Instituto Superior Técnico

Bologna Master Degree in Computer Science and Engineering

Data and Information Systems Management

Project: EREDES

2nd Semester 2023/2024

Group N.º 3:
David Cruz N.º 89377
Ana Baptista N.º 95529

Lisbon, March of 2024

1.

```
USE ProjectDB;

ALTER DATABASE ProjectDB ADD FILEGROUP PART_2020;
ALTER DATABASE ProjectDB ADD FILEGROUP PART_2021;
ALTER DATABASE ProjectDB ADD FILEGROUP PART_2022;
ALTER DATABASE ProjectDB ADD FILEGROUP PART_2023;
```

For each year-specific filegroup, a corresponding data file (.ndf) is created and stored in a specified directory.

```
alter database ProjectDB add file (
   name = PART_2020,
   filename = 'C:\ProjectDB-Partitions\PART_2020.ndf'
) to filegroup PART_2020;

alter database ProjectDB add file (
   name = PART_2021,
   filename = 'C:\ProjectDB-Partitions\PART_2021.ndf'
) to filegroup PART_2021;

alter database ProjectDB add file (
   name = PART_2022,
   filename = 'C:\ProjectDB-Partitions\PART_2022.ndf'
) to filegroup PART_2022;

alter database ProjectDB add file (
   name = PART_2023,
   filename = 'C:\ProjectDB-Partitions\PART_2023.ndf'
) to filegroup PART_2023;
```

In this case, rows for the years 2020, 2021, and 2022 are directed to specific partitions, with each subsequent partition holding the rows for the next year.

```
CREATE PARTITION FUNCTION ProjectDB_Part_Fun(CHAR (4))
AS RANGE LEFT FOR VALUES('2020', '2021', '2022');

CREATE PARTITION SCHEME ProjectDB_Partition
AS PARTITION ProjectDB_Part_Fun TO
(PART_2020, PART_2021, PART_2022, PART_2023);
```

With the partition function and scheme in place, the script creates the Energy.MonthlyConsumptionPartitioned table.

```
GO
```

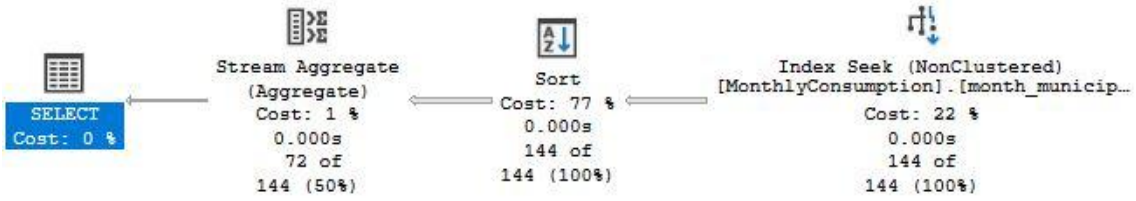
```
CREATE TABLE [Energy].[MonthlyConsumptionPartitioned](
	[Year] [char](4) NOT NULL,
	[Month] [char](2) NOT NULL,
	[Date] [char](7) NULL,
	[District] [varchar](255) NULL,
	[Municipality] [varchar](255) NULL,
	[Parish] [varchar](255) NULL,
	[VoltageLevel] [varchar](255) NOT NULL,
	[ActiveEnergy] [float] NOT NULL,
	[DistrictCode] [char](2) NULL,
	[DistrictMunicipalityCode] [char](4) NULL,
	[DistrictMunicipalityParishCode] [char](6) NOT NULL,
 CONSTRAINT [PK_MonthlyConsumptionPartitioned] PRIMARY KEY CLUSTERED
(
	[Year] ASC,
	[Month] ASC,
	[DistrictMunicipalityParishCode] ASC,
	[VoltageLevel] ASC
)
) ON ProjectDB_Partition([Year])
GO

insert into [Energy].[MonthlyConsumptionPartitioned] select *  from
[Energy].[MonthlyConsumption];
```

Finally, a query is provided to count the number of rows in each partition of the Energy.MonthlyConsumptionPartitioned table. This allows for verification that the data has been correctly partitioned and that the row counts match the expected distribution based on the Year column.

```
SELECT
   partition_number,
   rows
FROM sys.partitions
WHERE object_id = OBJECT_ID('[Energy].[MonthlyConsumptionPartitioned]');
```

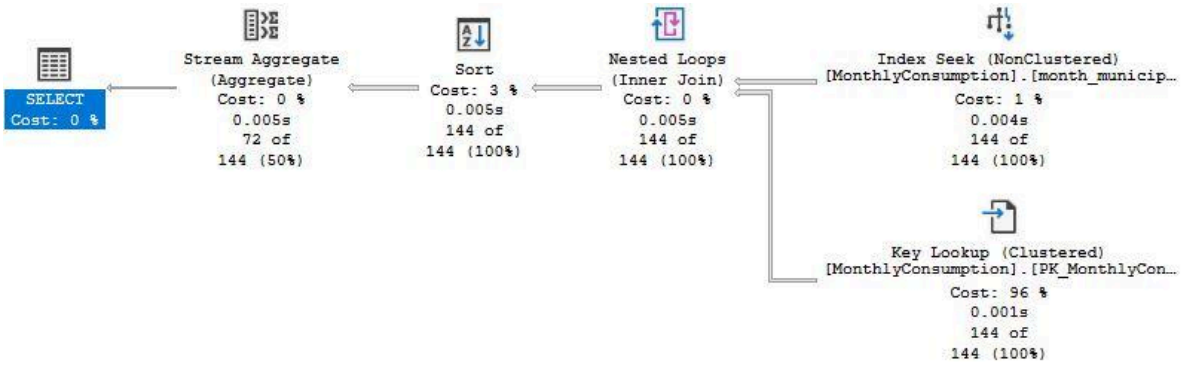
2.

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

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



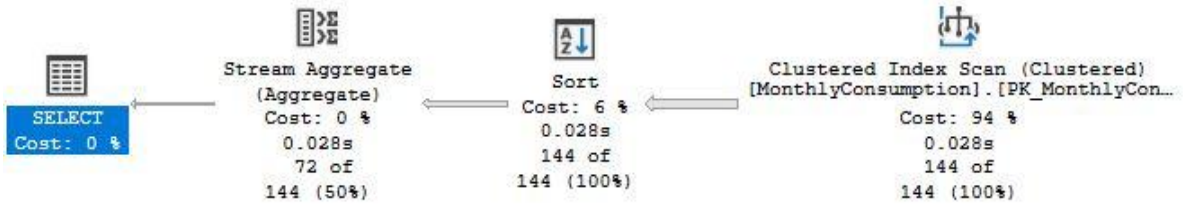
This index significantly improved query performance, achieving an Estimated Subtree Cost of just 0.0034533. The month_municipality index is efficient because it contains all necessary data to locate the rows, filter, and aggregate them without the need to resort to the primary key index. SQL Server recommended this index as the fastest option among the tested indexes. The disadvantage is that the month_municipality index is highly specific to this particular query, limiting its applicability to other potential queries. Additionally, it occupies more disk space compared to more generalized indexes.

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

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



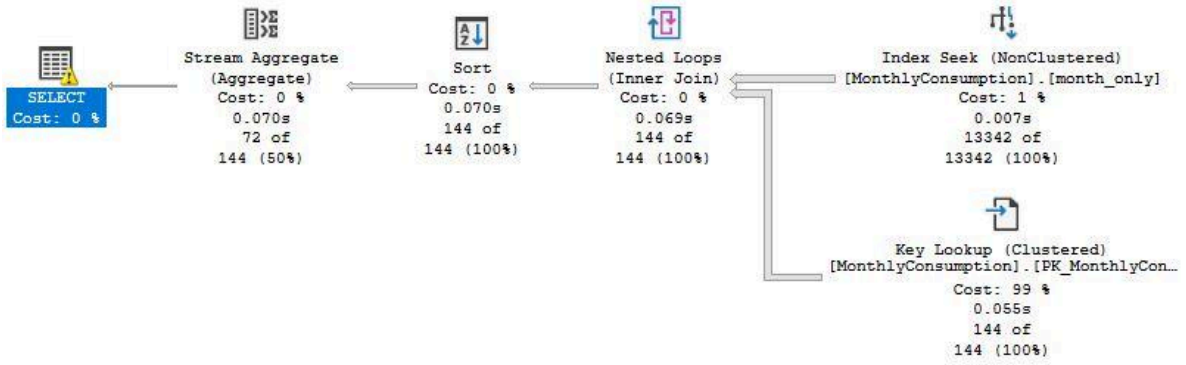
The Estimated Subtree Cost for an index seek using the month_municipality_only index was notably low at 0.0034037. However, to complete the operation, a key lookup using the PK_MonthlyConsumption index was necessary, with a cost of 0.354086, and the overall cost for the nested loop operation that combined the two was 0.357952. The month_municipality_only index is capable of efficiently locating the relevant rows by utilizing both the Month and Municipality columns. Yet, it lacks included columns for performing operations such as sorting (ORDER BY) and grouping (GROUP BY). This necessitates the use of the primary key index to accomplish these tasks. Although this index was only marginally faster than the one containing only the municipality, it was larger in size due to also including the month

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



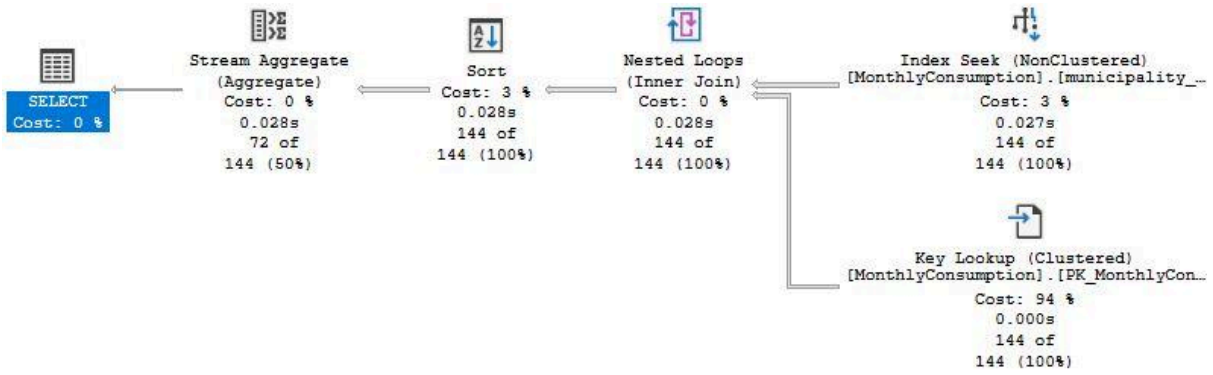
When utilizing the PK_MonthlyConsumption index, which is the primary key of the table the Estimated Subtree Cost associated with the clustered index scan was observed to be 2.44567. This index is inherent to every table and does not require additional storage, serving as the default method for organizing table data in a sorted manner based on the primary key. However, relying on the PK_MonthlyConsumption primary key index for this specific query resulted in a slower performance compared to the other indexing strategies previously discussed, except for the index that contains only the month.

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

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



The Estimated Subtree Cost for performing an index seek to find the month using the month_only index was relatively low at 0.067515. However, the subsequent key lookup to locate Lisbon using the PK_MonthlyConsumption primary key index incurred a much higher cost of 11.5323. Consequently, the overall cost for the nested loop operation that combined these two steps was 11.6556. The inefficiency in locating Lisbon indicates that this index is poorly suited for queries that require filtering by both month and municipality. In fact, this approach proved to be worse than using the primary key index alone, making it an impractical choice for optimizing this particular query.

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

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



The Estimated Subtree Cost for an index seek operation to locate the municipality using the municipality_only index was impressively low, at 0.0131579. However, to identify the specific month, a subsequent key lookup using the PK_MonthlyConsumption primary key index was required, which had a cost of 0.459656. The combined operations in the nested loop resulted in an overall cost of 0.474269. The effectiveness of the municipality_only index in this context can be attributed to its targeted focus on the municipality column, which was the most challenging aspect of the search. The small size of the index, due to it only containing a single column as the key, contributed to its speed. This index is potentially more versatile for other queries that require filtering by municipality but do not necessarily involve the month.

3.

```
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]
```

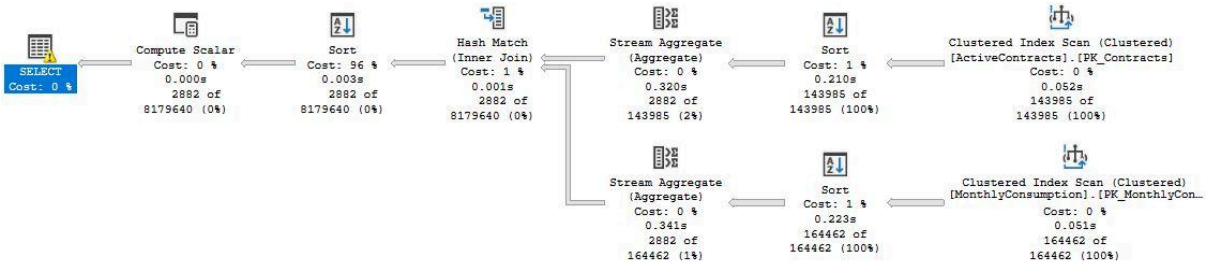
Execution Plans and Estimated Subtree Cost Study:

The results will be displayed in a descending order of performance, showcasing the most efficient outcomes first and proceeding to the least efficient.

When not specified, it chooses the option with the best estimated cost.

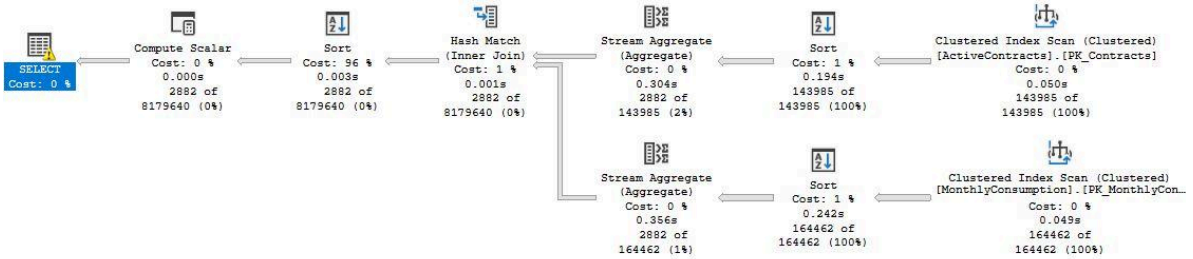
It is important to note how significantly worse the LOOP JOIN was compared to the other Hints, which all achieved similar costs.

- OPTION (HASH JOIN, ORDER GROUP)



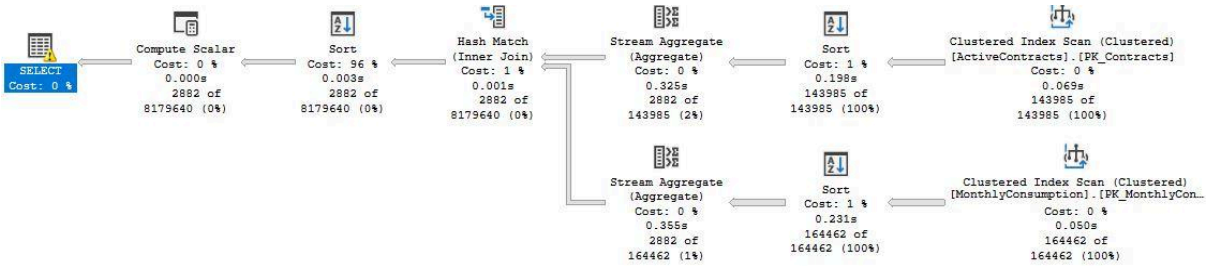
Estimated Subtree Cost: 523.135

- OPTION (HASH JOIN);



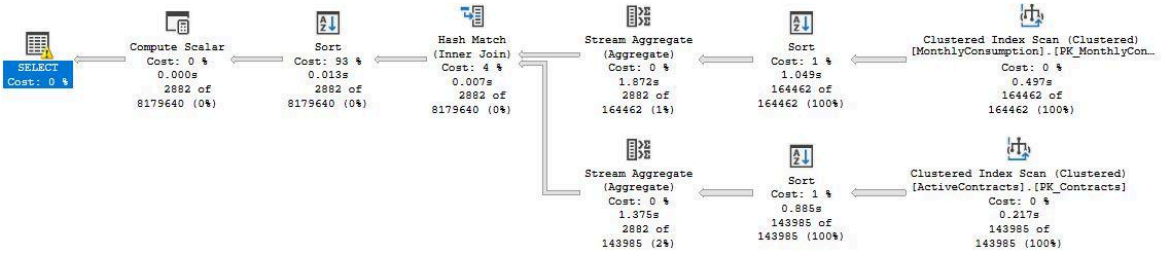
Estimated Subtree Cost: 523.135

- OPTION (ORDER GROUP);



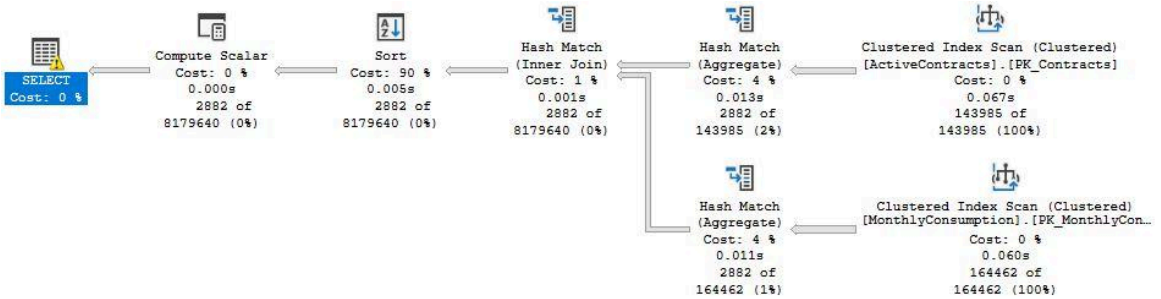
Estimated Subtree Cost: 523.135

- OPTION (FORCE ORDER);



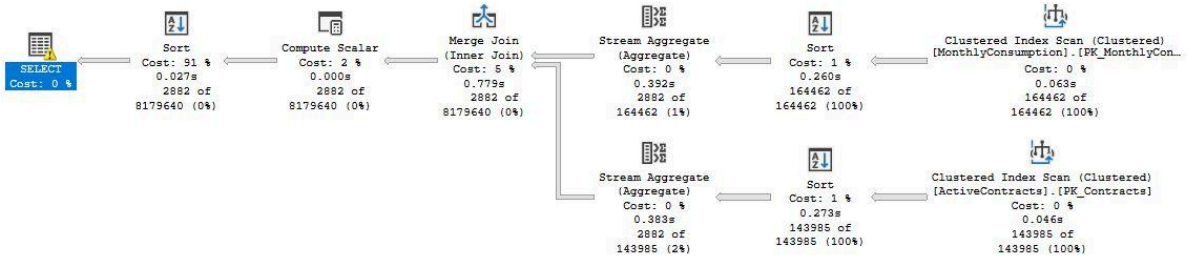
Estimated Subtree Cost: 538.359

- OPTION (HASH GROUP);



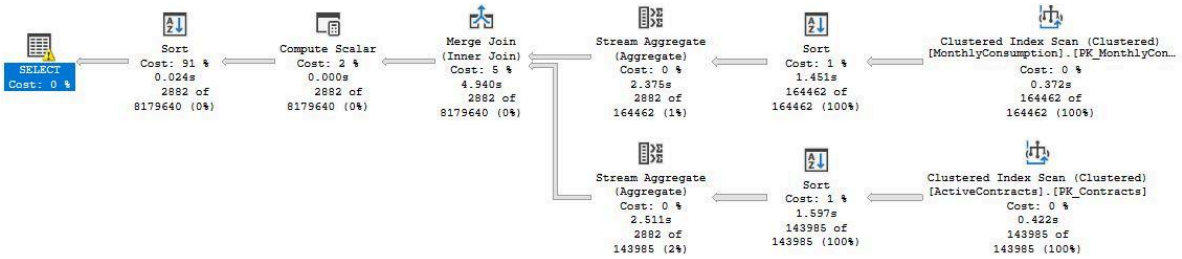
Estimated Subtree Cost: 555.603

- OPTION (MERGE JOIN);



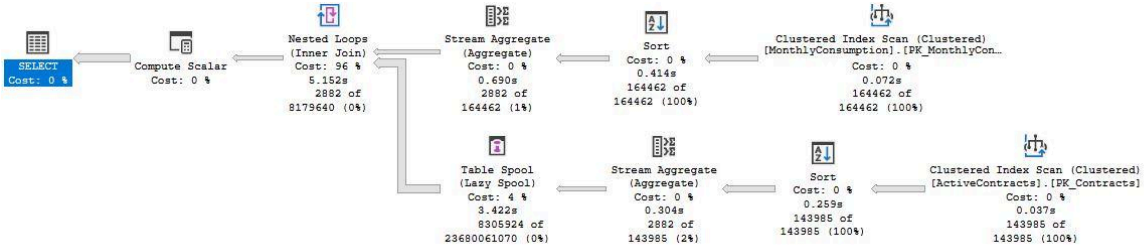
Estimated Subtree Cost: 583.995

- OPTION (MERGE JOIN, FORCE ORDER);



Estimated Subtree Cost: 583.995

- OPTION (LOOP JOIN);



Estimated Subtree Cost: 110389

4.

Creating the Materialized Energy View:

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

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

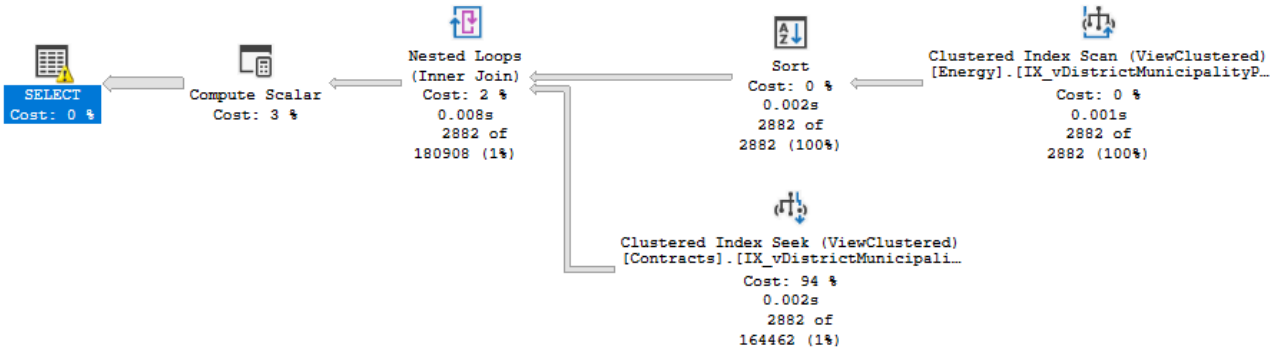
Creating the Materialized Contracts View:

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

CREATE UNIQUE CLUSTERED INDEX IX_vDistrictMunicipalityParishCode
ON Energy.Contracts (DistrictMunicipalityParishCode);
```

Execution Plans and Estimated Subtree Cost Study:

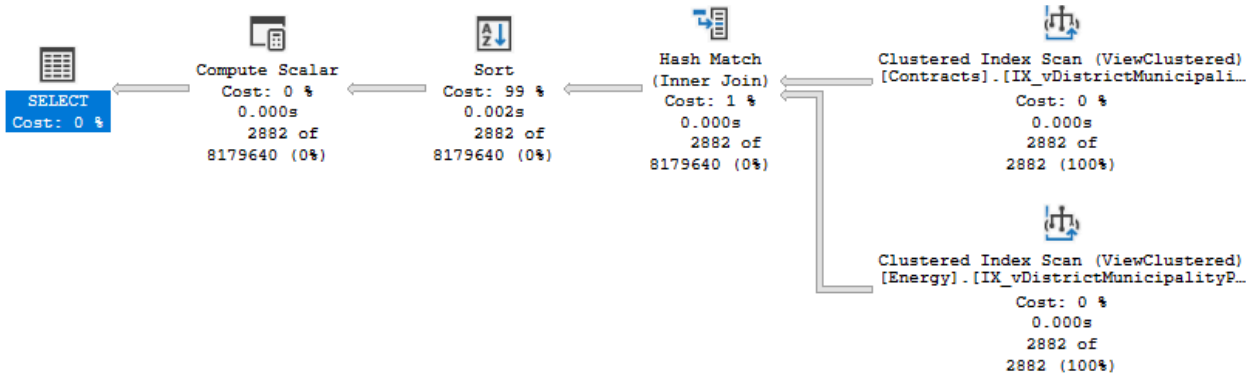
The results will be displayed in a descending order of performance, showcasing the most efficient outcomes first and proceeding to the least efficient.

- OPTION(LOOP JOIN):



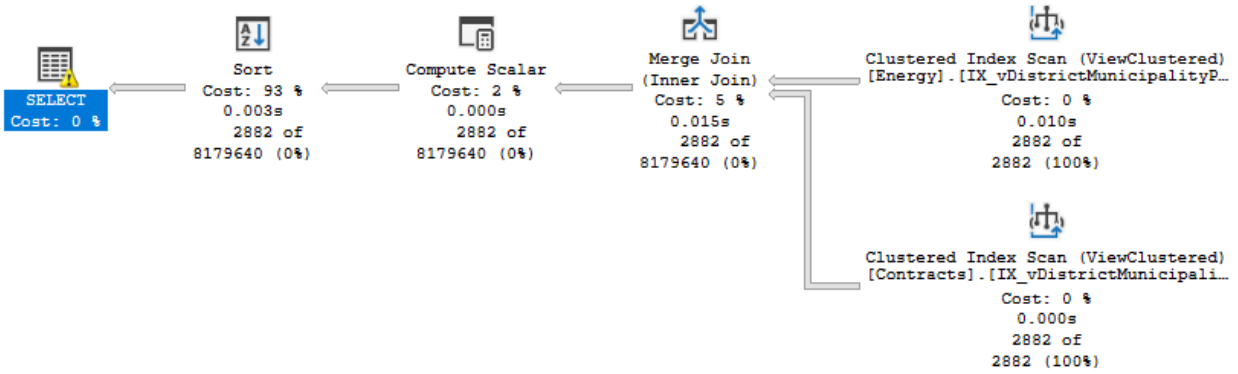
Estimated Subtree Cost: 27.646

- OPTION(HASH JOIN):



Estimated Subtree Cost: 506.052

- OPTION(MERGE JOIN):



Estimated Subtree Cost: 567.135

Because our nested subqueries use the GROUP BY clause, COUNT_BIG(*) needed to be present in our views.

This is due to a limitation regarding Indexed Views, quoting the Microsoft Learn - SQL Server regarding the creation of indexed views:

> "If GROUP BY is present, the VIEW definition must contain COUNT_BIG(*) and must not contain HAVING. These GROUP BY restrictions are applicable only to the indexed view definition. A query can use an indexed view in its execution plan even if it doesn't satisfy these GROUP BY restrictions." [3]

If executing the query without an OPTION clause, SQL Server will choose to apply the Nested Loop Join, which is in fact the one with the best performance in terms of cost. The creation of the materialized view decreased drastically the cost of our query on the LOOP JOIN, since the results don't have to be computed on the fly, and the indexes can just be scanned and seeked respectively, which makes the LOOP join highly efficient [6]; this join was by far the worst option without the materialized view. The HASH and MERGE Joins achieved similar results to the best results without a view.

5.

For starters a workload file needed to be created, using the queries from questions no. 's 2 and 3, we called it Project-Workload.sql (see attachment).

To get the recommendations desired two consecutive analyses with the Tuning Advisor were required. All analyses used the same setup, which consists of:

- Setting our mentioned workload as input file;
- Selecting projectDB database for analysis and tuning processes;
- No tuning time limit;
- PDS to use: Indexes and indexed views;
- Partitioning strategy: No partitioning;
- PDS to keep: Keep all existing PDS;

With the first Tuning Advisor run we got as recommendations the creation of a set of stats and the creation of our first recommended indexed view(highlighted), which corresponds to the Contracts subquery from question no. 3 :

Database Name	Object Name	Recommendation	Target of Recommendation	Details
ProjectDB	[Energy].[ActiveContracts]	create	_dta_stat_933578364_11_4_5_6	
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_stat_901578250_11_4_5	
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_stat_901578250_4_5_6_11	
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_stat_901578250_5_2_6_1	
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_stat_901578250_6_1_5	
ProjectDB		create	[Energy].[_dta_mv_1]	
ProjectDB	[Energy].[_dta_mv_1]	create	_dta_index_dta_mv_1_c_6_1221579390__K1_K2_K3_K4	clustered, unique

Through the report tab we can tell that these recommendations achieved an overall estimated percent improvement of 83.72%. The first statement, being the query from question no. 2, didn't get any percent improvement, as this analysis produced only statistic creations for MonthlyConsumption; the second statement got an improvement of 84.14% thanks to the new indexed view.

To apply this we saved these recommendations as a .sql file, opened it on Microsoft SQL Server Management Studio and executed it, and by refreshing the database we confirmed our 1st view was achieved.

Running the Tuning Advisor a second time on the updated database we get as recommendations the creation of an nonclustered index on MonthlyConsumption with the columns being Month, Municipality, Parish and Year; we also get our second recommended indexed view, corresponding to the Energy subquery from question no. 3:

Database Name	Object Name	Recommendation	Target of Recommendation	Details
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_index_MonthlyConsumption_6_901578250__K2_K5_K6_K1_8	
ProjectDB	[Energy].[MonthlyConsumption]	create	_dta_stat_901578250_6_5	
ProjectDB		create	[Energy].[_dta_mv_1_9987]	
ProjectDB	[Energy].[_dta_mv_1_9987]	create	_dta_index_dta_mv_1_9987_c_6_1317579732__K1_K2_K3_K4	clustered, unique

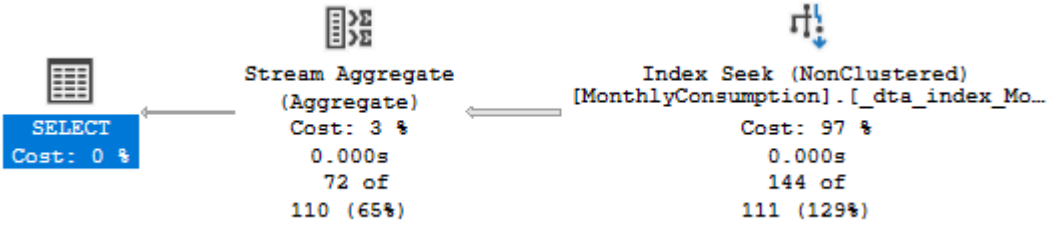
Through the report tab we can tell that these recommendations achieved an estimated percent improvement of 55.84%. This time the first statement got a 99.85% improvement, achieved specifically through the nonclustered index, the view didn't affect this query; the second statement got an improvement as well, being that of 48.81% through the indexed view creation.

Again, we save the recommendations and execute the sql on the management studio to confirm these changes. By running Tuning Advisor a third time no recommendations were achieved.

Execution Plans and Estimated Subtree Cost Study:

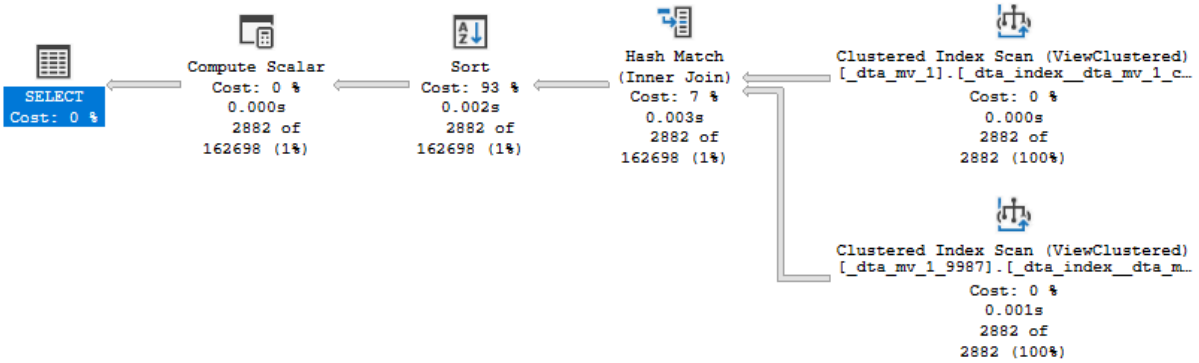
These changes obtained the results that follow. We can see that both queries were improved significantly from our previous Subtree Cost studies, so it proved beneficial to use the Tuning Advisor Software even though it took 2 runs and some trial and error to find the right recommendations.

- **Query from question no. 2:**



Estimated Subtree Cost: 0.004

- **Query from question no. 3:**



Estimated Subtree Cost: 6.987

Attachment - Project-Workload.sql:

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

USE ProjectDB;
GO
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];
GO
```

References:

1. Course Slides and Lab Assignments;
2. SQL Server - COUNT_BIG in Indexed View | Stack Overflow
3. Create Indexed Views - SQL Server | Microsoft Learn
4. SQL Server JOIN Hints | MSSQTips
5. Join Hints (Transact-SQL) - SQL Server | Microsoft Learn
6. SQL Server Nested Loop Join | sqlTechHead