Data and Information Systems Management

August 22, 2025

1 Query Optimization by Index Creation

The following query has been written to analyze the evolution of energy consumption over the years, for each parish for parishes with consumption above the average consumption of their respective municipality. For this task we perform a study of the Estimated Subtree Cost for every possible index that might be useful to optimize this query.

```
SELECT District, Municipality, Parish, Year, SUM(ActiveEnergy) AS
   ActiveEnergy
FROM Energy.MonthlyConsumption AS C
WHERE ActiveEnergy > (
SELECT AVG(ActiveEnergy)
FROM Energy.MonthlyConsumption AS D
WHERE D.Year = C.Year
AND D.Municipality = C.Municipality
GROUP BY Year, Municipality
)
GROUP BY District, Municipality, Parish, Year
ORDER BY Parish, Year;
```

Execution Plan and Index Analysis

The query in question performs a complex operation that requires:

- Calculating average energy consumption per municipality and year.
- Filtering parishes with above-average consumption.
- Aggregating consumption data by district, municipality, parish, and year.
- Sorting results by parish and year.

Before proposing any optimization strategy, we analyze the baseline execution plan:

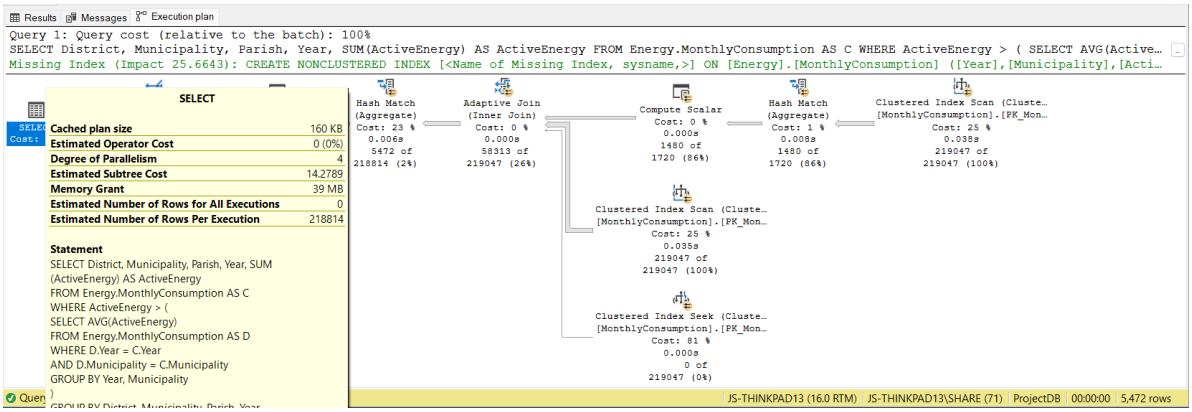

Figure 1: Initial execution plan obtained from the query.

Critical Performance Bottlenecks Identified:

- **Clustered Index Seek (Cost: 81%)** The most significant bottleneck occurs when comparing each row's ActiveEnergy against the calculated average. This operation requires lookup operations against the primary key that cannot efficiently filter based on our complex predicate.
- **Clustered Index Scan (Cost: 25%)** The query engine must scan the entire table to calculate aggregations and perform filtering. This scan operation demonstrates a fundamental inefficiency in the current index structure, as it lacks support for the specific data access patterns required by our query.
- **Hash Match (Cost: 23%)** The presence of hash aggregation indicates memory-intensive operations that could be optimized with proper indexing. The query engine must build hash tables in memory to perform the GROUP BY operations, which becomes costly as data volume increases.
- **Adaptive Join (Cost: 0%)** The optimizer dynamically selects between different join strategies, but its impact on performance is minimal in this context.

Proposed Indexes for Optimization

Index 1: Covering Index for Main Query Columns

```
-- Create the index
CREATE INDEX IX_Energy_MonthlyConsumption_Main
ON Energy.MonthlyConsumption (Parish, Year, Municipality, District,
    ActiveEnergy);

-- Drop the index
DROP INDEX IX_Energy_MonthlyConsumption_Main ON Energy.MonthlyConsumption;
```

This index applies the covering index pattern, which includes all columns required by the query to eliminate the need for key lookups or bookmark lookups against the clustered index.
A non-clustered index was chosen over a clustered index because:

- The existing clustered index likely supports other critical operations.
- This index structure specifically targets this analytical query pattern.
- The query benefits from the included columns rather than clustered organization.

Estimated Subtree Cost: 10.2074

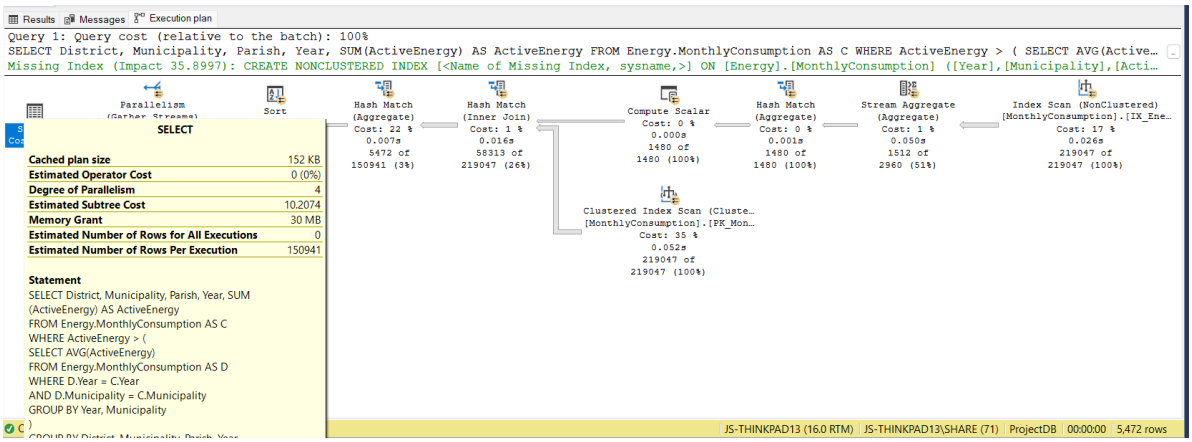

Figure 2: Execution plan applying index for main query columns.

The optimizer can now use the index for both data retrieval and sorting operations, eliminating the need for expensive sort operations.

Index 2: Index Focused on the Subquery Filter

```
-- Create the index
CREATE INDEX IX_Energy_MonthlyConsumption_AvgFilter
ON Energy.MonthlyConsumption (Year, Municipality, ActiveEnergy);

-- Drop the index
DROP INDEX IX_Energy_MonthlyConsumption_AvgFilter ON Energy.MonthlyConsumption
    ;
```

This index targets the specific access pattern of the subquery. By creating an index with Year and Municipality as leading columns followed by ActiveEnergy, we optimize for:

- Fast grouping operations in the subquery (GROUP BY Year, Municipality).
- Efficient aggregation of ActiveEnergy values for average calculations.
- Reduction of logical reads when calculating averages.

Estimated Subtree Cost: 11.1362

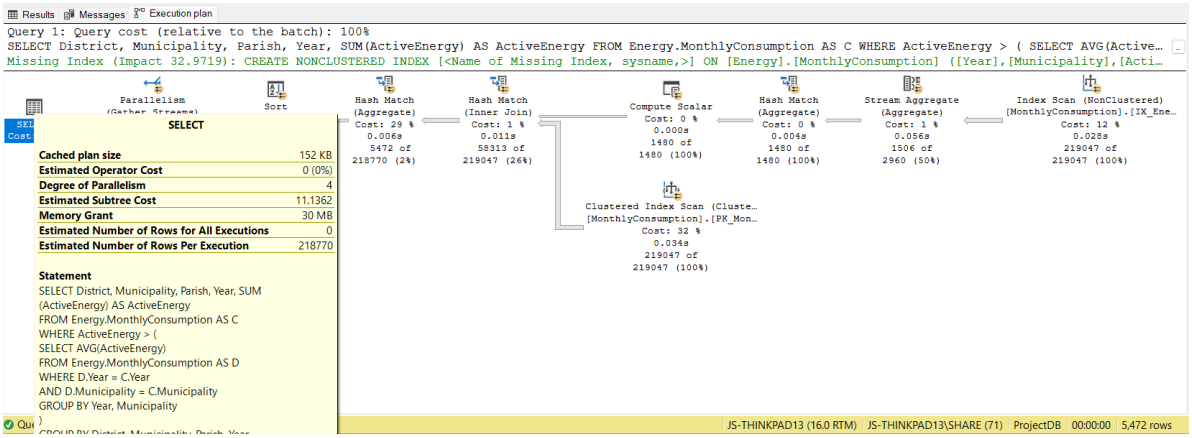

Figure 3: Execution plan applying index focused on subquery filter.

Index 3: Composite Index with Month Filter

```
-- Create the index
CREATE INDEX IX_Energy_MonthlyConsumption_MonthFilter
ON Energy.MonthlyConsumption (Month, Year, Municipality, Parish, District,
    ActiveEnergy);

-- Drop the index
DROP INDEX IX_Energy_MonthlyConsumption_MonthFilter ON Energy.
    MonthlyConsumption;
```

This index applies multiple optimization patterns simultaneously:

- Dimensional Hierarchy Organization: By ordering columns from Month → Year → Municipality → Parish → District, we create a natural hierarchy that aligns with common temporal and geographic analysis patterns.
- Selectivity Optimization: Including Month as the leading column increases the selectivity of the index, allowing the optimizer to work with smaller, more focused data segments.
- Data Distribution Awareness: The Month column likely has even distribution across 12 values, which helps the optimizer make better cardinality estimates and choose more efficient execution strategies.

Estimated Subtree Cost: 8.41069

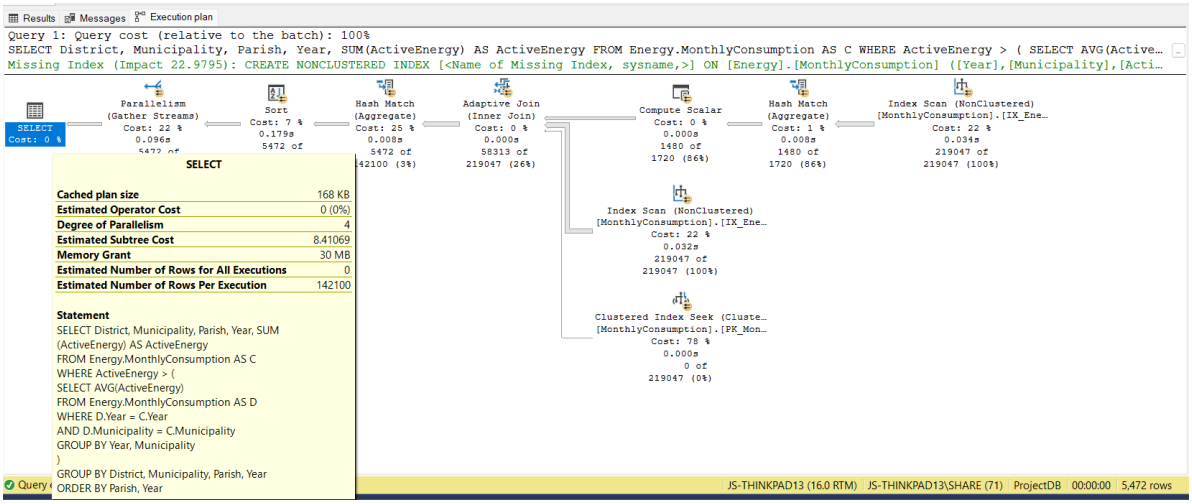

Figure 4: Execution plan applying composite index.

Index Name	Index Type	Subtree Cost	Improvement (%)
No Index (Baseline)	-	14.2789	-
IX_Energy_MonthlyConsumption_Main	Non-Clustered	10.2074	28.51%
IX_Energy_MonthlyConsumption_AvgFilter	Non-Clustered	11.1362	22.01%
IX_Energy_MonthlyConsumption_MonthFilter	Non-Clustered	8.41069	41.10%

Table 1: Execution Plan and Estimated Subtree Cost Analysis

2 Query Optimization with Materialized Views

The following query has been written to analyze the total energy consumption by parish, the total number of contracts by parish, and (based on those two) the average energy consumption per contract, in each parish.

```
SELECT Energy.DistrictMunicipalityParishCode,
       Energy.District,
       Energy.Municipality,
       Energy.Parish,
       Energy.ActiveEnergy,
       Contracts.NumberContracts,
       Energy.ActiveEnergy / Contracts.NumberContracts AS
          EnergyPerContract
FROM (SELECT DistrictMunicipalityParishCode,
             District,
             Municipality,
             Parish,
             SUM(ActiveEnergy) AS ActiveEnergy
      FROM Energy.MonthlyConsumption
      GROUP BY DistrictMunicipalityParishCode,
               District,
               Municipality,
               Parish) AS Energy,
```

```
     (SELECT DistrictMunicipalityParishCode,
             District,
             Municipality,
             Parish,
             SUM(NumberContracts) AS NumberContracts
      FROM Energy.ActiveContracts
      GROUP BY DistrictMunicipalityParishCode,
               District,
               Municipality,
               Parish) AS Contracts
WHERE Energy.DistrictMunicipalityParishCode =
      Contracts.DistrictMunicipalityParishCode
AND Energy.District IN ('Lisboa', 'Porto')
ORDER BY Energy.District,
         Energy.Municipality,
         Energy.Parish
```

To speed up the query, we create a materialized view for each nested subquery, and perform a study of the Estimated Subtree Cost for each possible join algorithm.

Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents. However, it is much cheaper in many cases to read the contents of a materialized view than to compute the contents of the view by executing the query defining the view. [AS19]

First we create the views for the Selection Energy and for the selection Contracts with the following queries.

```
CREATE VIEW dbo.vw_MonthlyConsumptionMaterialized
WITH SCHEMABINDING
AS
  SELECT
    DistrictMunicipalityParishCode,
    District,
    Municipality,
    Parish,
    COUNT_BIG(*) AS cnt,
    SUM(ActiveEnergy) AS ActiveEnergy
  FROM Energy.MonthlyConsumption
  GROUP BY
    DistrictMunicipalityParishCode,
    District,
    Municipality,
    Parish;
```

```
CREATE VIEW dbo.vw_ActiveContractsMaterialized
WITH SCHEMABINDING
AS
  SELECT
    DistrictMunicipalityParishCode,
    District,
    Municipality,
    Parish,
    COUNT_BIG(*) AS cnt,              -- Required for aggregate indexed views
    SUM(NumberContracts) AS NumberContracts
  FROM Energy.ActiveContracts
```

```
  GROUP BY
    DistrictMunicipalityParishCode,
    District,
    Municipality,
    Parish;
```

SQL Server does not allow the direct creation of materialized views so we simulate a materialized view using an indexed view. Indexed views require that you create the view with the WITH SCHEMABINDING option. One key requirement when using aggregation as in this case SUM is to include a counter column in the view's SELECT list. The code used for this process is:

```
CREATE UNIQUE CLUSTERED INDEX IX_vw_MonthlyConsumptionMaterialized
  ON dbo.vw_MonthlyConsumptionMaterialized
     (DistrictMunicipalityParishCode, District, Municipality, Parish);
```

```
CREATE UNIQUE CLUSTERED INDEX IX_vw_ActiveContractsMaterialized
  ON dbo.vw_ActiveContractsMaterialized
     (DistrictMunicipalityParishCode, District, Municipality, Parish);
```

Then we can see the views in the object explorer in SQL Server Studio. After creating this views, we execute again the initial query. In Figure 5 we can see the new execution plan. It reduces the total cost of the query by **52.5%**. The new total cost is **15.82**. The cost of the JOIN operation is higher but we eliminate the need to SORT, which was the most expensive task in the original query.

Nested-loop join

Now we have the views ordered by the sub-queries and the Server chooses to perform Nested Loop Join for this reason. When we switch to materialized views, the data being joined is pre-aggregated and stored physically. SQL Server use these new tools and available indexes on the materialized views to determine the most efficient join method. In this case, because the selections and data are smaller, it now chooses a nested loop join over a hash join that is generally better for larger, unsorted inputs.

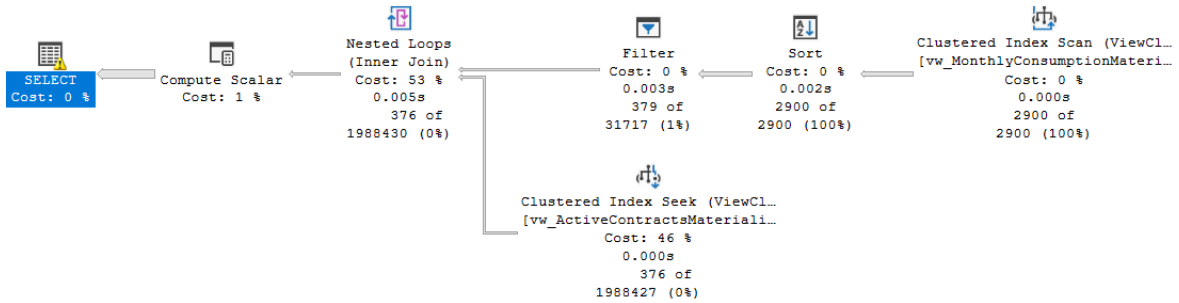

Figure 5: Execution Plan with Materialized View (Nested-Loop)

Hash join

If we force the Server to use the Hash Join, the execution plan looks very similar to the initial execution plan for question 3 (Figure 6). The difference is that now there is not aggregation step for the sub-queries as there are now materialized views in which the aggregation SUM is calculated. The total cost of the query now is **27.62**. Why is this SORT operation appears here

and not with the original query? The Server's query optimizer is doing something clever. When we execute the original query without any JOIN hints, SQL Server analyzes the indexes on your materialized views and realize inputs are already sorted in the order the query needs. In this case by forcing the HASH, we are not preserving order at all so it has to be order after.

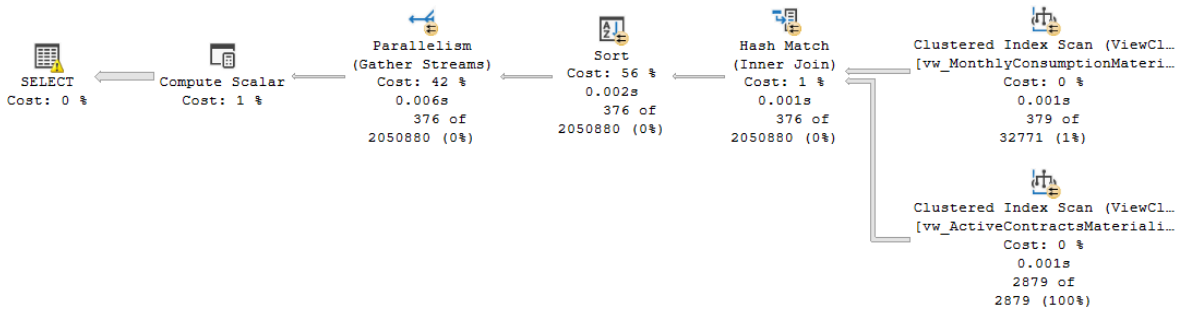

Figure 6: Execution Plan with Materialized View - Hash Join

Merge join

As in the previous here we have this additional SORT operation. The MERGE JOIN can preserve order if inputs are perfectly sorted on the join key, but for this case, ORDER BY is on different columns (District, Municipality, Parish), so it doesn't align and SQL Server adds a SORT. The total cost of the query is **28.20**.

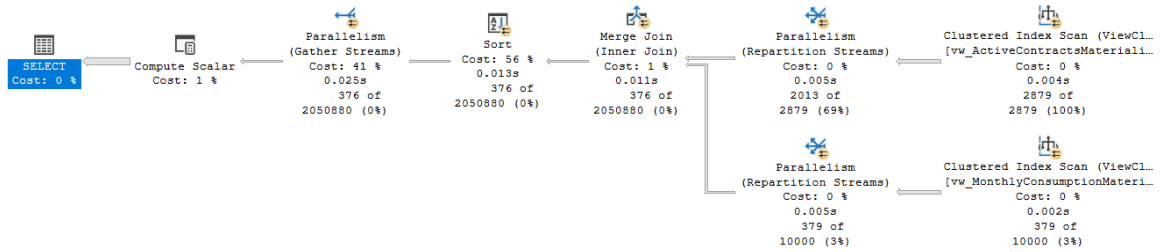

Figure 7: Execution Plan with Materialized View - Merge Join

3 Analysis with Database Engine Tuning Advisor

For this task we use the queries presented before as workload, we run Database Engine Tuning Advisor until it produces a set of recommendations that include, among other things, the two materialized views created and at least one index to optimize the query number 1. We also provide step-by-step instructions to produce those recommendations from the initial database.

Step-by-step instructions

1. Create a file called `workload.sql` containing the queries from tasks 2 and 3.
2. Launch the Database Engine Tuning Advisor from the Start menu and connect to SQL Server.
3. In the General tab, under Workload, select `File` and browse to the location of `workload.sql` created in step 1. In `Database for workload analysis`, select the project database.

4. In the Tuning Options tab, uncheck `Limit tuning time`. Under `Physical Design Structures (PDS) to use in database`, select `Indexes` and `Indexed Views`. Under `Physical Design Structures (PDS) to keep in database`, select `Do not keep any existing PDS`.

5. Click the `Start Analysis` button in the toolbar.

After applying these steps, the results are as follows:

- One index recommended to be created.
- Two materialized views are recommended.
- Estimated improvement of 36%.

The recommended index is shown in Figure 8, and it matches one of the indexes suggested in Question N°2. In figures 9 and 10 we can observe the recommended materialized views, similar to the ones proposed for both selections "Energy" and "Contracts" from query in Question N°3.

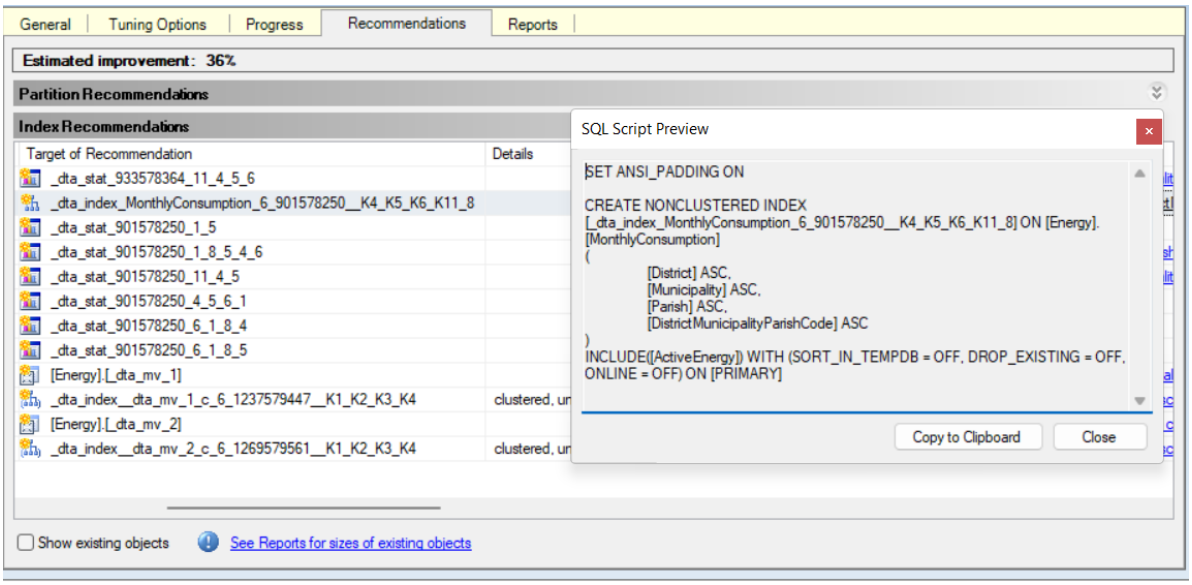

Figure 8: Recommended Index

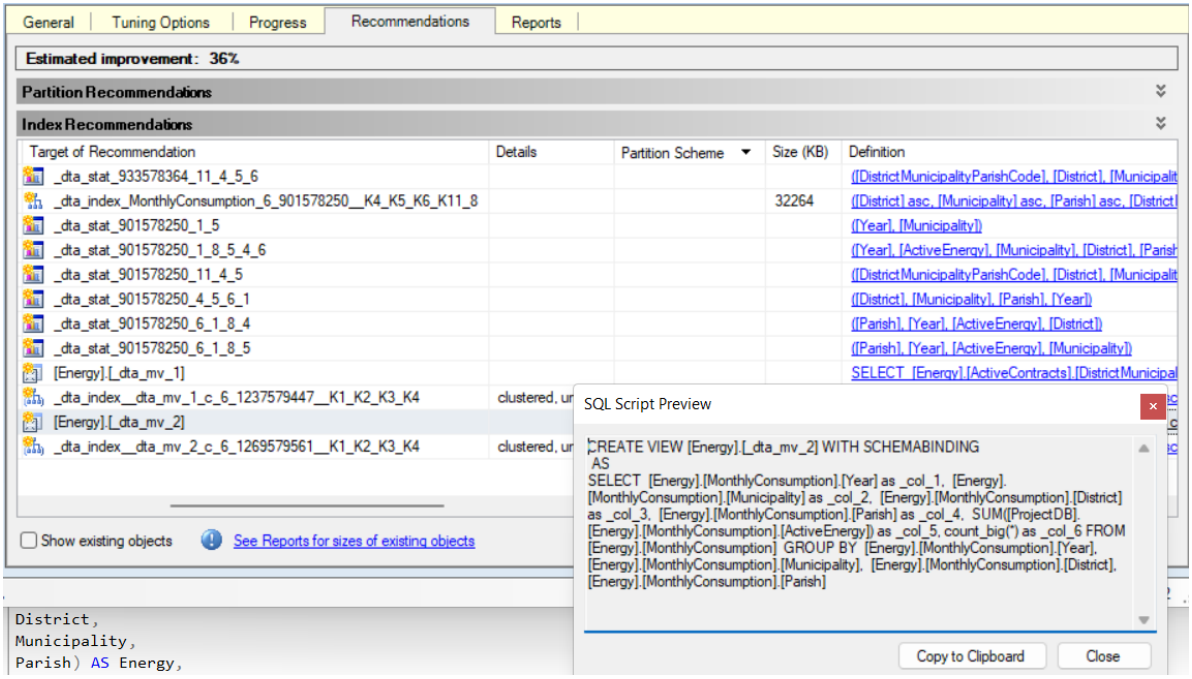

Figure 9: Recommended Materialized View 1 - Energy

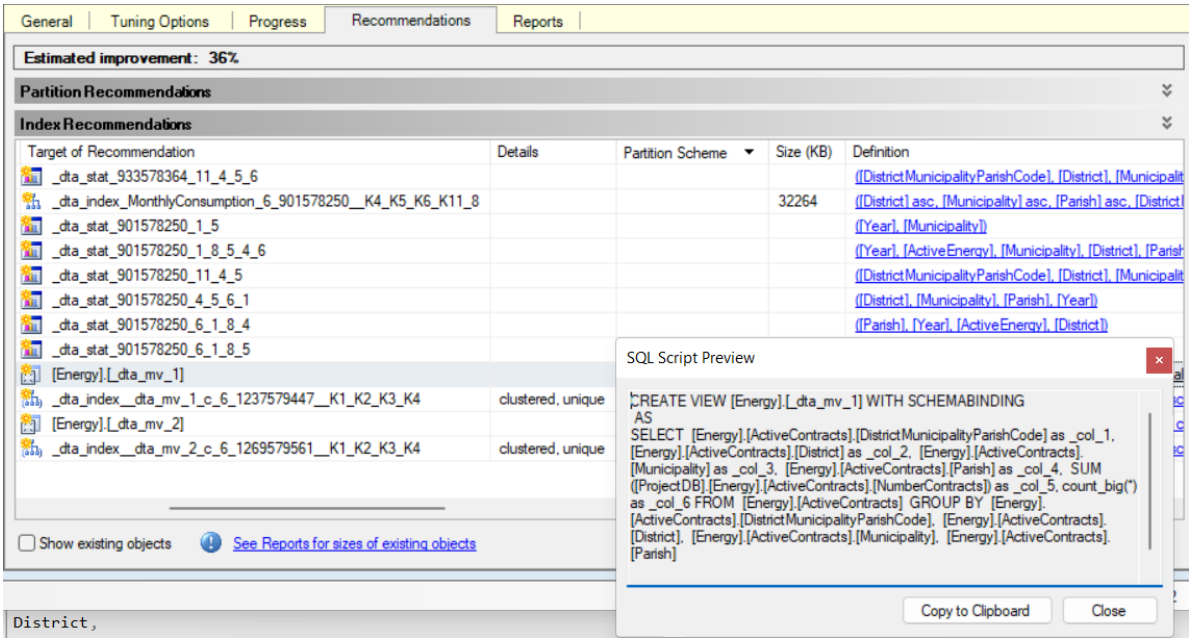

Figure 10: Recommended Materialized View 2 - Contracts

To measure the performance of the recommendations, next we apply them and observe the the new estimated Subtree Costs for each query. To do this, select the recommended changes (index and materialized views) and click `Apply` followed by `Apply Recommendations`.

Improvement Analysis

Query Question N°2

For the query in Question N°2, before applying recommendations we have the initial execution plan shown in Figure 11, same as we had before. The initial conditions we had for the query are the following:

- Estimated Subtree Cost: 14.2789.
- The execution plan relies heavily on clustered index scans (seen in the rightmost operations).A Hash Match join is used, which suggests that SQL Server is scanning large portions of the table instead of using an efficient seek operation.
- The number of rows processed is high (218,814 rows), indicating that the query has to traverse a significant portion of the dataset.

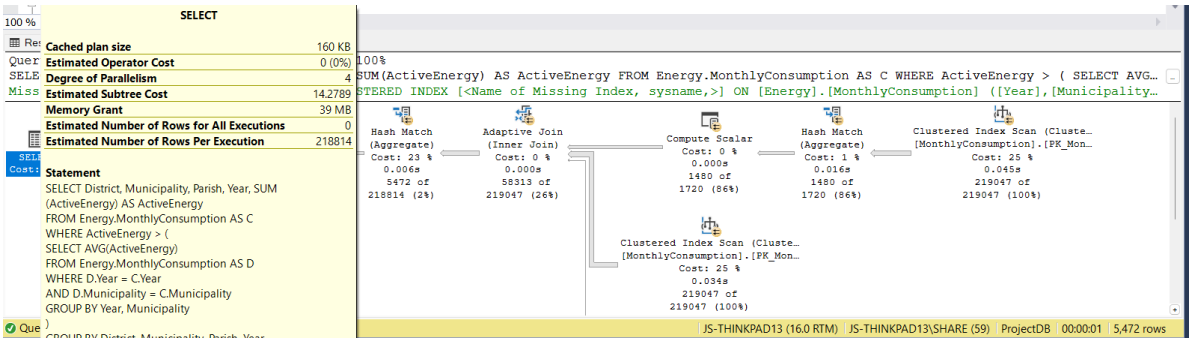

Figure 11: Execution Plan and Estimated Subtree Cost for the query in task 2 before recommendations

After applying the recommended index we have the execution plan shown in Figure 12. The final conditions are:

- Estimated Subtree Cost: 2.72708 (The Estimated Subtree Cost was reduced by approximately 80.89%, indicating a substantial improvement in query performance.)
- Instead of a Clustered Index Scan, we now see an Index Scan (NonClustered), which means SQL Server is using the newly created index to retrieve data more efficiently.
- The number of rows processed per execution dropped significantly to 14,500 compared to 218,814 previously.
- The query plan still contains a Hash Match join, but the overall cost is lower because SQL Server is scanning far fewer rows.

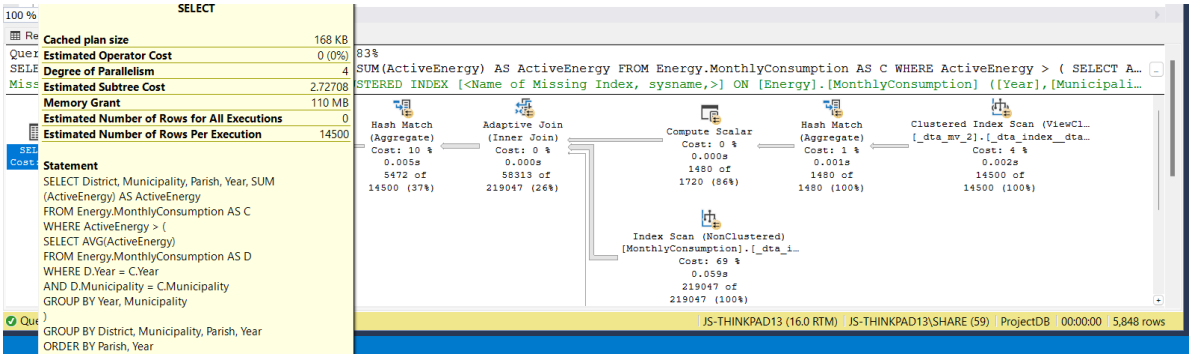

Figure 12: Execution Plan and Estimated Subtree Cost for the query in Question N°2 after recommendations

Indexes speed up data retrieval by reducing the amount of data scanned during query execution. Without an index, SQL Server performs a large table scan, which is inefficient for large datasets. This demonstrates how a well-designed index can significantly reduce query execution time by improving data access patterns.

Query Question N°3

For the query in Question N°3, before applying recommendations we have the initial execution plan shown in Figure 13, same as we had before. The initial conditions we had for the query are the following:

- Estimated Subtree Cost: 37.1006
- Estimated Number of Rows per Executions: 1986580
- The execution plan shows Clustered Index Scans, meaning the query
- is scanning entire tables instead of efficiently retrieving the required rows.
- The Hash Match join indicates that SQL Server is joining large amounts of data without an optimized index.
- The Memory Grant is 401MB, suggesting that SQL Server requires significant memory to process the query.

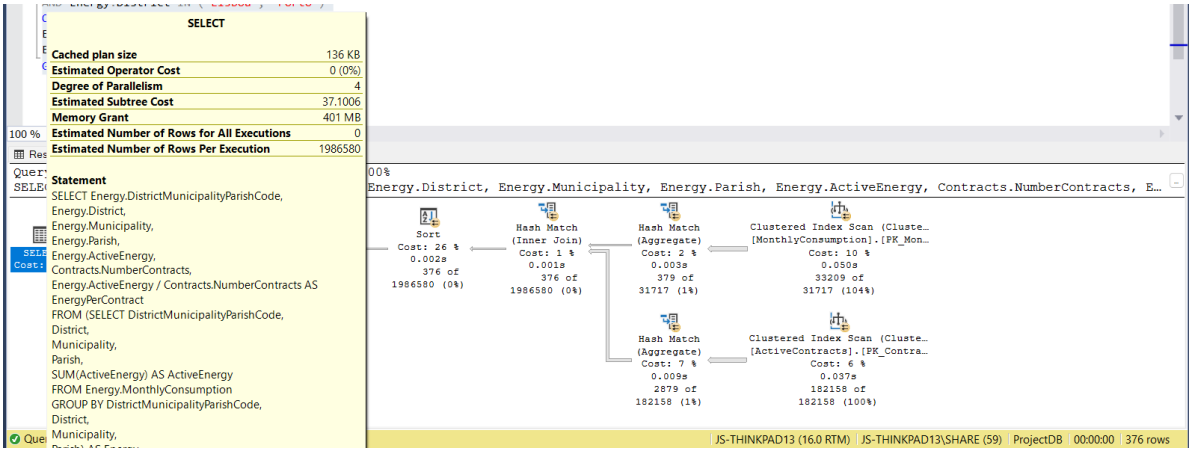

Figure 13: Execution Plan and Estimated Subtree Cost for the query in Question N°3 before recommendations

After applying the recommended views and the recommended index we have the execution plan shown in Figure 14. The difference in improvement from what we had in Question N°4 is that here we also applied the index and it impacted the query performance by a huge amount. The final conditions are:

- Drastic Reduction in Estimated Subtree Cost: The subtree cost dropped from 37.1006 to 0.562462. The Estimated Subtree Cost was reduced by approximately 98.48%, indicating a significant improvement in query efficiency.
- Significant Decrease in Estimated Rows: The number of rows processed was reduced from 1,986,580 to 20,198, implying better filtering or indexing.

- Reduced Cached Plan Size: The size of the cached plan decreased from 136 KB to 48 KB, which indicates a more streamlined execution plan. A Stream Aggregate is used instead of a Hash Match, indicating that SQL Server can now aggregate data more efficiently.

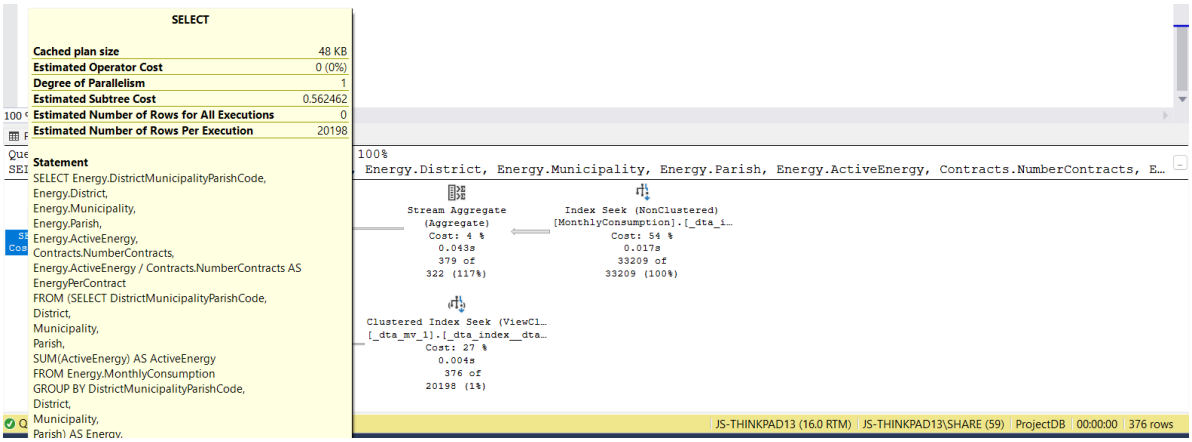

Figure 14: Execution Plan and Estimated Subtree Cost for the query in Question N°3 after recommendations

References

[AS19] S. Sudarshan Abraham Silberschatz, Henry F. Korth. *Database System Concepts*. McGrawHill, 2019.