A report on the Case Study for LotsGroup technical evaluation.

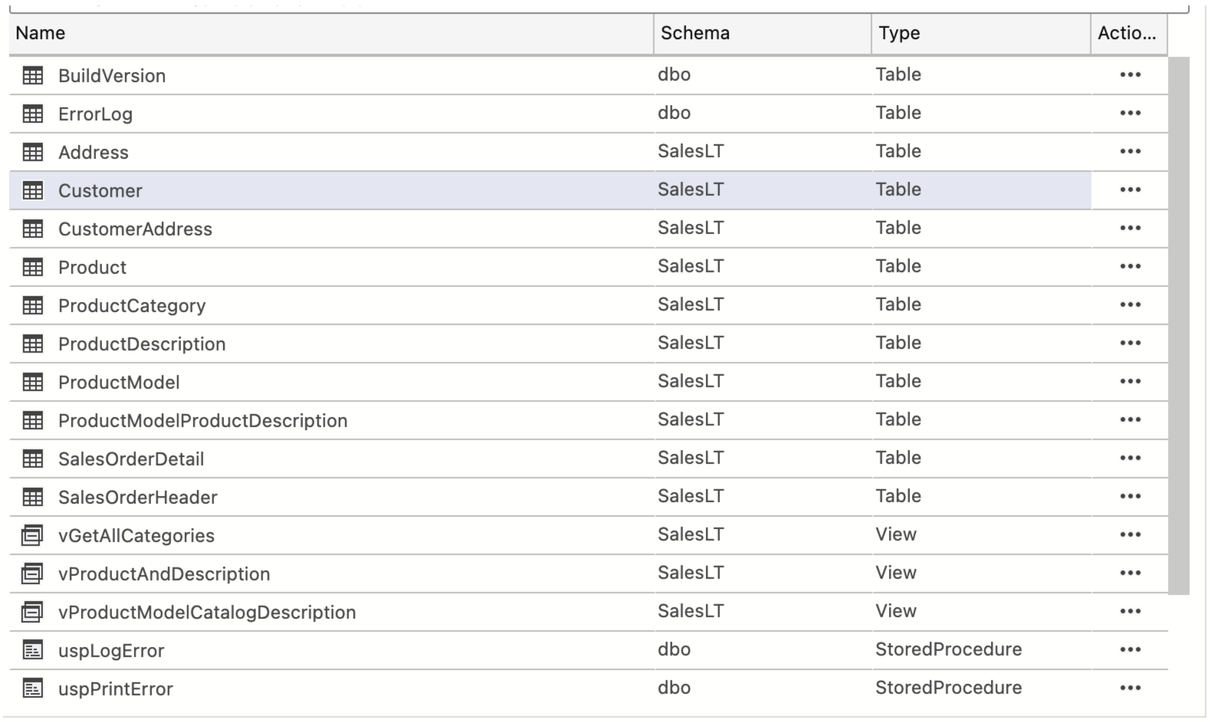
This is the standard Microsoft demo database consisting of sales of products to customers over time. Based on this database you will do some exercises. These are the tasks:

1) Start with a simple profiling of the data. Please report any irregular or inconsistent data.

In order to perform a profiling of the data and identify irregular or inconsistent data, I would need more specific information about the structure and content of the Microsoft demo database.

Could you provide some details about the tables, columns, and types of data present in the database? This would help me provide a more accurate analysis of the data quality issues.

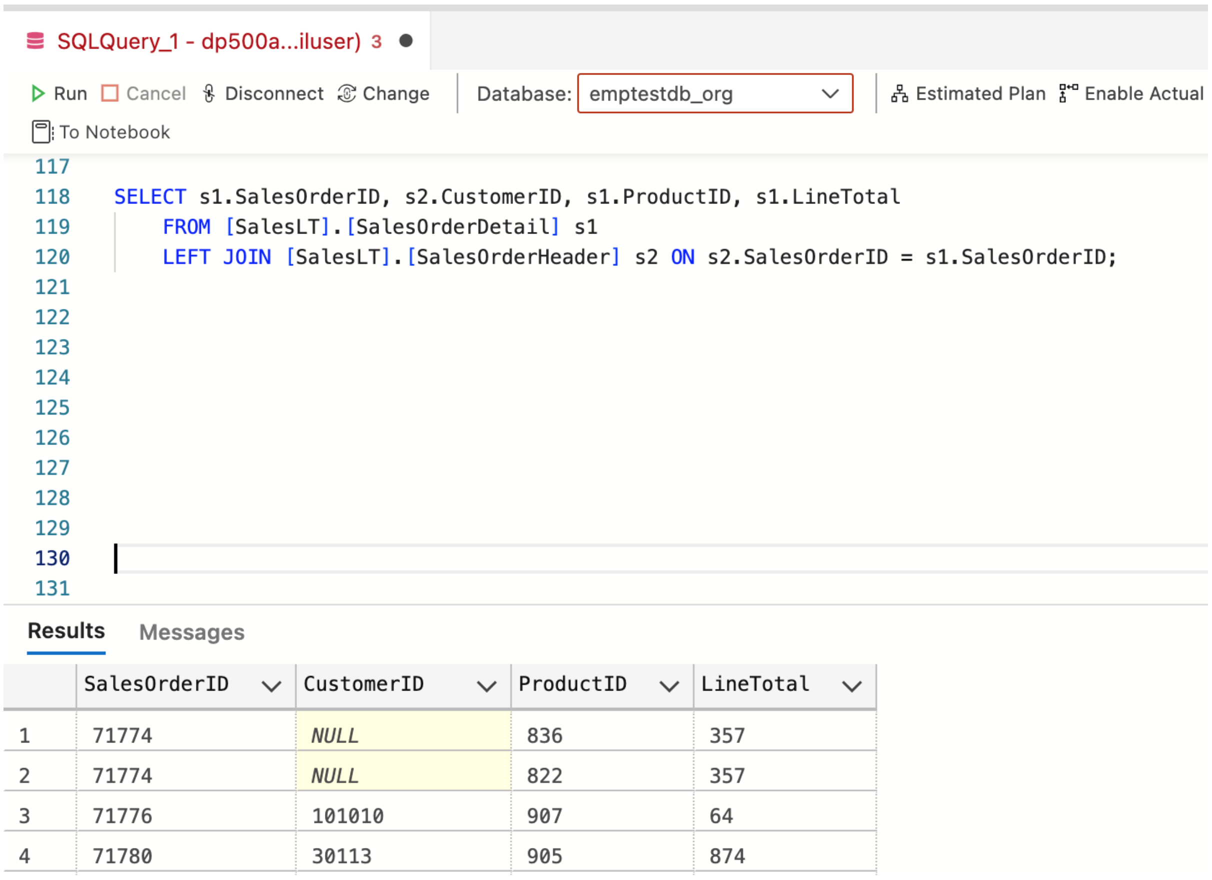
Access to lotsgroup:emptestdb_org has been granted successfully



Name	Schema	Type	Actio...
BuildVersion	dbo	Table	•••
ErrorLog	dbo	Table	•••
Address	SalesLT	Table	•••
Customer	SalesLT	Table	•••
CustomerAddress	SalesLT	Table	•••
Product	SalesLT	Table	•••
ProductCategory	SalesLT	Table	•••
ProductDescription	SalesLT	Table	•••
ProductModel	SalesLT	Table	•••
ProductModelProductDescription	SalesLT	Table	•••
SalesOrderDetail	SalesLT	Table	•••
SalesOrderHeader	SalesLT	Table	•••
vGetAllCategories	SalesLT	View	•••
vProductAndDescription	SalesLT	View	•••
vProductModelCatalogDescription	SalesLT	View	•••
uspLogError	dbo	StoredProcedure	•••
uspPrintError	dbo	StoredProcedure	•••

Important: while writing SQL scripts to create fact and dimension tables, it has been noticed NULL values to customers. It was assumed as a non-critical issue to investigate in another moment.

```sql
SELECT s1.SalesOrderID, s2.CustomerID, s1.ProductID, s1.LineTotal
FROM [SalesLT].[SalesOrderDetail] s1
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID;
```



```sql
SELECT s1.SalesOrderID, s2.CustomerID, s1.ProductID, s1.LineTotal
    FROM [SalesLT].[SalesOrderDetail] s1
    LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID;
```

	SalesOrderID	CustomerID	ProductID	LineTotal
1	71774	*NULL*	836	357
2	71774	*NULL*	822	357
3	71776	101010	907	64
4	71780	30113	905	874

2) Which are our top 3 customers based on sales (linetotal)

To determine the top 3 customers based on sales (linetotal), a SQL query has been written within an aggregate and rank of the customers by their total sales.

```
        SELECT TOP (3)
                c.[CustomerID], c.[FirstName], c.[LastName], c.[CompanyName]
                ,SUM(s2.[TotalDue]) AS total_sales
        FROM [SalesLT].[SalesOrderDetail] s1
        LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID =
s1.SalesOrderID
        LEFT JOIN [SalesLT].[Customer] c ON c.CustomerID = s2.CustomerID
        GROUP BY
                c.[CustomerID], c.FirstName, c.LastName, c.CompanyName
        ORDER BY
                total_sales DESC
```

In this query, [SalesLT] is the schema, while [Customer] and [SalesOrderDetails] are the target tables.

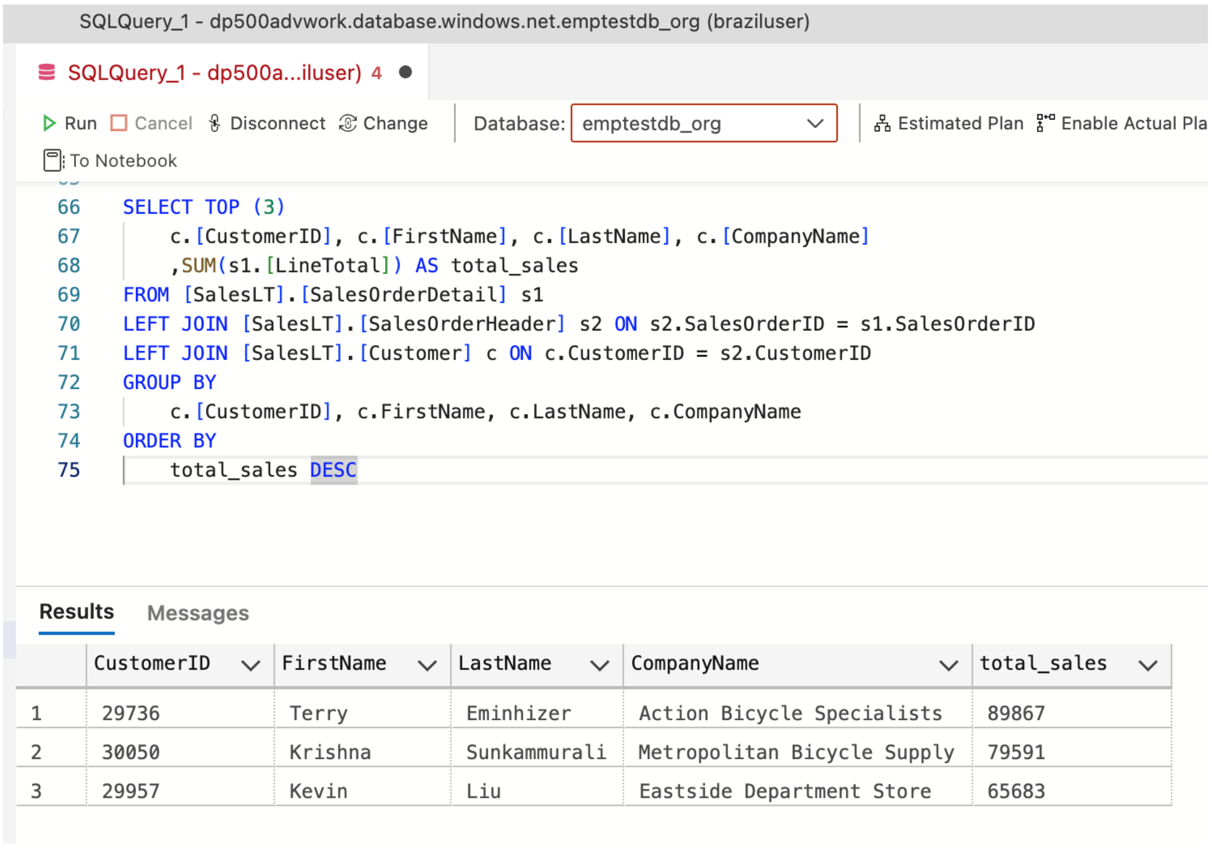
Two JOIN statements were written in order to retrieve detailed data of sales and customers, from SalesOrderDetail, SalesOrderHeader and Customer tables. SalesOrderHeader was the intermediary table to establish the relationship between LineTotal Customer_id. CustomerID was the key used in this relationship between SalesOrderHEader and Customer tables.

[CustomerID], [FirstName], [LastName], [CompanyName] are columns from Customer table

Furthermore, the SUM() aggregate function was used to calculate the total order that a respective customer has purchased in the whole period available/stored currently in the DataBase.

The query groups the data by customer ID, FirstName and LastName, and calculates the total sales for each customer, orders the results in descending order by total sales, and then limits the output to the top 3 rows.

The screenshot below shows evidence of execution, the query and its results.


SQLQuery_1 - dp500advwork.database.windows.net.emptestdb_org (braziluser)

```sql
SELECT TOP (3)
    c.[CustomerID], c.[FirstName], c.[LastName], c.[CompanyName]
    ,SUM(s1.[LineTotal]) AS total_sales
FROM [SalesLT].[SalesOrderDetail] s1
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID
LEFT JOIN [SalesLT].[Customer] c ON c.CustomerID = s2.CustomerID
GROUP BY
    c.[CustomerID], c.FirstName, c.LastName, c.CompanyName
ORDER BY
    total_sales DESC
```

	CustomerID	FirstName	LastName	CompanyName	total_sales
1	29736	Terry	Eminhizer	Action Bicycle Specialists	89867
2	30050	Krishna	Sunkammurali	Metropolitan Bicycle Supply	79591
3	29957	Kevin	Liu	Eastside Department Store	65683

3) We have realized that we have made a mistake. Our top 5 customers always get a discount of 10%. Please adjust these sales order lines and make a procedure that can be run to adjust this. Change the total sales for these customers to be 10% lower than now. The customers should be the top 5 customers based on the sales before discounts.

To adjust the sales order lines for the top 5 customers to apply a 10% discount, it has been written a similar SQL procedure from task 2, and added an extra column, which stores the Sales discounted total.

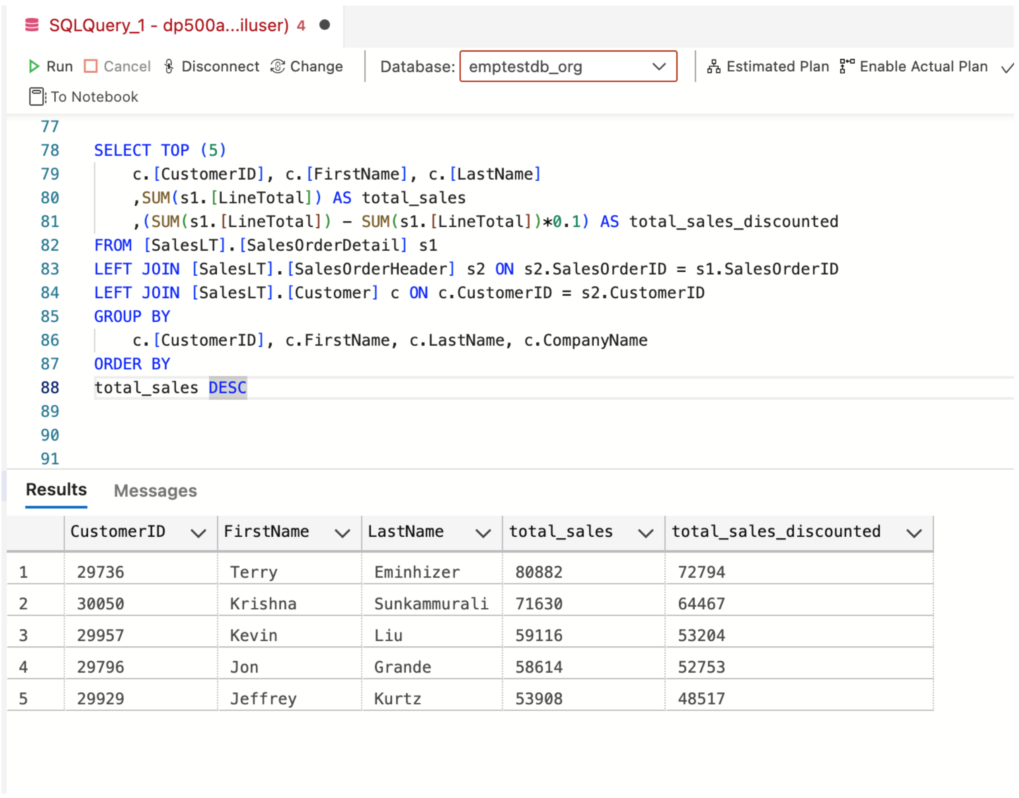
A general outline of the steps:

i. Identified the top 5 customers based on sales before discounts.

ii. Calculated the 10% discount for each customer's sales.

```
SELECT TOP (5)
c.[CustomerID], c.[FirstName], c.[LastName]
,SUM(s1.[LineTotal]) AS total_sales
,(SUM(s1.[LineTotal]) - SUM(s1.[LineTotal])*0.1) AS total_sales_discounted
FROM [SalesLT].[SalesOrderDetail] s1
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID =
s1.SalesOrderID
LEFT JOIN [SalesLT].[Customer] c ON c.CustomerID = s2.CustomerID
GROUP BY
c.[CustomerID], c.FirstName, c.LastName, c.CompanyName
ORDER BY
total_sales DESC
```

The screenshot below shows evidences of execution, the query and its results.



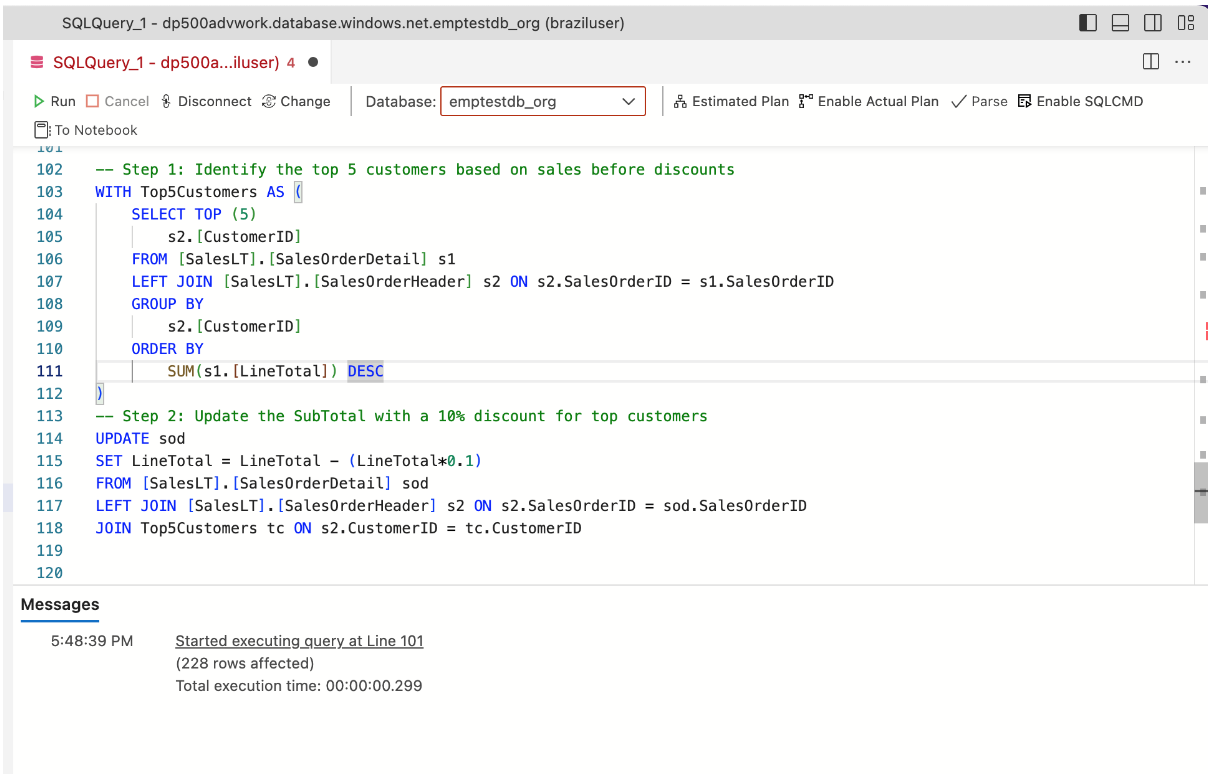
```
SELECT TOP (5)
    c.[CustomerID], c.[FirstName], c.[LastName]
    ,SUM(s1.[LineTotal]) AS total_sales
    ,(SUM(s1.[LineTotal]) - SUM(s1.[LineTotal])*0.1) AS total_sales_discounted
FROM [SalesLT].[SalesOrderDetail] s1
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID
LEFT JOIN [SalesLT].[Customer] c ON c.CustomerID = s2.CustomerID
GROUP BY
    c.[CustomerID], c.FirstName, c.LastName, c.CompanyName
ORDER BY
total_sales DESC
```

	CustomerID	FirstName	LastName	total_sales	total_sales_discounted
1	29736	Terry	Eminhizer	80882	72794
2	30050	Krishna	Sunkammurali	71630	64467
3	29957	Kevin	Liu	59116	53204
4	29796	Jon	Grande	58614	52753
5	29929	Jeffrey	Kurtz	53908	48517

Another query, slightly different from the query above, was written to actually update the sales order lines for these customers to apply the discount, as following below.

```
-- Step 1: Identify the top 5 customers based on sales before discounts
WITH Top5Customers AS (
SELECT TOP (5)
s2.[CustomerID]
FROM [SalesLT].[SalesOrderDetail] s1
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID
GROUP BY
s2.[CustomerID]
ORDER BY
SUM(s1.[LineTotal]) DESC
)
-- Step 2: Update the SubTotal with a 10% discount for top customers
UPDATE sod
SET LineTotal = LineTotal - (LineTotal*0.1)
FROM [SalesLT].[SalesOrderDetail] sod
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = sod.SalesOrderID
JOIN Top5Customers tc ON s2.CustomerID = tc.CustomerID
```

The screenshot below shows evidences of execution, the query and its results.



```
-- Step 1: Identify the top 5 customers based on sales before discounts
WITH Top5Customers AS (
    SELECT TOP (5)
        s2.[CustomerID]
    FROM [SalesLT].[SalesOrderDetail] s1
    LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = s1.SalesOrderID
    GROUP BY
        s2.[CustomerID]
    ORDER BY
        SUM(s1.[LineTotal]) DESC
)
-- Step 2: Update the SubTotal with a 10% discount for top customers
UPDATE sod
SET LineTotal = LineTotal - (LineTotal*0.1)
FROM [SalesLT].[SalesOrderDetail] sod
LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID = sod.SalesOrderID
JOIN Top5Customers tc ON s2.CustomerID = tc.CustomerID
```

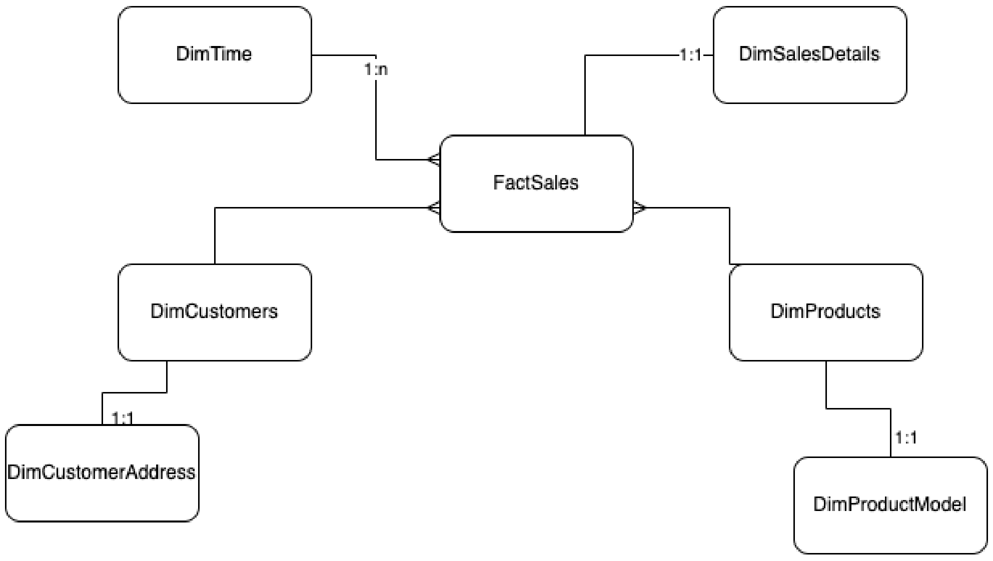
```
5:48:39 PM    Started executing query at Line 101
              (228 rows affected)
              Total execution time: 00:00:00.299
```

Additionally, it is important to note that none appropriate backups and a proper testing environment before making bulk updates were executed on the source database. It is notably recommendable to create backups and a proper test the environment before making such bulk updates

4) Make a star schema on this data. You can do either a virtual star schema or a physical one. The star schema should be in a schema called DW.

Designing a virtual star schema for the provided sales data. In a virtual schema, we create views that represent the star schema structure without physically restructuring the data.

In this case, I'll assume you have three main entities: Sales, Customers, and Products.



Regarding star schema, the scripts used to create DW schema, fact nad dimension tables were presented below. We might set up the virtual star schema using views in a schema named DW:

Fact table now has the advantage to store product SalesOrderID, CustomerID and ProductID in the same structure (view/table).

Furthermore, besides foreign keys, it was added one columns only: LineTotal containing the amount of the sales Order

In the future, after discussing the team, more fact and dimension tables might be amended. Every element in the DW might be adjusted later such as: the column names and join conditions to match the actual database structure.

Also, consider adding more dimensions if it has additional attributes that should be part of the star schema. Columns might be added or even removed.

-- Create the DW schema

```
CREATE SCHEMA DW;
```

-- Create a view for the fact table (Sales)

```
CREATE OR ALTER VIEW DW.FactSales AS
	SELECT s1.SalesOrderID, s2.CustomerID, s1.ProductID, s1.LineTotal
	FROM [SalesLT].[SalesOrderDetail] s1
	LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID =
	s1.SalesOrderID;



CREATE OR ALTER VIEW DW.DimSales AS
	SELECT s1.SalesOrderID, s1.SalesOrderDetailID, s2.SalesOrderNumber
	FROM [SalesLT].[SalesOrderDetail] s1
	LEFT JOIN [SalesLT].[SalesOrderHeader] s2 ON s2.SalesOrderID =
	s1.SalesOrderID;

-- Create a view for the dimension table Products
CREATE OR ALTER VIEW DW.DimProducts AS
	SELECT p.ProductID, p.Name
	FROM [SalesLT].[Product] p;

-- Create a view for the dimension table Customers
CREATE OR ALTER VIEW DW.DimCustomers AS
	SELECT c.CustomerID, c.FirstName, c.LastName
	FROM [SalesLT].[Customer] c;



-- Create a view for the dimension table Dates (Assuming date-related data is
available)
CREATE OR ALTER VIEW DW.DimTime AS
	SELECT
		ABS(CONVERT(INT, CONVERT(VARBINARY(4), HASHBYTES('SHA1',
		CONVERT(VARCHAR(19), t.OrderDate, 120))))) AS time_key,
		CONVERT(VARCHAR(10), t.OrderDate, 23) AS formatted_date,
		DATEPART(WEEKDAY, t.OrderDate) AS day_of_week,
		DATENAME(month, t.OrderDate) AS month_name
	FROM [SalesLT].[SalesOrderHeader] t;
```

The virtual star schema is set up in the DW schema. In this virtual star schema, only views were created that represent the fact table (FactSales) and dimension tables (DimCustomers, DimProducts, DimDates).

These views allow you to query the data as if it were structured in a star schema, even though the physical data storage remains unchanged.

5) Classify the customers in 2 groups. Those that have had more than 30 sales order rows and those that have 30 or less sales order rows. Add this information to the customer dimension.

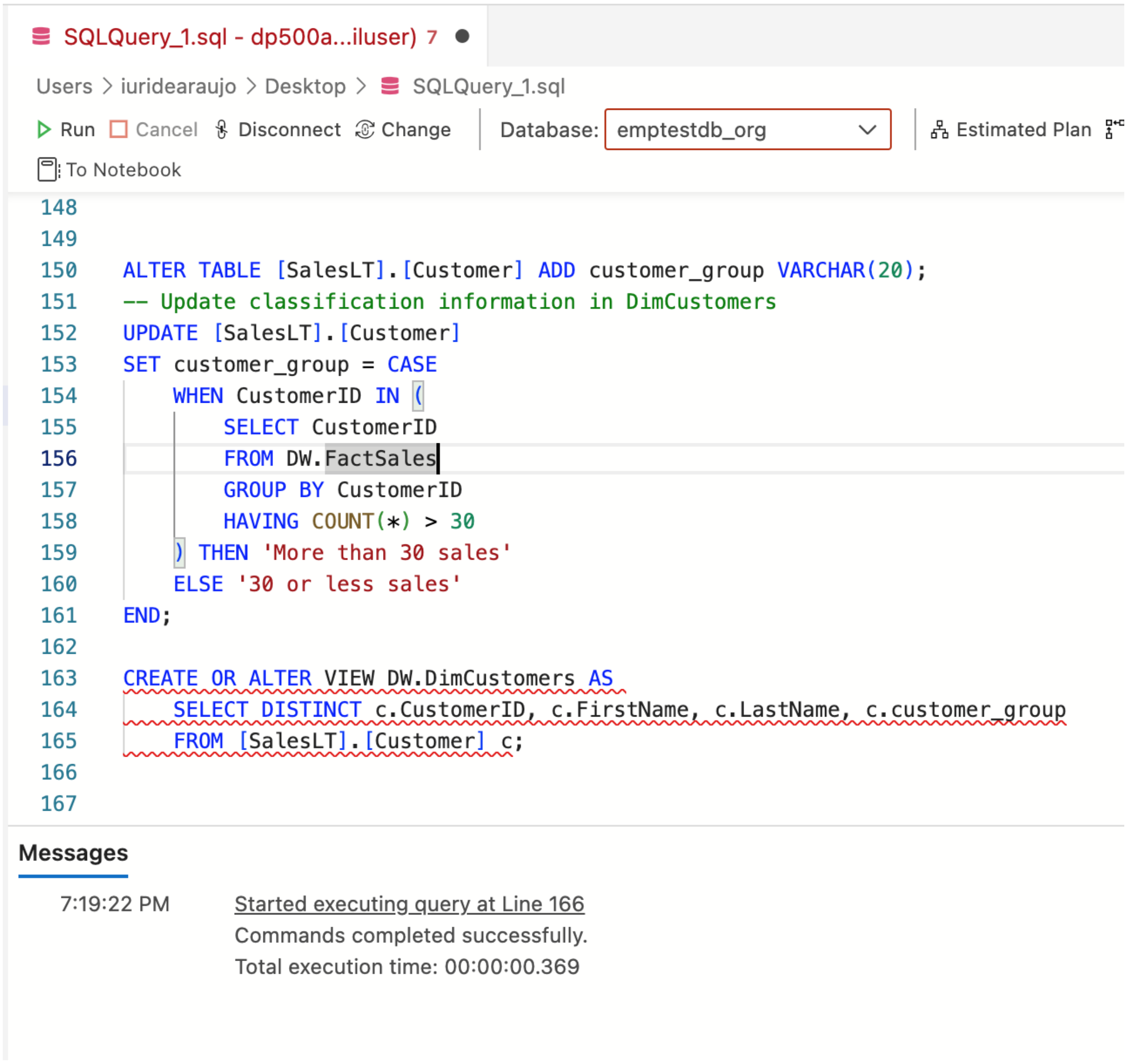
To classify customers into two groups based on the number of sales order rows (transactions) and add this information to the customer dimension, you can follow these steps:

1. Add a new column to the DimCustomers view to hold the classification information.
2. Update the DimCustomers view with the classification information using a CASE statement.
3. Adjust the join conditions and column selections in the other views to include this new classification column.

```
ALTER TABLE [SalesLT].[Customer] ADD customer_group VARCHAR(20);
-- Update classification information in DimCustomers
UPDATE [SalesLT].[Customer]
SET customer_group = CASE
WHEN CustomerID IN (
SELECT CustomerID
FROM DW.FactSales
GROUP BY CustomerID
HAVING COUNT(*) > 30
) THEN 'More than 30 sales'
ELSE '30 or less sales'
END;

CREATE OR ALTER VIEW DW.DimCustomers AS
SELECT DISTINCT c.CustomerID, c.FirstName, c.LastName, c.customer_group
FROM [SalesLT].[Customer] c;
```

The customer dimension now includes the classification information


SQLQuery_1.sql - dp500a...iluser) 7
Users > iuridearaujo > Desktop > SQLQuery_1.sql
Run
Cancel
Disconnect
Change
Database: emptestdb_org
Estimated Plan
To Notebook
Messages

```sql

ALTER TABLE [SalesLT].[Customer] ADD customer_group VARCHAR(20);
-- Update classification information in DimCustomers
UPDATE [SalesLT].[Customer]
SET customer_group = CASE
    WHEN CustomerID IN (
        SELECT CustomerID
        FROM DW.FactSales
        GROUP BY CustomerID
        HAVING COUNT(*) > 30
    ) THEN 'More than 30 sales'
    ELSE '30 or less sales'
END;

CREATE OR ALTER VIEW DW.DimCustomers AS
    SELECT DISTINCT c.CustomerID, c.FirstName, c.LastName, c.customer_group
    FROM [SalesLT].[Customer] c;

```

```
7:19:22 PM    Started executing query at Line 166
              Commands completed successfully.
              Total execution time: 00:00:00.369
```

In this example above, It was first added a new column called customer_group to the source table SalesLT. The customer_group column was updated, using a CASE statement to classify customers based on the count of sales order rows.

Finally, view DimCustomers was refreshed to include the new classification column.

6) **The last task is to just reason about how we could handle historical values** of this classification if we want to follow our customers and their behaviour. You don’t have to implement this in SQL. Just reason about it.

Handling historical values of customer classification is an important consideration when we aim to track and analyze customer behavior over time. I have researched the question topics and a few concepts on the internet, such as the ones listed below. I have splitted in main topics about the strategy that we could apply. They are steps to be considered when we start the discussion in the interview

Historical Tracking:

To maintain historical values of customer classification, you could create an additional table that stores changes in customer classification over time. This table could have columns like customer_id, classification, effective_date, and expiry_date.

Effective and Expiry Dates:

Each row in the historical tracking table would represent a change in classification. The effective_date would denote when the classification change took effect, and the expiry_date would indicate when the classification ceased to be valid.

Updating Historical Data:

When a customer's classification changes, you would insert a new row into the historical tracking table with the updated classification, effective date, and an appropriate expiry date (which might be the day before the next classification change).

Querying Historical Data:

When analyzing historical customer behavior, you would query this historical tracking table to retrieve the relevant classification for a given point in time. You'd need to join this table with other relevant tables using the customer_id and filtering based on the effective and expiry dates.

Reporting and Analysis:

By using this approach, you'd be able to see how customer behavior evolves over time in relation to changes in classification. You could analyze trends, changes in buying patterns, and the effects of different classifications on customer behavior.

Maintenance and Archiving:

Over time, the historical tracking table might accumulate a lot of data. You could consider archiving or purging older records to maintain efficient querying and reporting.

Consistency and Accuracy:

Careful handling is needed to ensure the historical data remains consistent and accurate. Any updates or changes to the main classification logic should also be reflected in the historical tracking records.

This approach involves more complexity than the simple classification update we discussed earlier. It requires careful consideration of database design, data management practices, and potentially modifying your application logic to accommodate this historical tracking. It can be a powerful way to analyze and understand long-term customer behavior patterns.