This post was published to Knowledge Base at 9:19:10 AM 03/05/2013

Columnstore Index Quirks

Category SQL Server

New with SQL Server 2012, they have now have columnstore indexes. In short, it is an index that stores everything in memory and does a quick reduction of any of the pages that it actually needs to read.

However, there are a few quirks that you need to be aware of when using it.

Some that we have found are as follows:

* Multiple COUNT(DISTINCT) within the same query will force the engine to go into row processing mode and become slow
* Mixing COUNT(DISTINCT) with other aggregations (i.e. SUM()) is slower than taking 2 trips to your fact table
* Having multiple aggregations in a query (other than COUNT(DISTINCT) will only affect performance because it is pulling more into memory – not a big hit)

Below are a few queries that we currently have and their respective execution times.

**Please Note**: All of these should return the same data set.

Below are some of our findings with their corresponding queries. Hopefully you will find these as interesting as I did ☺

I am running the data into temp tables that are all created equally and everything was done in a single batch so that all of the estimated costs per batch will be relative to everything else.

Test Case 1: SUM and COUNT(DISTINCT) in the same query

CREATE TABLE #TotalsCase

(

DepartmentKey SMALLINT

, Dollars MONEY

, ProductCount INT

)

-- How we normally code the counts

INSERT INTO #TotalsCase

SELECT DepartmentKey, SUM(Dollars), COUNT(DISTINCT ProductKey)

FROM FactSalesTimeframeDenormalised AS f

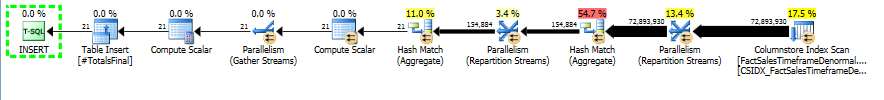
WHERE TimeFrameKey = **1**

GROUP BY DepartmentKey

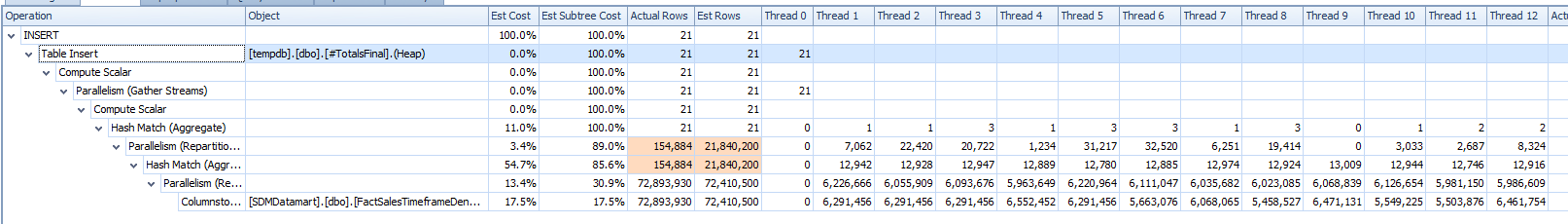
**Execution Information**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Est Cost** | **Duration** | **CPU** | **Est CPU Cost** | **Reads** | **Writes** | **Est IO Cost** | **Est Rows** | **Actual Rows** | **Degree of Parallelism** | **Parallel Operations** | **Sort Operations** | **Hash Match Operations** | **Index Scan Operations** | **Table Scan Operations** |
| 69.40% | 9,540 | 110,341 | 72.80% | 32 | 0 | 65.10% | 21 | 21 | 12 | 7 | 0 | 2 | 1 | 0 |

**Execution Plan**:



**Plan Tree**:



Test Case 2: SUM and COUNT(DISTINCT) within separate CTEs

CREATE TABLE #TotalsCTE

(

DepartmentKey SMALLINT

, Dollars MONEY

, ProductCount INT

)

-- Calculate the total dollars and the distinct products that were sold

-- separately (CTEs)

; WITH ProductCount AS (

SELECT DepartmentKey, PD = COUNT(DISTINCT ProductKey)

FROM FactSalesTimeframeDenormalised AS f

WHERE TimeFrameKey = **1** AND IsPromo = **1**

GROUP BY DepartmentKey

), TotalDollars AS (

SELECT DepartmentKey, Dollars = SUM(Dollars)

FROM FactSalesTimeframeDenormalised AS f

WHERE TimeFrameKey = **1**

GROUP BY DepartmentKey

)

INSERT INTO #TotalsCTE

SELECT f.DepartmentKey, SUM(Dollars), SUM(pd.PD)

FROM TotalDollars AS f

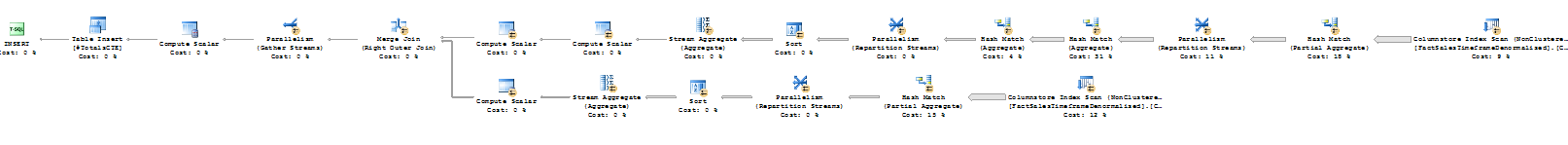
LEFT JOIN ProductCount AS pd ON f.DepartmentKey = pd.DepartmentKey

GROUP BY f.DepartmentKey

**Execution Information**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Est Cost** | **Duration** | **CPU** | **Est CPU Cost** | **Reads** | **Writes** | **Est IO Cost** | **Est Rows** | **Actual Rows** | **Degree of Parallelism** | **Parallel Operations** | **Sort Operations** | **Hash Match Operations** | **Index Scan Operations** | **Table Scan Operations** |
| 0.00% | 1,137 | 12,522 | 0.00% | 32 | 0 | 0.00% |  | 21 |  | 0 | 0 | 0 | 0 | 0 |

**Execution Plan**:



**Plan Tree**:

Wasn’t available.

Test Case 3: SUM and then UPDATE

CREATE TABLE #TotalsUpdate

(

DepartmentKey SMALLINT

, Dollars MONEY

, ProductCount INT

)

-- Get the total dollars

INSERT INTO #TotalsUpdate ( DepartmentKey, Dollars, ProductCount )

SELECT DepartmentKey, SUM(Dollars), **0**

FROM FactSalesTimeframeDenormalised AS f

WHERE TimeFrameKey = **1**

GROUP BY DepartmentKey

-- Then calculate the product count

; WITH ProductCount AS (

SELECT DepartmentKey, PD = COUNT(DISTINCT ProductKey)

FROM FactSalesTimeframeDenormalised AS f

WHERE TimeFrameKey = **1** AND IsPromo = **1**

GROUP BY DepartmentKey

)

UPDATE #TotalsUpdate SET ProductCount = PD

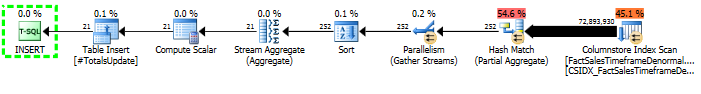
FROM ProductCount AS p

**Execution Information**: 2 part operation

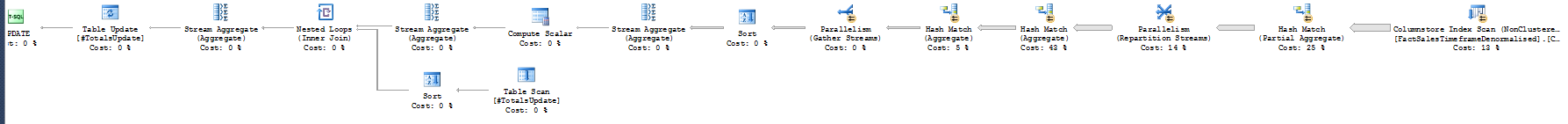
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Est Cost** | **Duration** | **CPU** | **Est CPU Cost** | **Reads** | **Writes** | **Est IO Cost** | **Est Rows** | **Actual Rows** | **Degree of Parallelism** | **Parallel Operations** | **Sort Operations** | **Hash Match Operations** | **Index Scan Operations** | **Table Scan Operations** |
| 4.80% | 434 | 4,274 | 3.00% | 32 | 0 | 12.20% | 21 | 21 | 12 | 3 | 1 | 1 | 1 | 0 |
| 0.00% | 707 | 7,581 | 0.00% | 24 | 0 | 0.00% |  | 21 |  | 0 | 0 | 0 | 0 | 0 |

**Execution Plan**:

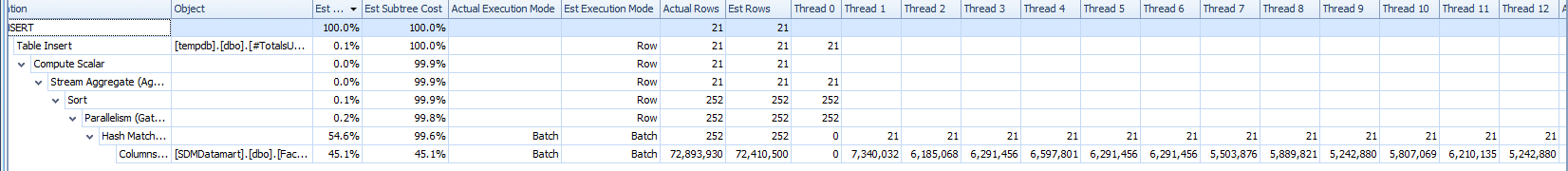
*Insert*:



*Update*:



**Plan Tree**: (Insert only)



As you can see, when you are using columnstore indexes, you don’t want to do both a COUNT(DISTINCT) in the same operation as a SUM because then, even though it gives a nice and linear execution plan, the cost of the entire operation is a lot bigger.

So, when you are using columnstore indexes, either use a CTE to split it up or use the insert, then update method for optimal performance ☺