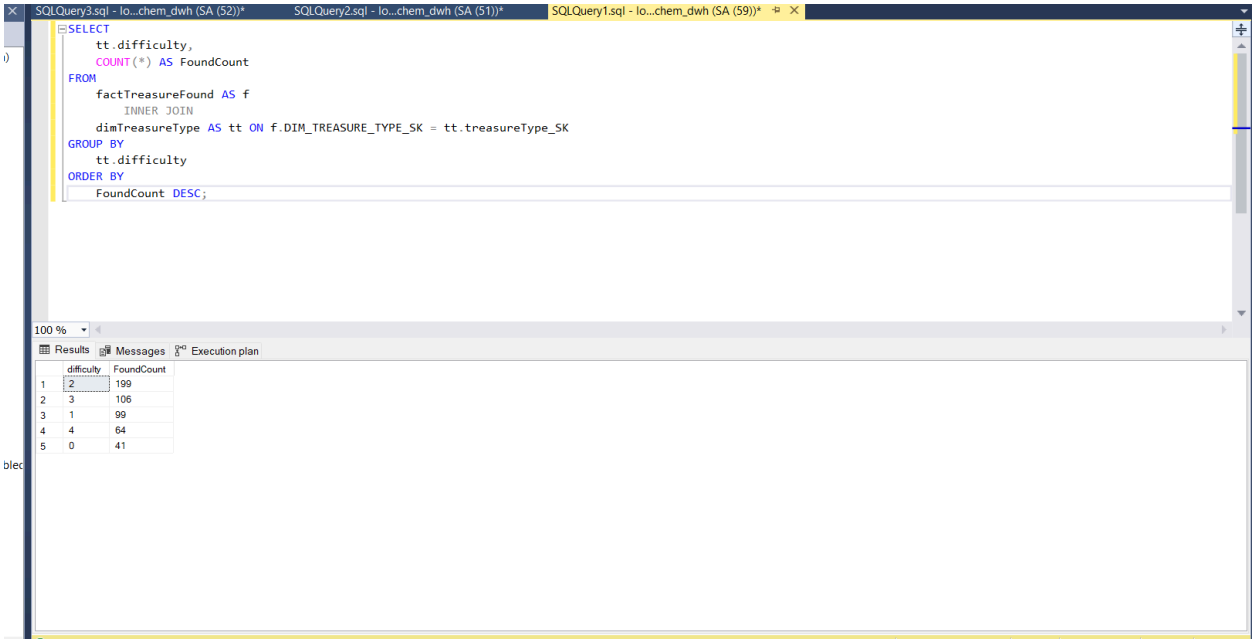
Optimization

[S1]: Create a logical index based on a research question
This is the research question:

```
-- Which caches are most popular based on difficulty level?
```

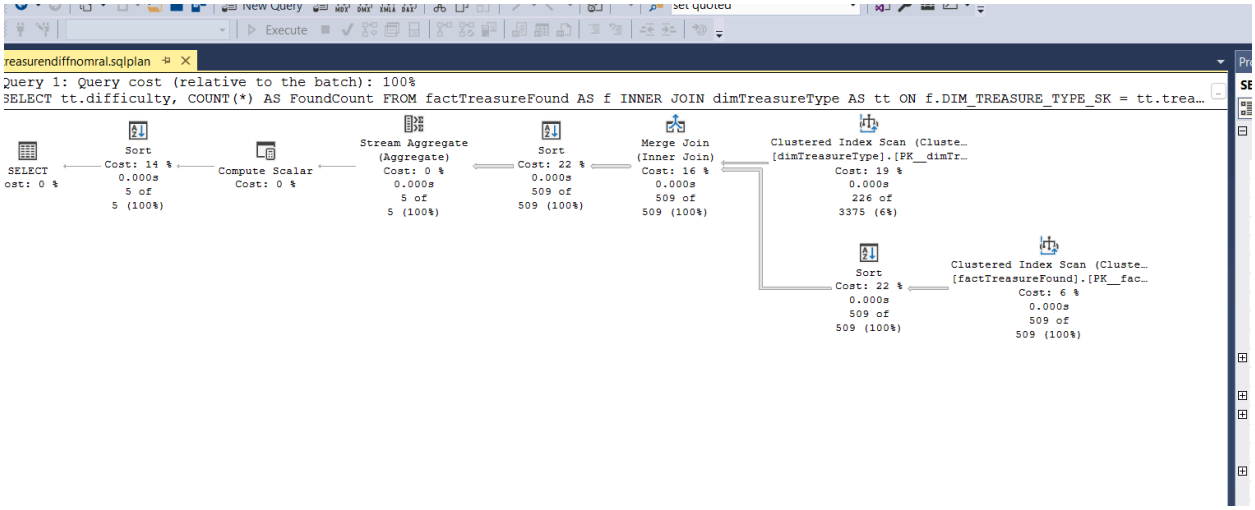
Below is the output of the said research question

Photo from sql manager showing the output of the basic query:



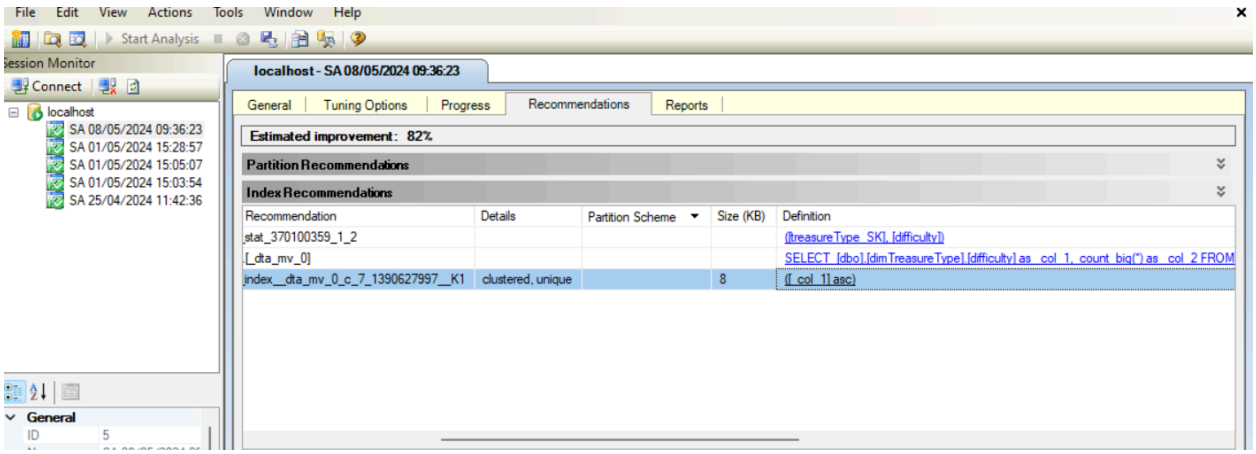
Based on the provided data, it appears that caches with a difficulty level of 2 are the most popular at the moment, with 199 found counts.
-- This suggests that caches with a moderate difficulty level are sought after by various people. The popularity decreases as the difficulty level increases,
-- with fewer caches found for difficulty levels 3, 1, 4, and 0, respectively.

Here is the execution plan for the research question query:

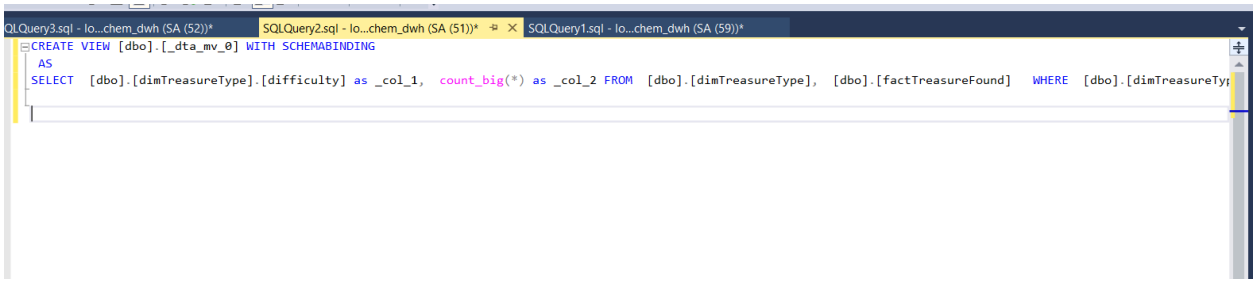


Before the index optimization, the query had many joins and sorting operations, leading to a complex execution plan with numerous links. These joins and sorting operations increase computational overhead and make the query execution less efficient.

Then we will be going to tuning advisor and login with same credentials and will first run the query if you have not done so yet, so that the tuning advisor knows from which database and table to run the recommendations on, for us we will be choosing the index options in the tuning options then we start the analysis.

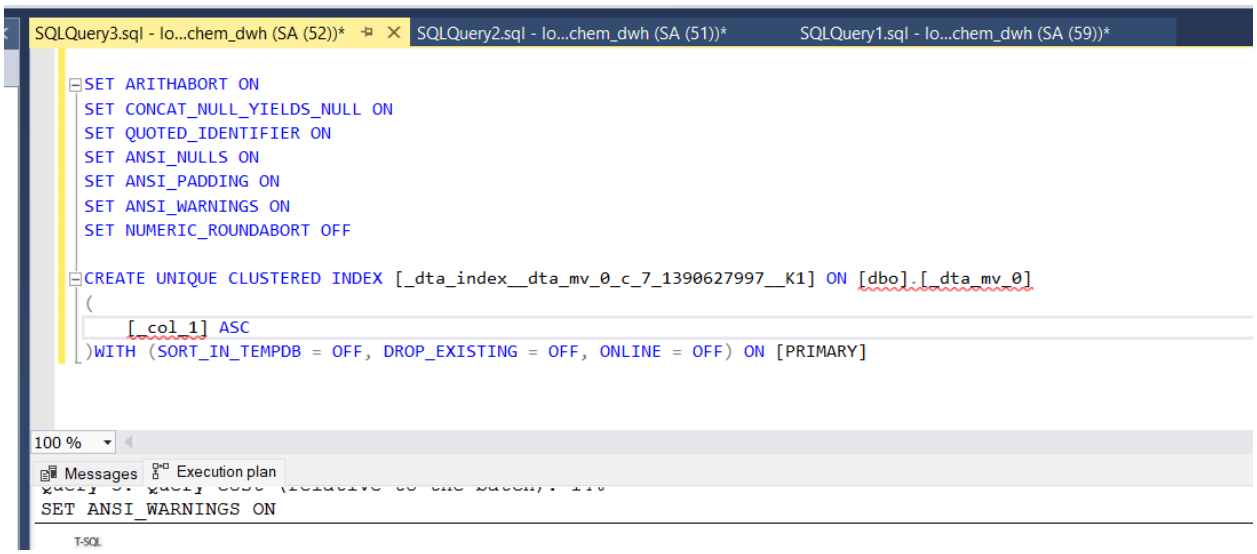


We will be choosing both the create view and the clustered index on said view: which are shown below



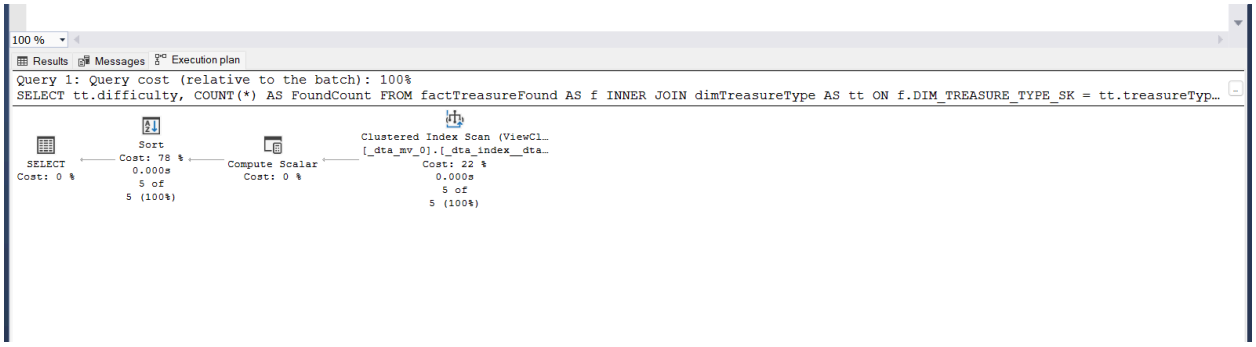
Above we create the view on the original query parameters

Then create a unique clustered index on that view:



We create it because:
Creating a unique clustered index on the view derived from the original query parameters boosts query performance by organizing data efficiently, speeding up data retrieval, enhancing join operations, and reducing I/O operations. This index assists the query optimizer in generating efficient execution plans, resulting in overall better database performance.

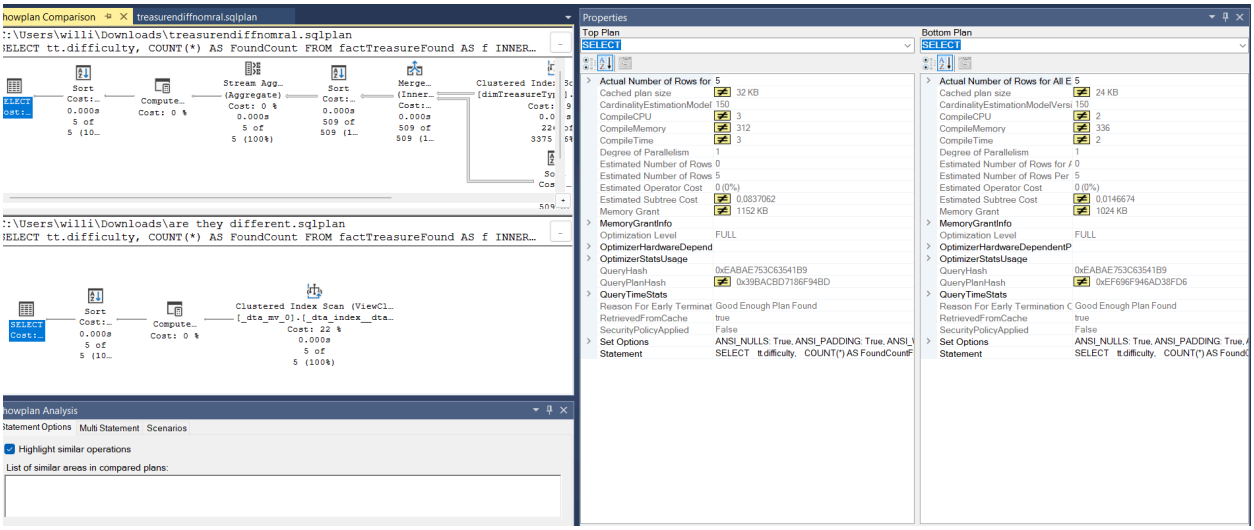
And run the query again:



Incorporating a clustered index materialized view into the original query simplifies execution by reducing the need for complex joins, sorting, and merging operations. With pre-computed data stored in the materialized view, the query optimizer can streamline execution, resulting in a more efficient query plan with fewer computational steps

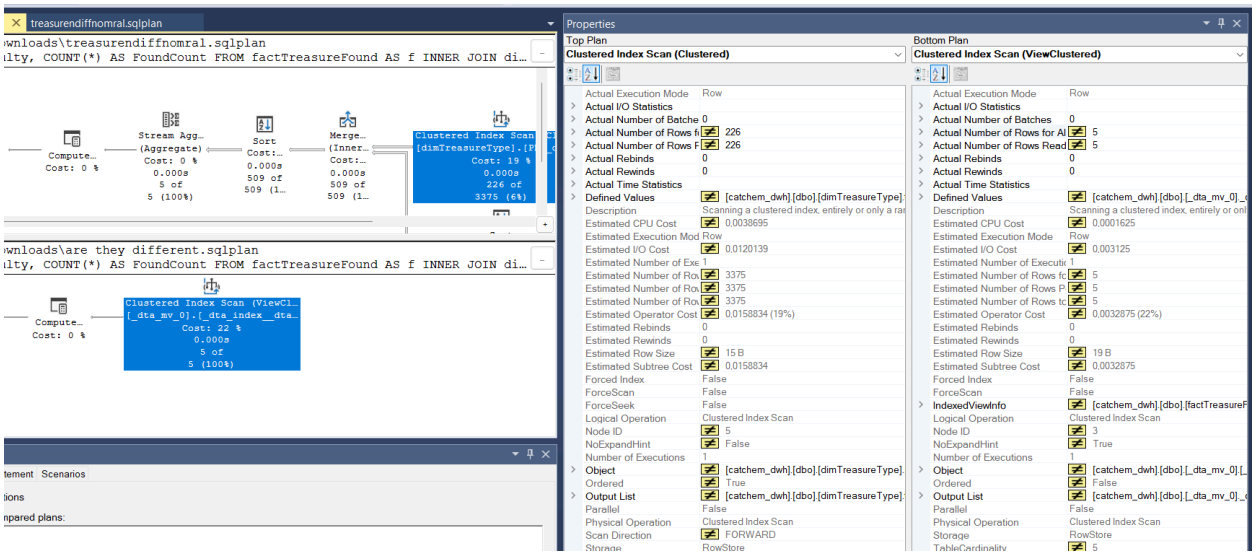
Also, the logical index on the original query helps streamline data access by providing a structured organization of the queried columns. This aids in faster data retrieval and improved query performance without physically altering the database schema.

Execution Plan comparison:



Based on this analysis, it appears that the query has been optimized effectively. By employing joins, indexes, and clusters, the query execution has notably improved in terms of speed and efficiency, resulting in a lower subtree cost.

Comparing the performance metrics before and after optimization, there's a significant reduction in the subtree cost with the optimized query. For instance, the subtree cost has decreased from 0.08 to 0.01 with the utilization of an index. This indicates a substantial enhancement in query execution efficiency.



The decrease in I/O from 0.01 to 0.003 signifies improved efficiency in data input/output operations. This optimization reduces the amount of data read from or written to storage, leading to faster query execution and lower resource usage.

Similarly, the decrease in CPU cost from 0.003 to 0.0001 indicates enhanced efficiency in CPU utilization during query execution. This optimization results in reduced processing time and resource consumption, contributing to overall faster query performance and improved system responsiveness.

In summary, the integration of joins, indexes, and clusters has significantly optimized the query, resulting in notable improvements in performance metrics such as subtree cost, CPU utilization, and I/O operations. The reduction in subtree cost highlights the effectiveness of these optimization techniques in enhancing overall query efficiency, while the decrease in CPU cost and I/O reflects improved resource utilization and faster data access.

Decision:
Considering the significant decreases observed in subtree cost, CPU cost, and I/O operations after implementing the logical index, it's evident that the index has effectively optimized the query's performance. The improvements in these key metrics indicate enhanced query efficiency and resource utilization.

Therefore, based on the observed benefits of the logical index, it is recommended to use the indexed version of the query for improved performance and resource efficiency.

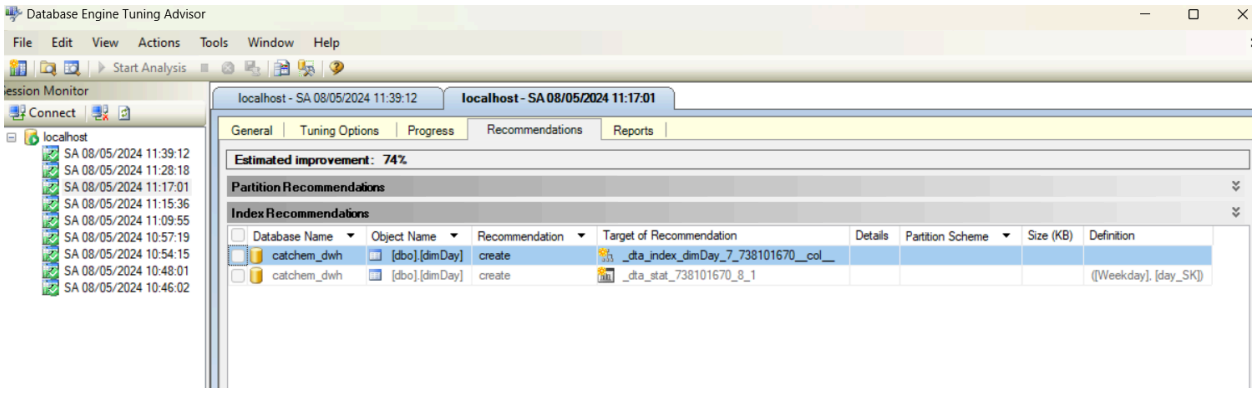
[S2] Column Storage Optimization

I chose the following research question and query for column storage optimization.
Q. Are more difficult caches done on weekends?

```
SELECT
    dt.difficulty,
    dd.Weekday,
    COUNT(ftf.TreasureFoundID) AS total_caches_searched
FROM
    catchem_dwh.dbo.factTreasureFound ftf
        JOIN
    catchem_dwh.dbo.dimTreasureType dt ON ftf.DIM_TREASURE_TYPE_SK = dt.treasureType_SK
        JOIN
    catchem_dwh.dbo.dimDay dd ON ftf.DIM_DAY_SK = dd.day_SK
GROUP BY
    dt.difficulty, dd.Weekday
ORDER BY
    dt.difficulty, dd.Weekday;
```

because it has aggregate expression in a select clause and joins with two other tables that will deal with potentially large amounts of datasets.

After I ran the query for the first time, the Turning Advisor gave the recommendation plans.



I implemented the following query to create column storage and create statistics for dimDay.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX [_dta_index_dimDay_7_738101670__col__] ON [dbo].[dimDay]
(
    [day_SK],
    [Date],
    [DayOfMonth],
    [Month],
    [Year],
    [DayOfWeek],
    [DayOfYear],
    [Weekday],
    [MonthName],
    [Season]
)WITH (DROP_EXISTING = OFF, COMPRESSION_DELAY = 0) ON [PRIMARY]

CREATE STATISTICS [_dta_stat_738101670_8_1] ON [dbo].[dimDay]([Weekday], [day_SK])
```

Top plan refers to the query **after optimization** and **Button Plan** is the **original** query.
Estimated Subtree Cost has been reduced from **0.143501** to **0.0265457**.
The top plan used 3 seconds of CPU time and 593KB of memory during compilation, whereas the bottom plan used 41 seconds of CPU time and 416 KB of memory, which indicates the top plan compiled faster and more efficiently.
Overall, the top plan seems to be more efficient than the bottom plan due to lower estimated costs, and faster compile time.