TASK 4: Data Warehouse Optimizations

Logical indexed view

Personal notes:

In data warehouses, you can use materialized views **to precompute and store aggregated data such as the sum of sales**.

The SELECT list in the materialized view definition needs to meet at least one of these two criteria: The SELECT list contains an aggregate function.
GROUP BY is used in the Materialized view definition and all columns in GROUP BY are included in the SELECT list.
Aggregate functions are required in the SELECT list of the materialized view definition.
Supported aggregations include MAX, MIN, AVG, COUNT, COUNT_BIG, SUM, VAR, STDEV.

—

For the logical indexed view I chose the query for my **third analytical question**, which is "Which delay type has the longest average delay ?".

Reason for choosing this query is that it **contains an aggregated calculation** (AVG) and it would be beneficial to not have to query every single delay row when requesting this information, since there are 632 thousand delays (currently).

```
SELECT ddt.delay_code, ddt.delay_description,
AVG(fd.EFFECTIVE_DELAY_TIME_MV) AS average_delay,
COUNT(ddt.delay_code) as delay_count

FROM [datawarehouse].[dbo].[fact_delay] fd
INNER JOIN [datawarehouse].[dbo].[dim_delay_type] ddt
ON fd.DIM_DELAY_TYPE_FK = ddt.delay_type_sk

GROUP BY ddt.delay_code, ddt.delay_description
```

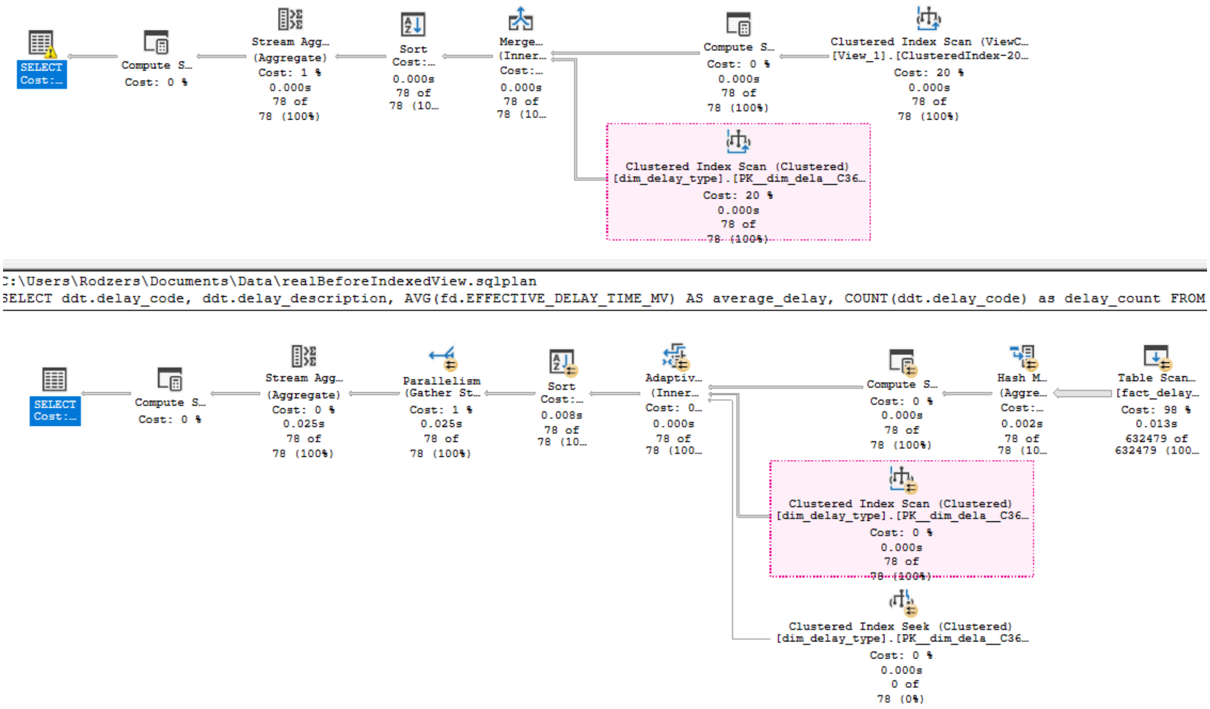
The script:

https://docs.google.com/document/d/1LUMJH2-f0gq6DbV1lXE7WqB30qmwrX-q5adBfuMQeWA/edit?usp=sharing

Or in the Task 4 folder.

Execution plans side by side

Top - With logical indexed view.
Bottom - Without logical indexed view.

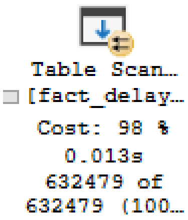


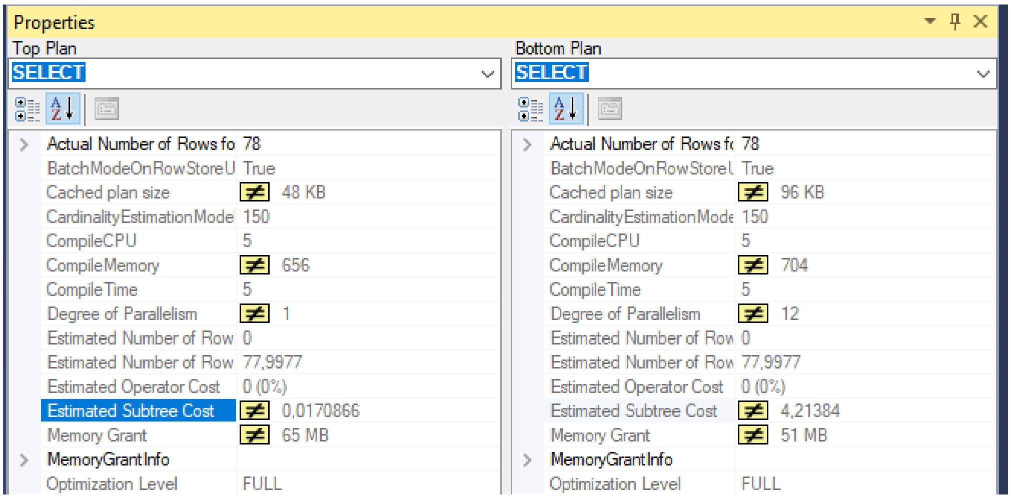
```
C:\Users\Rodzers\Documents\Data\realBeforeIndexedView.sqlplan
SELECT ddt.delay_code, ddt.delay_description, AVG(fd.EFFECTIVE_DELAY_TIME_MV) AS average_delay, COUNT(ddt.delay_code) as delay_count FROM
```

Conclusion

With the help of the logical indexed view, I was able to eliminate the most resource intensive operation, which was the fact_delay table scan.

It’s relative cost to the whole query was 98% and it was due to the fact that it had to read the entire table which at the time of querying was 632’479 rows.





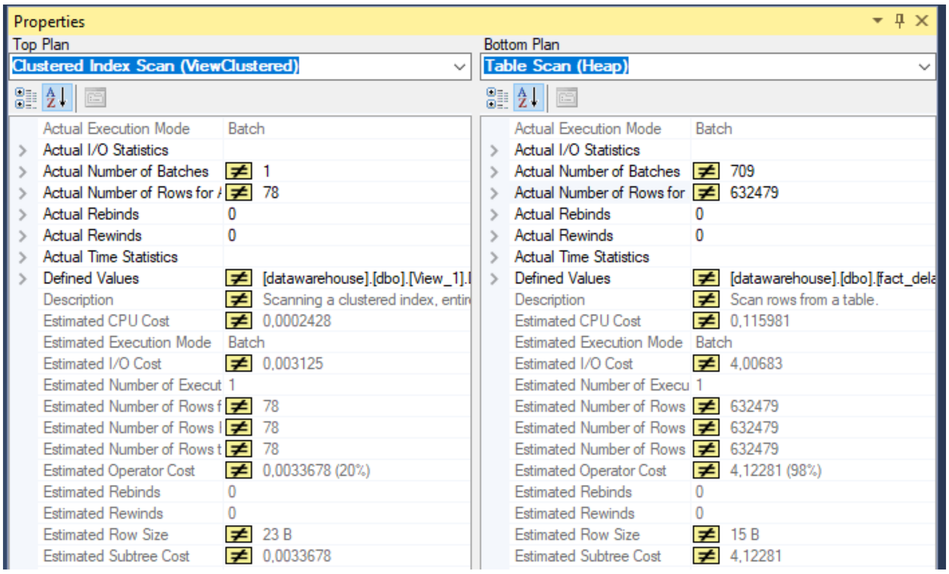
Properties

Top Plan	SELECT	Bottom Plan	SELECT
Actual Number of Rows fo	78	Actual Number of Rows fo	78
BatchModeOnRowStoreU	True	BatchModeOnRowStoreU	True
Cached plan size	≠ 48 KB	Cached plan size	≠ 96 KB
CardinalityEstimationMode	150	CardinalityEstimationMode	150
CompileCPU	5	CompileCPU	5
CompileMemory	≠ 656	CompileMemory	≠ 704
CompileTime	5	CompileTime	5
Degree of Parallelism	≠ 1	Degree of Parallelism	≠ 12
Estimated Number of Row	0	Estimated Number of Row	0
Estimated Number of Row	77,9977	Estimated Number of Row	77,9977
Estimated Operator Cost	0 (0%)	Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	≠ 0,0170866	Estimated Subtree Cost	≠ 4,21384
Memory Grant	≠ 65 MB	Memory Grant	≠ 51 MB
MemoryGrantInfo		MemoryGrantInfo	
Optimization Level	FULL	Optimization Level	FULL

Thanks to this optimization, we can see that the overall query cost has dropped from 4.21384 to 0.0170866, which in my opinion is a **massive** improvement.

And the host resources were reduced as well with the Degree of Parallelism dropping from 12 all the way to 1.

And that is because the table_scan operation was replaced with my newly created Clustered Index.



Properties

Top Plan	Clustered Index Scan (ViewClustered)	Bottom Plan	Table Scan (Heap)
Actual Execution Mode	Batch	Actual Execution Mode	Batch
Actual I/O Statistics		Actual I/O Statistics	
Actual Number of Batches	≠ 1	Actual Number of Batches	≠ 709
Actual Number of Rows for /	≠ 78	Actual Number of Rows for	≠ 632479
Actual Rebinds	0	Actual Rebinds	0
Actual Rewinds	0	Actual Rewinds	0
Actual Time Statistics		Actual Time Statistics	
Defined Values	≠ [datawarehouse].[dbo].[View_1].[	Defined Values	≠ [datawarehouse].[dbo].[fact_dela
Description	≠ Scanning a clustered index, entir	Description	≠ Scan rows from a table.
Estimated CPU Cost	≠ 0,0002428	Estimated CPU Cost	≠ 0,115981
Estimated Execution Mode	Batch	Estimated Execution Mode	Batch
Estimated I/O Cost	≠ 0,003125	Estimated I/O Cost	≠ 4,00683
Estimated Number of Execut	1	Estimated Number of Execu	1
Estimated Number of Rows f	≠ 78	Estimated Number of Rows	≠ 632479
Estimated Number of Rows I	≠ 78	Estimated Number of Rows	≠ 632479
Estimated Number of Rows t	≠ 78	Estimated Number of Rows	≠ 632479
Estimated Operator Cost	≠ 0,0033678 (20%)	Estimated Operator Cost	≠ 4,12281 (98%)
Estimated Rebinds	0	Estimated Rebinds	0
Estimated Rewinds	0	Estimated Rewinds	0
Estimated Row Size	≠ 23 B	Estimated Row Size	≠ 15 B
Estimated Subtree Cost	≠ 0,0033678	Estimated Subtree Cost	≠ 4,12281

Some notable observations:

- Number of rows that it has to scan is now 78 instead of 632’479.
- Estimated CPU Cost dropped from 0.115981 to 0.0002428.
- Estimated I/O Cost dropped from 4.0683 to 0.003125.
- And the Estimated Operator Cost for acquiring the necessary data is now 0.0033678 instead of 4.12281

Would I implement this logical indexed view ?

That depends on how often I will use this query and whether this information is vital to my ‘business’ / ‘company’.

If it’s only being used once a month (or even less frequently) for preparing presentations / summaries within the company, then there’s no reason in my opinion to waste additional disk space and resources.

Otherwise if the query serves some valid purpose, then why not, the benefits would outweigh the disadvantages.

Column storage

Personal notes:

Column storage is optimized for fast retrieval of columns of data, which is useful for analytical applications, due to the fact that it drastically reduces the overall disk I/O requirements and reduces the amount of data that you need to load from disk.

It’s often used in data warehouses where it’s necessary to send large amounts of data from multiple sources for these previously mentioned BI applications.

It was designed to accelerate data warehousing queries, which require scanning, aggregation and filtering of large amounts of data, or joining multiple tables like a star schema.

Requirements:

- Limited amount of columns.

Therefore the best place to implement this columnstore storage is in the **fact_delay** table of my datawarehouse.

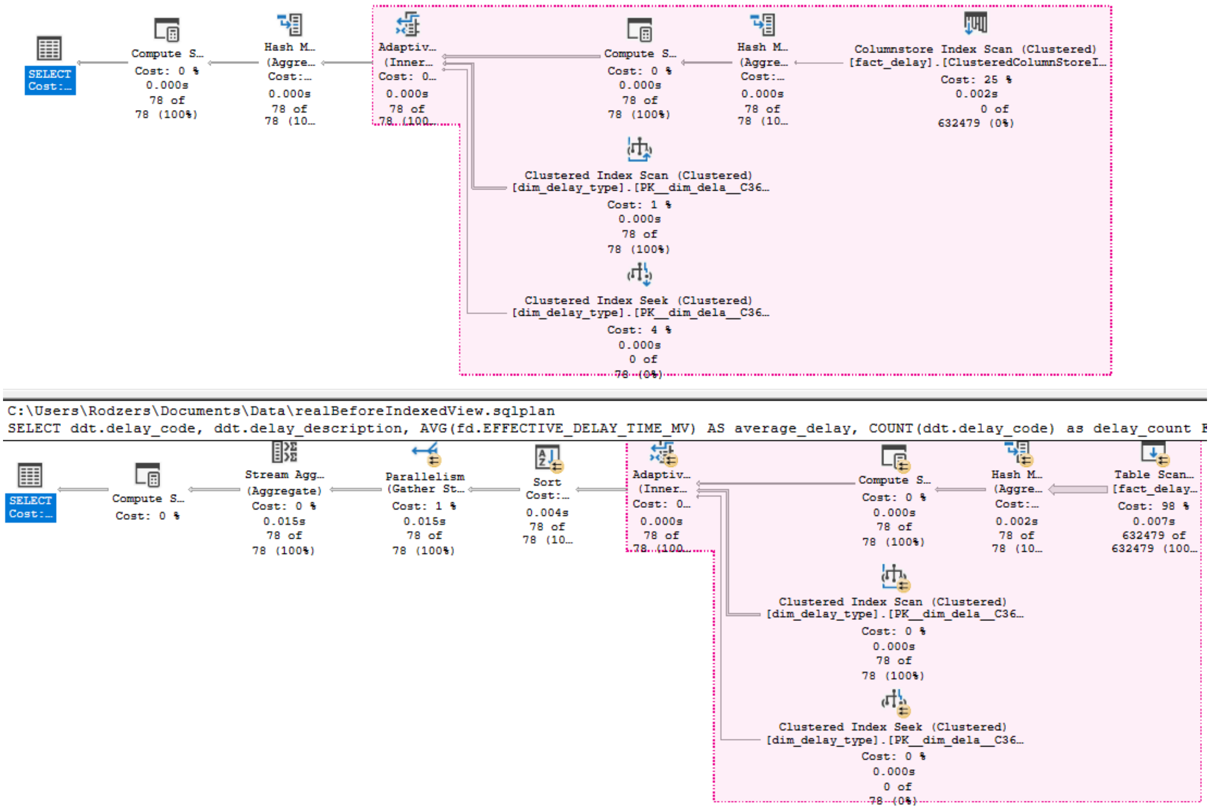
The script:

https://docs.google.com/document/d/1FDfXJOAlcQTMtr40EqH3UJzzjqGn5uDoDT5-xqlE4WQ/edit?usp=sharing

Or in the Task 4 folder.

Execution plans side by side

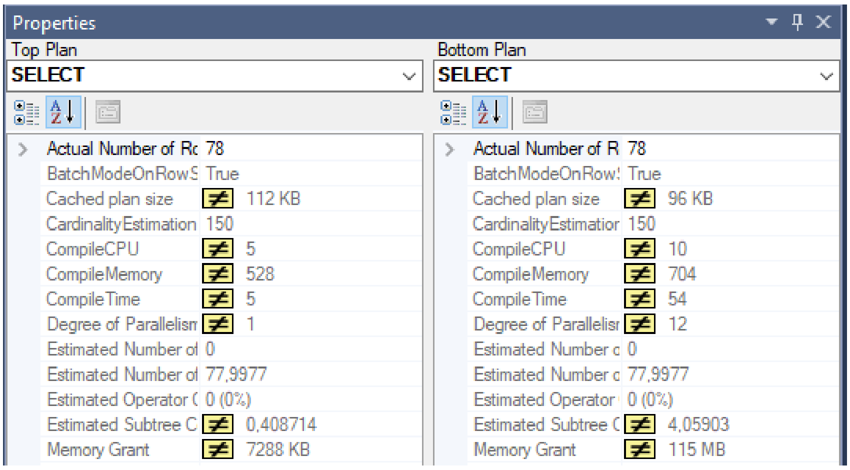
Top - With column store index.
Bottom - Without column store index.



```
C:\Users\Rodzers\Documents\Data\realBeforeIndexedView.sqlplan
SELECT ddt.delay_code, ddt.delay_description, AVG(fd.EFFECTIVE_DELAY_TIME_MV) AS average_delay, COUNT(ddt.delay_code) as delay_count F
```

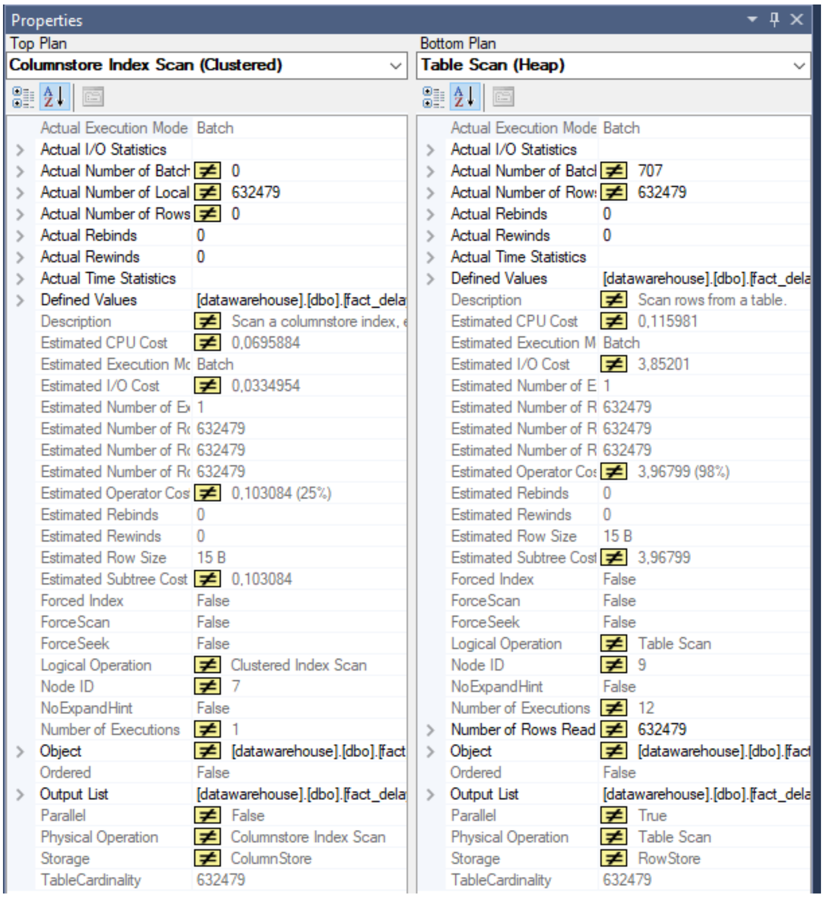
Conclusion

Once again you can see large overall improvements.


Properties

Top Plan			Bottom Plan		
SELECT			SELECT		
Actual Number of Rc		78	Actual Number of R		78
BatchModeOnRowS		True	BatchModeOnRow:		True
Cached plan size	≠	112 KB	Cached plan size	≠	96 KB
CardinalityEstimation		150	CardinalityEstimatior		150
CompileCPU	≠	5	CompileCPU	≠	10
CompileMemory	≠	528	CompileMemory	≠	704
CompileTime	≠	5	CompileTime	≠	54
Degree of Parallelisr	≠	1	Degree of Parallelisr	≠	12
Estimated Number of		0	Estimated Number c		0
Estimated Number of		77,9977	Estimated Number c		77,9977
Estimated Operator (		0 (0%)	Estimated Operator		0 (0%)
Estimated Subtree C	≠	0,408714	Estimated Subtree C	≠	4,05903
Memory Grant	≠	7288 KB	Memory Grant	≠	115 MB

But not as much as for the Logical Indexed View.



Properties

Top Plan

Columnstore Index Scan (Clustered)

Actual Execution Mode	Batch
Actual I/O Statistics	
Actual Number of Batch	≠ 0
Actual Number of Local	≠ 632479
Actual Number of Rows	≠ 0
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[datawarehouse].[dbo].[fact_dela
Description	≠ Scan a columnstore index, e
Estimated CPU Cost	≠ 0,0695884
Estimated Execution Mc	Batch
Estimated I/O Cost	≠ 0,0334954
Estimated Number of Ex	1
Estimated Number of Rc	632479
Estimated Number of Rc	632479
Estimated Number of Rc	632479
Estimated Operator Cos	≠ 0,103084 (25%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	15 B
Estimated Subtree Cost	≠ 0,103084
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	≠ Clustered Index Scan
Node ID	≠ 7
NoExpandHint	False
Number of Executions	≠ 1
Object	≠ [datawarehouse].[dbo].[fact
Ordered	False
Output List	[datawarehouse].[dbo].[fact_dela
Parallel	≠ False
Physical Operation	≠ Columnstore Index Scan
Storage	≠ ColumnStore
TableCardinality	632479

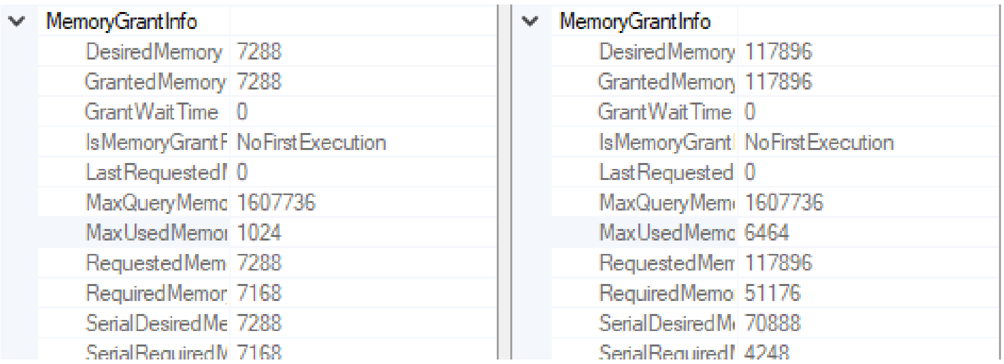
Bottom Plan

Table Scan (Heap)

Actual Execution Mode	Batch
Actual I/O Statistics	
Actual Number of Batcl	≠ 707
Actual Number of Row:	≠ 632479
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[datawarehouse].[dbo].[fact_dela
Description	≠ Scan rows from a table.
Estimated CPU Cost	≠ 0,115981
Estimated Execution M	Batch
Estimated I/O Cost	≠ 3,85201
Estimated Number of E	1
Estimated Number of R	632479
Estimated Number of R	632479
Estimated Number of R	632479
Estimated Operator Co:	≠ 3,96799 (98%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	15 B
Estimated Subtree Cost	≠ 3,96799
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	≠ Table Scan
Node ID	≠ 9
NoExpandHint	False
Number of Executions	≠ 12
Number of Rows Read	≠ 632479
Object	≠ [datawarehouse].[dbo].[fact
Ordered	False
Output List	[datawarehouse].[dbo].[fact_dela
Parallel	≠ True
Physical Operation	≠ Table Scan
Storage	≠ RowStore
TableCardinality	632479

The Estimated Operator Costs dropped from 3.96799 to 0.103084.

And this time you can also see massive improvements in the memory sector, where the query execution only required 7168 KB of memory instead of 51176 KB and used 1024 KB instead of 6464.



MemoryGrantInfo	
DesiredMemory	7288
GrantedMemory	7288
GrantWaitTime	0
IsMemoryGrantF	NoFirstExecution
LastRequestedI	0
MaxQueryMemo	1607736
MaxUsedMemoi	1024
RequestedMem	7288
RequiredMemor	7168
SerialDesiredMe	7288
SerialRequiredIV	7168

MemoryGrantInfo	
DesiredMemory	117896
GrantedMemory	117896
GrantWaitTime	0
IsMemoryGrant	NoFirstExecution
LastRequested	0
MaxQueryMem	1607736
MaxUsedMemo	6464
RequestedMen	117896
RequiredMemo	51176
SerialDesiredM	70888
SerialRequiredI	4248

Would I implement the column storage specifically for this query ?

For this specific query, no, but overall for an OLAP system, more specifically my fact_delay table, yes.