AZURE CODING CHALLENGE

HARSHINI V

QUESTION:

Create a cluster & attach the notebook to the cluster and run all commands in the notebook & creates a DataFrame from a Databricks dataset& Create a Visualizations in Databricks notebooks

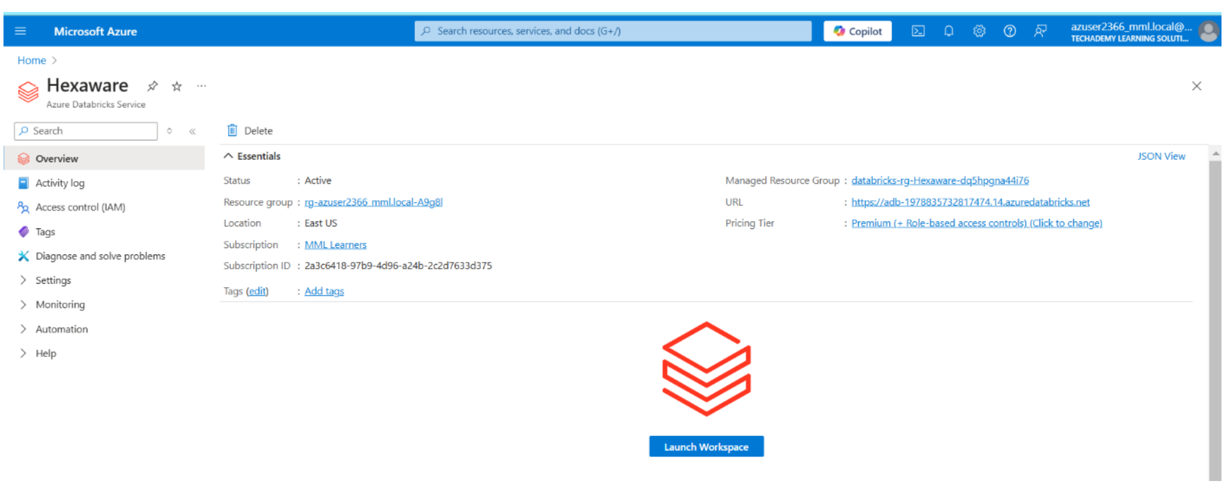
INTRODUCTION:

This document provides a comprehensive overview of the process undertaken to analyse a retail sales dataset using Azure Databricks. The tasks involved setting up a cluster, uploading and preparing the dataset, executing SQL queries for insights, and visualizing the results. The objective was to utilize Databricks' capabilities to process and analyse data efficiently, deriving actionable insights from the dataset.

The dataset, retail_sales_dataset.csv, includes details such as transaction IDs, dates, customer demographics, product categories, quantities sold, prices, and total amounts. This analysis explores sales trends, customer behavior, and product performance, using SQL queries and visualizations.

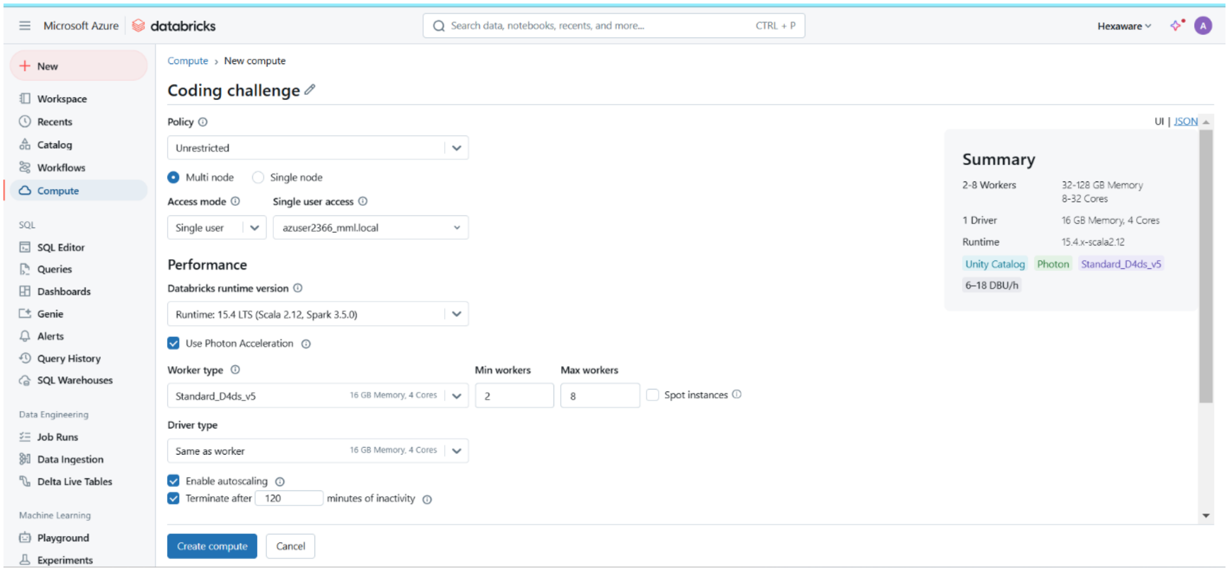
SET UP AZURE DATABRICKS WORKSPACE:

To begin, an **Azure Databricks Workspace** was created in the **Azure Portal**. This workspace serves as the environment for running notebooks, managing clusters, and scheduling jobs. By creating the workspace, you gain the ability to leverage Spark clusters for distributed data processing and the collaborative features of Databricks notebooks.

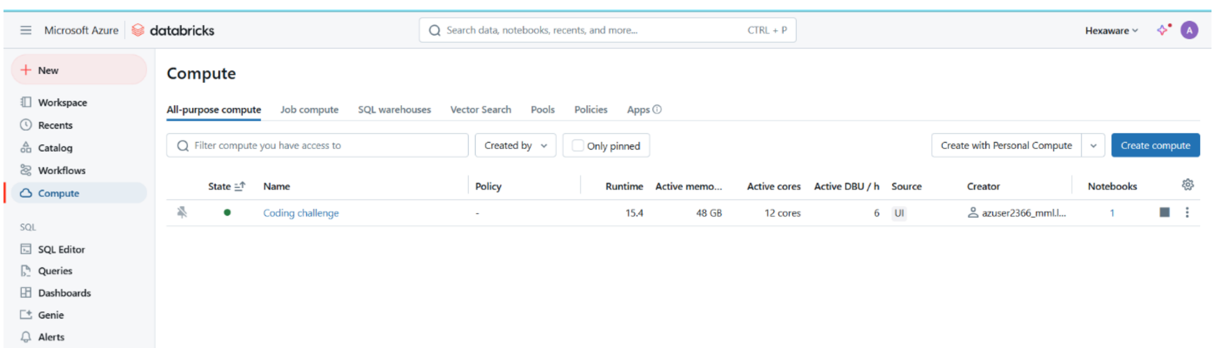


CLUSTER CREATION:

The process begins by navigating to the **Compute** tab in Azure Databricks, which is the section used for managing clusters. A cluster acts as the computation engine for processing data and running notebooks in Databricks. To create a new cluster, the **Create Cluster** button is clicked, and configuration details such as the cluster name, runtime version, and size are specified.

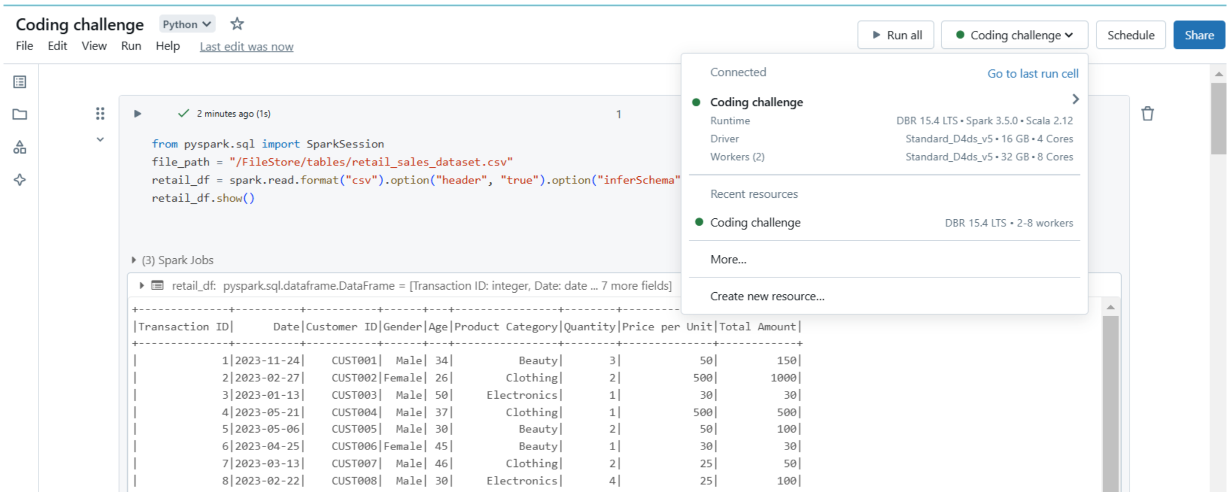


Once the configurations are set, the cluster is started, making it ready for executing data processing tasks and queries. The cluster's resources, such as memory and cores, ensure efficient execution of data workloads.



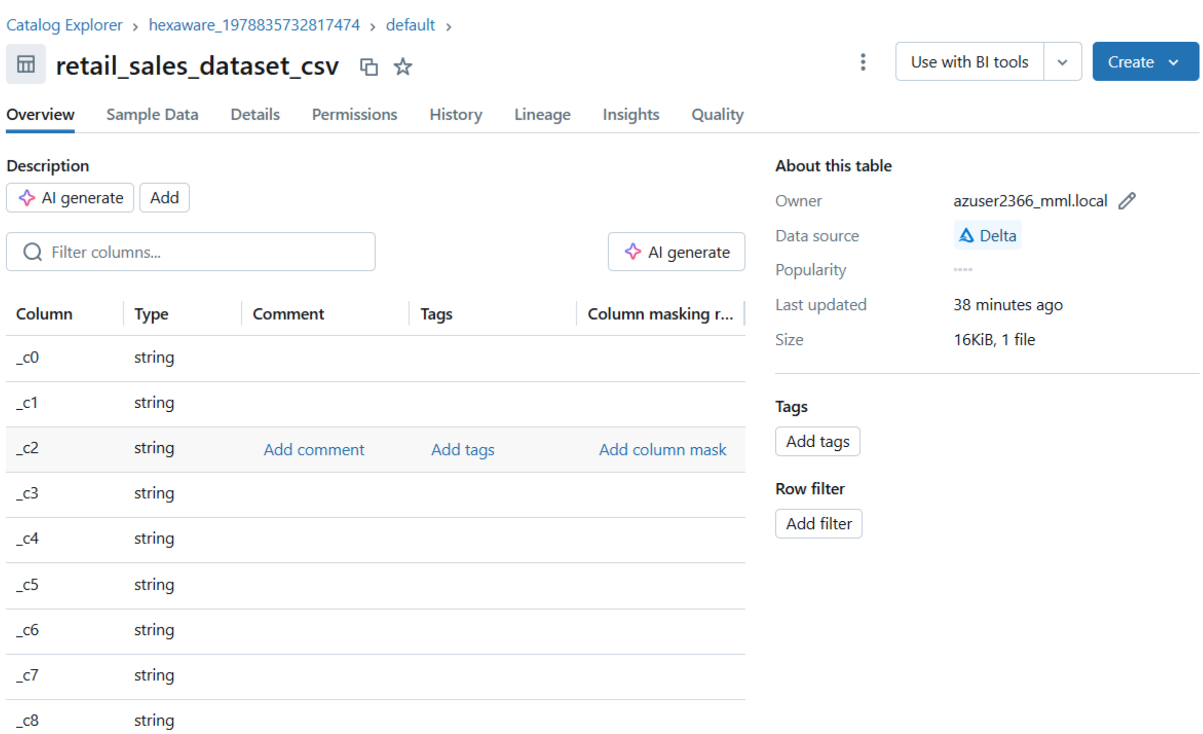
NOTEBOOK CREATION AND ATTACHING TO A CLUSTER:

In Azure Databricks, a notebook serves as an interactive workspace where users can write and execute code, analyze data, and visualize results. To create a notebook, navigate to the Workspace tab and select the option to create a new notebook. A suitable name is assigned to the notebook, and Python is chosen as the default language for execution. Once created, the notebook needs to be attached to a cluster to utilize its computational resources. This is done by selecting the desired cluster from the dropdown menu available in the notebook interface. Attaching the notebook to a cluster ensures that all commands and queries run efficiently, leveraging the power and configuration of the cluster for seamless data processing and analysis.



DATASET UPLOAD AND DATA LOADING:

To begin working with a dataset in Azure Databricks, the first step is to upload the dataset. This is done by navigating to the **Data** tab in the Databricks workspace, where the option to **Add Data** is selected. Under the DBFS (Databricks File System) section, the **Upload File** option is chosen, allowing the user to upload the dataset file. Once the dataset is uploaded, the file path is noted (e.g., /FileStore/tables/retail_sales_dataset.csv), which will be used for accessing the data. To load the dataset into a Spark DataFrame, the file path is provided in the following code, which reads the CSV file and infers the schema automatically. The dataset is loaded using the .load() function, and the show() method is used to display a preview of the first few rows, confirming that the data is loaded successfully.



DATA PREPARATION:

After loading the dataset into a Spark DataFrame, the next step is to prepare it for SQL queries. To do this, the DataFrame is registered as a temporary SQL view, allowing SQL commands to be executed directly on the dataset within Databricks.

Just now (1s) 1 Python

```
from pyspark.sql import SparkSession
file_path = "/FileStore/tables/retail_sales_dataset.csv"
retail_df = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load(file_path)
retail_df.show()
```

(3) Spark Jobs

retail_df: pyspark.sql.dataframe.DataFrame = [Transaction ID: integer, Date: date ... 7 more fields]

```
+--------------+----------+-----------+------+---+----------------+--------+--------------+------------+
|Transaction ID|      Date|Customer ID|Gender|Age|Product Category|Quantity|Price per Unit|Total Amount|
+--------------+----------+-----------+------+---+----------------+--------+--------------+------------+
|             1|2023-11-24|    CUST001|  Male| 34|          Beauty|       3|            50|         150|
|             2|2023-02-27|    CUST002|Female| 26|        Clothing|       2|           500|        1000|
|             3|2023-01-13|    CUST003|  Male| 50|     Electronics|       1|            30|          30|
|             4|2023-05-21|    CUST004|  Male| 37|        Clothing|       1|           500|         500|
|             5|2023-05-06|    CUST005|  Male| 30|          Beauty|       2|            50|         100|
|             6|2023-04-25|    CUST006|Female| 45|          Beauty|       1|            30|          30|
|             7|2023-03-13|    CUST007|  Male| 46|        Clothing|       2|            25|          50|
|             8|2023-02-22|    CUST008|  Male| 30|     Electronics|       4|            25|         100|
|             9|2023-12-13|    CUST009|  Male| 63|     Electronics|       2|           300|         600|
|            10|2023-10-07|    CUST010|Female| 52|        Clothing|       4|            50|         200|
|            11|2023-02-14|    CUST011|  Male| 23|        Clothing|       2|            50|         100|
|            12|2023-10-30|    CUST012|  Male| 35|          Beauty|       3|            25|          75|
|            13|2023-08-05|    CUST013|  Male| 22|     Electronics|       3|           500|        1500|
|            14|2023-01-17|    CUST014|  Male| 64|        Clothing|       4|            30|         120|
|            15|2023-01-16|    CUST015|Female| 42|     Electronics|       4|           500|        2000|
|            16|2023-02-17|    CUST016|  Male| 19|        Clothing|       3|           500|        1500|
|            17|2023-04-22|    CUST017|Female| 27|        Clothing|       4|            25|         100|
|            18|2023-04-30|    CUST018|Female| 47|     Electronics|       2|            25|          50|
```

This is accomplished using the createOrReplaceTempView() method, which registers the DataFrame as a temporary table named "retail_sales". Once the DataFrame is registered as a SQL view, SQL queries can be run on the data, enabling efficient analysis and insights extraction using SQL syntax within the notebook.

04:41 PM (<1s) 2

```
retail_df.createOrReplaceTempView("retail_sales")
```

QUERY EXECUTION:

Seven SQL queries were executed to derive insights from the dataset:

1. Total Sales by Product Category:

+ Code + Text

04:45 PM (1s) 3

```
%sql
SELECT `Product Category`, SUM(`Total Amount`) AS Total_Sales
FROM retail_sales
GROUP BY `Product Category`
ORDER BY Total_Sales DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Product Category: string, Total_Sales: long]

Table +

	Product Category	Total_Sales
1	Electronics	156905
2	Clothing	155580
3	Beauty	143515

3 rows | 1.37 seconds runtime Refreshed 13 minutes ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query calculates the total sales for each product category by summing the Total Amount for each category. The results are then sorted in descending order to highlight the product categories with the highest sales.

2. Top 5 Customers by Total Spend:

```
%sql
SELECT `Customer ID`, SUM(`Total Amount`) AS Total_Spent
FROM retail_sales
GROUP BY `Customer ID`
ORDER BY Total_Spent DESC
LIMIT 5;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Customer ID: string, Total_Spent: long]

	Customer ID	Total_Spent
1	CUST412	2000
2	CUST072	2000
3	CUST875	2000
4	CUST946	2000
5	CUST093	2000

5 rows | 0.52 seconds runtime   Refreshed 12 minutes ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query identifies the top 5 customers who spent the most by summing their respective Total Amount. The query groups the data by Customer ID and orders the results in descending order of total spending. A limit of 5 is applied to display only the top customers.

3. Average Sales by Gender:

```
%sql
SELECT 'Gender', AVG(`Total Amount`) AS Avg_Sales
FROM retail_sales
GROUP BY 'Gender';
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Gender: string, Avg_Sales: double]

	Gender	Avg_Sales
1	Gender	456

1 row | 0.82 seconds runtime   Refreshed 8 minutes ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query calculates the average sales per gender. It groups the dataset by the Gender column and computes the average of the Total Amount for each gender category. This provides insight into the average spending patterns based on gender.

4. Average Quantity Sold by Age Group:

```
%sql
SELECT CASE
        WHEN Age < 20 THEN 'Under 20'
            WHEN Age BETWEEN 20 AND 40 THEN '20-40'
                WHEN Age BETWEEN 41 AND 60 THEN '41-60'
                    ELSE '60+'
                        END AS Age_Group,
                            AVG(Quantity) AS Avg_Quantity
                            FROM retail_sales
                            GROUP BY Age_Group;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Age_Group: string, Avg_Quantity: double]

Table

	Age_Group	Avg_Quantity
1	20-40	2.535377358490566
2	60+	2.4193548387096775
3	Under 20	2.642857142857143
4	41-60	2.5011337868480727

4 rows | 0.58 seconds runtime   Refreshed 5 minutes ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

In this query, customers are grouped into different age categories (Under 20, 20-40, 41-60, 60+) based on their age. The average quantity of products sold for each age group is calculated. This helps in understanding the purchasing behavior across different age ranges.

5. Sales Contribution by Gender:

```
%sql
SELECT 'Gender', SUM(`Total Amount`) AS Total_Sales,
(SUM(`Total Amount`) * 100 / (SELECT SUM(`Total Amount`) FROM retail_sales)) AS Contribution_Percentage
FROM retail_sales
GROUP BY 'Gender'
ORDER BY Contribution_Percentage DESC;
```

(4) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Gender: string, Total_Sales: long ... 1 more field]

Table

	Gender	Total_Sales	Contribution_Percentage
1	Gender	456000	100

1 row | 1.14 seconds runtime   Refreshed 4 minutes ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query calculates the total sales by gender and determines the percentage contribution of each gender to the overall sales. The SUM of Total Amount is calculated for each gender, and a subquery is used to calculate the total sales across all genders, allowing the calculation of the contribution percentage.

6. First 10 Product Categories and Quantities:

```
%sql
SELECT `Product Category`, Quantity
FROM retail_sales
LIMIT 10;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Product Category: string, Quantity: integer]

	Product Category	Quantity
1	Beauty	3
2	Clothing	2
3	Electronics	1
4	Clothing	1
5	Beauty	2
6	Beauty	1
7	Clothing	2
8	Electronics	4
9	Electronics	2
10	Clothing	4

10 rows | 0.38 seconds runtime   Refreshed 1 minute ago

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query retrieves the first 10 rows from the dataset, displaying the Product Category and Quantity. This query is useful for examining a small sample of the data to ensure its accuracy and structure.

7. Product Category with Highest Sales Quantity:

```
%sql
SELECT `Product Category`, SUM(Quantity) AS Total_Quantity
FROM retail_sales
WHERE Quantity IS NOT NULL
GROUP BY `Product Category`
ORDER BY Total_Quantity DESC
LIMIT 1;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Product Category: string, Total_Quantity: long]

	Product Category	Total_Quantity
1	Clothing	894

1 row | 0.58 seconds runtime   Refreshed now

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

This query identifies the product category that has the highest sales quantity. It sums the Quantity for each product category, filters out any rows where Quantity is null, and then orders the results by total quantity in descending order. The LIMIT 1 ensures that only the top product category is displayed.

VISUALIZATION:

1. Total Sales by Product Category

To create a visualization, the **Visualization** button was clicked, and a **Bar Chart** was selected to represent the total sales per product category. The axes were adjusted such that the Product Category was set as the y-axis and Total_Sales was set as the x-axis to clearly show the distribution of sales across different product categories.

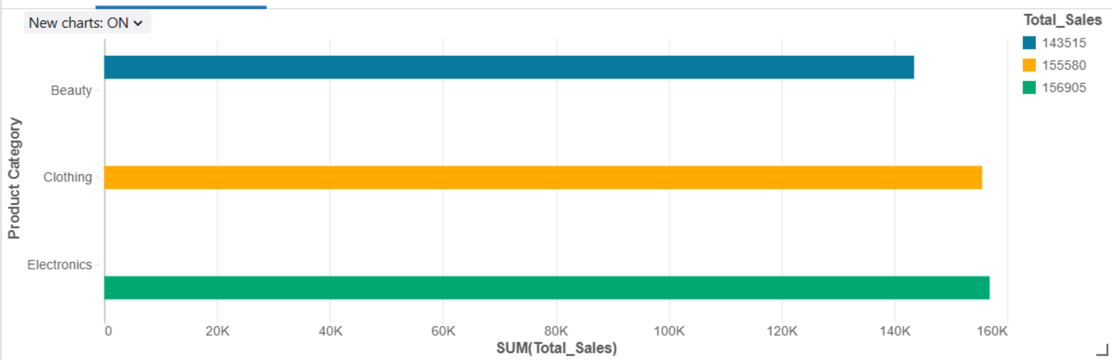
```
%sql
SELECT 'Product Category', SUM('Total Amount') AS Total_Sales
FROM retail_sales
GROUP BY 'Product Category'
ORDER BY Total_Sales DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Product Category: string, Total_Sales: long]

Table Visualization 1 +

New charts: ON



2. Top 5 Customers by Total Spend

A **Bar Chart** was selected to display the total amount spent by each of the top 5 customers. This chart was set up with Customer ID on the x-axis and Total_Spent on the y-axis, effectively highlighting the customers with the highest total spend.

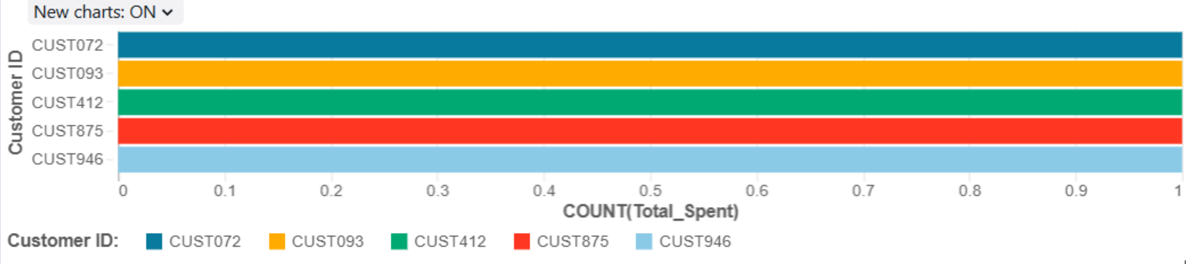
```
%sql
SELECT 'Customer ID`, SUM(`Total Amount`) AS Total_Spent
FROM retail_sales
GROUP BY `Customer ID`
ORDER BY 'Total_Spent' DESC
LIMIT 5;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Customer ID: string, Total_Spent: long]

Table Visualization 1 +

New charts: ON



Edit Visualization 5 rows

This result is stored as _sqldf and can be used in other Python and SQL cells.

3. Average Sales by Gender

For visualizing average sales by gender, the SQL query was executed to compute the average Total Amount grouped by Gender. Once the query ran successfully, the **Visualization** button was clicked, and a **Pie Chart** was chosen. The chart was configured to display Gender as the label and Avg_Sales as the value, giving a clear distribution of average sales across different genders.

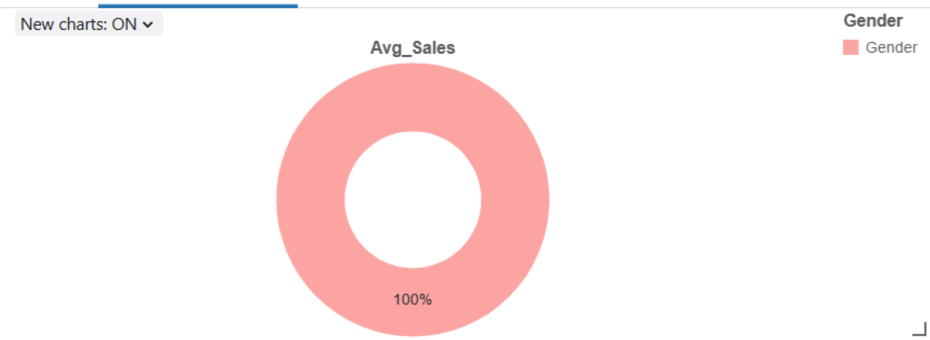
```sql
%sql
SELECT 'Gender', AVG('Total Amount`) AS Avg_Sales
FROM retail_sales
GROUP BY 'Gender';
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Gender: string, Avg_Sales: double]

Table Visualization 1

New charts: ON



Edit Visualization 1 row

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

4. Average Quantity Sold by Age Group

The query to calculate the average quantity sold by different age groups was executed by creating a case-based categorization of age. After running the query, the **Visualization** button was clicked, and a **Line Chart** was selected to show the relationship between age groups and average quantity sold. This chart provides a clear visual of how sales quantities vary across different age ranges.

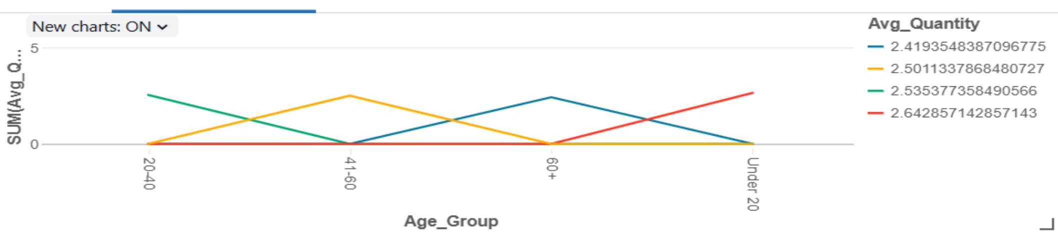
```sql
%sql
SELECT CASE
    WHEN Age < 20 THEN 'Under 20'
        WHEN Age BETWEEN 20 AND 40 THEN '20-40'
            WHEN Age BETWEEN 41 AND 60 THEN '41-60'
                ELSE '60+'
                    END AS Age_Group,
                        AVG(Quantity) AS Avg_Quantity
                        FROM retail_sales
                        GROUP BY Age_Group;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Age_Group: string, Avg_Quantity: double]

Table Visualization 1

New charts: ON



Edit Visualization 4 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

5. Sales Contribution by Gender

Once the query was executed, the **Visualization** button was clicked and a **Table** visualization was selected. This table displayed the sales contribution by gender, with Gender as the label and Contribution_Percentage as the value, offering an easy-to-read format to understand each gender's percentage of total sales.

```
%sql
SELECT 'Gender', SUM('Total Amount`) AS Total_Sales,
(SUM(`Total Amount`) * 100 / (SELECT SUM(`Total Amount`) FROM retail_sales)) AS Contribution_Percentage
FROM retail_sales
GROUP BY 'Gender'
ORDER BY Contribution_Percentage DESC;
```

▸ (4) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [Gender: string, Total_Sales: long ... 1 more field]

Table | Visualization 1 | +

Gender	Total_Sales	Contribution_Percentage
Gender	456000	100.00

Edit Visualization | 1 row

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

6. First 10 Product Categories and Quantities

The query to fetch the first 10 product categories and their respective quantities was executed. Afterward, the **Visualization** button was clicked, and a **Pie Chart** was chosen to display the quantity of each product category. The Product Category was set as the label and Quantity as the value in the pie chart, providing a visual breakdown of product quantities for the first 10 categories.

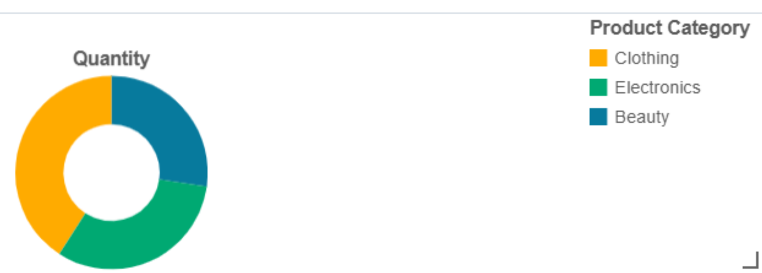
```
%sql
SELECT 'Product Category`, Quantity
FROM retail_sales
LIMIT 10;
```

▸ (1) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [Product Category: string, Quantity: integer]

Table | Visualization 1 | +

New charts: ON



Edit Visualization | 10 rows

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

7. Product Category with Highest Sales Quantity

After running the query, the **Visualization** button was clicked, and a **Pie Chart** or **Donut Chart** was selected for visualization. Since there was only one result, this chart displayed the product category with the highest quantity, providing a focused, clear visual of the data.

```
%sql
SELECT 'Product Category`, SUM(Quantity) AS Total_Quantity
FROM retail_sales
WHERE Quantity IS NOT NULL
GROUP BY `Product Category`
ORDER BY Total_Quantity DESC
LIMIT 1;
```