PROJECT REPORT

Data Administration in Information Systems

MEIC - 2nd Semester 2022/2023

Matheus de Souza Trindade, Student ID. 105471

Maria João Madeira Duarte, Student ID. 90415

Challenges

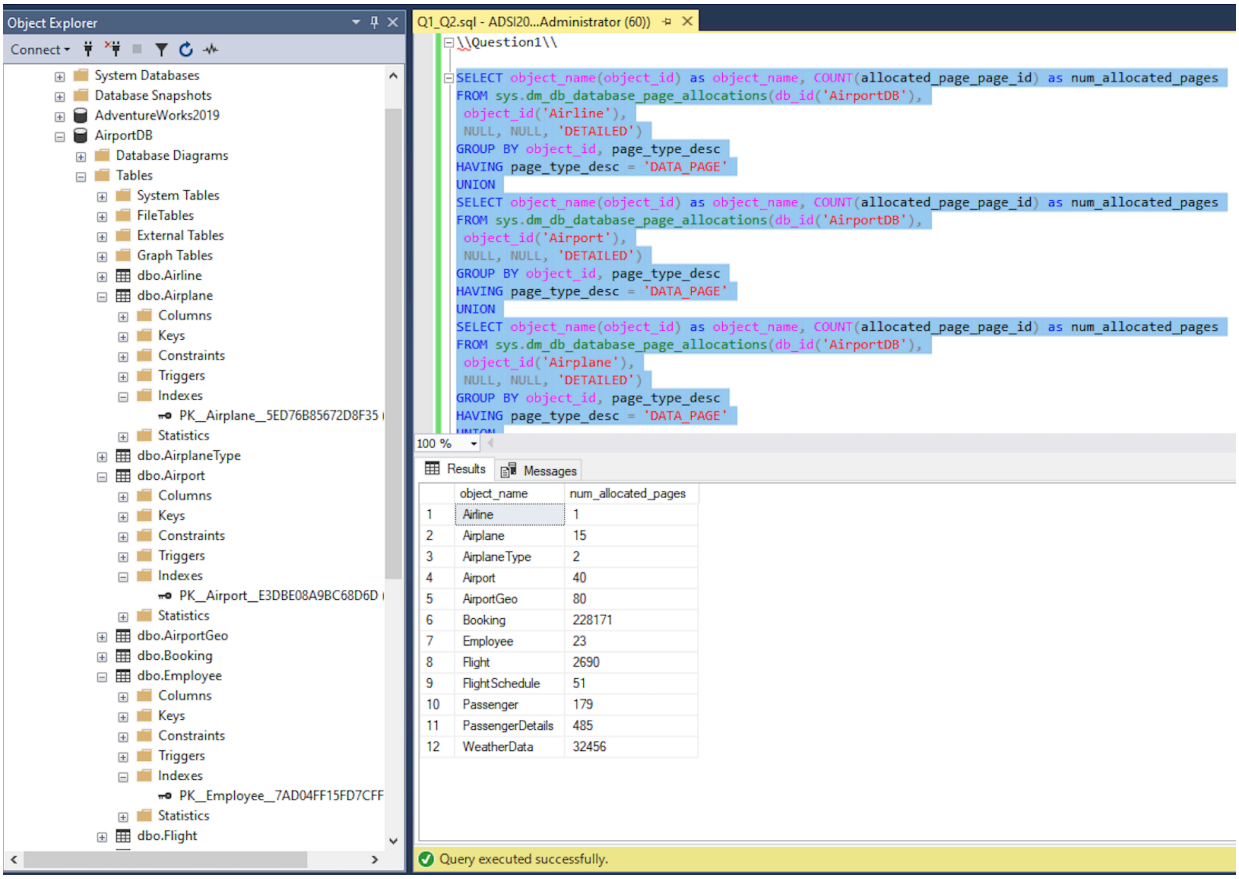
1. In a single query, find the number of data pages allocated by each table in the database. Present the query and the results. (Lab02)

Initially, it was implemented the following query to deliver the number of data pages allocated by each table in the database. It delivered the desired results, but the query's size ended up being too big and required an optimization.

```
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Airline'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Airport'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Airplane'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('AirplaneType'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Airport'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
```

```
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('AirportGeo'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Booking'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
 UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Employee'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Flight'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('FlightSchedule'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('Passenger'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('WeatherData'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
UNION
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
```

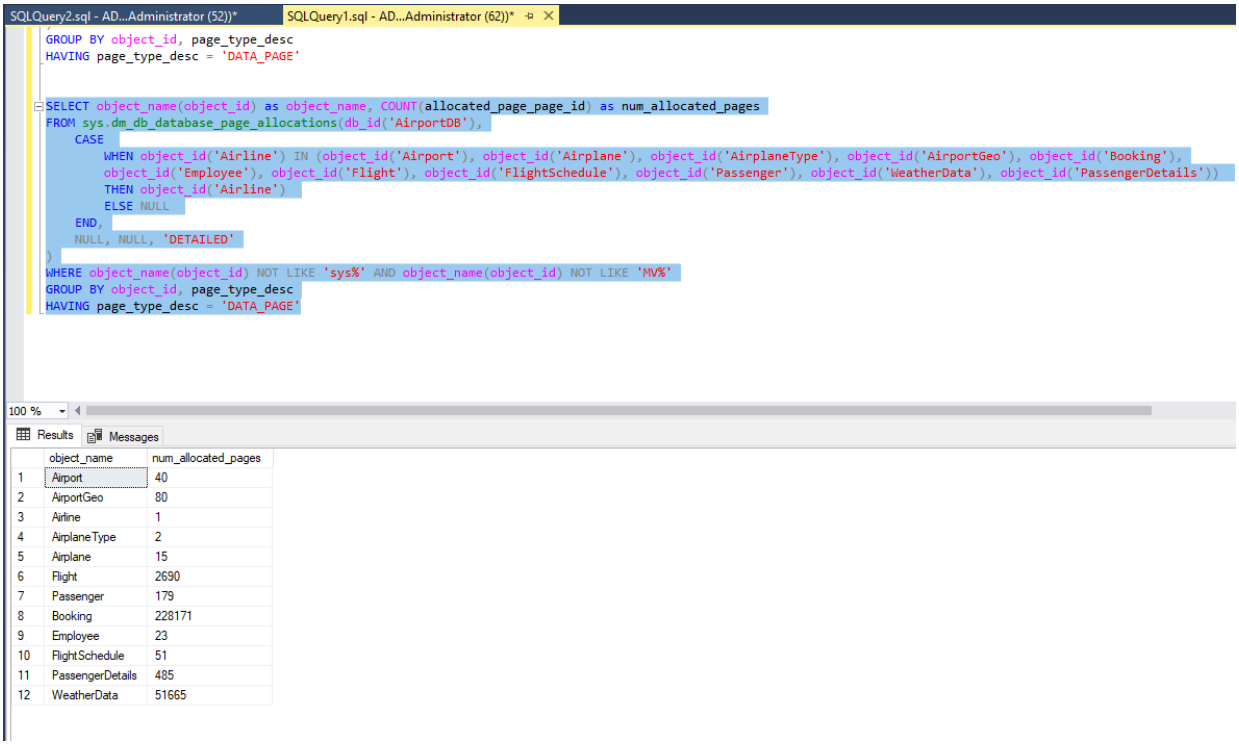
```
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
 object_id('PassengerDetails'),
 NULL, NULL, 'DETAILED')
GROUP BY object_id, page_type_desc
HAVING page_type_desc = 'DATA_PAGE'
```



The query was optimized, using the CASE statement to dynamically select the object to query based on whether the Airline table exists in the database, some tables starting with 'sys' and 'MV' had to be removed from the final results, and this was a strategy to retrieve information from all tables in a more simplified (shorter query) way, as follows:

```
SELECT object_name(object_id) as object_name,
COUNT(allocated_page_page_id) as num_allocated_pages
FROM sys.dm_db_database_page_allocations(db_id('AirportDB'),
    CASE
        WHEN object_id('Airline') IN (object_id('Airport'),
object_id('Airplane'), object_id('AirplaneType'),
object_id('AirportGeo'), object_id('Booking'),
```

```
        object_id('Employee'), object_id('Flight'),
    object_id('FlightSchedule'), object_id('Passenger'),
    object_id('WeatherData'), object_id('PassengerDetails'))
            THEN object_id('Airline')
            ELSE NULL
        END,
        NULL, NULL, 'DETAILED'
    )
    WHERE object_name(object_id) NOT LIKE 'sys%' AND
    object_name(object_id) NOT LIKE 'MV%'
    GROUP BY object_id, page_type_desc
    HAVING page_type_desc = 'DATA_PAGE'
```



	object_name	num_allocated_pages
1	Airport	40
2	AirportGeo	80
3	Airline	1
4	AirplaneType	2
5	Airplane	15
6	Flight	2690
7	Passenger	179
8	Booking	228171
9	Employee	23
10	FlightSchedule	51
11	PassengerDetails	485
12	WeatherData	51665

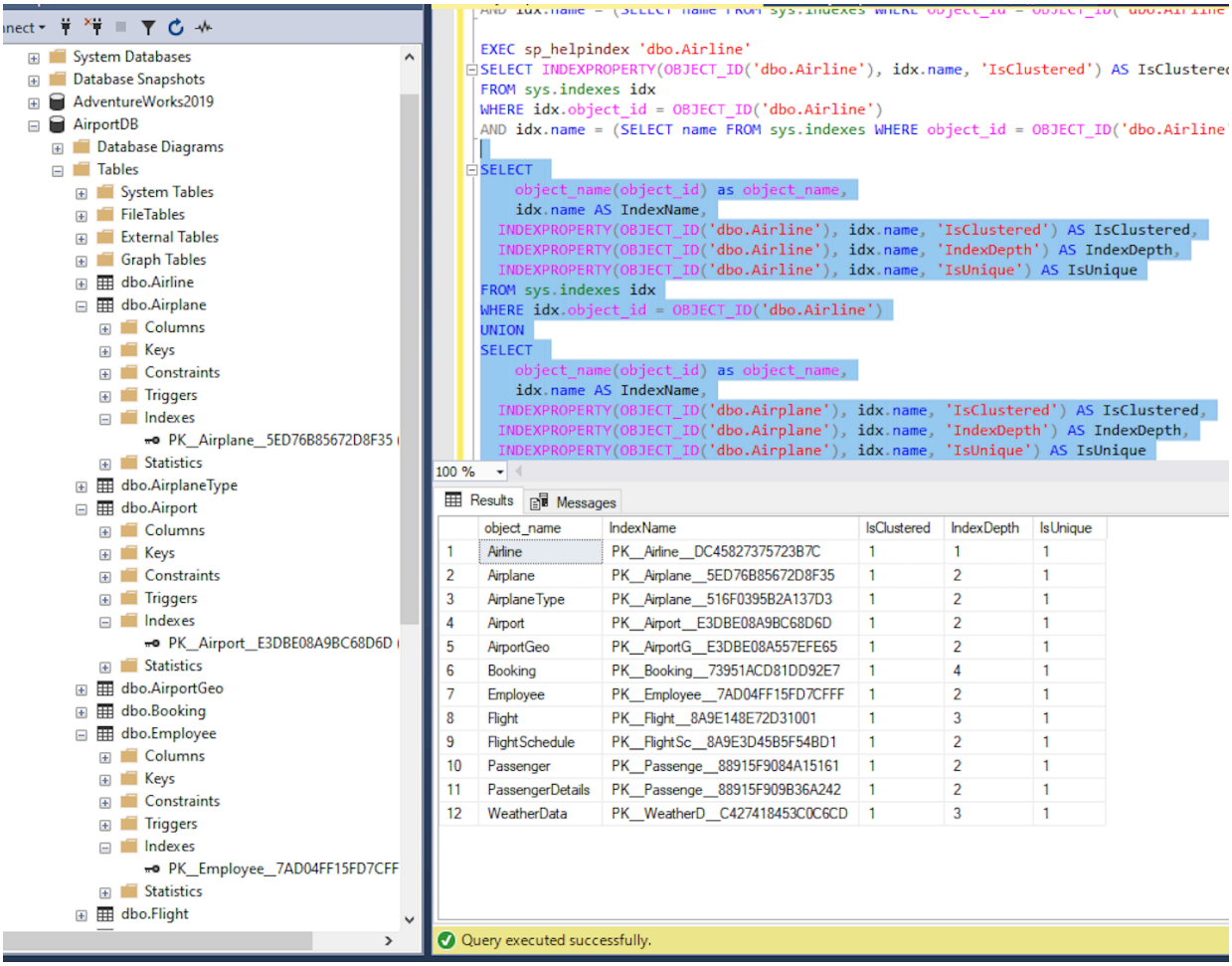
2. In a single query, find all the indexes that exist in each table of the AirportDB database, together with the properties IsClustered, IsUnique, IndexDepth for each index. Present the query and the results. (Lab03).

Initially, it was implemented the following query to deliver the desired results. It worked, but the query's size ended up being too big and required an optimization.

```
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Airline'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Airline'), idx.name, 'IndexDepth')
AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Airline'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Airline')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Airplane'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Airplane'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Airplane'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Airplane')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.AirplaneType'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.AirplaneType'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.AirplaneType'), idx.name,
'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.AirplaneType')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Airport'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Airport'), idx.name, 'IndexDepth')
AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Airport'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Airport')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.AirportGeo'), idx.name,
'IsClustered') AS IsClustered,
```

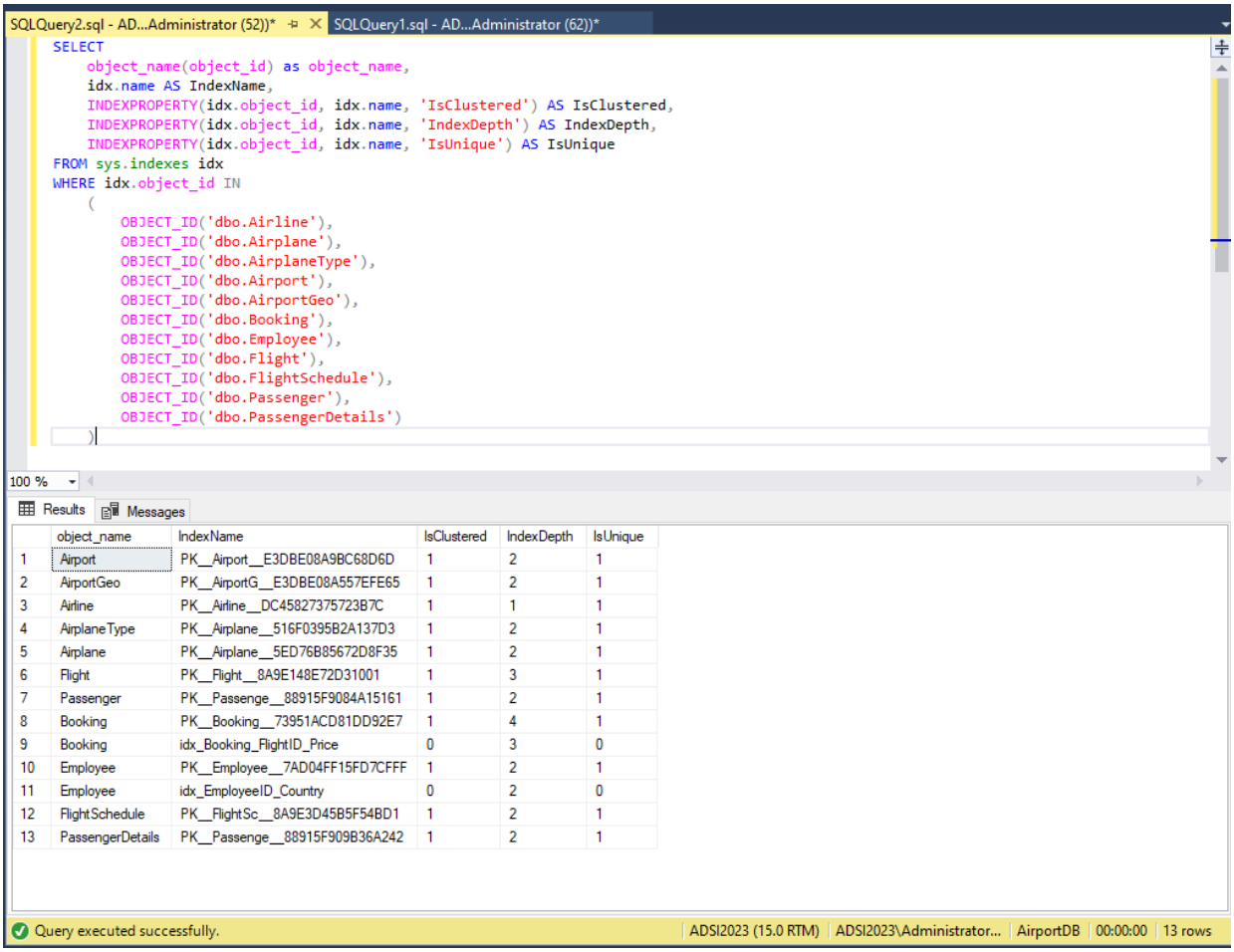
```
  INDEXPROPERTY(OBJECT_ID('dbo.AirportGeo'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.AirportGeo'), idx.name,
'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.AirportGeo')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Booking'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Booking'), idx.name, 'IndexDepth')
AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Booking'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Booking')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Employee'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Employee'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Employee'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Employee')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Flight'), idx.name, 'IsClustered')
AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Flight'), idx.name, 'IndexDepth')
AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Flight'), idx.name, 'IsUnique') AS
IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Flight')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.FlightSchedule'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.FlightSchedule'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.FlightSchedule'), idx.name,
'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.FlightSchedule')
UNION
SELECT
```

```
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.Passenger'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.Passenger'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.Passenger'), idx.name, 'IsUnique')
AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.Passenger')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.PassengerDetails'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.PassengerDetails'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.PassengerDetails'), idx.name,
'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.PassengerDetails')
UNION
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
  INDEXPROPERTY(OBJECT_ID('dbo.WeatherData'), idx.name,
'IsClustered') AS IsClustered,
  INDEXPROPERTY(OBJECT_ID('dbo.WeatherData'), idx.name,
'IndexDepth') AS IndexDepth,
  INDEXPROPERTY(OBJECT_ID('dbo.WeatherData'), idx.name,
'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id = OBJECT_ID('dbo.WeatherData')
```



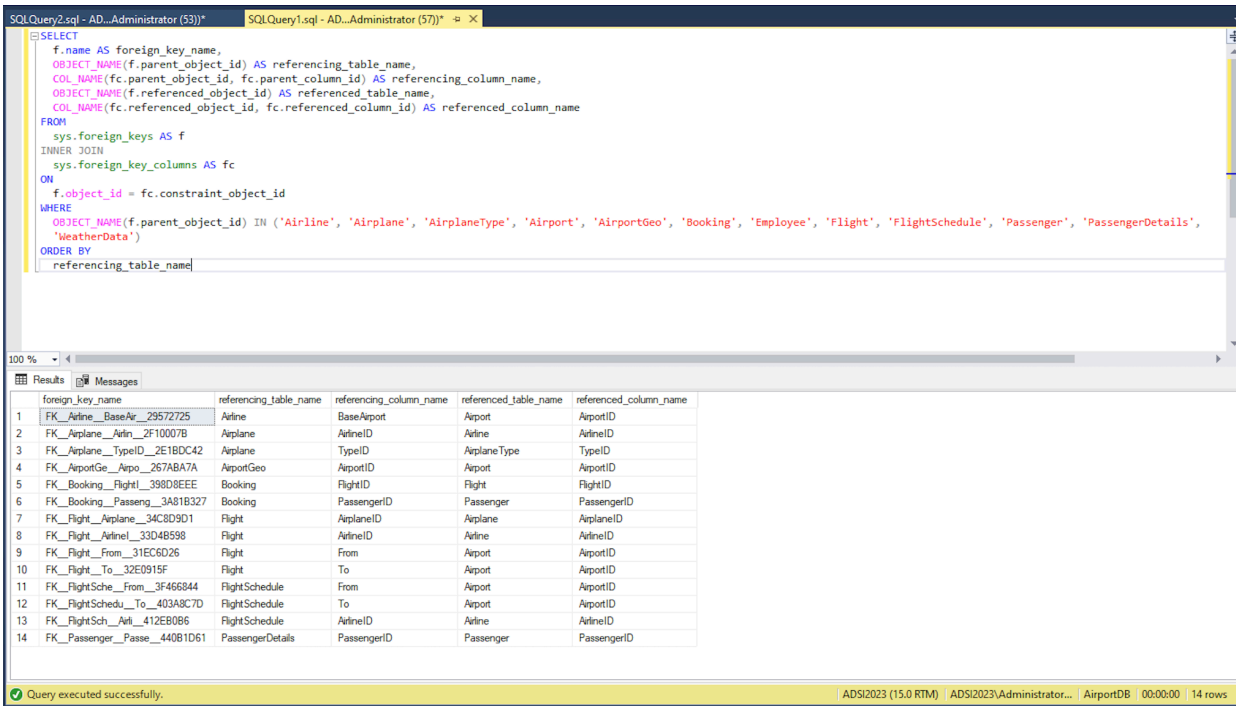
The query was optimized using “WHERE / IN” to group all ‘object_id’s, as follows:

```
SELECT
 object_name(object_id) as object_name,
 idx.name AS IndexName,
 INDEXPROPERTY(idx.object_id, idx.name, 'IsClustered') AS
IsClustered,
 INDEXPROPERTY(idx.object_id, idx.name, 'IndexDepth') AS
IndexDepth,
 INDEXPROPERTY(idx.object_id, idx.name, 'IsUnique') AS IsUnique
FROM sys.indexes idx
WHERE idx.object_id IN
 (
       OBJECT_ID('dbo.Airline'),
       OBJECT_ID('dbo.Airplane'),
       OBJECT_ID('dbo.AirplaneType'),
       OBJECT_ID('dbo.Airport'),
       OBJECT_ID('dbo.AirportGeo'),
       OBJECT_ID('dbo.Booking'),
       OBJECT_ID('dbo.Employee'),
       OBJECT_ID('dbo.Flight'),
       OBJECT_ID('dbo.FlightSchedule'),
       OBJECT_ID('dbo.Passenger'),
       OBJECT_ID('dbo.PassengerDetails')
 )
```



3. In a single query, find all foreign keys (FKs) between tables in the database, indicating the table and column where the FK comes from, and the table and column where the FK points to. Present the query and the results. (Lab03)

```
SELECT
  f.name AS foreign_key_name,
  OBJECT_NAME(f.parent_object_id) AS referencing_table_name,
  COL_NAME(fc.parent_object_id, fc.parent_column_id) AS
referencing_column_name,
  OBJECT_NAME(f.referenced_object_id) AS referenced_table_name,
  COL_NAME(fc.referenced_object_id, fc.referenced_column_id) AS
referenced_column_name
FROM
  sys.foreign_keys AS f
INNER JOIN
  sys.foreign_key_columns AS fc
ON
  f.object_id = fc.constraint_object_id
WHERE
  OBJECT_NAME(f.parent_object_id) IN ('Airline', 'Airplane',
'AirplaneType', 'Airport', 'AirportGeo', 'Booking', 'Employee',
'Flight', 'FlightSchedule', 'Passenger', 'PassengerDetails',
'WeatherData')
ORDER BY
  referencing_table_name
```



```
SELECT
  f.name AS foreign_key_name,
  OBJECT_NAME(f.parent_object_id) AS referencing_table_name,
  COL_NAME(fc.parent_object_id, fc.parent_column_id) AS referencing_column_name,
  OBJECT_NAME(f.referenced_object_id) AS referenced_table_name,
  COL_NAME(fc.referenced_object_id, fc.referenced_column_id) AS referenced_column_name
FROM
  sys.foreign_keys AS f
INNER JOIN
  sys.foreign_key_columns AS fc
ON
  f.object_id = fc.constraint_object_id
WHERE
  OBJECT_NAME(f.parent_object_id) IN ('Airline', 'Airplane', 'AirplaneType', 'Airport', 'AirportGeo', 'Booking', 'Employee', 'Flight', 'FlightSchedule', 'Passenger', 'PassengerDetails',
  'WeatherData')
ORDER BY
  referencing_table_name
```

	foreign_key_name	referencing_table_name	referencing_column_name	referenced_table_name	referenced_column_name
1	FK__Airline__BaseAir__29572725	Airline	BaseAirport	Airport	AirportID
2	FK__Airplane__Airlin__2F10007B	Airplane	AirlineID	Airline	AirlineID
3	FK__Airplane__TypeID__2E1BDC42	Airplane	TypeID	AirplaneType	TypeID
4	FK__AirportGe__Airpo__267ABA7A	AirportGeo	AirportID	Airport	AirportID
5	FK__Booking__Flight__398D8EEE	Booking	FlightID	Flight	FlightID
6	FK__Booking__Passeng__3A81B327	Booking	PassengerID	Passenger	PassengerID
7	FK__Flight__Airplane__34C8D9D1	Flight	AirplaneID	Airplane	AirplaneID
8	FK__Flight__Airline__33D4B598	Flight	AirlineID	Airline	AirlineID
9	FK__Flight__From__31EC6D26	Flight	From	Airport	AirportID
10	FK__Flight__To__32E0915F	Flight	To	Airport	AirportID
11	FK__FlightSche__From__3F466844	FlightSchedule	From	Airport	AirportID
12	FK__FlightSchedu__To__403A8C7D	FlightSchedule	To	Airport	AirportID
13	FK__FlightSch__Airli__412EB0B6	FlightSchedule	AirlineID	Airline	AirlineID
14	FK__Passenger__Passe__440B1D61	PassengerDetails	PassengerID	Passenger	PassengerID

4. We want to analyze the total revenue by airline with the following query:

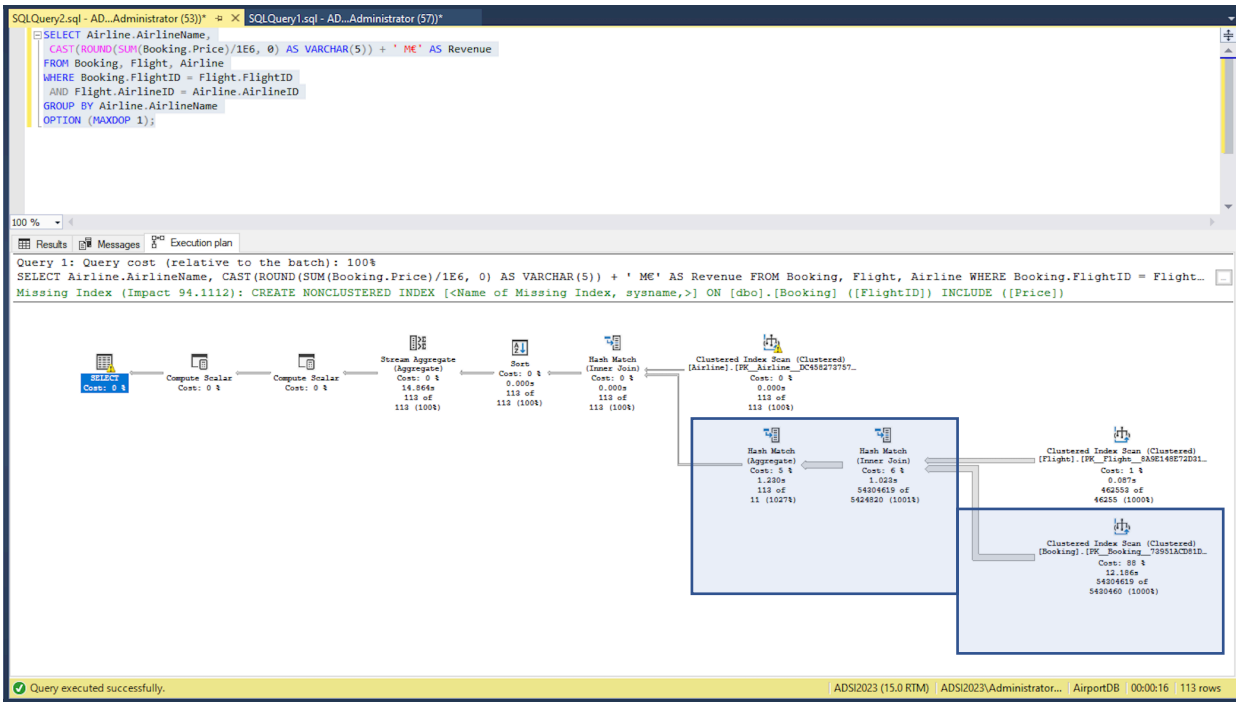
```
SELECT Airline.AirlineName,
       CAST(ROUND(SUM(Booking.Price)/1E6, 0) AS VARCHAR(5)) + ' M€' AS Revenue
FROM Booking, Flight, Airline
WHERE Booking.FlightID = Flight.FlightID
  AND Flight.AirlineID = Airline.AirlineID
GROUP BY Airline.AirlineName
OPTION (MAXDOP 1);
```

- SQL Server will recommend the creation of an index to optimize this query. If we create such index, how does the execution plan change? Present the following:
- The code to create and drop the suggested index. (Lab04)

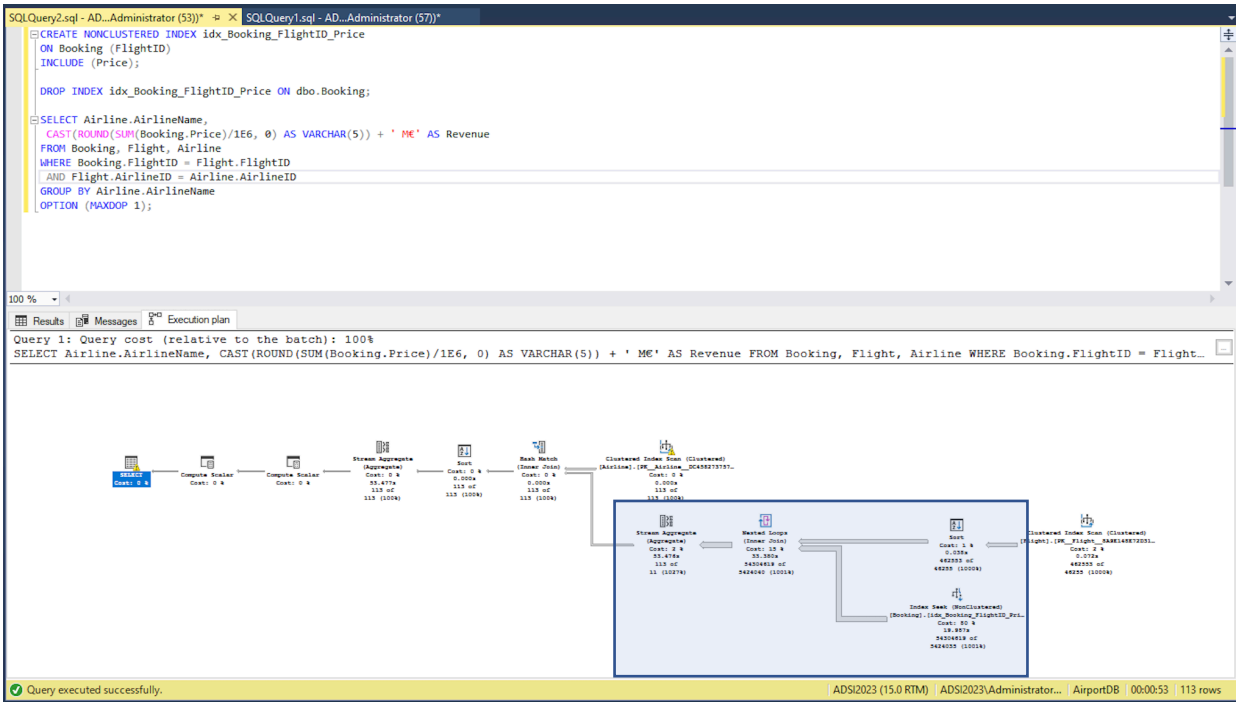
```
<Code to create Index>
CREATE NONCLUSTERED INDEX idx_Booking_FlightID_Price
ON dbo.Booking (FlightID)
INCLUDE (Price);
<Code to drop index>
DROP INDEX idx_Booking_FlightID_Price ON dbo.Booking;
```

- A picture of the old execution plan and of the new one, with a visual indication of the differences between them.

Old Execution Plan



New Execution Plan



- An explanation, in your own words, of what is being done differently in the new execution plan in comparison to the old one.

Initially it was performed a clustered index scan on the primary keys of the tables Flight and Booking, and after that, an inner join -hash match in both tables, then aggregate, another inner join - Hash Match between these results and primary key of the table Airline, and then sorting.

In order to improve the performance of this query, SQL Server advised to create a non-clustered index on the table 'dbo.Booking', column 'FlightID', including the column 'price'.

After the creation of the non-clustered index “idx_Booking_FlightID_Price”, it's performed an index seek in the index “idx_Booking_FlightID_Price” from table Booking, and then a nested loop with sorted primary key of the table Flight. This was aggregated and then applied a Hash Match between this result and the primary key of Airline.

What changes is that the SQL Server choses to perform a Nested Loop instead of a Hash Match whenever it's using the created non-clustered index. That can be related to the fact that in this way, the system can avoid building a large hash table in “FlightID” key to perform the join, what occupies a lot of memory. By using the Nested Loop, the system can iterate through the rows of the table Flight, that is smaller, and execute a lookup in the table Booking, and then can access directly the “Price” value in the non-clustered index for each row that matches, instead of having to access it in the Booking table.

5. The previous query used a query hint. What is this query hint being used for? Remove the query hint. How does your previous analysis change?

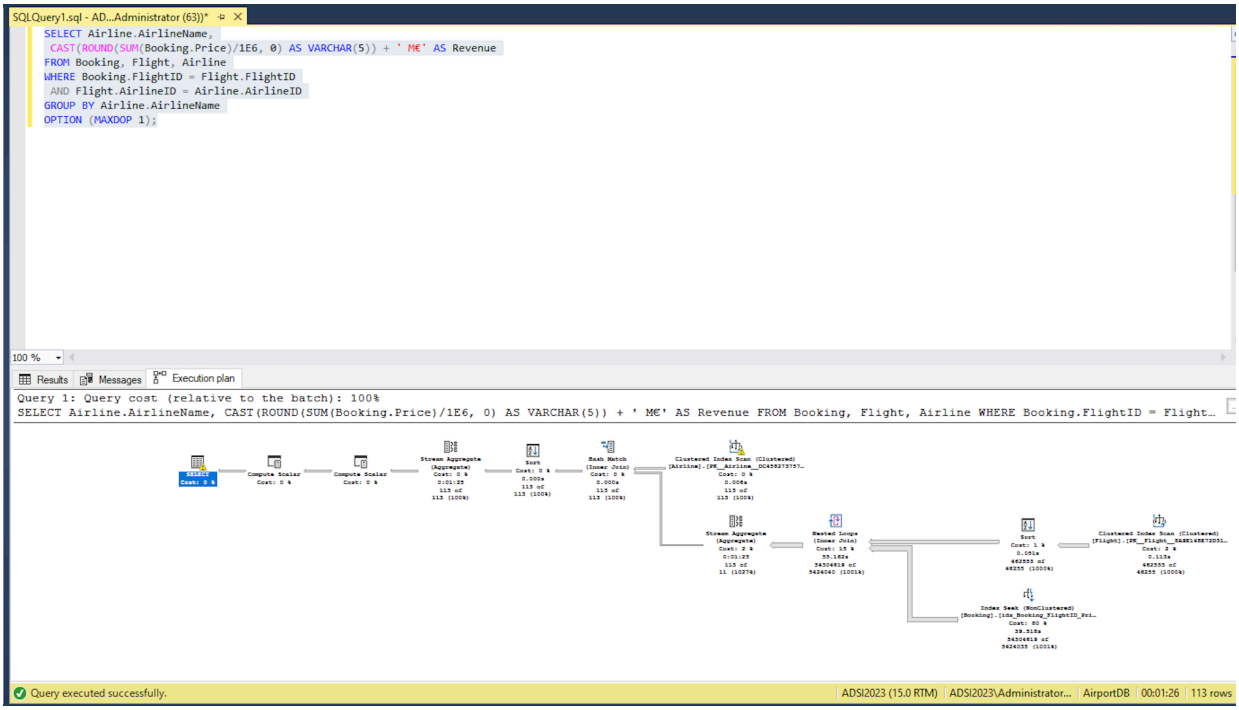
- An explanation of what the query hint is doing.

The query hint "OPTION (MAXDOP 1)" is limiting the maximum degree of parallelism used by the query to 1, that means the number of processors or cores that can be used to execute a single query. This strategy could be useful if the database server was under heavy load, or if there were other queries running concurrently that could interfere with this query.

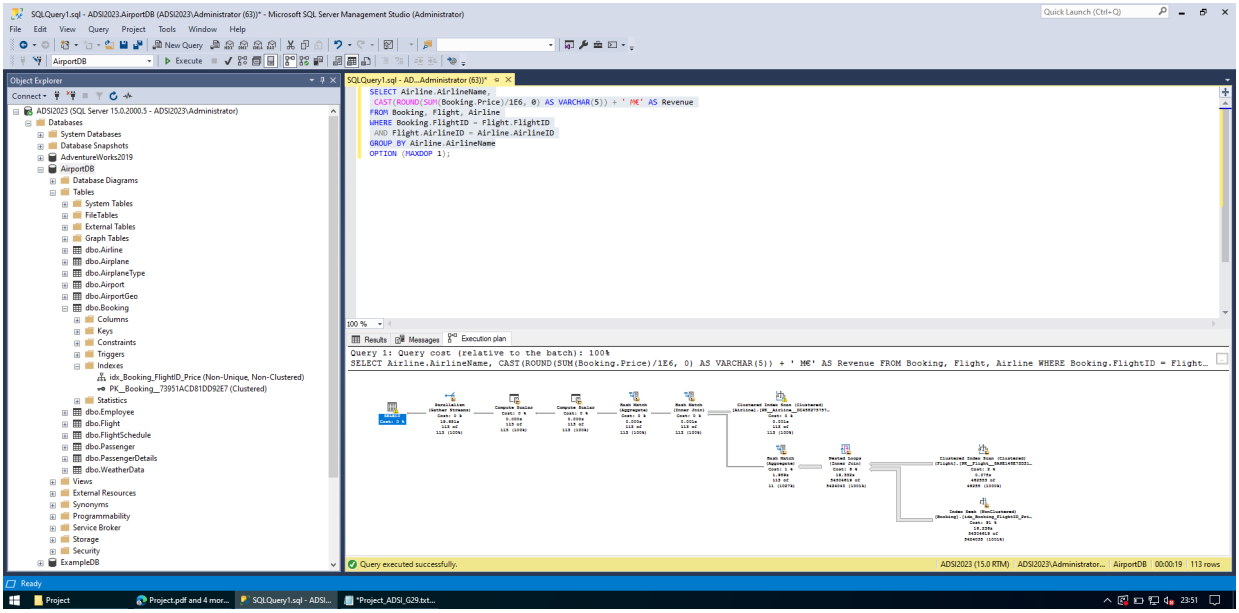
In a summary, when a query runs in parallel, two things can happen: improvement or loss of performance. In the first scenario, it can be divided into multiple tasks that can be executed simultaneously by different processors or cores, benefiting the performance. But it could also slow down the query or cause other issues. In such cases, limiting the DOP can be useful.

- A picture of the old execution plan and of the new one, with a visual indication of the differences between them.

Old Execution Plan



New Execution Plan



- An explanation, in your own words, of what is being done differently in the new execution plan in comparison to the old one.

Initially, it's performed an index seek in the non-clustered index “idx_Booking_FlightID_Price” from table Booking, and then a nested loop with the sorted primary key of the table Flight. This was aggregated and then applied a Hash Match between this result and the primary key of Airline.

In the execution without the hint, it’s noticed the following differences:

1. The system stops performing the sort in the table “Flight”, executing the Nest Loop directly with the Primary Key of the table Flight.
2. After that, it uses a Hash Match to aggregate the resulting join instead of the previous method of Steam Aggregate.
3. After performing the next Hash Match between the results of the previous execution with the table “Airline”, it executes a third Hash Match to introduce a new Aggregation, that wasn’t observed before.
4. In the final step, it includes a Parallelism to gather streams.

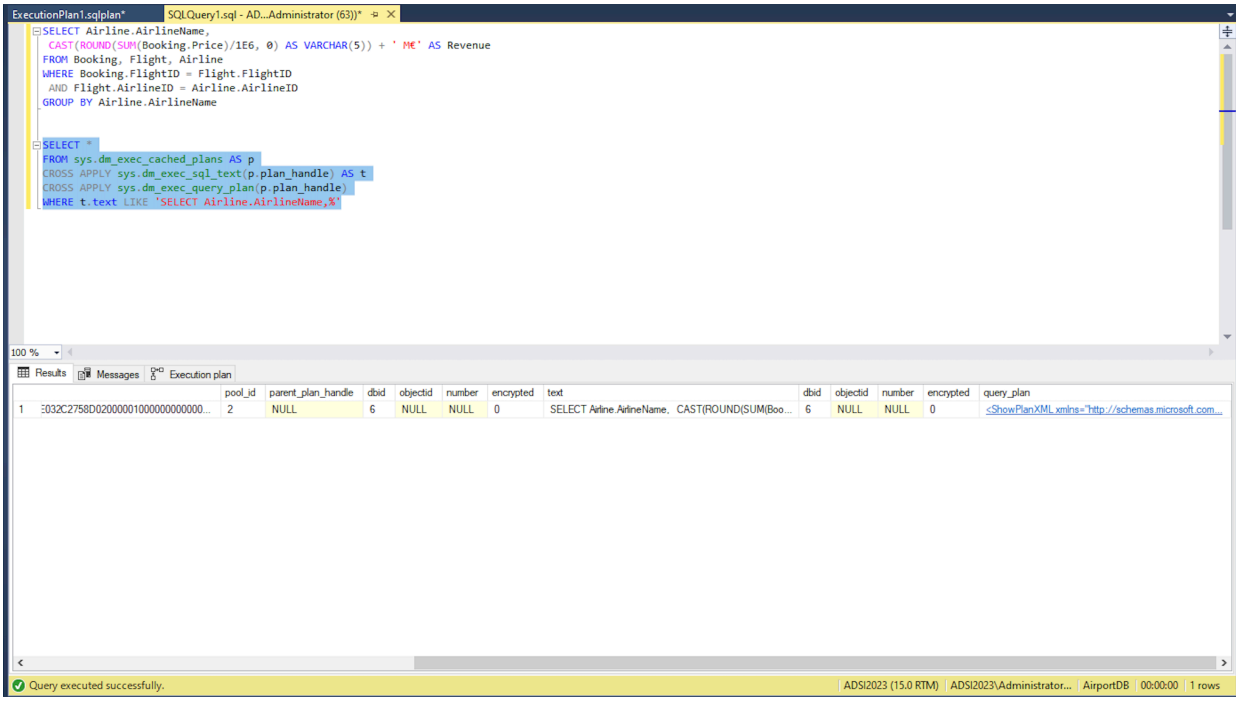
We conclude that the main difference in the execution plan after removing the hint is precisely parallelism. The database engine says performance would improve if parallelism is prioritized and that’s the reason why it ends up choosing that route. For the Hash Match operator, this means that each processor core can work on a separate portion of the data, performing the necessary aggregation operations in parallel. Once all the parallel processing is completed, the results are combined to produce the final aggregated result.

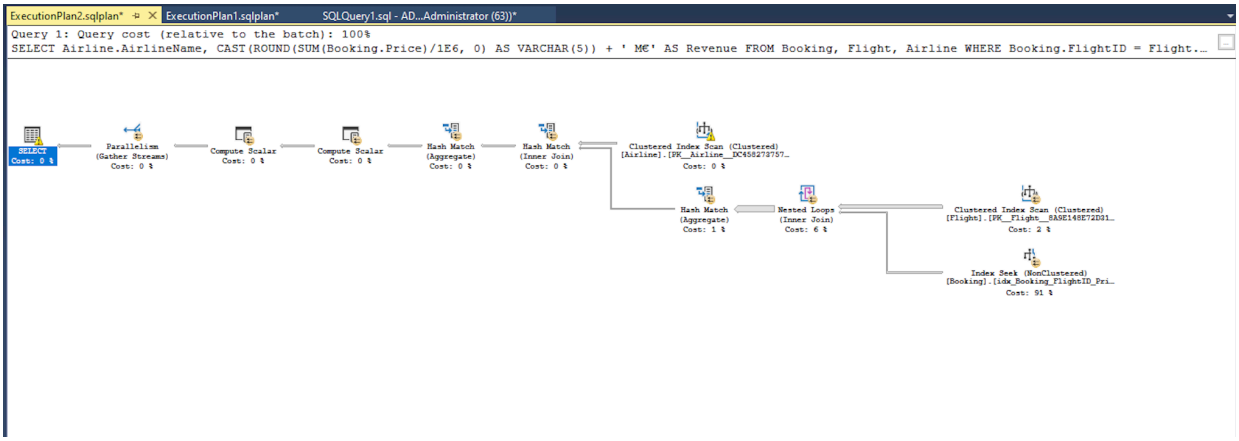
6. Write a query to retrieve, from the plan cache, all the execution plans for the query above, in the version without the query hint. Present the following: (Lab05)

- The query and the results.

```
SELECT *
FROM sys.dm_exec_cached_plans AS p
CROSS APPLY sys.dm_exec_sql_text(p.plan_handle) AS t
CROSS APPLY sys.dm_exec_query_plan(p.plan_handle)
WHERE t.text LIKE 'SELECT Airline.AirlineName,%'
```

- A picture of the execution plan (or one of the execution plans) that you used the most.





- Explain how you found the execution plan that you used the most.

By using the function sys.dm_exec_cached_plans, and specifying WHERE t.text LIKE 'SELECT Airline.AirlineName,%', the query executed returned a row for the only query plan that is cached by SQL Server for the query started as 'SELECT Airline.AirlineName,%'.

7. The Weather column in the WeatherData table has some terms that need to be translated from German to English (e.g. Schneefall ,Üí Snowfall, Regen ,Üí Rain, etc.). Is there any way to run these updates in parallel, with multiple transactions updating different records concurrently? Present the following: (Class 6 – Transactions) (Lab 6)

- A description of your attempt(s) to do this, including all the SQL code that you have used.

In order to simulate multiple transactions updating different records concurrently, it was initially opened two different query windows. In the first query window, it was written two queries to update the records Schneefall -> Snowfall, Regen -> Rain in the WeatherData Table, as follows:

```
BEGIN TRANSACTION;

UPDATE WeatherData
SET Weather = 'Rain'
WHERE Weather = 'Regen';

UPDATE WeatherData
SET Weather = 'Snowfall'
WHERE Weather = 'Schneefall';
```

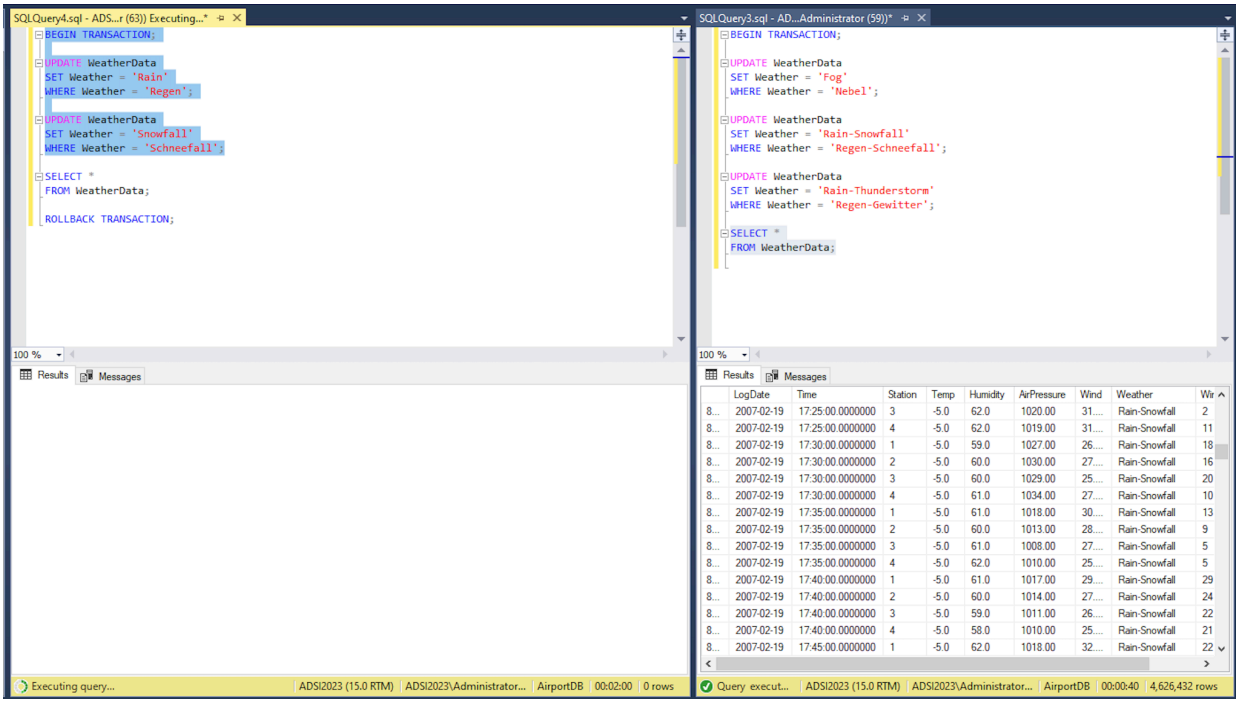
In the second query window it was written three queries to update the records Regen-Schneefall -> Rain-Snowfall, Regen-Gewitter -> Rain-Thunderstorm, Nebel -> Fog, in the WeatherData Table, as follows:

```
BEGIN TRANSACTION;

UPDATE WeatherData
SET Weather = 'Fog'
WHERE Weather = 'Nebel';

UPDATE WeatherData
SET Weather = 'Rain-Snowfall'
WHERE Weather = 'Regen-Schneefall';

UPDATE WeatherData
SET Weather = 'Rain-Thunderstorm'
WHERE Weather = 'Regen-Gewitter';
```

The second query described above was executed first, without commit. After it was executed, it was verified if the updates were successful:

```
SELECT *
FROM WeatherData;
```

The items were updated, and then it was executed the query one, trying to update Schneefall and Regen, but this query didn't conclude.



- If it didn't work, a detailed and technical explanation of why it cannot be achieved.
- It if worked, a detailed and technical explanation of what needs to be done to achieve it.
  Note: Be prepared to investigate what is happening in the database system in terms of locking and consider the use of techniques that go beyond what we addressed in this course.

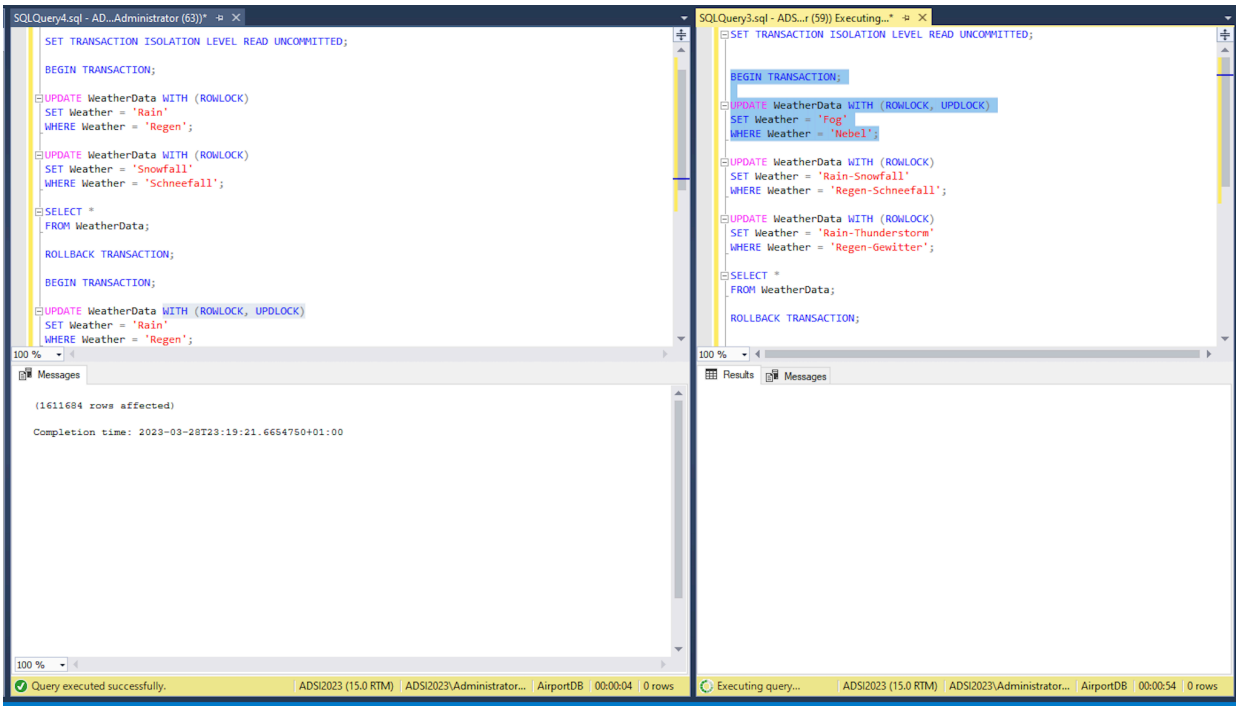
In MS SQL Server, the standard locking procedure to avoid that concurrent transactions overwrite each other's changes is based on "lock granularity". Lock granularity is essentially the amount of data locked as part of a query or update to provide complete isolation and serialization for the transaction, and it involves the use of shared and exclusive locks.

Whenever a transaction writes, modifies data, it will have an exclusive lock on the data, and this prevents other transactions from reading or modifying this data until the first transaction releases the lock that was set.

SQL Server can acquire locks at the row or page level, depending on the size of the data being modified. If a transaction updates a single row in a table, SQL Server will probably put a lock on that row, but in the other hand, if a transaction updates a larger set of rows or pages, the system can perform a lock escalation and put locks at the table level. According to the Microsoft definition, “Lock escalation is the process of converting many fine-grain locks into fewer coarse-grain locks, reducing system overhead while increasing the probability of concurrency contention.”

In this sense, The Query 1 couldn't write because SQL Server placed an exclusive lock in the entire table while running Query 2, that was modifying the data. Since query 2 was not committed, the transaction didn't end and it didn't release the lock.

In order to verify a possible exception to this lock escalation, it was tried changing the isolation level of the queries to snapshot and uncommitted reads. It was also tried using 'ROWLOCK', and timestamps, but no success was achieved, since Query 1 was never concluded without committing Query 2.



8. Create a materialized view of the employees who are also passengers. Use this view to count how many of such employees exist per country. Make sure that the view is used in the execution plan. Present the SQL instructions to create the view, the query, and the results.

Initially, the attempt to create the materialized view was through the following query:

```
CREATE VIEW MV_passenger_employee AS
SELECT e.FirstName, e.LastName, e.Country, e.EmployeeID
FROM Employee AS e, Passenger AS p
WHERE e.FirstName = p.FirstName AND e.LastName = p.LastName
```

```
ORDER BY e.EmployeeID
```

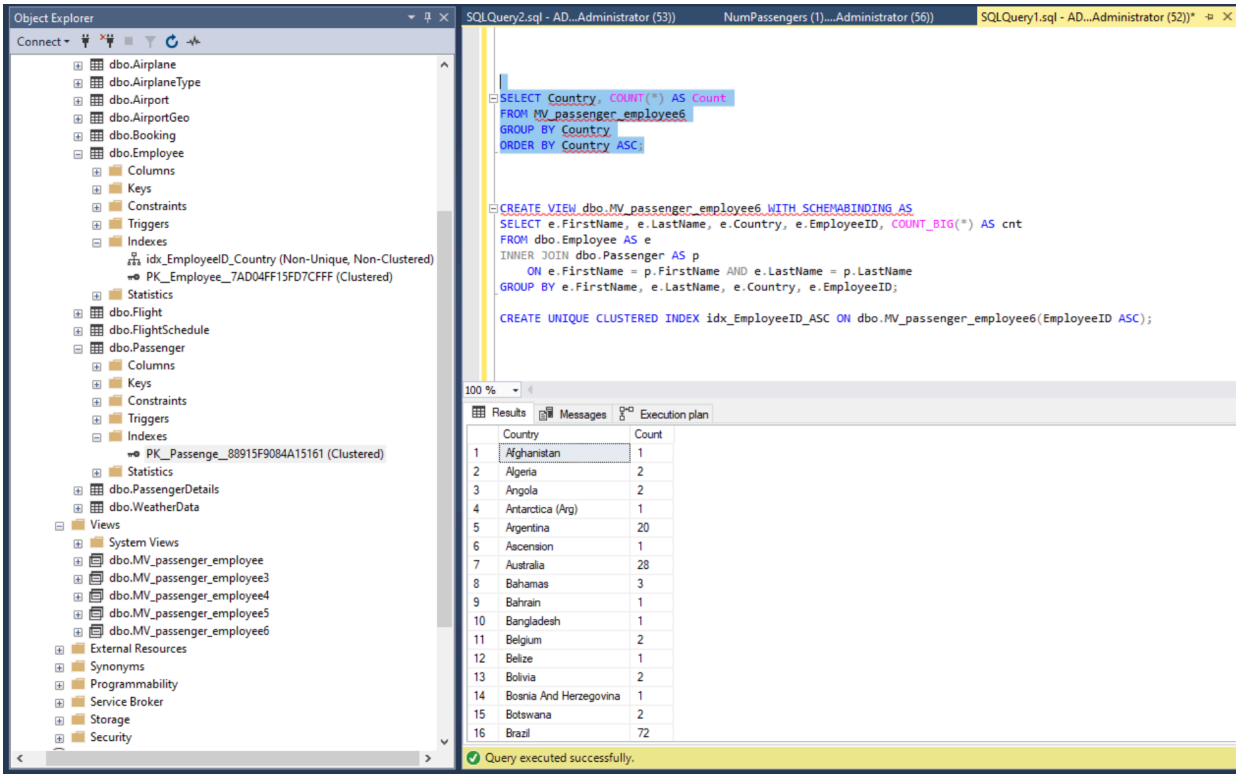
But then, it was realized two problems: the materialized view presented repeated EmployeeID values, what means that the JOIN between tables was not being performed properly, and the Execution Plan still preferred to perform a Hash Match between the tables Employee and Passenger instead of using the view.

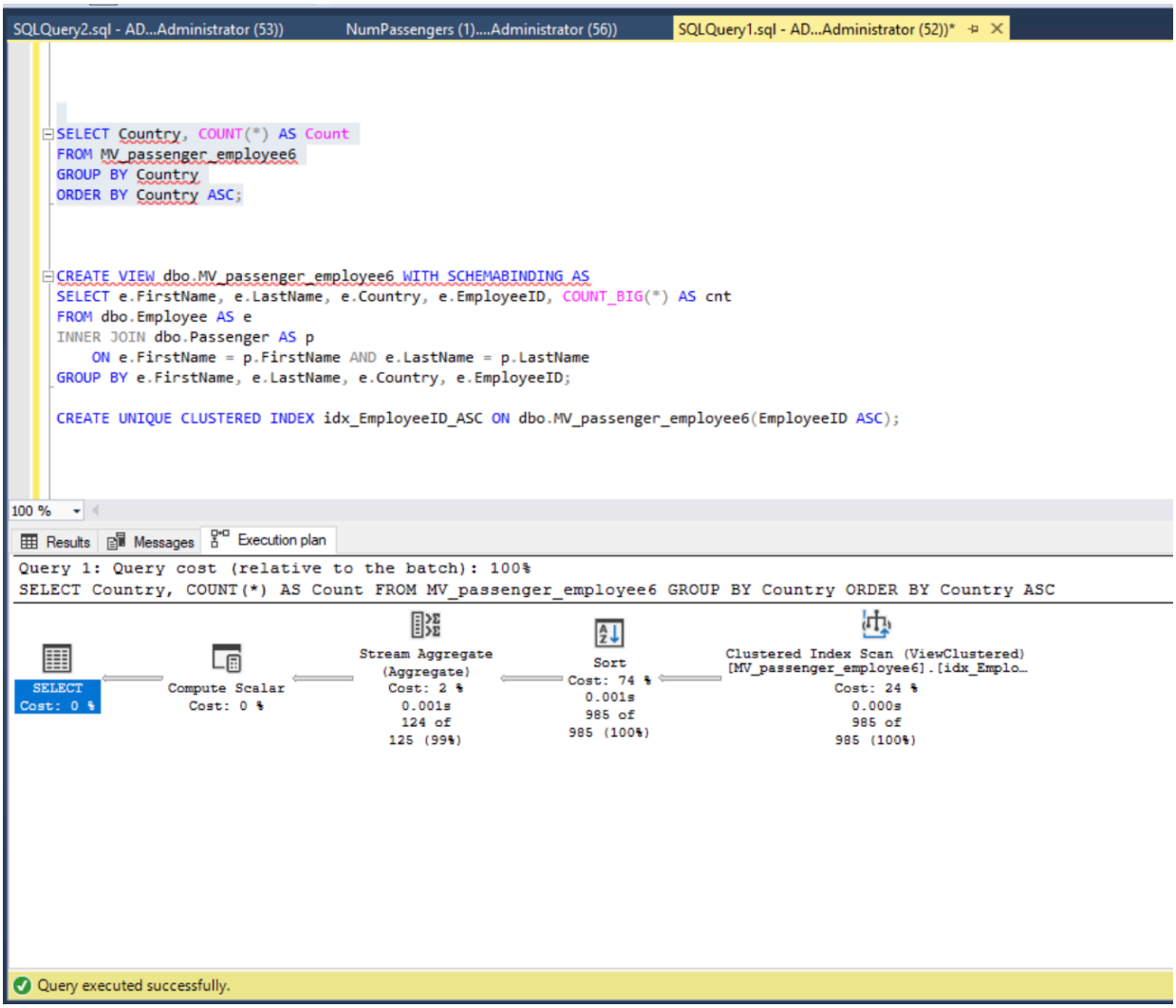
At this point it was realized that no clustered index was created and that's what materializes the view. During the attempt of creating the clustered index, the SQL Server provided a few change requirements to the query, such as guaranteeing the SCHEMABINDING, and adding Count Big to the SELECT list. After fixing the issues, the query for the materialized view creation and clustered index creation was the following:

```
CREATE VIEW dbo.MV_passenger_employee6 WITH SCHEMABINDING AS
SELECT e.FirstName, e.LastName, e.Country, e.EmployeeID,
COUNT_BIG(*) AS cnt
FROM dbo.Employee AS e
INNER JOIN dbo.Passenger AS p
    ON e.FirstName = p.FirstName AND e.LastName = p.LastName
GROUP BY e.FirstName, e.LastName, e.Country, e.EmployeeID;

CREATE UNIQUE CLUSTERED INDEX idx_EmployeeID_ASC ON
dbo.MV_passenger_employee6(EmployeeID ASC);
```

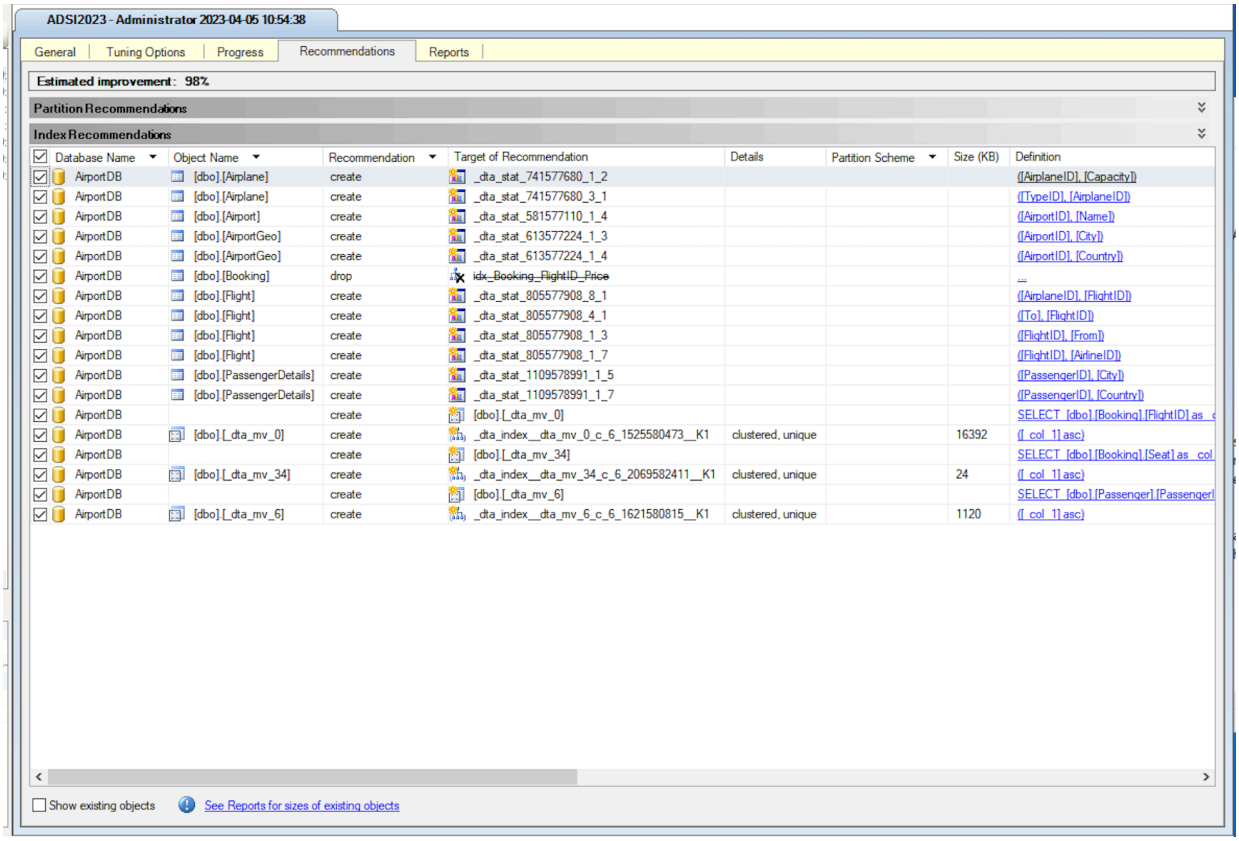
The query to count how many of such employees exist per country, using the created materialized view was the following:

```
SELECT Country, COUNT(*) AS Count
FROM MV_passenger_employee6
GROUP BY Country
ORDER BY Country ASC;
```


Object Explorer
Connect
dbo.Airplane
dbo.AirplaneType
dbo.Airport
dbo.AirportGeo
dbo.Booking
dbo.Employee
Columns
Keys
Constraints
Triggers
Indexes
idx_EmployeeID_Country (Non-Unique, Non-Clustered)
PK__Employee__7AD04FF15FD7CFFF (Clustered)
Statistics
dbo.Flight
dbo.FlightSchedule
dbo.Passenger
Columns
Keys
Constraints
Triggers
Indexes
PK__Passenge__88915F9084A15161 (Clustered)
Statistics
dbo.PassengerDetails
dbo.WeatherData
Views
System Views
dbo.MV_passenger_employee
dbo.MV_passenger_employee3
dbo.MV_passenger_employee4
dbo.MV_passenger_employee5
dbo.MV_passenger_employee6
External Resources
Synonyms
Programmability
Service Broker
Storage
Security
SQLQuery2.sql - AD...Administrator (53))
NumPassengers (1)....Administrator (56))
SQLQuery1.sql - AD...Administrator (52))*
SELECT Country, COUNT(*) AS Count
FROM MV_passenger_employee6
GROUP BY Country
ORDER BY Country ASC;
CREATE VIEW dbo.MV_passenger_employee6 WITH SCHEMABINDING AS
SELECT e.FirstName, e.LastName, e.Country, e.EmployeeID, COUNT_BIG(*) AS cnt
FROM dbo.Employee AS e
INNER JOIN dbo.Passenger AS p
ON e.FirstName = p.FirstName AND e.LastName = p.LastName
GROUP BY e.FirstName, e.LastName, e.Country, e.EmployeeID;
CREATE UNIQUE CLUSTERED INDEX idx_EmployeeID_ASC ON dbo.MV_passenger_employee6(EmployeeID ASC);
100 %
Results
Messages
Execution plan
Country
Count
1 Afghanistan 1
2 Algeria 2
3 Angola 2
4 Antarctica (Arg) 1
5 Argentina 20
6 Ascension 1
7 Australia 28
8 Bahamas 3
9 Bahrain 1
10 Bangladesh 1
11 Belgium 2
12 Belize 1
13 Bolivia 2
14 Bosnia And Herzegovina 1
15 Botswana 2
16 Brazil 72
Query executed successfully.


SQLQuery2.sql - AD...Administrator (53))
NumPassengers (1)....Administrator (56))
SQLQuery1.sql - AD...Administrator (52))*
SELECT Country, COUNT(*) AS Count
FROM MV_passenger_employee6
GROUP BY Country
ORDER BY Country ASC;
CREATE VIEW dbo.MV_passenger_employee6 WITH SCHEMABINDING AS
SELECT e.FirstName, e.LastName, e.Country, e.EmployeeID, COUNT_BIG(*) AS cnt
FROM dbo.Employee AS e
INNER JOIN dbo.Passenger AS p
ON e.FirstName = p.FirstName AND e.LastName = p.LastName
GROUP BY e.FirstName, e.LastName, e.Country, e.EmployeeID;
CREATE UNIQUE CLUSTERED INDEX idx_EmployeeID_ASC ON dbo.MV_passenger_employee6(EmployeeID ASC);
100 %
Results
Messages
Execution plan
Query 1: Query cost (relative to the batch): 100%
SELECT Country, COUNT(*) AS Count FROM MV_passenger_employee6 GROUP BY Country ORDER BY Country ASC
SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 2 %
0.001s
124 of
125 (99%)
Sort
Cost: 74 %
0.001s
985 of
985 (100%)
Clustered Index Scan (ViewClustered)
[MV_passenger_employee6].[idx_Emplo...
Cost: 24 %
0.000s
985 of
985 (100%)
Query executed successfully.

9. The file “NumPassengers.sql” contains a series of queries to compute the number of passengers according to different criteria. Use the Database Engine Tuning Advisor to get recommendations on how to optimize these queries. Provide a summary of these recommendations. In particular: (Lab10)



- Provide a summary, in your own words, of the indexes that DTA recommends creating.

Each materialized view created requires a clustered index in order to be materialized. So, it was also displayed recommendation for creation of 3 new index:

```
Index dta_mv_0

Column 1 of MV1 [_dta_mv_0] - (index of the [FlightID] column in
the table Booking)

Index dta_mv_34

Column 1 of MV2 [_dta_mv_34] - (index of the [Seat] column in the
Booking table)

Index dta_mv_6
```

```
Column 1 of MV3 [_dta_mv_6] - (index of the [PassengerID] column
in the table Passenger)
```

- Similarly, provide a summary of the materialized views that DTA recommends creating.

```
MV1 query:

CREATE VIEW [dbo].[_dta_mv_0] WITH SCHEMABINDING
 AS
SELECT [dbo].[Booking].[FlightID] as _col_1,
count_big([AirportDB].[dbo].[Booking].[PassengerID]) as _col_2,
count_big(*) as _col_3 FROM  [dbo].[Booking]
GROUP BY  [dbo].[Booking].[FlightID]
```

This query creates a view [_dta_mv_0]. It includes a SELECT that retrieves data from the table Booking and performs some calculations on the data. The SELECT retrieves three columns of data:

_col_1: This column includes the attribute from [FlightID] column.

_col_2: This column uses the COUNT_BIG function to count the number of rows in the table where the [PassengerID] column isn't null, meaning that it's assigned to a FlightID.

_col_3: This column shows the total number of bookings made for each flight, regardless of having Passengers assigned to the flight or not (Counts all rows from Booking).

The GROUP BY groups the data by the attribute [FlightID], making the view return one row for each value in the [FlightID] column, and the values in the _col_2 and _col_3 are then aggregated based on the grouping.

The SCHEMABINDING option binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition.

```
MV2 query:

CREATE VIEW [dbo].[_dta_mv_34] WITH SCHEMABINDING
 AS
SELECT [dbo].[Booking].[Seat] as _col_1,
count_big([AirportDB].[dbo].[Booking].[PassengerID]) as _col_2,
count_big(*) as _col_3 FROM  [dbo].[Booking]
GROUP BY [dbo].[Booking].[Seat]
```

This query creates a view [_dta_mv_34]. It includes a SELECT that retrieves data from the table Booking and performs some calculations on the data. The SELECT retrieves three columns of data:

_col_1: This column includes the attribute from [Seat] column.

_col_2: This column uses the COUNT_BIG function to count the number of rows in the table where the [PassengerID] column is assigned to a Seat.

_col_3: This column shows the total number of bookings made for each unique seat number, regardless of having Passengers assigned to the seat or not (Counts all rows from Booking).

The GROUP BY groups the data by the attribute [Seat], making the view return one row for each value in the [Seat] column, and the values in the _col_2 and _col_3 are then aggregated based on the grouping.

The SCHEMABINDING option binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition.

MV3 query:

```
CREATE VIEW [dbo].[_dta_mv_6] WITH SCHEMABINDING
 AS
SELECT  [dbo].[Passenger].[PassengerID] as _col_1,
[dbo].[Booking].[PassengerID] as _col_2,
count_big(*) as _col_3 FROM  [dbo].[Passenger],  [dbo].[Booking]
WHERE
[dbo].[Passenger].[PassengerID] = [dbo].[Booking].[PassengerID]
GROUP BY
[dbo].[Passenger].[PassengerID],  [dbo].[Booking].[PassengerID]
```

This query creates a view [_dta_mv_6]. It includes a SELECT that retrieves data from the tables “Booking” and “Passenger” and performs some calculations on the data. The SELECT retrieves three columns of data:

_col_1: This column includes the values from [PassengerID] column, from the Passenger table.

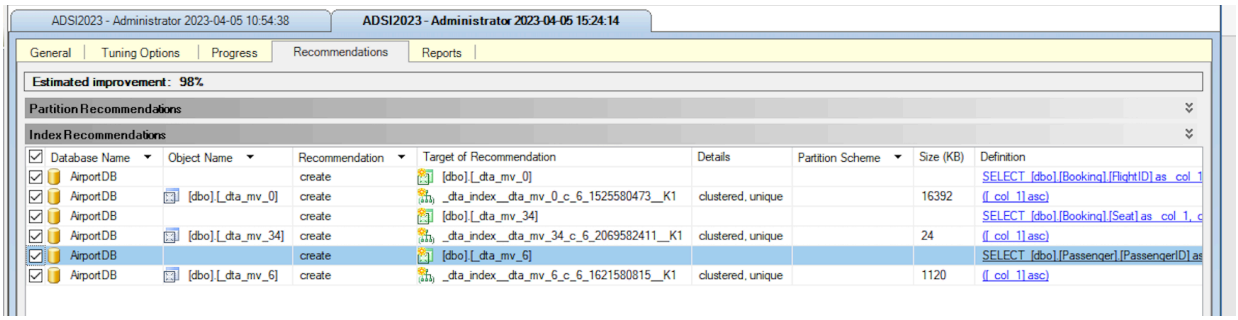
_col_2: This column includes the values from [PassengerID] column, from the Booking table.

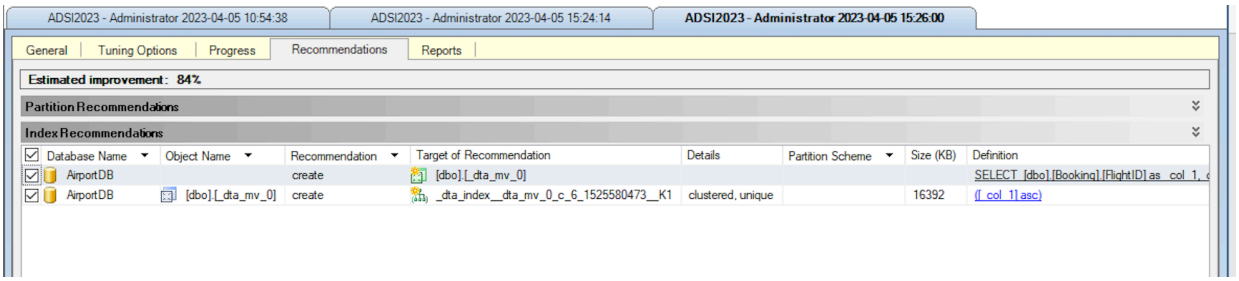
_col_3: This column uses the COUNT_BIG function to count the total number of rows in the result set. This result set is also defined by a WHERE clause, that specifies that the joining condition between tables Passenger and Booking should be on the column PassengerID, where the values of this column are the same in both tables. This provides a result set that includes then only the rows where a passenger has made a booking, and then, it shows the count of the number of bookings made by each passenger, where the passenger has made at least one booking.

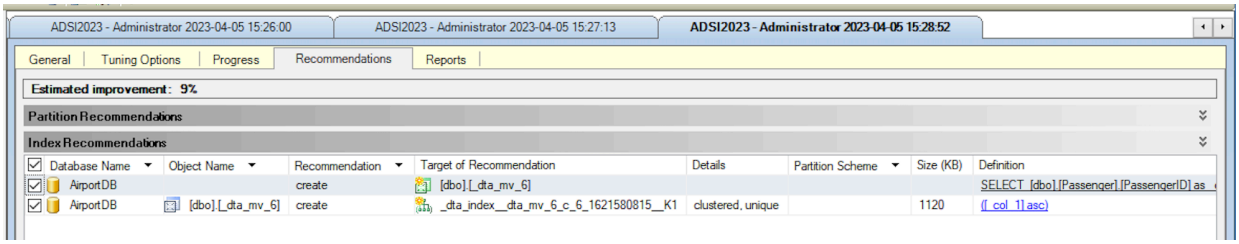
The GROUP BY groups the data by the combination of the attribute [PassengerID] from both the tables Passenger and Booking, returning one row for each combination.

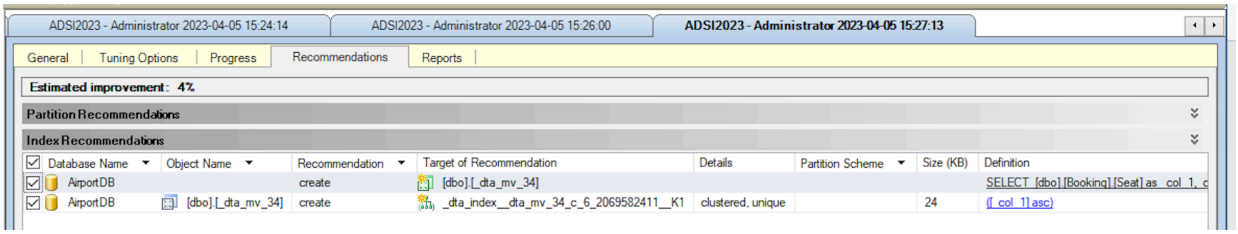
- Analyze and discuss which of these recommendations have the most impact on improving the performance for this workload, and why.

The total estimated improvement calculated by the DTA was 98%. The recommendation that has the greatest impact on improving the performance of this workload is the creation of the materialized view mv_0 (and its corresponding index), with 84% of estimated improvement, meanwhile the creation of mv_34 represents 4% of improvement and the creation of mv_6, 9% of improvement.









It's possible to notice that the materialized view mv_0 has a size of 16392KB, while mv_34 has 24KB and mv_6, 1120KB. That shows that when creating a materialized view with a clustered index on the attribute “FlightID”, instead of “Seats”, or “PassengersID”, it stored much more data in the view.

The view mv_0 has the highest estimated improvement of 84%. This view stores the total number of bookings made for each flight, and has a clustered index on the "FlightID" attribute, what suggests that the query workload includes queries that also aggregate booking data by flight, and that this view can improve performance by pre-aggregating this data. Since mv_0 stores a lot of data, it can be expected that many queries can benefit from using this view.

The view mv_6 has the second highest estimated improvement of 9%. This view stores the count of the number of bookings made by each passenger who has made at least one booking. By that, we can infer that the query workload includes queries that filter and aggregate booking data by passenger, but since mv_6 stores way less data than mv_0, it's possible that fewer queries can benefit from using this view.

In the last place, the materialized view mv_34 has the lowest estimated improvement percentage of 4%. This view stores the total number of bookings made for each unique seat number and has a clustered index on the "Seat" attribute, that indicates that the query workload includes queries that filter and aggregate booking data by seat. However, since this view stores very little data, it's possible that few queries can benefit from using this view, and then, the improvement for those queries is small.

Bibliographic References

Challenge 1

- Data Administration in Information Systems. Lab 2: Storage and file structure
- https://learn.microsoft.com/en-us/sql/t-sql/language-elements/case-transact-sql?view=sql-server-ver16. Retrieved April 9, 2023

Challenge 2

- Data Administration in Information Systems. Lab 3: Indexing

Challenge 3

- Data Administration in Information Systems. Lab 3: Indexing

Challenge 4

- https://learn.microsoft.com/en-us/answers/questions/85209/tipping-point-nested-loop-join-and-hash-join?sort=oldest. Retrieved March 30, 2023

Challenge 5

- https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-degree-of-parallelism-server-configuration-option?view=sql-server-ver16. Retrieved April 2, 2023

Challenge 6

- Data Administration in Information Systems. Lab 5: Query optimization

Challenge 7

- https://www.oreilly.com/library/view/microsoft-sql-server/9780133408539/ch33lev1sec7.html. Retrieved April 4, 2023.
- https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver16. Retrieved April 4, 2023
- Data Administration in Information Systems. Lab 6: Transactions and isolation levels.
- Data Administration in Information Systems. Transactions and Concurrency (Class 6).

Challenge 8

- Data Administration in Information Systems. Lab 8: Schema tuning.

Challenge 9

- https://learn.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver16. Retrieved April 5, 2023.
- Data Administration in Information Systems. Lab 10: Index tuning