TÉCNICO LISBOA

Data Administration in Information Systems Project Report

Marta Marques Félix
ist 199276

Nuno Miguel Ribeiro
ist 199293

2nd Semester

1 Task 1

The first task required the implementation of a partition scheme for the *MonthlyConsumption* table, in a way that the records for each year are stored in their own partition.

1.1 Find the years in the *MonthlyConsumption* Table

We started by discovering the years using the query:

```
SELECT DISTINCT [YEAR]
FROM [ProjectDB].[Energy].[MonthlyConsumption]
```

The output from that query is in the table 1.

	Year
1	2020
2	2021
3	2022
4	2023

Table 1: Results from query 1.1

1.2 Create the partition function and scheme for each year

So we knew there were only 4 different years (2020, 2021, 2022, 2023), and that we needed 4 partitions. The following SQL queries create the Partition Function responsible to divide the data into the different partitions and create the scheme that maps those partitions with the partition function.

```
CREATE PARTITION FUNCTION MonthlyConsumption_Years (CHAR(4))
AS RANGE RIGHT FOR VALUES (2021, 2022, 2023);

CREATE PARTITION SCHEME MonthlyConsumption_PartitionScheme
AS PARTITION MonthlyConsumption_Years ALL
TO ([Primary]);
```

1.3 Alter the *MonthlyConsumption* table to a new index

Alter the existing table to use the partition.

```
ALTER TABLE Energy.MonthlyConsumption DROP CONSTRAINT [PK_MonthlyConsumption];

CREATE CLUSTERED INDEX [IX_MonthlyConsumption_Partitioned] ON Energy.MonthlyConsumption(Year)
```

```
ON MonthlyConsumption_PartitionScheme(Year);
```

1.4 Verify number of records per partition

Run the last query to count the number of records loaded into each partition

```
SELECT
    $PARTITION.MonthlyConsumption_Years(Year) AS PartitionNumber,
    COUNT(*) AS NumberOfRecords
FROM
    Energy.MonthlyConsumption
GROUP BY
    $PARTITION.MonthlyConsumption_Years(Year);
```

The table 2 stores the results of the number of records on each one of the four partitions. Partition Number 1 is for year 2020, Partition Number 2 is for year 2021, Partition Number 3 is for year 2022 and Partition Number 4 is for year 2023.

Partition Number	Number of Records
1	8882
2	53304
3	53394
4	48882

Table 2: Results from query 1.4

2 Task 2

The query below:

```
SELECT [Parish], [Year], SUM([ActiveEnergy]) AS [ActiveEnergy]
FROM [Energy].[MonthlyConsumption]
WHERE [Municipality] = 'Lisboa'
 AND [Month] = '06'
GROUP BY [Parish], [Year]
ORDER BY [Parish], [Year]
```

Selects the Parish, Year, and sums of the ActiveEnergy for each group of Parish and Year in the *MonthlyConsumption* table. It filters rows where Municipality is equal to 'Lisboa' and Month is equal to '06', the month of June. Then, the results are grouped by Parish and Year, and ordered by Parish and Year as well. We will analyze the impact different indices have on the cost of this query.

2.1 Running the Query with index on the primary key

```
SELECT [Parish], [Year], SUM([ActiveEnergy]) AS [ActiveEnergy]
FROM [Energy].[MonthlyConsumption]
WHERE [Municipality] = 'Lisboa'
 AND [Month] = '06'
GROUP BY [Parish], [Year]
ORDER BY [Parish], [Year]
```

In Figure 1 we have the execution plan of the query above.

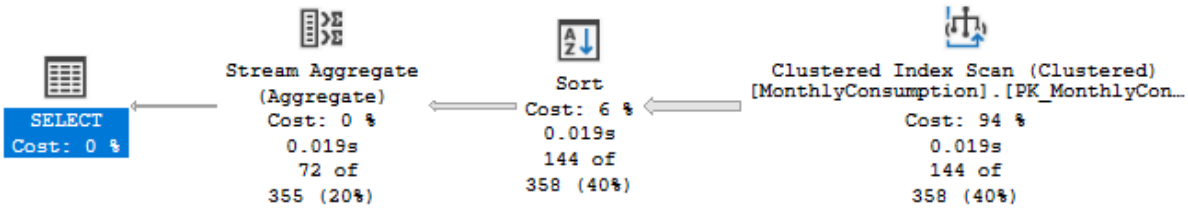

Figure 1: Execution plan of 2.1

Estimated Tree Cost: 2.606

Taking a look at the execution plan, we see that the system starts by doing an Clustered Index Scan on the *PK_MonthlyConsumption*, the primary key. Then it Sorts, which is the ORDER BY of the query, arranges the rows in order. After that, the Stream Aggregate, which is the GROUP BY, groups the rows. With a Clustered Index Scan, the system will have to read through entire table to get to relevant rows.

However, this could be more efficient. In the subsequent sections, we will study how different indices affect the performance, and which ones are the best.

2.2 Running the Query with IX_Municipality

The query is searching by two columns: Municipality and Month, so I makes sense to create an index on them. First, we will try to run the query with an index in a single column, Municipality, that is created with:

```
CREATE INDEX IX_Municipality ON [Energy].[MonthlyConsumption] ([Municipality]);
```

The execution plan follows the same lines as the previous one. However, the Estimated Tree Cost is significantly larger.

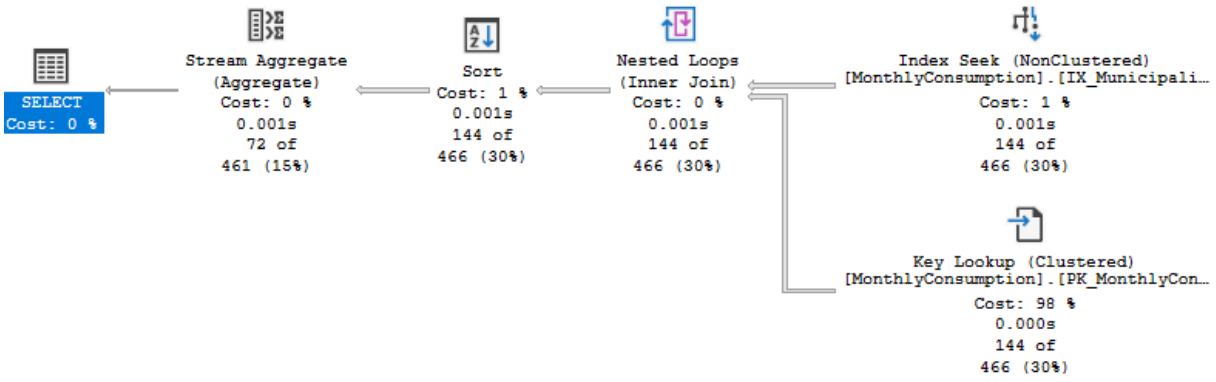

Figure 2: Execution Plan of 2.2

Estimated Tree Cost: 1.457

This index decreases the cost, because our query filters by two columns, Month and Municipality,and we have an index that satisfies part of that, one column, in this case Municipality.

In the Key Lookup step, the system looks up information about Parish and ActiveEnergy, which are needed for the SELECT, GROUP BY and ORDER BY.

Dropping the index, so we can explore other options, with:

```
DROP INDEX IX_Municipality ON [Energy].[MonthlyConsumption];
```

2.3 Running the Query with IX_Month

Trying the same strategy, this time only on the Month column:

```
CREATE INDEX IX_Month ON [Energy].[MonthlyConsumption] (Month);
```

This index is very inefficient, so the system does not want to use it. To force it with a little change to query, a WITH clause.

```
SELECT [Parish], [Year], SUM([ActiveEnergy]) AS [ActiveEnergy]
FROM [Energy].[MonthlyConsumption]
WITH (INDEX(IX_Month))
WHERE [Municipality] = 'Lisboa'
 AND [Month] = '06'
GROUP BY [Parish], [Year]
ORDER BY [Parish], [Year]
```

And so we find a worst performing index, as we can see by the high Estimated Tree Cost of this execution, Figure 3.

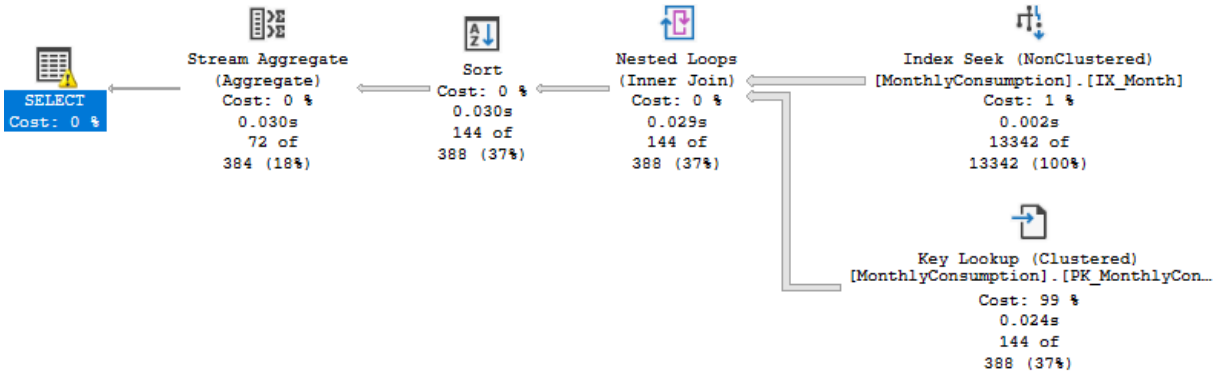

Figure 3: Execution Plan of 2.3

Estimated Tree Cost: 11.679

This index performs worse than the one only on Municipality, because Months is a column with few distinct values (only 12). Due do this, many rows share the same Month and the index does not narrow down enough, meaning that there are still many rows that need scanning.

Dropping the index:

```
DROP INDEX IX_Month ON [Energy].[MonthlyConsumption];
```

2.4 Running the Query with IX_Month_Municipality

Seen as the query searches by two columns, it's logical the index should be on those two columns, and it's expected to yield better results. This index on both Month and Municipality is created with :

```
CREATE NONCLUSTERED INDEX IX_Month_Municipality ON [Energy].[MonthlyConsumption]
    ([Month],[Municipality]);
```

In Figure 4 we have the execution plan of the query above, where the Estimated Tree Cost increase significantly, in comparison to the one in Figure 4.

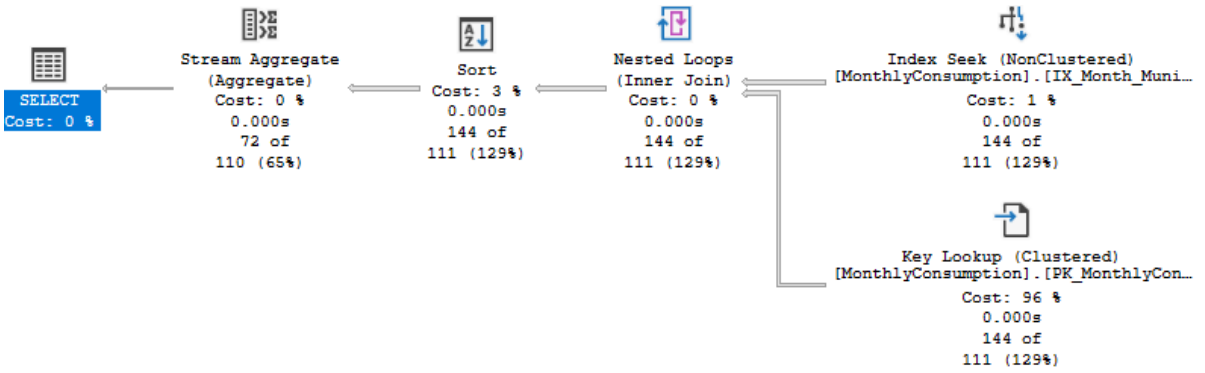

Figure 4: Execution plan of 2.4

Estimated Tree Cost: 0.37

The improvement is huge. The index that was created is a Composite Index, which means that it is created on two columns, providing even faster access to data entries by utilizing multiple columns within an index.

Drop the index, so we can continue the analysis.

```
DROP INDEX IX_Month_Municipality ON [Energy].[MonthlyConsumption];
```

2.5 Running the Query with IX_Municipality_Month

In composite indices, the order of the columns can have impacts on the performance. Switching the order, we will now try creating on Municipality first and then Month, to see if we get any improvements.

```
CREATE NONCLUSTERED INDEX IX_Municipality_Month ON [Energy].[MonthlyConsumption]
    ([Municipality],[Month]);
```

The query's execution plan, depicted in Figure 5, closely resembles the previous one shown in Figure 4. Additionally, the estimated tree cost is only marginally higher.

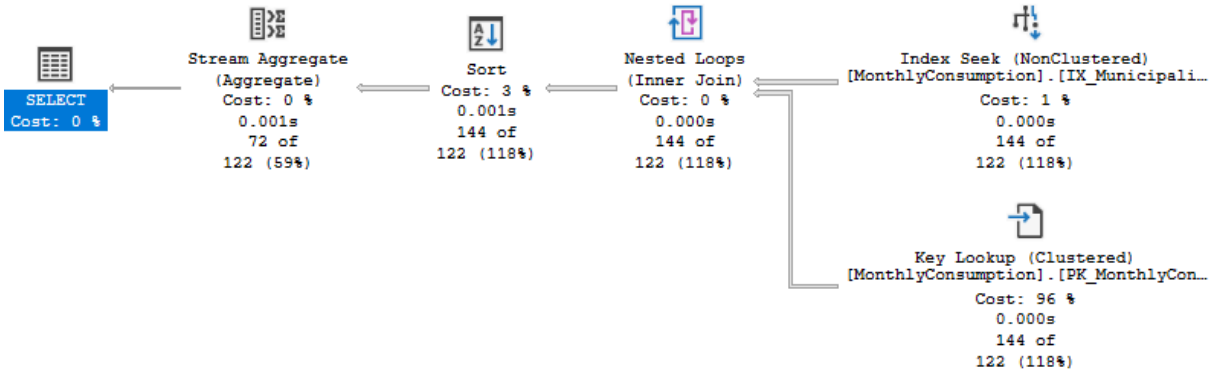

Figure 5: Execution plan of 2.5

Estimated Tree Cost: 0.408

Switching the order of the columns made the index slightly worse, as it affects how the index sorts and searches the columns within them.

Dropping the index with:

```
DROP INDEX IX_Municipality_Month ON [Energy].[MonthlyConsumption];
```

2.6 Running the Query with Index IX_Month_Municipality_Includes

It's possible to use the Includes Clause to add columns to the leaf level of our non-clustered index, which means the system doesn't need to lookup the data base again, improving performance. This is called, a Covering Index. Also, the system won't search by the columns in the Includes. We create the index using:

```
CREATE NONCLUSTERED INDEX IX_Month_Municipality_Includes ON [Energy].[MonthlyConsumption]
    ([Month],[Municipality]) INCLUDE ([Parish], [ActiveEnergy]);
```

In Figure 6 we have the execution plan that resulted from the query.

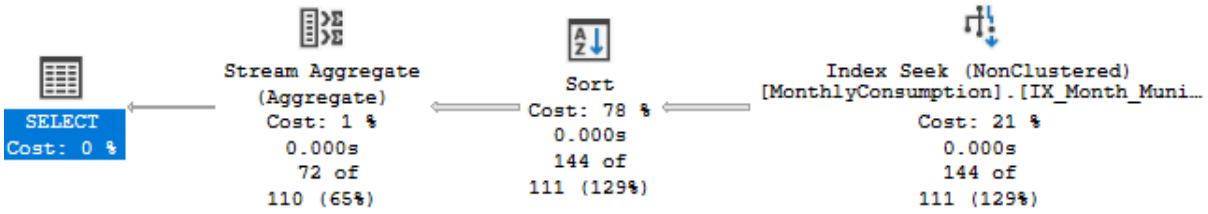

Figure 6: Execution plan of 2.6

Estimated Tree Cost: 0.0168

This time on the Execution Plan there isn't an Index Scan, instead there is a Index Seek. In the Index Seek, the system uses the b-tree structure of the index to seek directly to matching records. In this case, an Index Seek is faster than an Index Scan. Since we have the information from the Includes in the index, the system doesn't have to look them up, improving performance.

To drop the Index so we can analyze the others, we use:

```
DROP INDEX IX_Month_Municipality_Includes ON [Energy].[MonthlyConsumption];
```

2.7 Running the Query with IX_Municipality_Month_Parish_ActiveEnergy

The index we are creating now contains every column the last one has, but instead of Including Parish and Active Energy, the columns are on the index:

```
CREATE NONCLUSTERED INDEX IX_Month_Municipality_Parish_ActiveEnergy ON
    [Energy].[MonthlyConsumption] ([Month],[Municipality], [Parish],[ActiveEnergy]);
```

This execution, Figure 7 has the Estimated Tree Cost is 0.0161, meaning it's the one that performs the best.

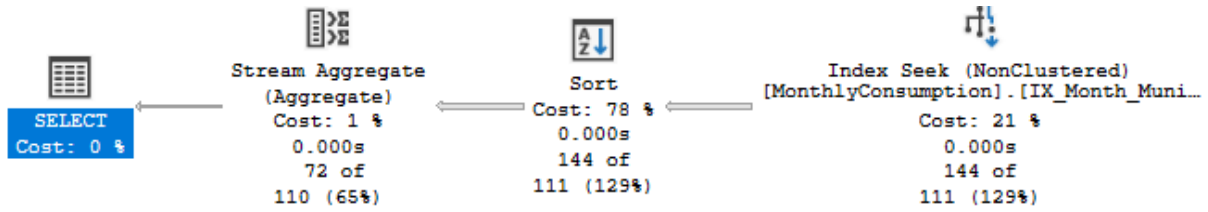

Figure 7: Execution Plan of 2.7

Estimated Tree Cost: 0.0161

This is explained by having an index that includes almost every column used in the query. Although this index may be better in this case, it may overkill if the query isn't using columns [Month],[Municipality], [Parish], [ActiveEnergy] at the same time.

To drop the Index so we can move on:

```
DROP INDEX IX_Month_Municipality_Parish_ActiveEnergy ON [Energy].[MonthlyConsumption];
```

2.8 Running the Query with IX_Municipality_Month_Parish_ActiveEnergy_Year

However, Year is used in the SELECT, GROUP BY and ORDER BY, so now the index we are creating now contains every column:

```
CREATE NONCLUSTERED INDEX IX_Month_Municipality_Parish_ActiveEnergy_Year ON
    [Energy].[MonthlyConsumption] ([Month],[Municipality], [Parish],[ActiveEnergy], [Year]);
```

This execution, Figure 8 has the Estimated Tree Cost is 0.0167.

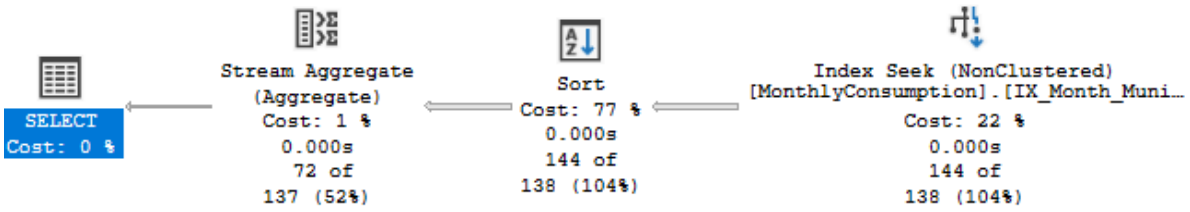

Figure 8: Execution Plan of 2.8

Estimated Tree Cost: 0.0167

Turns out this index is slightly worse than the previous, but still very good. Worth noting that, seeing every combination of these 5 columns, we could reach an index that performs even better.

We drop it with:

```
DROP INDEX IX_Month_Municipality_Parish_ActiveEnergy_Year ON [Energy].[MonthlyConsumption];
```

3 Task 3

3.1 Introduction

In this exercise, we will need to analyze multiple algorithms for JOIN and GROUP BY operations and how to combine.

3.1.1 Algorithms for JOIN Operations

Join operations are fundamental in database management systems for combining data from multiple tables. Several algorithms facilitate this process:

- **Nested Loops Joins:** This straightforward method is efficient when one table is small and appropriately indexed for the join condition.
- **Merge Join:** Both tables must be sorted based on the join key beforehand. This approach is beneficial for large databases and performs optimally with proper indexing.
- **Hash Join:** Constructing a hash table on the smaller input table enables efficient inner joins. This algorithm shines when one table significantly outweighs the other in size.

3.1.2 Algorithms for GROUP BY Operations

Grouping data is essential for summarizing and aggregating information. Various algorithms cater to this requirement:

- **Hash Aggregate:** Hashing the grouping columns and subsequently aggregating the data based on hash values. It excels when dealing with numerous groups and uneven data distributions.
- **Stream Aggregate:** This algorithm processes data in a single pass, calculating aggregates on the fly. It proves efficient when the entire result set cannot fit into memory.

By understanding these algorithms, we can optimize query performance and enhance database management efficiency.

3.2 Default Execution: Hash Join with Stream Aggregation

We start by executing the given query to analyse the default join and group-by algorithms defined by the software:

```
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]
ORDER BY [Energy].[District],
 [Energy].[Municipality],
 [Energy].[Parish];
```

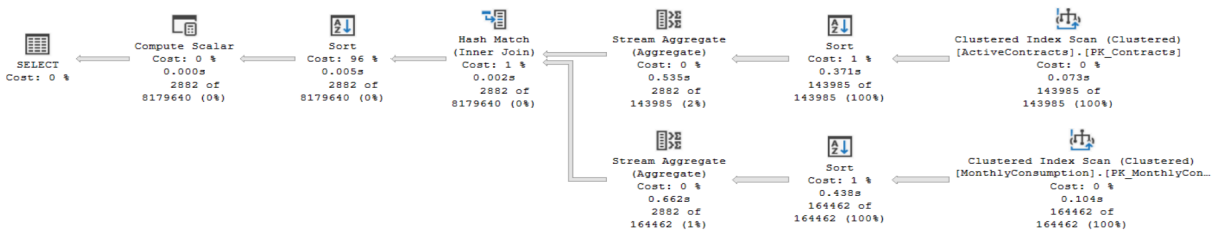

Figure 9: Execution Plan of 3.1

Estimated Tree Cost: 529.548

Upon reviewing the execution plan of this query, it becomes evident that the program opted for the Stream Aggregate algorithm for the GROUP BY operation and the Hash Join algorithm for the JOIN operation.

3.3 Hash Joins with Hash Groups

To compel the program to employ the Hash Match Aggregation for the Group By operations, it is necessary to append the following line to the original query:

```
OPTION (HASH GROUP);
```

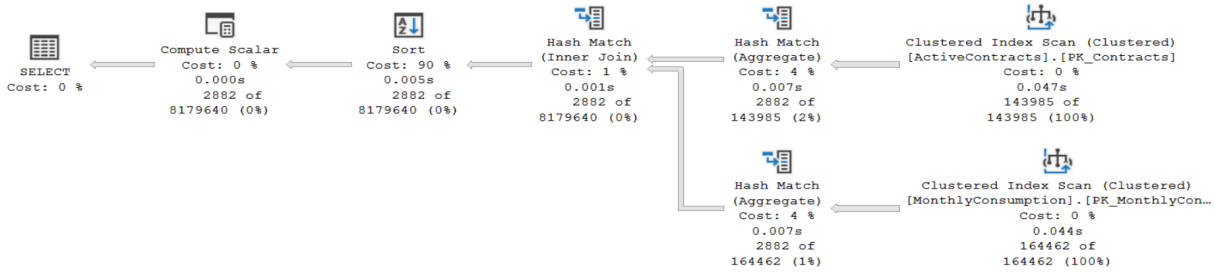

Figure 10: Execution Plan of 3.2

Estimated Tree Cost: 562.016

In this scenario, the program initiates by executing the Clustered Index Scan, then transitions directly to the Hash Match Aggregation, eliminating the need for any intermediary sorting step. Subsequently, the JOIN operation employs the Hash Match algorithm, akin to the example in section 3.1, and the execution concludes with the Sort step.

3.4 Nested Loops Joins with Order Group

This is the simplest join algorithm. It loops through each row of the first table and matches it with each row of the second table. It's efficient for small tables or when joining on indexed columns.
We force the query use this join with:

```
OPTION (LOOP JOIN);
```

OPTION (LOOP JOIN, ORDER GROUP) could be used, but by default the system will Merge group with this query and join.

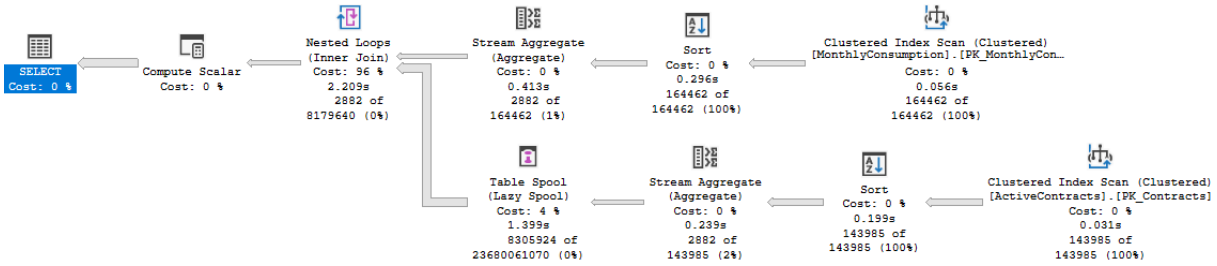

Figure 11: Execution Plan of 3.3

Estimated Tree Cost: 110389

Similar to the previous execution, the system does a Clustered Index Scan, Sort, Stream Aggregate on the PK_MonthlyConsumption, on MonthlyConsumption table. On PK_Contracts, on ActiveContracts table, there is also Clustered Index Scan, Sort, Stream Aggregate.

We also get a new step, Table Spool. That is a performance step, the operator keeps a copy of all data in a temporary database, so that it can return extra copies of rows without having to re-execute operators.

After that, the Nested Loop Join happens, comparing each record in MonthlyConsumption to every record in ActiveContracts. No Sort is needed after.

3.5 Nested Loops Joins with Hash Aggregate

To force the query to use the Nested Loop Join and the Hash Group, we need to add the following clause:

```
OPTION (LOOP JOIN, HASH GROUP);
```

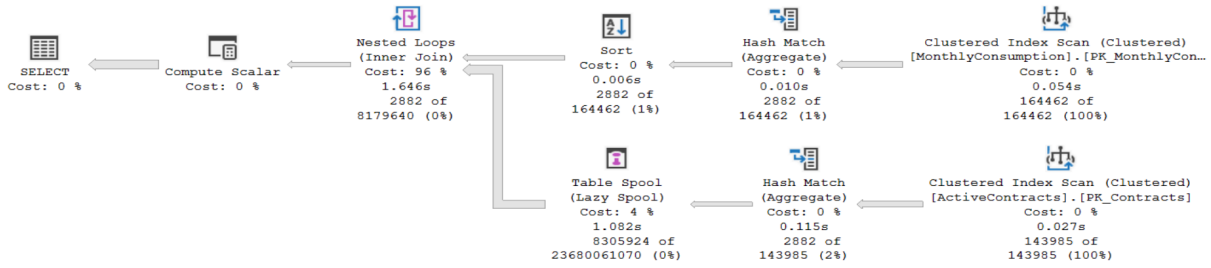

Figure 12: Execution Plan of 3.4

Estimated Tree Cost: 110443

In this execution, the system begins by performing a Clustered Index Scan on the MonthlyConsumption table, followed by a Hash Match Aggregation. Subsequently, it proceeds to execute a Sort operation on the data from the MonthlyConsumption table.

On the ActiveContracts table, the system again initiates with a Clustered Index Scan, then directly proceeds to a Hash Match Aggregation. Additionally, it executes a Table Spool step, also known as Lazy Spool.

Finally, the execution continues with the Nested Loop Join algorithm. Notably, in this particular execution, the system does not involve any sorting of data from the ActiveContracts table.

3.6 Merge Join with Order Groups

To test the merge algorithm for the JOIN Operation along with the stream aggregation algorithm for the GROUP BY, the following clause needs to be added:

```
OPTION (MERGE JOIN, ORDER GROUP);
```

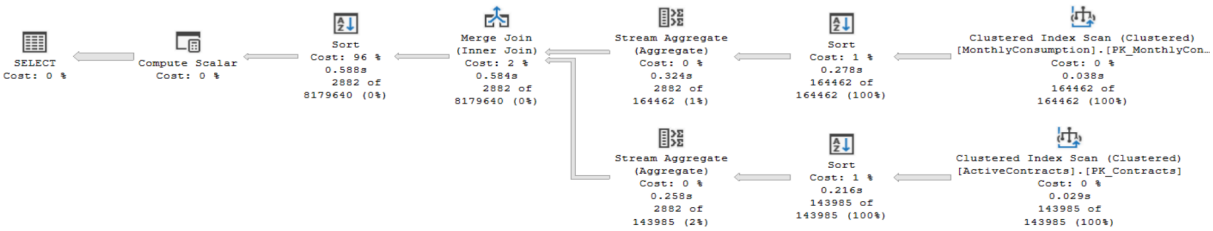

Figure 13: Execution Plan of 3.5

Estimated Tree Cost: 1415.7

This execution commences with the Clustered Index Scan, Sort, and Stream Aggregation operations on the data retrieved from both the MonthlyConsumption and ActiveContracts tables.

Subsequently, for the JOIN operation, the system employs the merge algorithm. This algorithm pairs rows from two appropriately sorted input tables, leveraging their sort order.

Following this, the system repeats the Sort step to process the output from the Merge Join.

3.7 Merge Join with Hash Aggregation

Finally, to evaluate the merge algorithm for the JOIN Operation alongside the hash match aggregation algorithm for the GROUP BY, the following clause must be incorporated:

```
OPTION (MERGE JOIN, HASH GROUP);
```

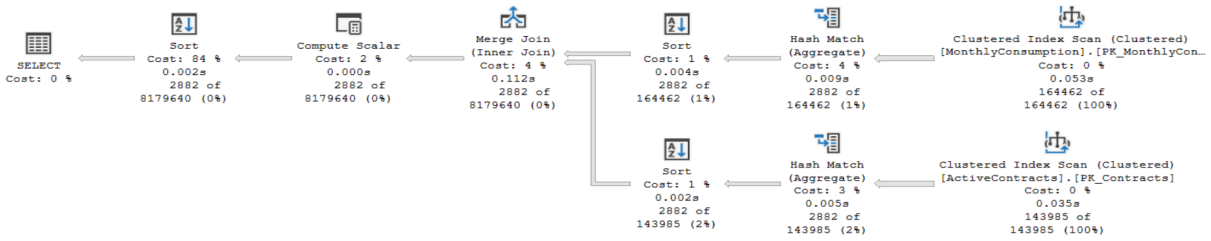

Figure 14: Execution Plan of 3.6

Estimated Tree Cost: 635.31

This execution closely resembles the 3.5). It starts with a Clustered Index Scan, followed by a Sort step after the GROUP BY operation, utilizing the hash aggregation algorithm on both the MonthlyConsumption and ActiveContracts tables.

The final steps mirror those outlined previously, with the execution of the JOIN employing the merge algorithm, culminating in a final Sort step.

3.8 Resume

The comparison of results from all previous combinations can be found in table 3.

Join Algorithm	Group By Algorithm	Estimated Tree Cost
Hash Join	Stream Aggregation	529.548
Hash Join	Hash Aggregation	562.016
Nested Loop Join	Stream Aggregation	110389
Nested Loop Join	Hash Aggregation	110443
Merge Join	Stream Aggregation	1415.7
Merge Join	Hash Aggregation	635.31

Table 3: Comparison form results of Q3

As evident from the table 3, the tool defaulted to selecting the combination of algorithms that yielded the lower Estimated Tree Cost, which in this case is

4 Task 4

4.1 Identify the nested queries

We commenced by identifying the nested subqueries within the main query analyzed in the preceding question.

The initial subquery pertains to the table Energy.MonthlyConsumption, tasked with calculating the SUM of Active Energy by parish and organizing the outcomes alphabetically by district, municipality, and parish.

```
SELECT [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish],
 SUM([ActiveEnergy]) AS [ActiveEnergy]
 FROM [Energy].[MonthlyConsumption]
 GROUP BY [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish]
```

The second subquery is associated with the table Energy.ActiveContracts and is responsible for computing the SUM of contracts, also by parish and organizing the results by district, municipality, and parish.

```
SELECT [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish],
 SUM([NumberContracts]) AS [NumberContracts]
 FROM [Energy].[ActiveContracts]
 GROUP BY [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish]
```

4.2 Creating the views

For the materialized views' creation, we used **SCHEMABIDING** tag to prevent the modification of the schema of the underlying objects that would affect the view, and optimize the performance of the query, and COUNT BIG tag was used instead of the COUNT, due to limitations raised by the Microsoft SQL Server Management Studio. To finally materialize the view, we need to create a clustered index on it.

4.2.1 View: vContractsByLocation

The query used to create the view of the Energy.ActiveContracts table was:

```
CREATE VIEW Energy.vContractsByLocation WITH SCHEMABINDING AS
SELECT [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish],
 SUM([NumberContracts]) AS [NumberContracts],
 COUNT_BIG(*) AS [ResultCounting]
 FROM [Energy].[ActiveContracts]
 GROUP BY [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish]
```

And for the index of this view:

```
CREATE UNIQUE CLUSTERED INDEX IX_vContractsByLocation
ON Energy.vContractsByLocation(DistrictMunicipalityParishCode);
```

4.2.2 View: vActiveEnergyByLocation

The SQL query used to create the second view associated with the table Energy.MonthlyConsumption:

```
CREATE VIEW Energy.vActiveEnergyByLocation WITH SCHEMABINDING AS
SELECT [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish],
 SUM([ActiveEnergy]) AS [ActiveEnergy],
 COUNT_BIG(*) AS [ResultCounting]
 FROM [Energy].[MonthlyConsumption]
 GROUP BY [DistrictMunicipalityParishCode],
 [District],
 [Municipality],
 [Parish];
```

And the index of the second view:

```
CREATE UNIQUE CLUSTERED INDEX IX_vActiveEnergyByLocation
ON Energy.vActiveEnergyByLocation(DistrictMunicipalityParishCode);
```

4.3 Join Algorithms

In our experimentation with various join algorithms using the provided views, we executed the original query from Task 3 while utilizing the OPTION clause to enforce the use of specific algorithms. We then analyzed the execution paths of each execution.

4.3.1 Default: Nested Loops Join

The tool defaults to using the Nested Loops Join algorithm for joining data from the views.

In the Nested Loops Join algorithm, the execution process follows a distinctive pattern, as depicted below:

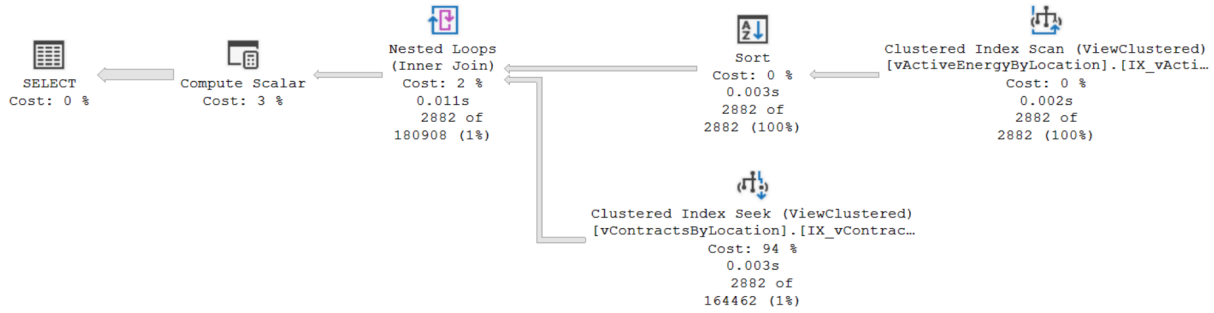

Figure 15: Execution Plan of 4.1

Estimated Tree Cost: 27.6459

In this algorithm, one table, typically referred to as the outer table, is sequentially scanned row by row. For each row encountered, a lookup operation is carried out on the other table, known as the inner table, utilizing an index seek.

The inclusion of a sorting step after the clustered index scan signifies that the rows are processed sequentially and organized before conducting the look-ups. This sorting step is crucial for efficiently correlating rows from the outer table with their counterparts in the inner table throughout the process.

4.3.2 Merge Join

The Merge Join algorithm efficiently combines data from two sorted inputs.

```
OPTION(MERGE JOIN);
```

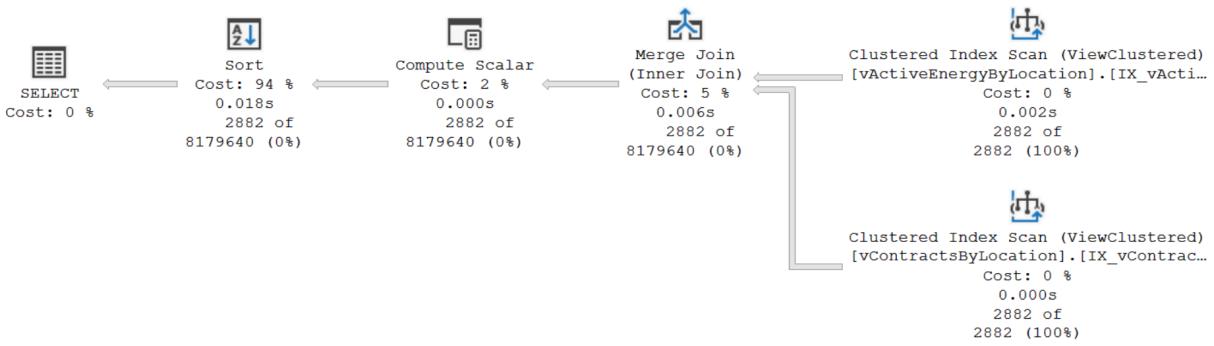

Figure 16: Execution Plan of 4.2

Estimated Tree Cost: 573.548

In this case, the execution plan shows that both tables are searched through Clustered Index Scans, and the join operation immediately follows, with the only Sort operation occurring at the end of the execution.

This algorithm yielded the highest Estimated Tree Cost. This can be attributed to the large size of the analyzed tables, which may be too big for this type of algorithm even when sorted. Additionally, limitations in the resources of the Virtual Machine may contribute to the higher Estimated Tree Cost, especially when compared to the Nested Loop Algorithm.

4.3.3 Hash Join

```
OPTION(HASH JOIN);
```

The execution path of the Hash Join algorithm closely resembles that of the Merge Join.

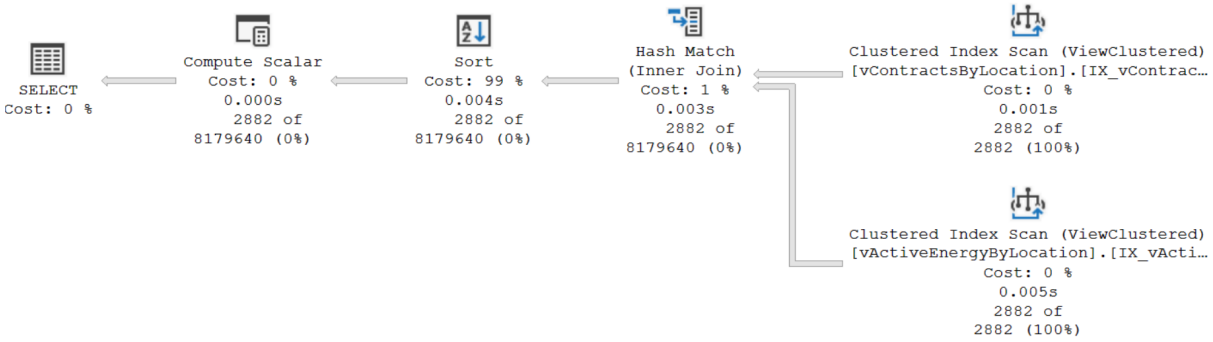

Figure 17: Execution Plan of 4.3

Estimated Tree Cost: 512.465

Here, the high Estimated Tree Cost can be justified due to memory pressure. The Hash algorithm typically requires sufficient memory to store the Hash tables, and with a larger dataset, this necessitates the usage of disk-based hash joins, thereby increasing the Estimated Tree Cost. Another contributing factor may be Hash Collisions, caused by the existence of a large number of distinct values or poor hash functions, which degrade the efficiency of the hash algorithm and increase both the estimated cost and computational overhead.

4.4 Resume

The comparison between the estimated Tree costs of the previous combinations is in table 4.

Join Algorithm	Estimated Tree Cost
Nested Loop Join	27.6459
Merge Join	573.548
Hash Join	512.465

Table 4: Comparison form results of Q4

By using views, the cost decrease donsistently, the tool defaults to selecting the option with the smallest Estimated Tree Cost, as seen previously.

5 Task 5

Firstly, we created the workload file with the queries from question 2 and 3. The file looks like:

```
SELECT [Parish], [Year], SUM([ActiveEnergy]) AS [ActiveEnergy]
FROM [Energy].[MonthlyConsumption]
```

```
WHERE [Municipality] = 'Lisboa'
 AND [Month] = '06'
GROUP BY [Parish], [Year]
ORDER BY [Parish], [Year]
```

```
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]
ORDER BY [Energy].[District],
 [Energy].[Municipality],
 [Energy].[Parish]
```

Running it, we the following execution plan:

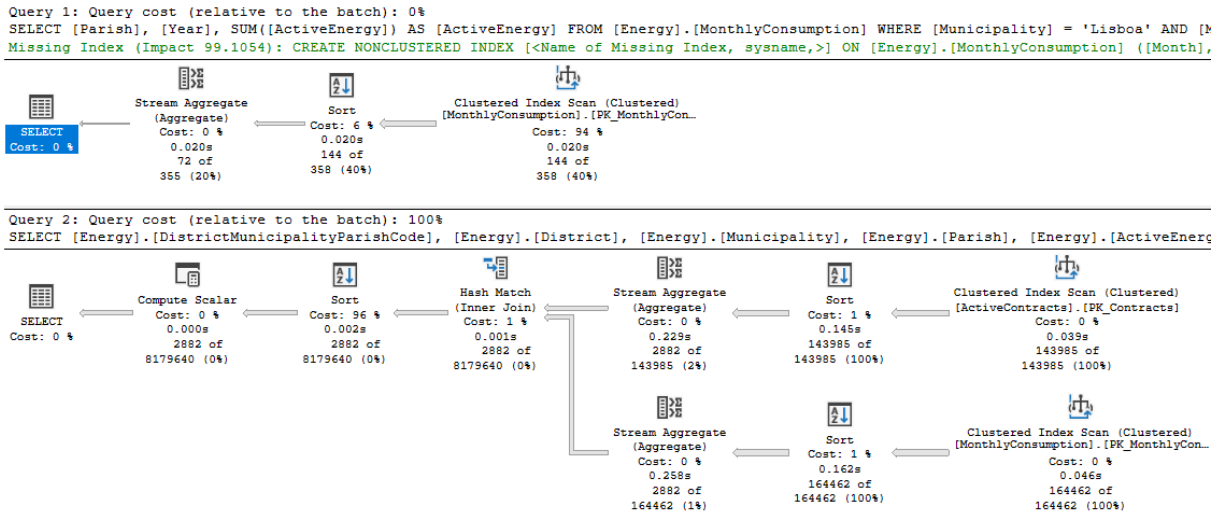

Figure 18: Execution Plan of 5 Before Database Engine Tuning Advisor

Estimated Tree Cost of Query 1: 2.608 **Estimated Tree Cost of Query 2:** 523.14

We input the file into Database Engine Tuning Advisor, selecting the correct database, and, in the Tuning Options, unchecked Limit Tuning Time, in Physical Design Structures (PDS) to use in database, we selected Indexes and indexed views and in the Physical Design Structures (PDS) to keep in database, selected keep all existing PDS. After it finished its analysis, we got the following recommendations:

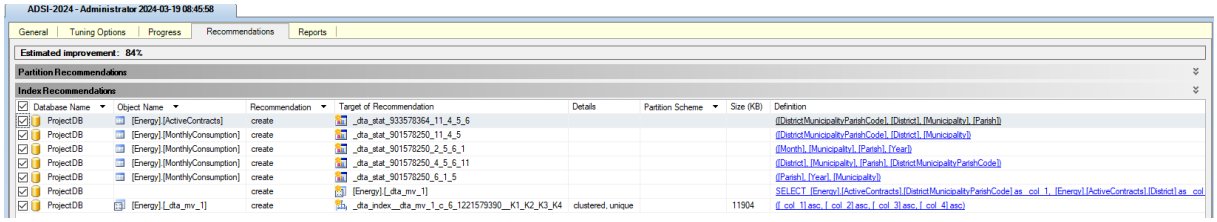

Figure 19: Execution Plan of 5.1

The recommendations include creating statistics in various columns, and a the following materialized view:

```
CREATE VIEW [Energy].[_dta_mv_1] WITH SCHEMABINDING
 AS
SELECT [Energy].[ActiveContracts].[DistrictMunicipalityParishCode] as _col_1,
    [Energy].[ActiveContracts].[District] as _col_2, [Energy].[ActiveContracts].[Municipality]
    as _col_3, [Energy].[ActiveContracts].[Parish] as _col_4,
    SUM([ProjectDB].[Energy].[ActiveContracts].[NumberContracts]) as _col_5, count_big(*) as
    _col_6 FROM [Energy].[ActiveContracts] GROUP BY
    [Energy].[ActiveContracts].[DistrictMunicipalityParishCode],
    [Energy].[ActiveContracts].[District], [Energy].[ActiveContracts].[Municipality],
    [Energy].[ActiveContracts].[Parish]
```

Index:

```
SET ARITHABORT ON
SET CONCAT_NULL_YIELDS_NULL ON
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF

CREATE UNIQUE CLUSTERED INDEX [_dta_index__dta_mv_1_c_6_1221579390__K1_K2_K3_K4] ON
    [Energy].[_dta_mv_1]
(
   [_col_1] ASC,
   [_col_2] ASC,
   [_col_3] ASC,
   [_col_4] ASC
)WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
```

This is one of the materialized views created in exercise 4, more specifically the one created on Energy.ActiveContracts table, on 4.2.1.

After applying the recommendations, we decided to run Database Engine Tuning Advisor again to see if there were more improvements. When the new analysis finished, we got even more recommendations:

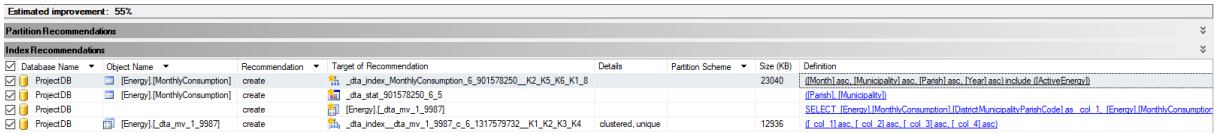

Figure 20: Execution Plan of 5.2

It is suggested a creation of a view and the index to materialize it, which is same one we created in 4.2.2:

```
CREATE VIEW [Energy].[_dta_mv_1_9987] WITH SCHEMABINDING
 AS
SELECT [Energy].[MonthlyConsumption].[DistrictMunicipalityParishCode] as _col_1,
    [Energy].[MonthlyConsumption].[District] as _col_2,
    [Energy].[MonthlyConsumption].[Municipality] as _col_3,
    [Energy].[MonthlyConsumption].[Parish] as _col_4, count_big(*) as _col_5,
    SUM([ProjectDB].[Energy].[MonthlyConsumption].[ActiveEnergy]) as _col_6 FROM
    [Energy].[MonthlyConsumption] GROUP BY
    [Energy].[MonthlyConsumption].[DistrictMunicipalityParishCode],
    [Energy].[MonthlyConsumption].[District], [Energy].[MonthlyConsumption].[Municipality],
    [Energy].[MonthlyConsumption].[Parish]
```

```
SET ARITHABORT ON
SET CONCAT_NULL_YIELDS_NULL ON
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON
SET NUMERIC_ROUNDABORT OFF

CREATE UNIQUE CLUSTERED INDEX [_dta_index__dta_mv_1_9987_c_6_1317579732__K1_K2_K3_K4] ON
    [Energy].[_dta_mv_1_9987]
(
   [_col_1] ASC,
   [_col_2] ASC,
   [_col_3] ASC,
   [_col_4] ASC
)WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
```

Lastly, we also get recommended an index that optimizes query 2.

```
SET ANSI_PADDING ON

CREATE NONCLUSTERED INDEX [_dta_index_MonthlyConsumption_6_901578250__K2_K5_K6_K1_8] ON
    [Energy].[MonthlyConsumption]
(
   [Month] ASC,
   [Municipality] ASC,
   [Parish] ASC,
   [Year] ASC
)
INCLUDE([ActiveEnergy]) WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON
    [PRIMARY]
```

We applied these recommendations as well. Now, going back to SQL Server, we run the query again to see the improvements.

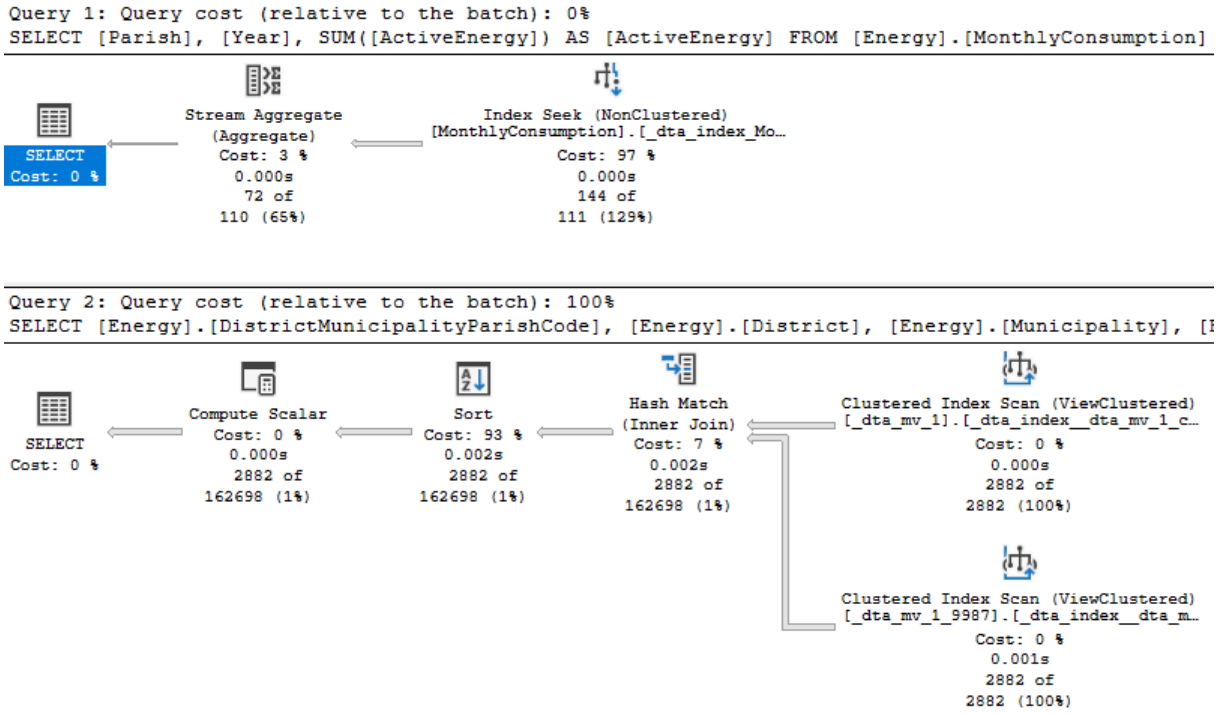

Figure 21: Execution Plan of 5 After Database Engine Tuning Advisor

Estimated Tree Cost of Query 1: 0.00357 **Estimated Tree Cost of Query 2:** 6.987

We see that the recommendations lower the cost of both queries by a significant ammount, making them more efficient.