014. The percentages shown (which are always based on the estimated cost, even in an actual execution plan) seem to indicate that the rewrite will be almost twice as slow as the simple original version of the query. (The optimizer also suggests adding a nonclustered rowstore index to improve performance. This is a bad suggestion. If you add the suggested index it is not even used, and when you add a hint to force the use of this index it reduces performance.)



Zoom in | Open in new window

Figure 11-3: Execution plans for the simple OR and the long rewrite

The estimated query has always been a dangerous and misleading tuning tool; this is even worse when columnstore indexes are involved. The estimated cost is based on assumptions and these assumptions don't sufficiently take into account the savings of reading from columnstore indexes and of executing in batch mode. That is why, instead of looking at the percentages shown above, I prefer to go by the output generated by `SET STATISTICS IO`. As shown in figure 11-4, estimated costs are indeed completely wrong in this case: the elapsed time is reduced from 0.75 to just 0.25 seconds, and the CPU time used is down from over 0.8 seconds to less than 0.2.



Zoom in | Open in new window

Figure 11-4: Actual elapsed and CPU time of the two queries

Rewriting the query improves performance of this particular query, but that will not always be the case; it depends on selectivity of the filters and on the data distribution. For other conditions, the original version of the query using a simple `OR` would be faster than the rewrite. You should always test both variations on realistic test data and only use the rewrite if it improves performance in your specific case. Also, because data can change over time, it is important to periodically revisit these queries to verify if the implemented version is still the best option.

Probably the most surprising limitation of batch mode execution in SQL Server 2012 is that it cannot handle aggregates without a `GROUP BY` in the query. So when a data analyst runs a few quick queries to get for instance minimum and maximum value in a column or number of rows, just to get a first impression of the data, those queries will run slower than the much more complicated queries used for actual reports. The reason for that speed difference is that the reports have `GROUP BY` in their queries so that aggregation (and most of the rest of the execution plan) can be performed in batch mode; the simple query used to get a first feel for the data has no `GROUP BY` and hence falls back to row mode.

It is possible to work around this limitation by using local-global aggregation: first grouping by any available column for the local aggregation, and then doing global aggregation without `GROUP BY` on those results. But in this specific case there is a far easier way to work around it: just add a "fake" `GROUP BY` clause, using an expression that is redundant because it always returns the same value, but that the optimizer doesn't recognize as redundant. Surprisingly enough, simply subtracting a numeric column from itself is sufficient for this purpose. (Do make sure to use a column that doesn't allow `NULL` values, otherwise you'll end up with two rows in the result set).

```
USE ContosoRetailDW;
GO

SET STATISTICS TIME ON;
```