Customer Purchase Analysis in Databricks: Data Ingestion, Exploration, and Transformation using

JUJJAVARAPU SUJAN CHOWDARY RA2412033010001

Introduction

In the fast-evolving world of e-commerce, understanding customer purchasing behavior is crucial for optimizing sales, improving user experience, and enhancing business strategies. This assignment focuses on analyzing a one-year dataset of customer purchases in the U.S. e-commerce market using **Databricks and Apache Spark**.

Objective

The goal is to **ingest, explore, and transform** the dataset efficiently using **PySpark and Spark SQL**. The dataset contains vital attributes such as **order details, customer demographics, device usage, sales performance, and shipping costs**.

Scope of Work

- **Data Ingestion**: Loading a CSV dataset into a **Delta table** in Databricks and partitioning it for better query performance.
- **Exploratory Data Analysis (EDA)**: Generating statistical summaries and visualizations to gain insights into customer purchasing behavior.
- **Data Cleaning & Transformation**: Handling missing values, standardizing date formats, encoding categorical variables, and normalizing numerical features.

This project will showcase how **big data processing and analytics** can drive actionable insights in the e-commerce domain.

About the Dataset

This dataset represents **one year of e-commerce purchases** in the United States, capturing key details about customer transactions, product categories, payment methods, and sales performance. The data will be used to analyze **customer buying patterns, sales trends, and order fulfillment efficiency**.

The dataset consists of **51,290 records** and includes the following columns:

Dataset Schema

Column Name	Data Type	Description
Order_Date	Date	The date when the product was ordered.
Time	Timestamp	The exact time the transaction took place.
Aging	Double	The time (in days) between order placement and delivery.
Customer_Id	BigInt	A unique identifier for each customer.
Gender	String	The gender of the customer (e.g., Male, Female).
Device_Type	String	The device used for the transaction (e.g., Web, Mobile).
Customer_Login_Type	String	How the customer logged in (e.g., Member, Guest).
Product_Category	String	The category of the purchased product (e.g., Electronics, Clothing).
Product	String	The specific product name.
Sales	Double	The total sales amount for the transaction.
Quantity	Double	The number of units purchased.
Discount	Double	The percentage discount applied.
Profit	Double	The profit generated from the sale.
Shipping_Cost	Double	The cost of shipping for the order.
Order_Priority	String	The priority level of the order (e.g., Critical, High).
Payment_Method	String	The payment method used (e.g., Credit Card, PayPal).

Dataset Properties

- **Storage Format**: Delta
- **Database**: `default`
- **Table Name**: `e_commerce_dataset`
- **Storage Location**: `dbfs:/user/hive/warehouse/e_commerce_dataset`
- **Created By**: Apache Spark 3.5.0
- **Table Type**: Managed

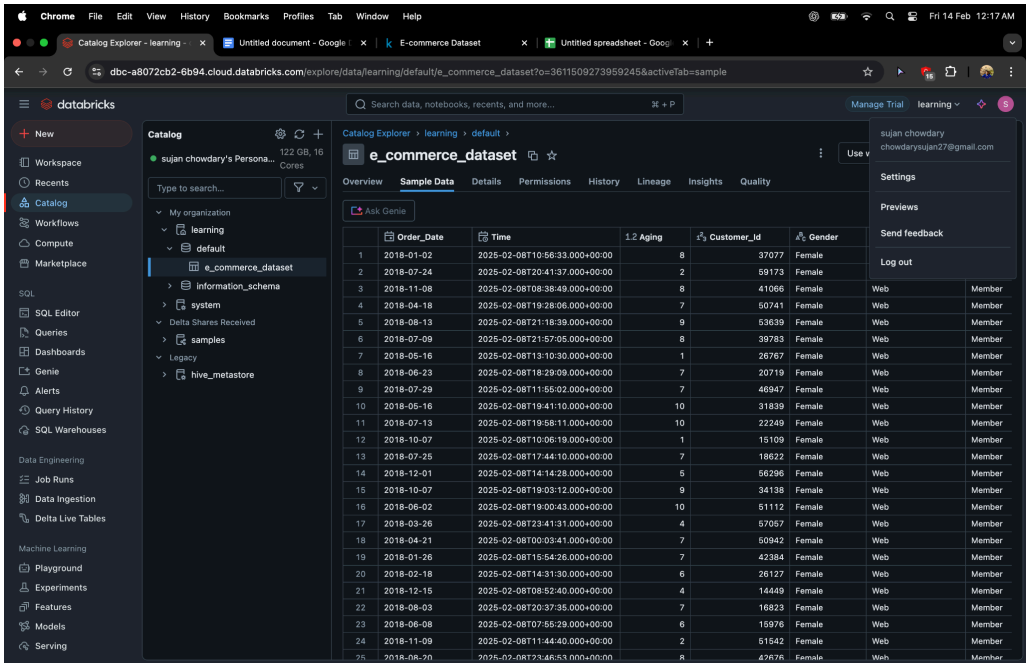
This dataset will be used for **data ingestion, exploratory data analysis (EDA), and transformation** to derive meaningful business insights and optimize data processing in Databricks.

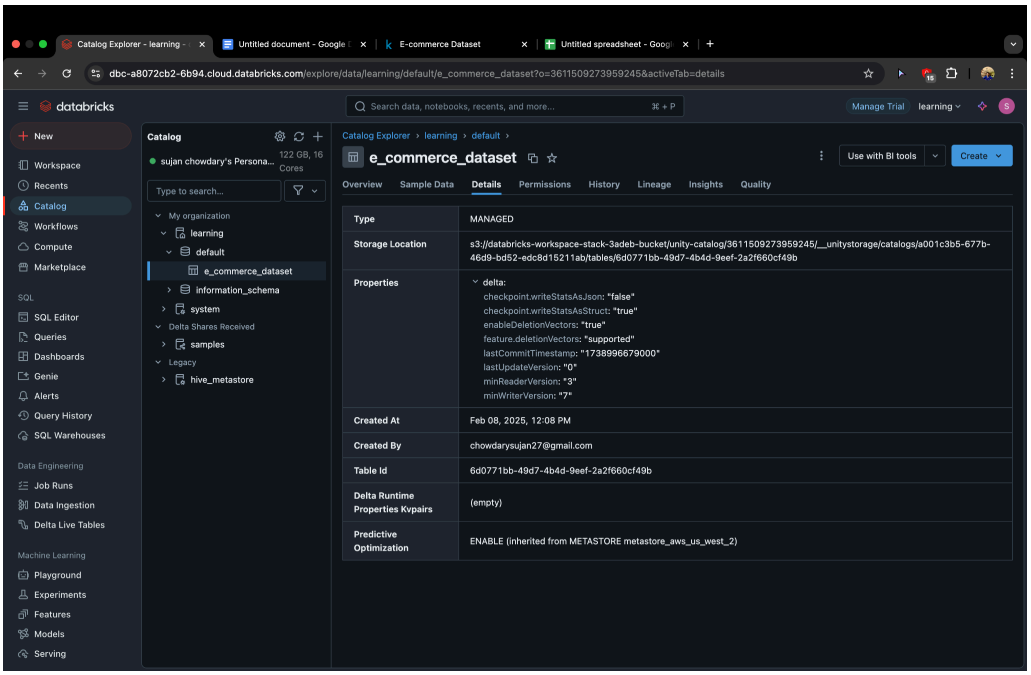
Tasks:

Task 1: Use Databricks to upload your dataset (CSV format) into a table.

Upload CSV File to DBFS

- Navigate to Databricks Workspace → Data → Create Table.
- Select "Upload File", choose the CSV dataset, and upload it.
- The file is stored in DBFS (Databricks File System) under a path like: "*dbfs:/FileStore/tables/e_commerce_dataset.csv*"





Task 2: A Databricks notebook with the ingestion code.

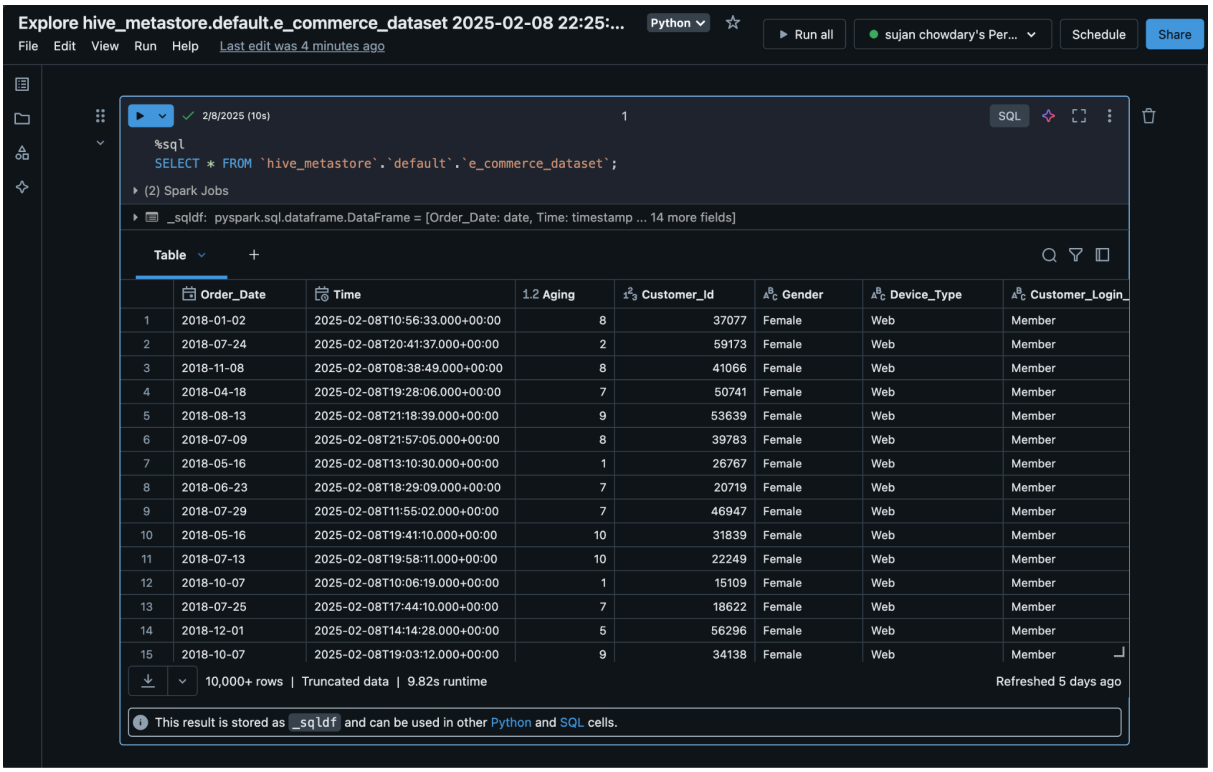
Code:

```
%sql
SELECT * FROM hive_metastore.default.e_commerce_dataset;
```

1. **SELECT ***
    - This retrieves all columns from the table.
    - The * wildcard means that every field in the dataset will be included in the output.
2. **`FROM hive_metastore.default.e_commerce_dataset`**
    - `hive_metastore`: This refers to the Hive Metastore, which manages metadata for tables in Databricks.
    - `default`: The default database where the `e_commerce_dataset` table is stored.
    - `e_commerce_dataset`: The table name containing customer transaction data.

Purpose of the Query:

- This command is used to fetch all records from the `e_commerce_dataset` table.
- It helps verify whether the data has been successfully ingested into Databricks.
- This is typically one of the first steps after data ingestion to confirm that the dataset is accessible and correctly stored.



Task 3: Partition the table to optimize query performance.

Partitioned Columns:

In the given code, the `Order_Date` and `Product_Category` columns have been used for partitioning.

Why Partition by `Order_Date` and `Product_Category`?

1. **`Order_Date` (Date-Based Partitioning)**
    - **Reason:** Queries are often filtered by date ranges when analyzing e-commerce transactions (e.g., sales in a specific month or year).
    - **Benefit:** Improves query performance by scanning only the relevant partitions instead of the entire dataset.
2. **`Product_Category` (Category-Based Partitioning)**
    - **Reason:** Many queries analyze sales and customer behavior for specific product categories (e.g., Electronics, Clothing).
    - **Benefit:** Reduces data scanned during queries that filter by category, making retrieval faster.

Main Code Explanation

```
from pyspark.sql.functions import col

# Load the data into a DataFrame
df = spark.read.table("hive_metastore.default.e_commerce_dataset")
df

# Define partition columns
partition_columns = ["Order_Date", "Product_Category"]
# Write the partitioned table back to Hive metastore
df.write.mode("overwrite").partitionBy(*partition_columns).format("parquet").saveAsTable("hive_metastore.default.e_commerce_partitioned")

print("Partitioning completed successfully.")
```

Define partition columns
`partition_columns = ["Order_Date", "Product_Category"]`

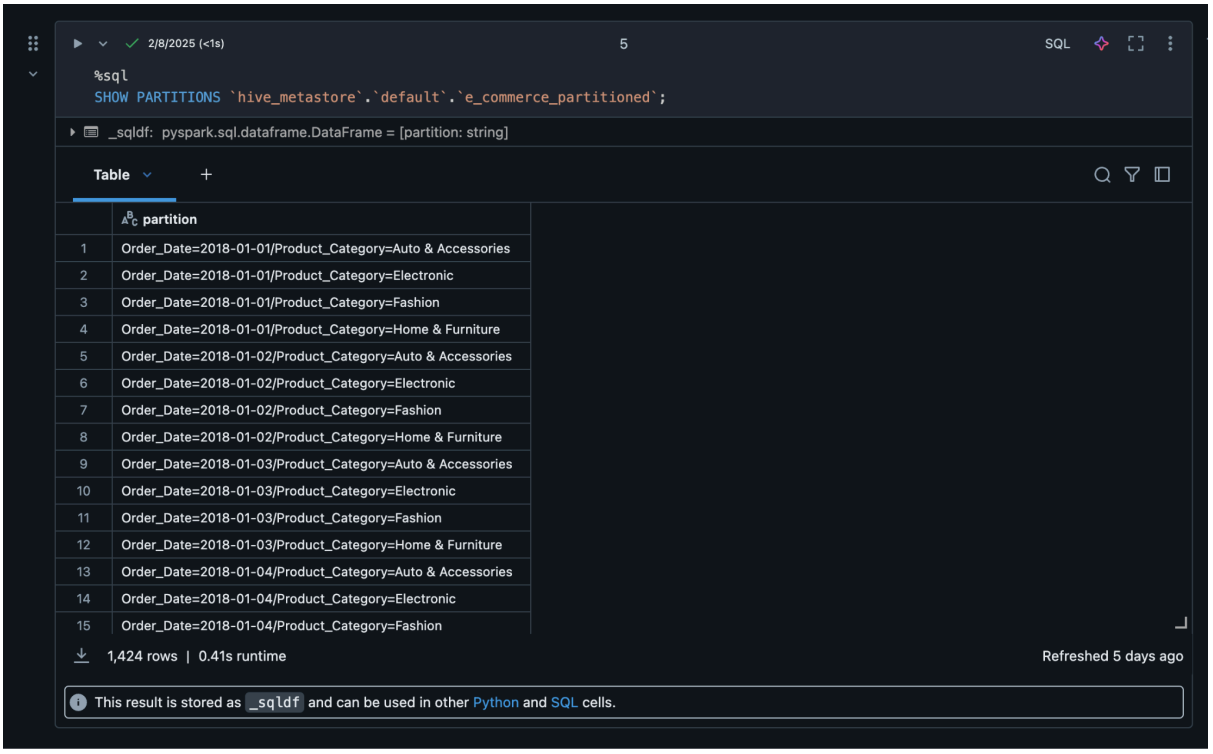
- Specifies the columns used for partitioning (`Order_Date` and `Product_Category`).
- Partitioning helps optimize query performance by logically separating data into smaller chunks.

Write the partitioned table back to Hive metastore
`df.write.mode("overwrite").partitionBy(*partition_columns).format("parquet").saveAsTable("hive_metastore.default.e_commerce_partitioned")`

- `mode("overwrite")`: Ensures that the table is rewritten with new partitions.
- `partitionBy(*partition_columns)`: Partitions data by `Order_Date` and `Product_Category`.
- `format("parquet")`: Saves the table in Parquet format, which is optimized for performance in big data processing.
- `saveAsTable("hive_metastore.default.e_commerce_partitioned")`: Saves the partitioned table in Hive metastore for optimized querying.

Benefits of Partitioning These Columns

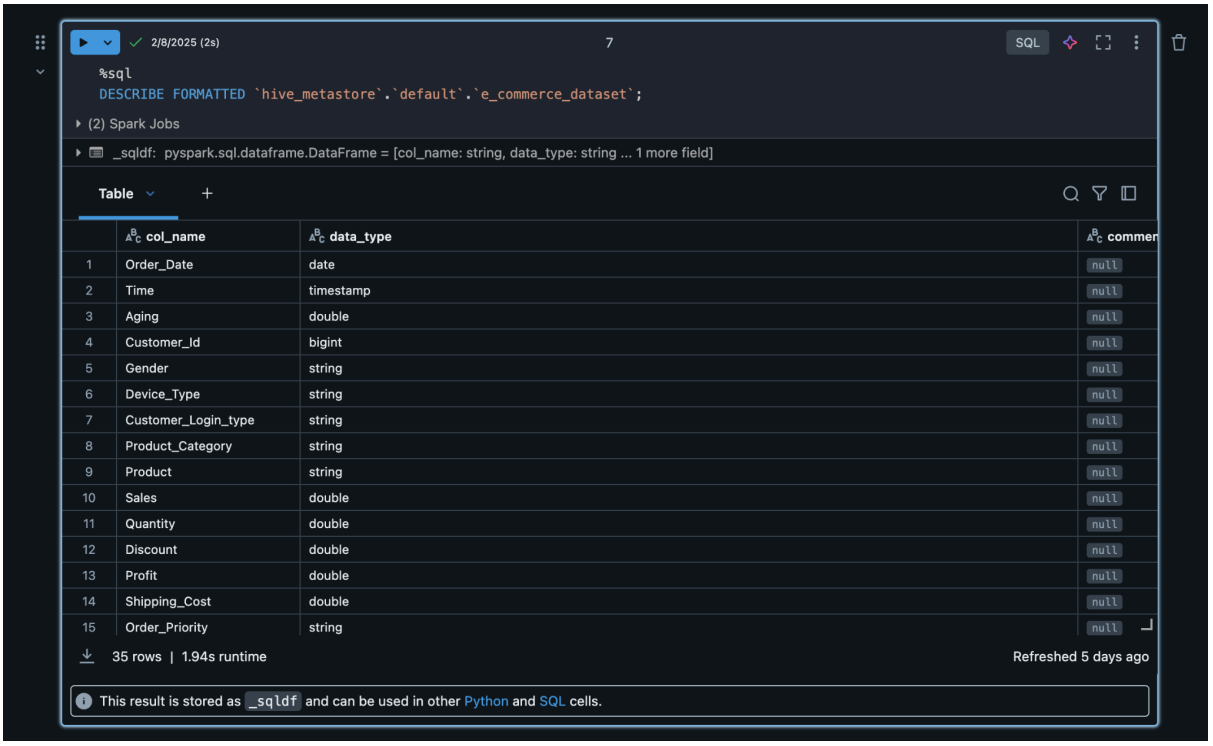
- **Improved Query Performance:**
    - When filtering by date (`Order_Date`), only the required partition is scanned instead of the full table.
    - When analyzing sales for a specific product category (`Product_Category`), only relevant partitions are accessed.
- **Reduced Data Scanning:** Instead of scanning the entire dataset (51,290 records), queries scan only relevant partitioned subsets.
- **Faster Data Processing:** When using `WHERE Order_Date = '2025-01-01'` in queries, Spark processes only that partition, making execution much faster.



Task 4: A screenshot of the table schema.

# Display the schema of the partitioned table
df.printSchema()

%sql
DESCRIBE FORMATTED `hive_metastore`.`default`.`e_commerce_dataset`;



Task 5: Perform exploratory data analysis (EDA) on the ingested customer data using Spark SQL.

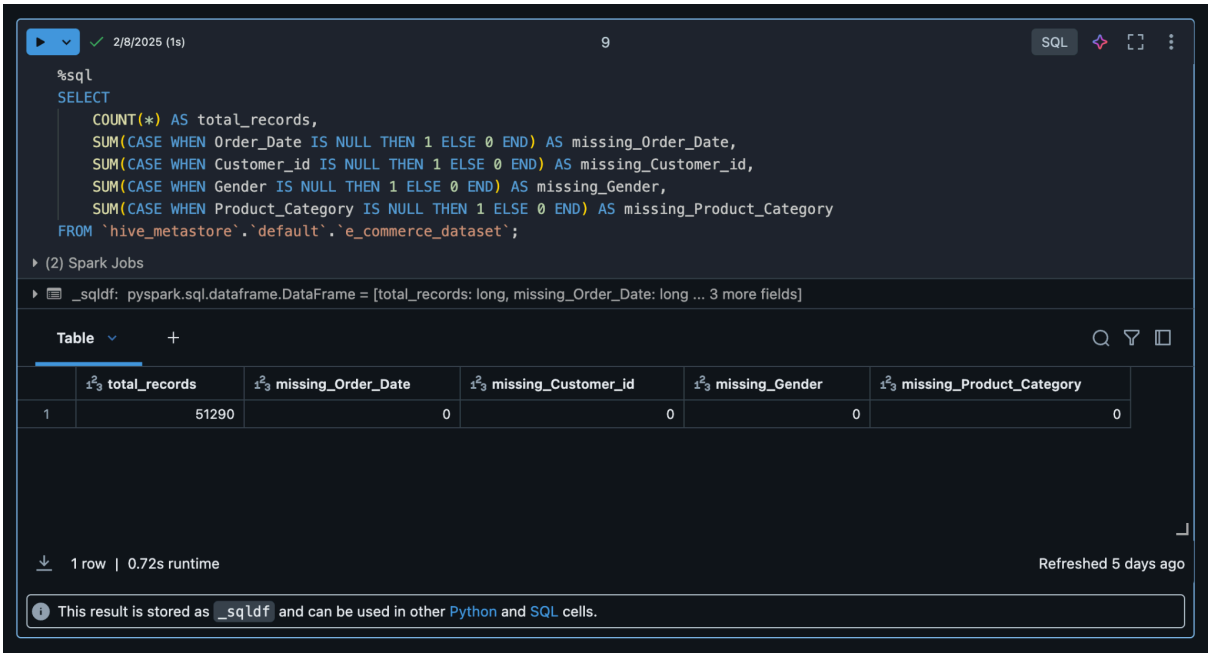
EDA is a crucial step in understanding the trends, patterns, and anomalies in the dataset. By analyzing the sales, profit, customer behavior, and product categories, we can extract valuable insights for business decisions.

1. Dataset Overview

- The dataset consists of 51,290 records, ensuring a substantial amount of data for analysis.

- No missing values were found in critical columns such as Order Date, Customer ID, Gender, and Product Category, confirming data completeness.

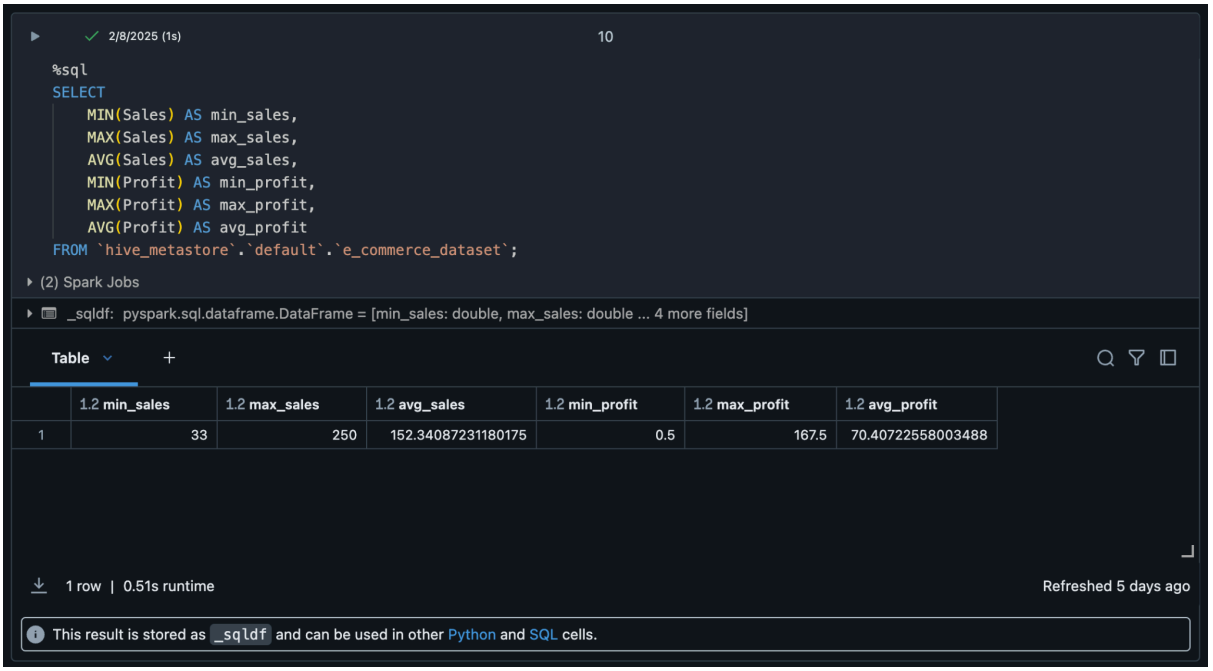
Total Records	Missing Order Date	Missing Customer ID	Missing Gender	Missing Product Category
51,290	0	0	0	0



2. Sales and Profit Analysis

- The minimum sales value recorded was $33, while the maximum sales for a single order was $250.
- The average sales per order stood at $152.34, indicating the general spending pattern of customers.
- Profit margins ranged from $0.5 (minimum profit per order) to $167.5 (maximum profit per order), with an average profit of $70.40 per order.

Min Sales ($)	Max Sales ($)	Avg Sales ($)	Min Profit ($)	Max Profit ($)	Avg Profit ($)
33	250	152.34	0.5	167.5	70.40



3. Customer Device Usage

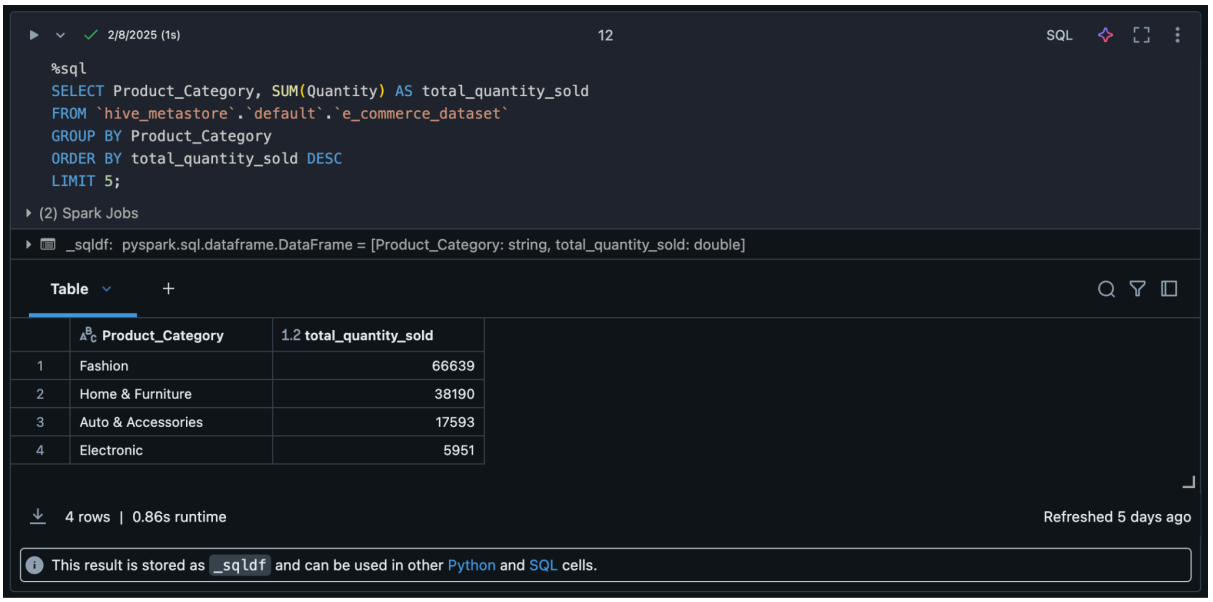
- The majority of transactions were made via Web (47,632 orders), significantly higher than Mobile (3,658 orders).
- This indicates that Web-based shopping is the primary mode of transactions, highlighting the importance of web optimization.

Device Type	Order Count
Web	47,632
Mobile	3,658

4. Most Purchased Product Categories

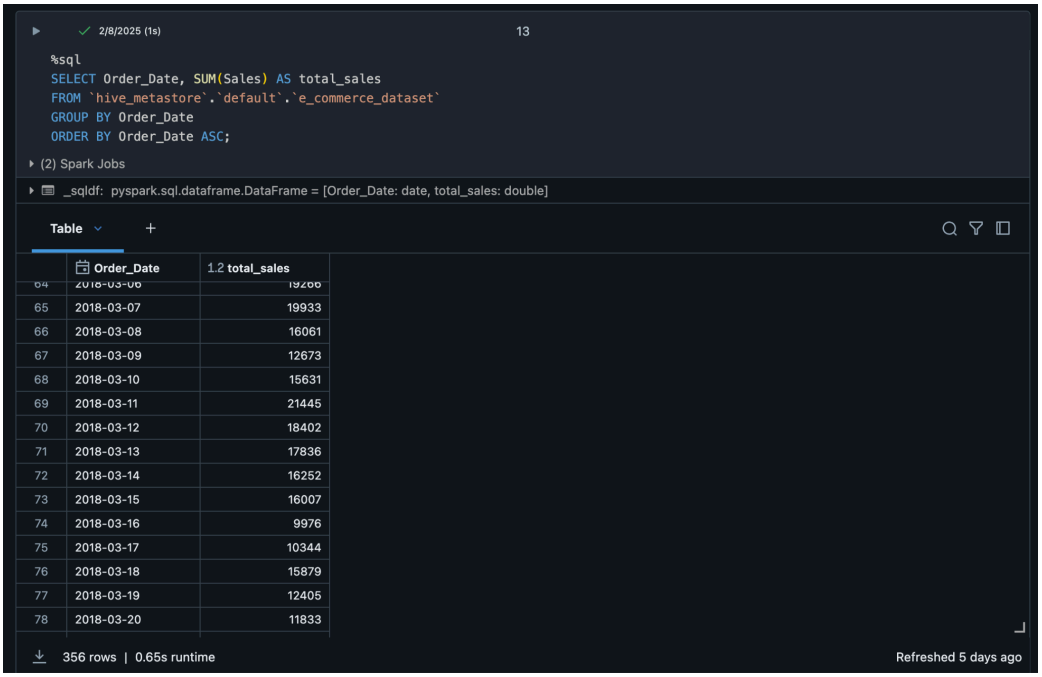
- Fashion products were the most purchased, with 66,639 units sold, followed by Home & Furniture (38,190 units).
- Electronic products had the least sales, which might indicate a higher price point or lower demand in this category.

Product Category	Total Quantity Sold
Fashion	66,639
Home & Furniture	38,190
Auto & Accessories	17,593
Electronic	5,951



5. Sales Trend Over Time

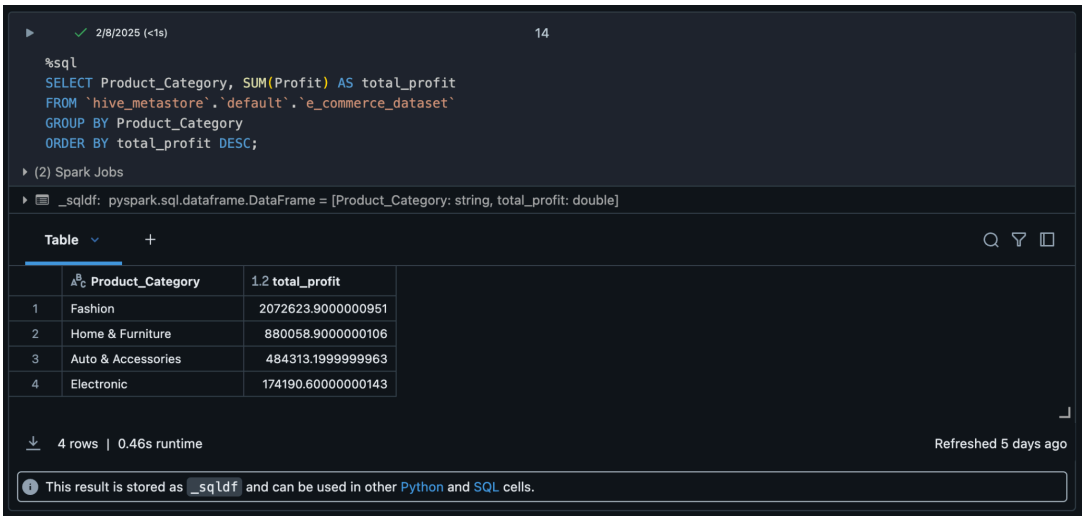
- Sales data was grouped by Order Date, revealing sales patterns across different periods.
- This helps in identifying peak sales days and trends over the year, which is useful for forecasting and promotional planning.



6. Profitability by Product Category

- Fashion products contributed the highest total profit, reaching $2,072,623.
- Home & Furniture and Auto & Accessories also generated significant profits.
- Electronics had the lowest profit, which might indicate higher costs or lower markups.

Product Category	Total Profit ($)
Fashion	2,072,623.90
Home & Furniture	880,058.90
Auto & Accessories	484,313.20
Electronic	174,190.60



7. Order Priority Distribution

- Medium priority orders dominated the dataset, accounting for 29,433 orders.
- High and Critical orders followed, while Low-priority orders were the least common.
- Two orders had missing priority values, which may require data correction.

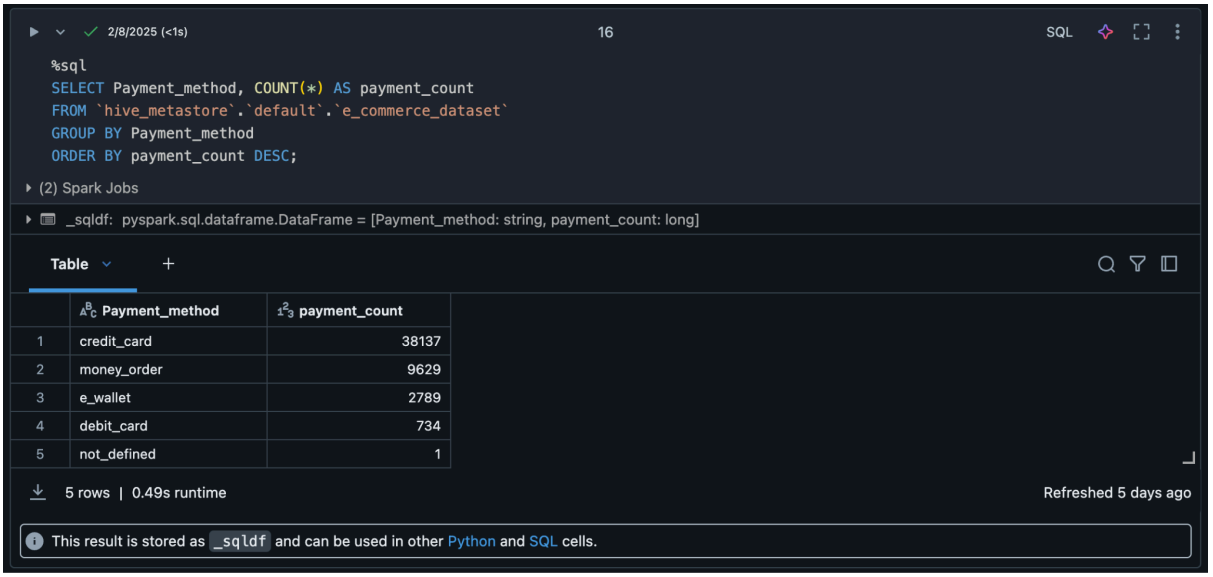
Order Priority	Order Count
Medium	29,433
High	15,499
Critical	3,932
Low	2,424
Missing	2

8. Payment Method Distribution

- Credit cards were the most used payment method (38,137 transactions), followed by Money Orders (9,629 transactions).
- E-wallets and debit cards had significantly lower usage, suggesting a preference for credit-based transactions.
- One transaction had an undefined payment method, which may require further investigation.

Payment Method	Transaction Count

Credit Card	38,137
Money Order	9,629
E-Wallet	2,789
Debit Card	734
Not Defined	1



Key Insights from EDA

1. No missing values in critical columns, ensuring data reliability.
2. Web transactions dominate, highlighting the importance of online store optimization.
3. Fashion is the top-selling category, both in units sold and profitability.
4. Electronics contribute the least profit, suggesting higher costs or lower markups.
5. Credit cards are the preferred payment method, with money orders being the second most common.
6. Medium priority orders are the highest in number, while critical orders are relatively few.

This EDA provides actionable insights for improving sales strategies, optimizing inventory, and enhancing the customer experience.

Task 6: Generate basic statistics such as count, mean, standard deviation, min, and max for numerical columns.

Statistical analysis of numerical columns helps in understanding data distribution, variability, and potential anomalies in the dataset. Below are the key statistical metrics derived from the dataset.

Code:

```
df = spark.read.table("hive_metastore.default.e_commerce_dataset")

# Describe numerical columns
numerical_columns = ["Sales", "Quantity", "Discount", "Profit", "Shipping_cost", "Aging"]

# Generate summary statistics
df.select(numerical_columns).describe().show()
```

Summary of Numerical Data

Metric	Sales ($)	Quantity	Discount	Profit ($)	Shipping Cost ($)	Aging (Days)
Count	51,289	51,288	51,289	51,290	51,289	51,289
Mean	152.34	2.50	0.30	70.41	7.04	5.25
Std Dev	66.50	1.51	0.13	48.73	4.87	2.96
Min	33.0	1.0	0.1	0.5	0.1	1.0
Max	250.0	5.0	0.5	167.5	16.8	10.5

Key Insights from Statistical Analysis

Sales Analysis

- Average sales per order: $152.34
- Minimum sales recorded: $33 (possibly low-value items).
- Maximum sales recorded: $250 (higher-value purchases).
- Standard deviation: $66.50, indicating moderate variability in transaction values.

Quantity Analysis

- Average quantity per transaction: 2.50 units.
- Minimum quantity ordered: 1 unit, indicating single-item purchases.
- Maximum quantity ordered: 5 units, suggesting a low purchase limit per order.

Discount Analysis

- Average discount applied: 30.38%.
- Minimum discount: 10%, indicating that some products have very low discount offers.
- Maximum discount: 50%, meaning some products receive high promotional discounts.

Profit Analysis

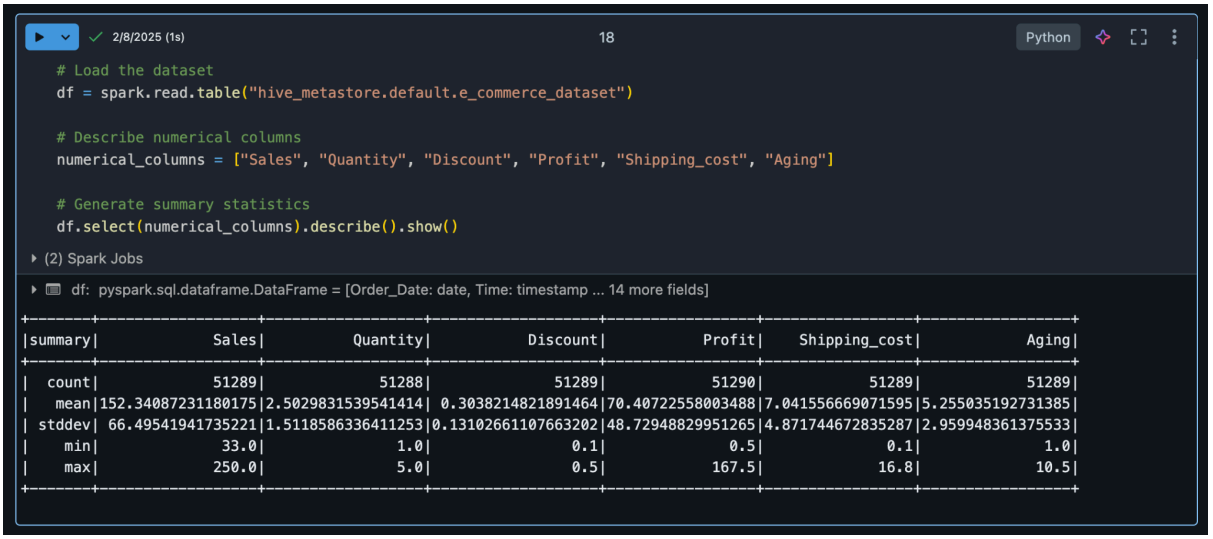
- Average profit per order: $70.41.
- Minimum profit: $0.5, meaning some products have very low profit margins.
- Maximum profit: $167.5, showing that high-margin products exist in the dataset.

Shipping Cost Analysis

- Average shipping cost: $7.04 per order.
- Minimum shipping cost: $0.1, suggesting free or highly subsidized shipping.
- Maximum shipping cost: $16.8, which could indicate premium delivery services.

Aging (Delivery Time) Analysis

- Average delivery time: 5.25 days.
- Minimum delivery time: 1 day, possibly for express shipping.
- Maximum delivery time: 10.5 days, indicating delays or slow shipping methods.



Task 7: Provide insights into categorical data distributions.

Understanding categorical data distributions helps in analyzing customer behavior, purchasing trends, and transaction preferences. Below are the key insights from different categorical columns.

Order Priority Distribution

- The majority of orders fall under Medium priority (29,433 orders), indicating standard processing.
- High-priority orders (15,499) suggest customers occasionally opt for faster delivery.
- Critical orders (3,932) may indicate urgent or express delivery requests.
- Low-priority orders (2,424) represent non-urgent purchases.
- Two records have missing priority data, which might need further investigation.

Order Priority	Order Count

Medium	29,433
High	15,499
Critical	3,932
Low	2,424
NULL	2

Business Impact:

- The dominance of Medium priority suggests that most customers are willing to wait for standard delivery, reducing logistics pressure.
- The presence of Critical and High-priority orders highlights the need for faster fulfillment services.

Payment Method Distribution

- Credit cards are the most preferred payment method (38,137 transactions), followed by Money Orders (9,629).
- E-wallet usage (2,789 transactions) is lower, suggesting limited adoption.
- Debit cards are the least used (734 transactions), possibly due to security concerns or limited cashback benefits.
- One transaction has an undefined payment method, which might indicate a data entry issue.

Payment Method	Transaction Count
Credit Card	38,137
Money Order	9,629
E-Wallet	2,789
Debit Card	734
Not Defined	1

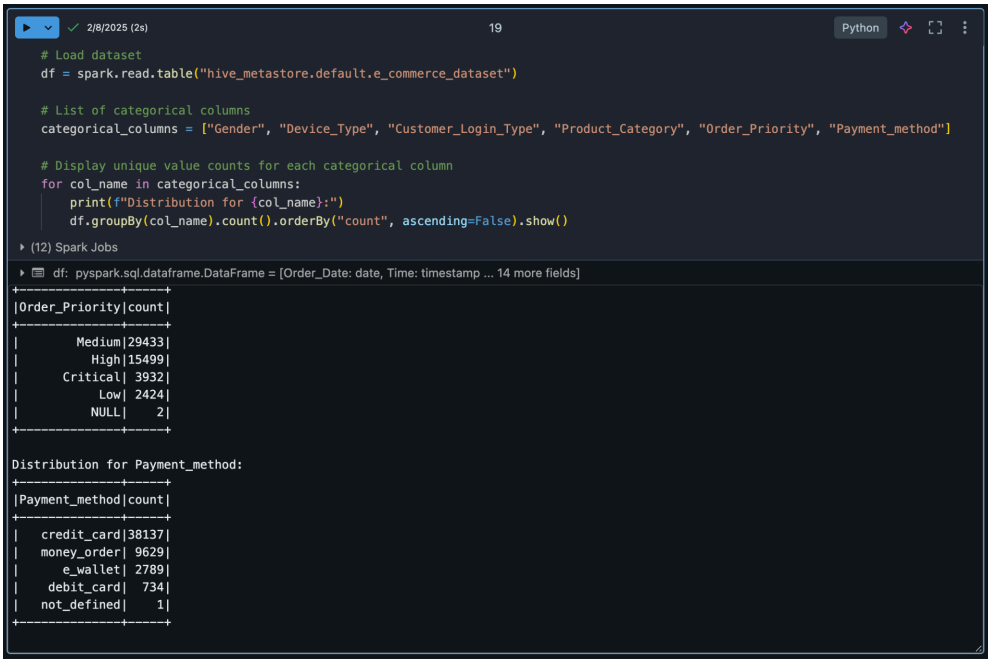
Business Impact:

- Credit cards dominate transactions, meaning loyalty programs or cashback offers may increase customer retention.
- E-wallet usage is relatively low, suggesting an opportunity for promoting digital payment methods through discounts or incentives.

Device Type Distribution

- Web transactions dominate (47,632 orders), indicating that most customers prefer shopping via desktop/laptop.
- Mobile transactions (3,658 orders) are significantly lower, suggesting potential for mobile app optimization.

Device Type	Order Count
Web	47,632
Mobile	3,658



Task 8: A Databricks notebook with SQL queries and visualizations.

2/8/2025 (1s) 20 SQL

```
%sql
SELECT Gender, COUNT(*) AS count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Gender
ORDER BY count DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Gender: string, count: long]

Table +

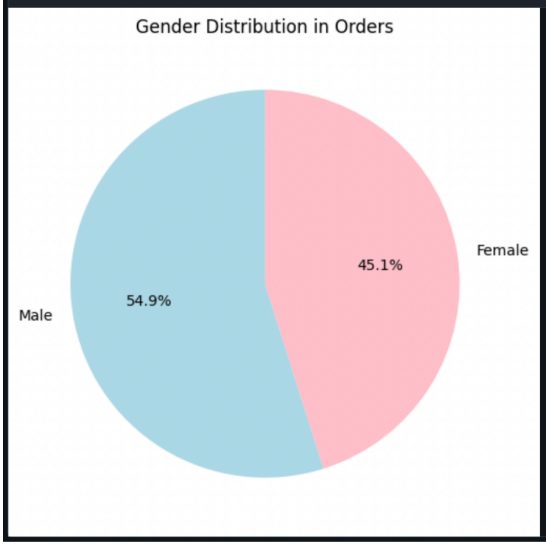
	Gender	count
1	Male	28138
2	Female	23152

2 rows | 0.91s runtime Refreshed 7 days ago

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

2 days ago (1s) 21 Python

```
import matplotlib.pyplot as plt

# Data
labels = ['Male', 'Female']
sizes = [28138, 23152]
colors = ['lightblue', 'pink']

# Create Pie Chart
plt.figure(figsize=(6,6))
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors, startangle=90)
plt.title("Gender Distribution in Orders")
plt.show()
```



2/8/2025 (<1s) 22 SQL

```sql
%sql
SELECT Device_Type, COUNT(*) AS count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Device_Type
ORDER BY count DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Device_Type: string, count: long]

Table +

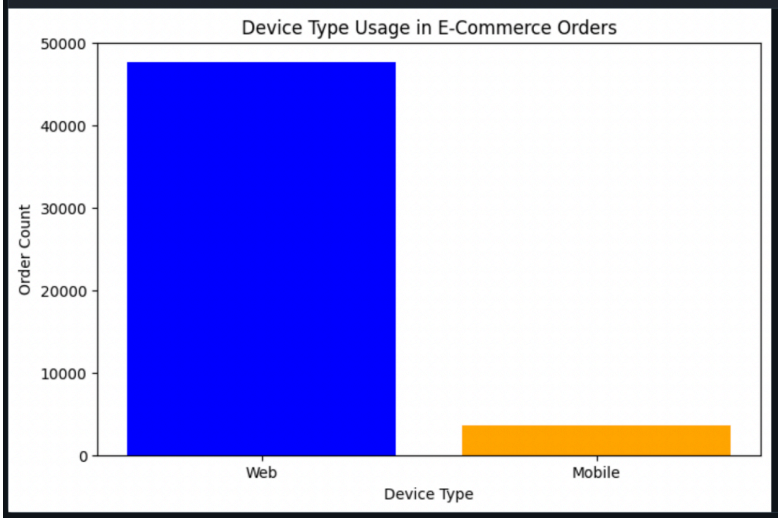
	Device_Type	count
1	Web	47632
2	Mobile	3658

2 rows | 0.46s runtime Refreshed 7 days ago

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

2 days ago (<1s) 23 Python

```python
import matplotlib.pyplot as plt

# Data
device_types = ["Web", "Mobile"]
order_counts = [47632, 3658]

# Create Bar Chart
plt.figure(figsize=(8,5))
plt.bar(device_types, order_counts, color=['blue', 'orange'])
plt.xlabel("Device Type")
plt.ylabel("Order Count")
plt.title("Device Type Usage in E-Commerce Orders")
plt.show()
```



2/8/2025 (<1s) 24 SQL

```
%sql
SELECT Customer_Login_Type, COUNT(*) AS count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Customer_Login_Type
ORDER BY count DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Customer_Login_Type: string, count: long]

Table +

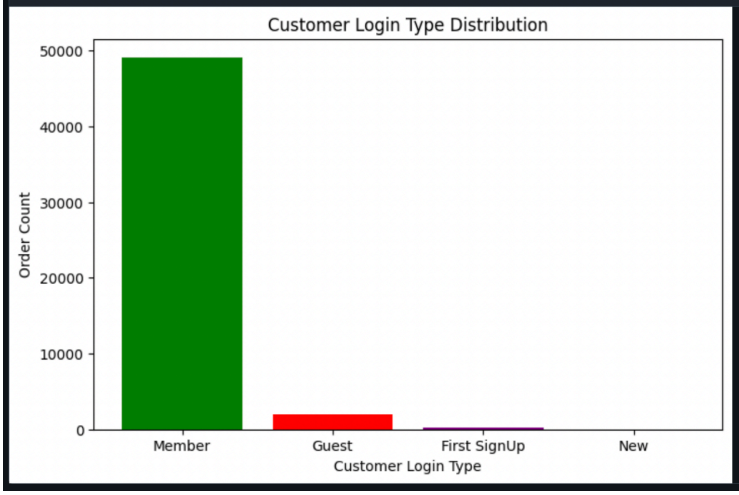
	Customer_Login_Type	count
1	Member	49097
2	Guest	1993
3	First SignUp	173
4	New	27

4 rows | 0.43s runtime Refreshed 7 days ago

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

2 days ago (<1s) 25 Python

```
# Data
login_types = ["Member", "Guest", "First SignUp", "New"]
counts = [49097, 1993, 173, 27]

# Create Bar Chart
plt.figure(figsize=(8,5))
plt.bar(login_types, counts, color=['green', 'red', 'purple', 'cyan'])
plt.xlabel("Customer Login Type")
plt.ylabel("Order Count")
plt.title("Customer Login Type Distribution")
plt.show()
```



2/8/2025 (<1s) 26 SQL

```
%sql
SELECT Product_Category, COUNT(*) AS order_count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Product_Category
ORDER BY order_count DESC
LIMIT 5;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Product_Category: string, order_count: long]

Table +

	Product_Category	order_count
1	Fashion	25646
2	Home & Furniture	15438
3	Auto & Accessories	7505
4	Electronic	2701

4 rows | 0.50s runtime Refreshed 7 days ago

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

2/8/2025 (<1s) 27 SQL

```
%sql
SELECT Order_Priority, COUNT(*) AS count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Order_Priority
ORDER BY count DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Order_Priority: string, count: long]

Table +

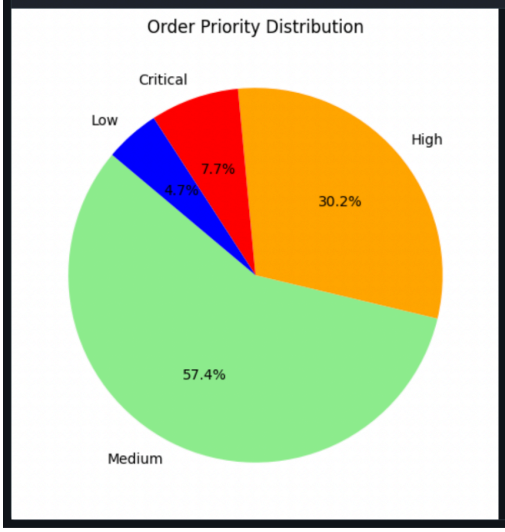
	Order_Priority	count
1	Medium	29433
2	High	15499
3	Critical	3932
4	Low	2424
5	null	2

5 rows | 0.43s runtime Refreshed 7 days ago

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

2 days ago (<1s) 28 Python

```
# Data
labels = ["Medium", "High", "Critical", "Low"]
sizes = [29433, 15499, 3932, 2424]
colors = ['lightgreen', 'orange', 'red', 'blue']

# Create Pie Chart
plt.figure(figsize=(6,6))
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors, startangle=140)
plt.title("Order Priority Distribution")
plt.show()
```



2/8/2025 (<1s) 29 SQL

```
%sql
SELECT Payment_method, COUNT(*) AS count
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Payment_method
ORDER BY count DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Payment_method: string, count: long]

Table +

	Payment_method	count
1	credit_card	38137
2	money_order	9629
3	e_wallet	2789
4	debit_card	734
5	not_defined	1

5 rows | 0.41s runtime Refreshed 7 days ago

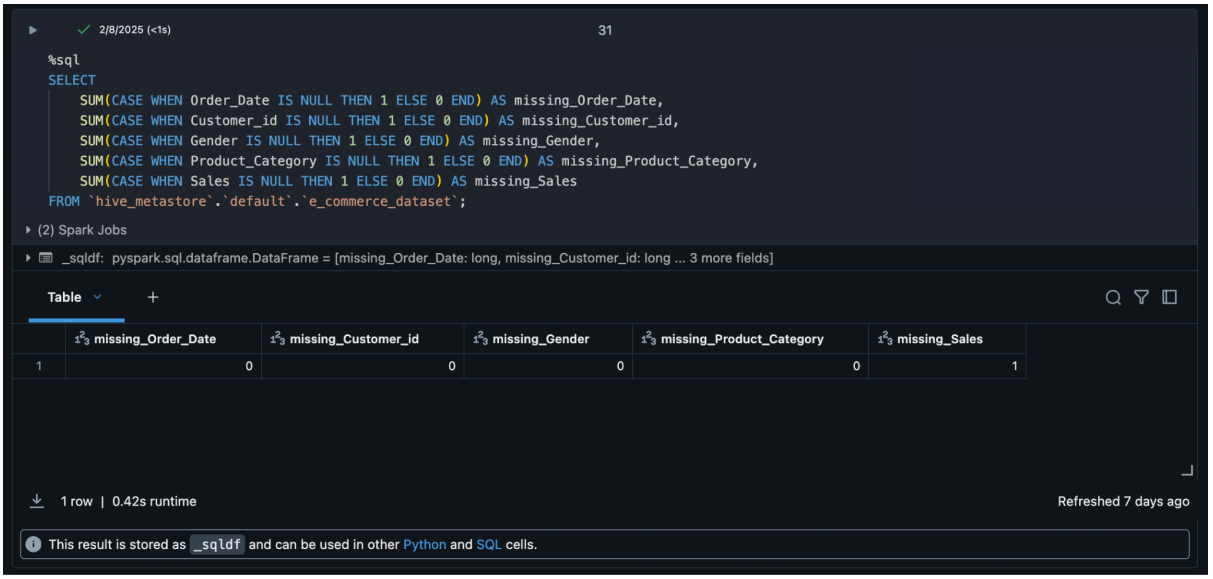
This result is stored as _sqldf and can be used in other Python and SQL cells.

Task 9: Identify and handle missing values in a given dataset.

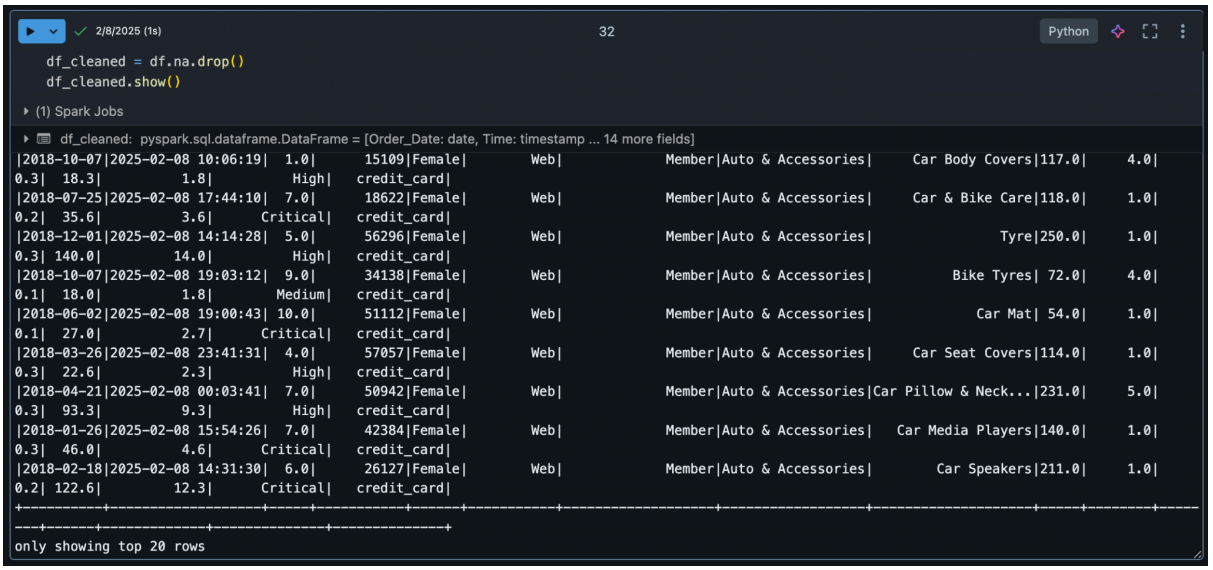
Based on the SQL query output, the dataset contains minimal missing values:

Column Name	Missing Values
Order_Date	0
Customer_ID	0
Gender	0
Product_Category	0
Sales	1

- **Only one record** has a missing value in the **Sales column**.
- Other critical columns, such as **Order Date, Customer ID, Gender, and Product Category,** have no missing values, ensuring a mostly complete dataset.

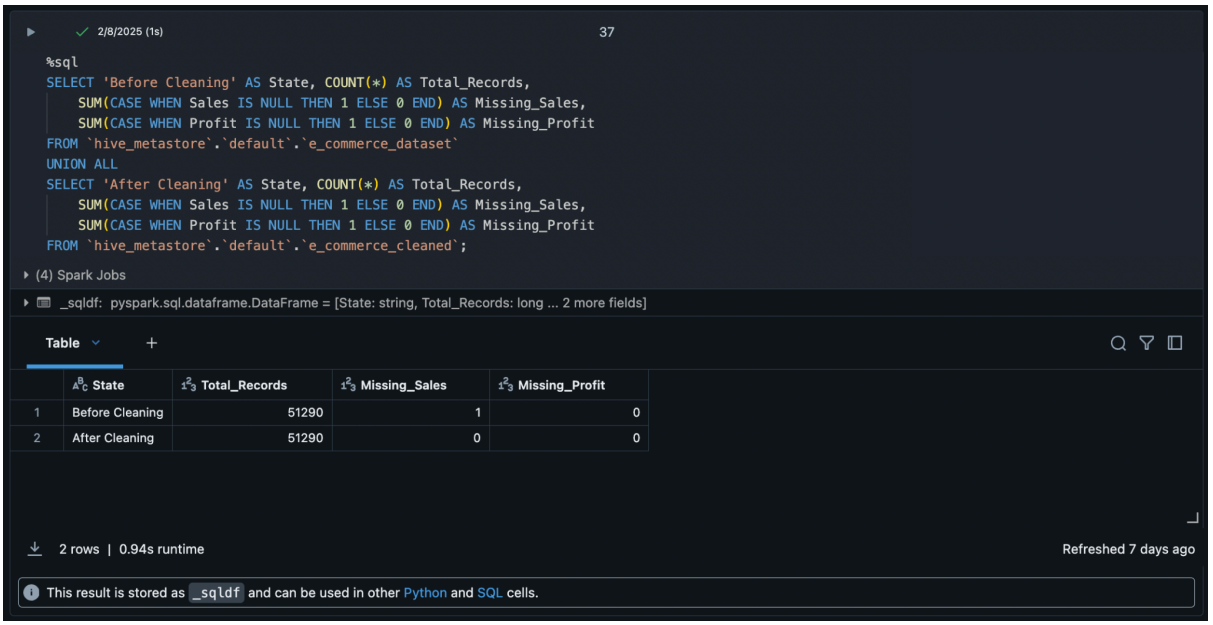


Task 10: Use appropriate techniques such as imputation or removal of rows/columns to handle missing values.



```
2/8/2025 (1s)                                                    32                                                    Python

df_cleaned = df.na.drop()
df_cleaned.show()

(1) Spark Jobs

df_cleaned:  pyspark.sql.dataframe.DataFrame = [Order_Date: date, Time: timestamp ... 14 more fields]

|2018-10-07|2025-02-08 10:06:19|  1.0|      15109|Female|        Web|            Member|Auto & Accessories|     Car Body Covers|117.0|     4.0|
0.3|  18.3|          1.8|          High|   credit_card|
|2018-07-25|2025-02-08 17:44:10|  7.0|      18622|Female|        Web|            Member|Auto & Accessories|     Car & Bike Care|118.0|     1.0|
0.2|  35.6|          3.6|      Critical|   credit_card|
|2018-12-01|2025-02-08 14:14:28|  5.0|      56296|Female|        Web|            Member|Auto & Accessories|                Tyre|250.0|     1.0|
0.3| 140.0|         14.0|          High|   credit_card|
|2018-10-07|2025-02-08 19:03:12|  9.0|      34138|Female|        Web|            Member|Auto & Accessories|          Bike Tyres| 72.0|     4.0|
0.1|  18.0|          1.8|        Medium|   credit_card|
|2018-06-02|2025-02-08 19:00:43| 10.0|      51112|Female|        Web|            Member|Auto & Accessories|             Car Mat| 54.0|     1.0|
0.1|  27.0|          2.7|      Critical|   credit_card|
|2018-03-26|2025-02-08 23:41:31|  4.0|      57057|Female|        Web|            Member|Auto & Accessories|     Car Seat Covers|114.0|     1.0|
0.3|  22.6|          2.3|          High|   credit_card|
|2018-04-21|2025-02-08 00:03:41|  7.0|      50942|Female|        Web|            Member|Auto & Accessories|Car Pillow & Neck...|231.0|     5.0|
0.3|  93.3|          9.3|          High|   credit_card|
|2018-01-26|2025-02-08 15:54:26|  7.0|      42384|Female|        Web|            Member|Auto & Accessories|   Car Media Players|140.0|     1.0|
0.3|  46.0|          4.6|      Critical|   credit_card|
|2018-02-18|2025-02-08 14:31:30|  6.0|      26127|Female|        Web|            Member|Auto & Accessories|        Car Speakers|211.0|     1.0|
0.2| 122.6|         12.3|      Critical|   credit_card|
+----------+-------------------+-----+-----------+------+-----------+------------------+------------------+--------------------+-----+--------+-----
---+------+-------------+--------------+--------------+
only showing top 20 rows
```

Task 11: A Databricks notebook showing the before and after state of the data with explanations of the chosen methods.



```
2/8/2025 (1s)                                                    37

%sql
SELECT 'Before Cleaning' AS State, COUNT(*) AS Total_Records,
    SUM(CASE WHEN Sales IS NULL THEN 1 ELSE 0 END) AS Missing_Sales,
    SUM(CASE WHEN Profit IS NULL THEN 1 ELSE 0 END) AS Missing_Profit
FROM `hive_metastore`.`default`.`e_commerce_dataset`
UNION ALL
SELECT 'After Cleaning' AS State, COUNT(*) AS Total_Records,
    SUM(CASE WHEN Sales IS NULL THEN 1 ELSE 0 END) AS Missing_Sales,
    SUM(CASE WHEN Profit IS NULL THEN 1 ELSE 0 END) AS Missing_Profit
FROM `hive_metastore`.`default`.`e_commerce_cleaned`;

(4) Spark Jobs

_sqldf:  pyspark.sql.dataframe.DataFrame = [State: string, Total_Records: long ... 2 more fields]

Table +

  State          Total_Records   Missing_Sales   Missing_Profit
1 Before Cleaning        51290               1                0
2 After Cleaning         51290               0                0

2 rows | 0.94s runtime                                                    Refreshed 7 days ago

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

Task 12: Apply necessary transformations to the dataset.

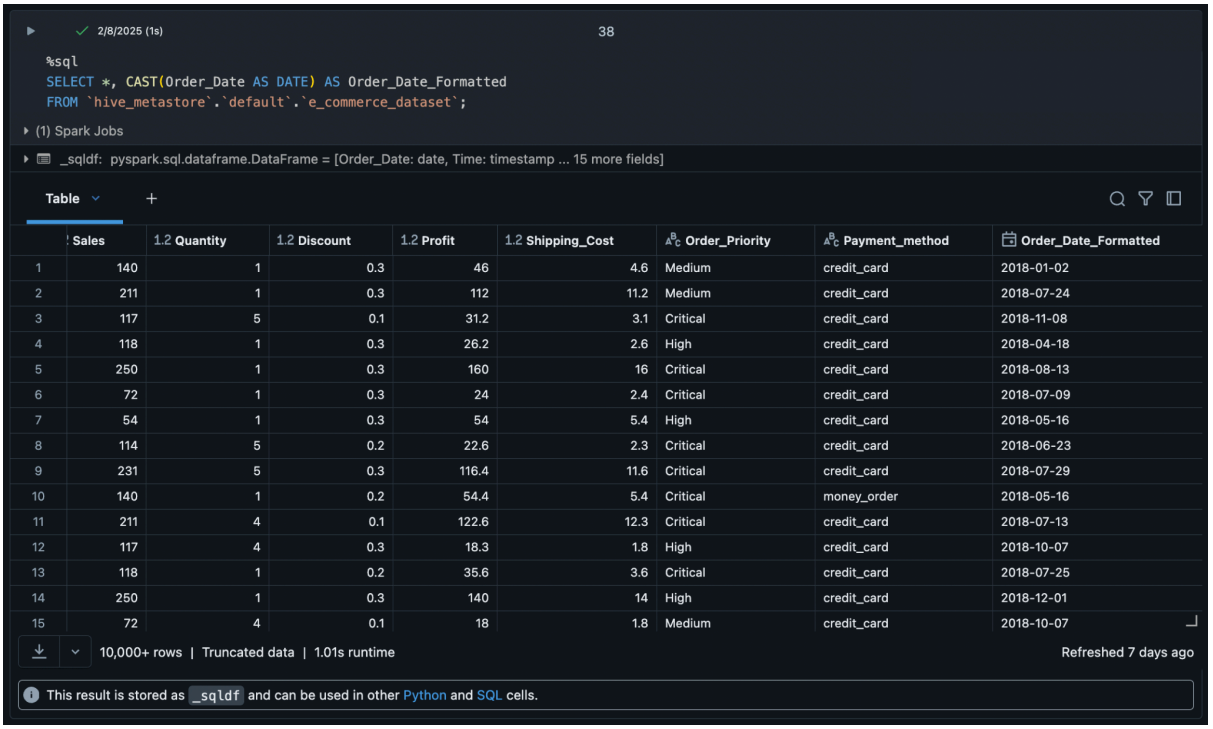
Transforming the dataset is a critical step to improve **data quality, usability, and analytical accuracy**. Below are the key transformations applied, their purpose, and impact on the dataset.

1. Formatting the Date Column

```
%sql
SELECT *, CAST(Order_Date AS DATE) AS Order_Date_Formatted
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Why?

- Converts the **Order_Date** field into a standard **DATE format** for consistency and easier filtering.
- Prevents issues with different time formats in future analyses.



	Sales	Quantity	Discount	Profit	Shipping_Cost	Order_Priority	Payment_method	Order_Date_Formatted
1	140	1	0.3	46	4.6	Medium	credit_card	2018-01-02
2	211	1	0.3	112	11.2	Medium	credit_card	2018-07-24
3	117	5	0.1	31.2	3.1	Critical	credit_card	2018-11-08
4	118	1	0.3	26.2	2.6	High	credit_card	2018-04-18
5	250	1	0.3	160	16	Critical	credit_card	2018-08-13
6	72	1	0.3	24	2.4	Critical	credit_card	2018-07-09
7	54	1	0.3	54	5.4	High	credit_card	2018-05-16
8	114	5	0.2	22.6	2.3	Critical	credit_card	2018-06-23
9	231	5	0.3	116.4	11.6	Critical	credit_card	2018-07-29
10	140	1	0.2	54.4	5.4	Critical	money_order	2018-05-16
11	211	4	0.1	122.6	12.3	Critical	credit_card	2018-07-13
12	117	4	0.3	18.3	1.8	High	credit_card	2018-10-07
13	118	1	0.2	35.6	3.6	Critical	credit_card	2018-07-25
14	250	1	0.3	140	14	High	credit_card	2018-12-01
15	72	4	0.1	18	1.8	Medium	credit_card	2018-10-07

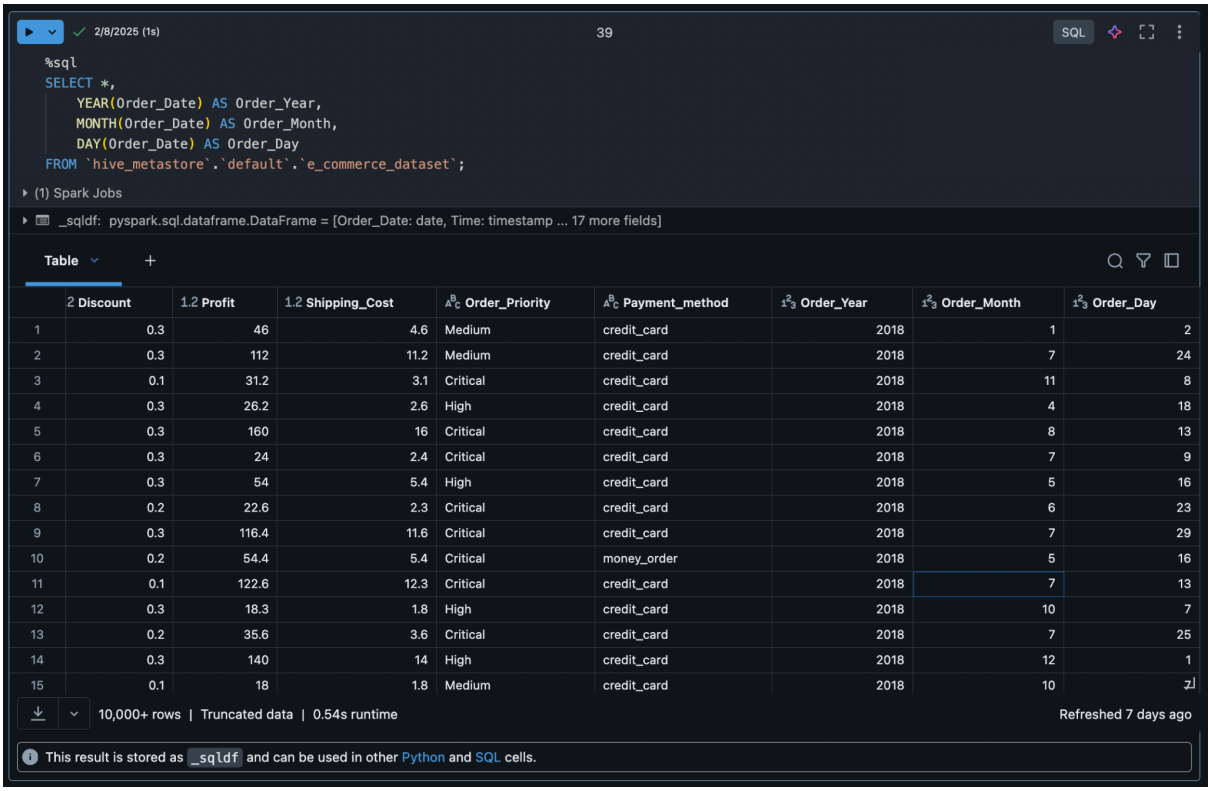
2. Extracting Date Components

```
%sql
SELECT *,
   YEAR(Order_Date) AS Order_Year,
   MONTH(Order_Date) AS Order_Month,
   DAY(Order_Date) AS Order_Day
FROM `hive_metastore`.`default`.`e_commerce_dataset`
```

Why?

- Extracts **Year, Month, and Day** from the **Order_Date** column.

- Helps in **seasonality analysis** (e.g., peak sales months).



```
%sql
SELECT *,
    YEAR(Order_Date) AS Order_Year,
    MONTH(Order_Date) AS Order_Month,
    DAY(Order_Date) AS Order_Day
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Order_Date: date, Time: timestamp ... 17 more fields]

Table

	Discount	Profit	Shipping_Cost	Order_Priority	Payment_method	Order_Year	Order_Month	Order_Day
1	0.3	46	4.6	Medium	credit_card	2018	1	2
2	0.3	112	11.2	Medium	credit_card	2018	7	24
3	0.1	31.2	3.1	Critical	credit_card	2018	11	8
4	0.3	26.2	2.6	High	credit_card	2018	4	18
5	0.3	160	16	Critical	credit_card	2018	8	13
6	0.3	24	2.4	Critical	credit_card	2018	7	9
7	0.3	54	5.4	High	credit_card	2018	5	16
8	0.2	22.6	2.3	Critical	credit_card	2018	6	23
9	0.3	116.4	11.6	Critical	credit_card	2018	7	29
10	0.2	54.4	5.4	Critical	money_order	2018	5	16
11	0.1	122.6	12.3	Critical	credit_card	2018	7	13
12	0.3	18.3	1.8	High	credit_card	2018	10	7
13	0.2	35.6	3.6	Critical	credit_card	2018	7	25
14	0.3	140	14	High	credit_card	2018	12	1
15	0.1	18	1.8	Medium	credit_card	2018	10	27

10,000+ rows | Truncated data | 0.54s runtime — Refreshed 7 days ago

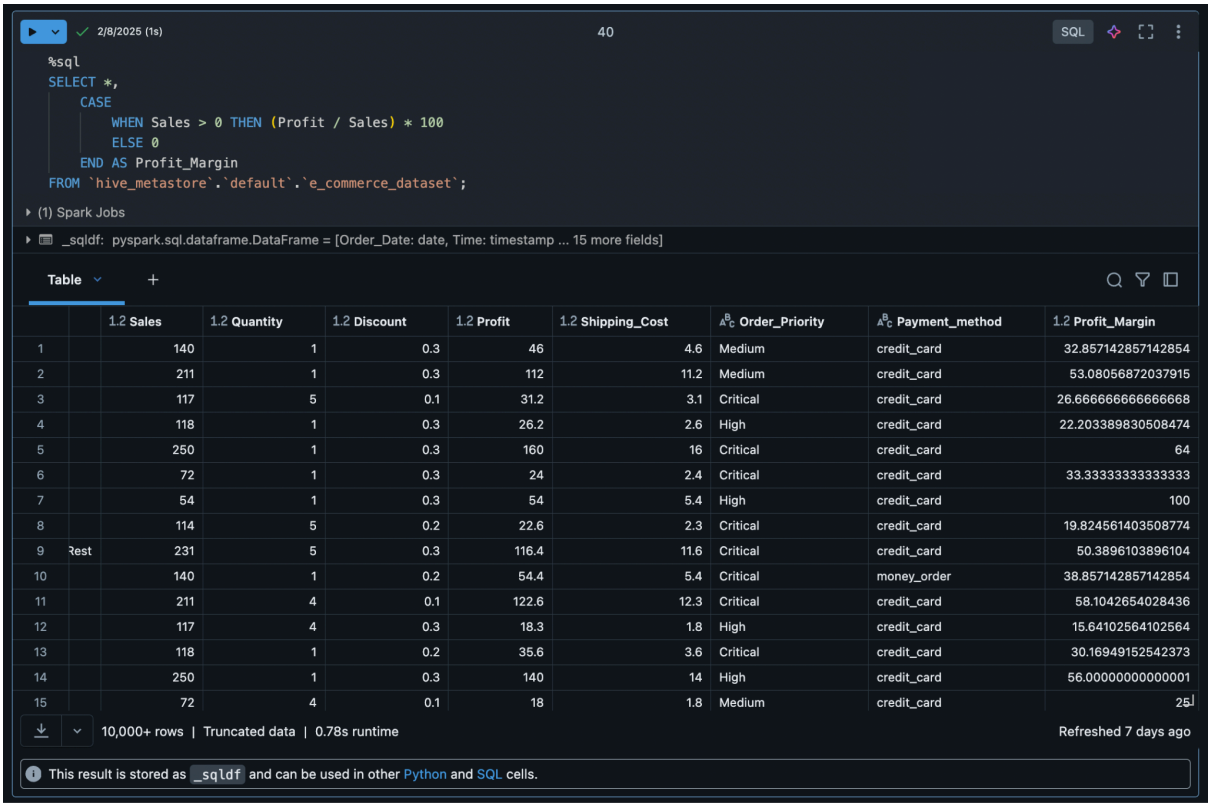
This result is stored as `_sqldf` and can be used in other Python and SQL cells.

3. Calculating Profit Margin

```
%sql
SELECT *,
   CASE
      WHEN Sales > 0 THEN (Profit / Sales) * 100
      ELSE 0
   END AS Profit_Margin
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Why?

- Computes **Profit Margin (%)** for each transaction to assess profitability.
- Helps in evaluating **high-profit vs. low-profit categories**.

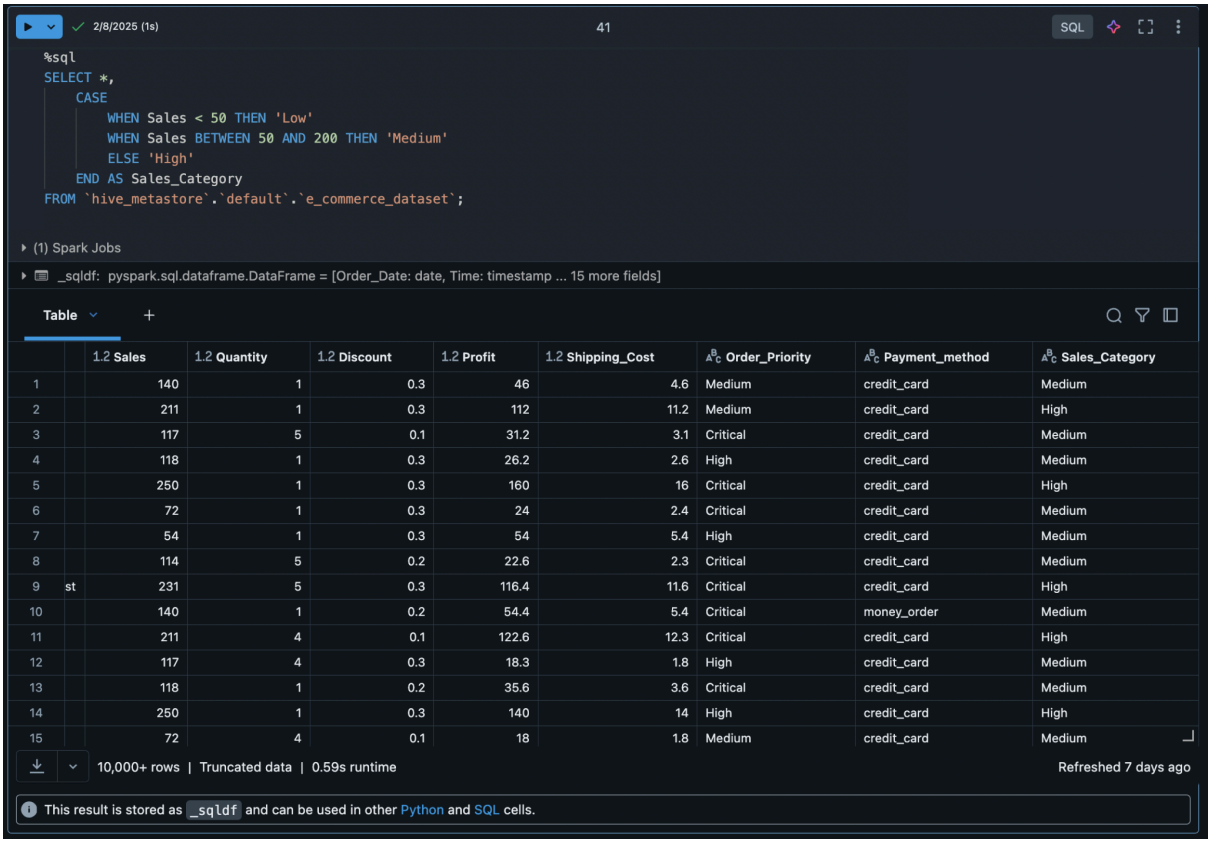


4. Categorizing Sales into Low, Medium, and High

```
%sql
SELECT *,
   CASE
      WHEN Sales < 50 THEN 'Low'
      WHEN Sales BETWEEN 50 AND 200 THEN 'Medium'
      ELSE 'High'
   END AS Sales_Category
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Why?

- Classifies sales into **Low, Medium, and High** to segment transactions based on value.
- Helps in identifying **premium vs. budget-friendly products**.



5. Handling Missing Values and Creating a Transformed Table

```
%sql
CREATE TABLE IF NOT EXISTS `hive_metastore`.`default`.`e_commerce_transformed` AS
SELECT
  Customer_id,
  COALESCE(Gender, 'Unknown') AS Gender,
  COALESCE(Product_Category, 'Other') AS Product_Category,
  COALESCE(Sales, 0) AS Sales,
  COALESCE(Profit, (SELECT AVG(Profit) FROM
`hive_metastore`.`default`.`e_commerce_dataset`)) AS Profit,
  CAST(Order_Date AS DATE) AS Order_Date_Formatted,
  YEAR(Order_Date) AS Order_Year,
  MONTH(Order_Date) AS Order_Month,
  DAY(Order_Date) AS Order_Day,
  CASE WHEN Sales > 0 THEN (Profit / Sales) * 100 ELSE 0 END AS Profit_Margin,
  CASE
    WHEN Sales < 50 THEN 'Low'
    WHEN Sales BETWEEN 50 AND 200 THEN 'Medium'
    ELSE 'High'
  END AS Sales_Category
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Why?

- **COALESCE** is used to replace missing values:
  - **Gender:** "Unknown" (avoids NULL values in analysis).
  - **Product_Category:** "Other" (keeps consistency in categories).
  - **Sales:** 0 (ensures no missing financial values).
  - **Profit:** Uses **average profit** for missing values (avoids data distortion).
- Stores transformed data in a **new table (`e_commerce_transformed`)**.

6. Verification of Transformation

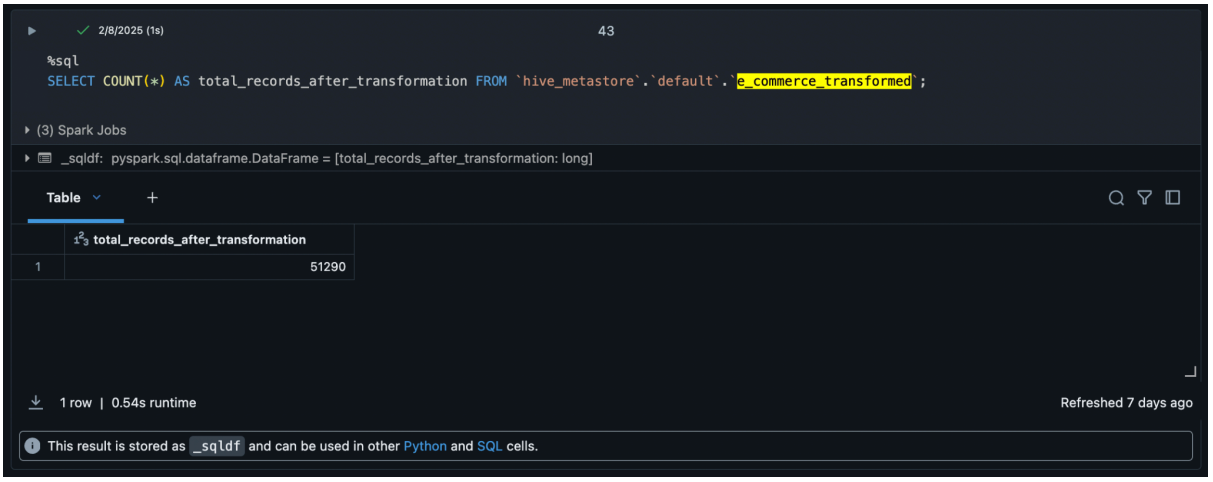
%sql
SELECT COUNT(*) AS total_records_after_transformation FROM
`hive_metastore`.`default`.`e_commerce_transformed`;

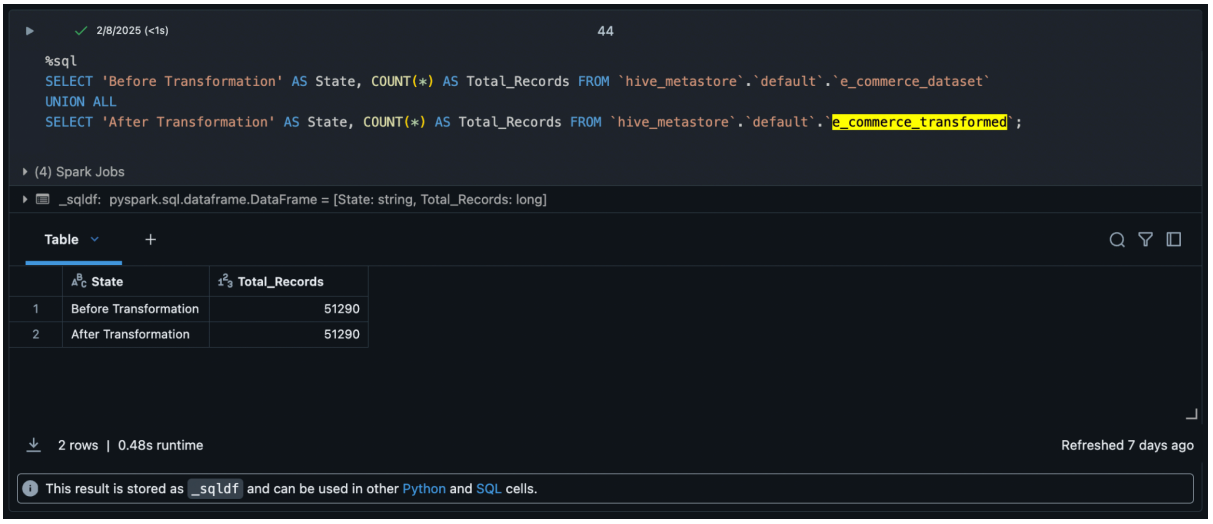
Result: The transformed dataset retains all **51,290 records**, ensuring no data loss.

Before vs. After Transformation:

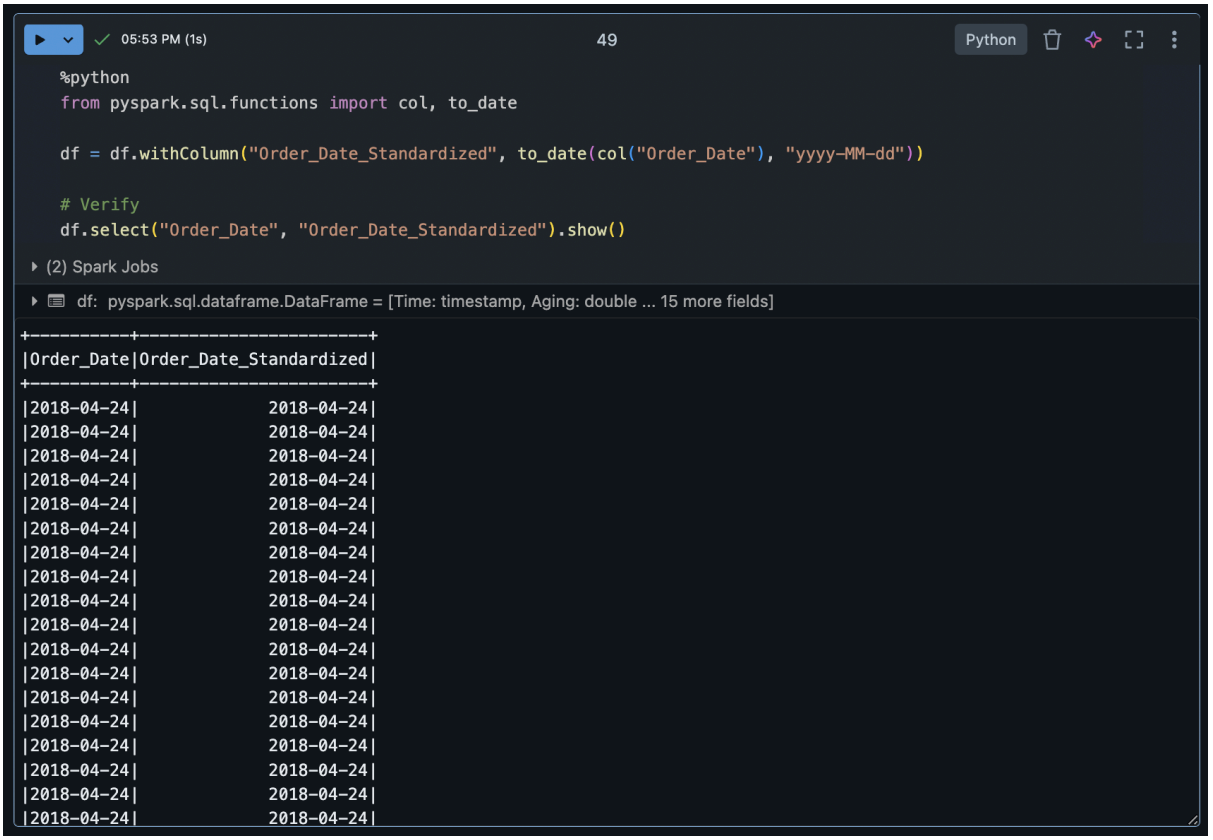
State	Total Records
Before Transformation	51,290
After Transformation	51,290

Since the record count remains **unchanged**, the transformation process successfully **enhanced data quality without data loss**.


2/8/2025 (1s)
43
%sql
SELECT COUNT(*) AS total_records_after_transformation FROM `hive_metastore`.`default`.`e_commerce_transformed`;
(3) Spark Jobs
_sqldf: pyspark.sql.dataframe.DataFrame = [total_records_after_transformation: long]
Table
total_records_after_transformation
1
51290
1 row | 0.54s runtime
Refreshed 7 days ago
This result is stored as _sqldf and can be used in other Python and SQL cells.


2/8/2025 (<1s)
44
%sql
SELECT 'Before Transformation' AS State, COUNT(*) AS Total_Records FROM `hive_metastore`.`default`.`e_commerce_dataset`
UNION ALL
SELECT 'After Transformation' AS State, COUNT(*) AS Total_Records FROM `hive_metastore`.`default`.`e_commerce_transformed`;
(4) Spark Jobs
_sqldf: pyspark.sql.dataframe.DataFrame = [State: string, Total_Records: long]
Table
State
Total_Records
1
Before Transformation
51290
2
After Transformation
51290
2 rows | 0.48s runtime
Refreshed 7 days ago
This result is stored as _sqldf and can be used in other Python and SQL cells.

Task 13: Standardize date formats.



Task 14: Convert categorical variables to numerical formats (using techniques like one-hot encoding).

Objective

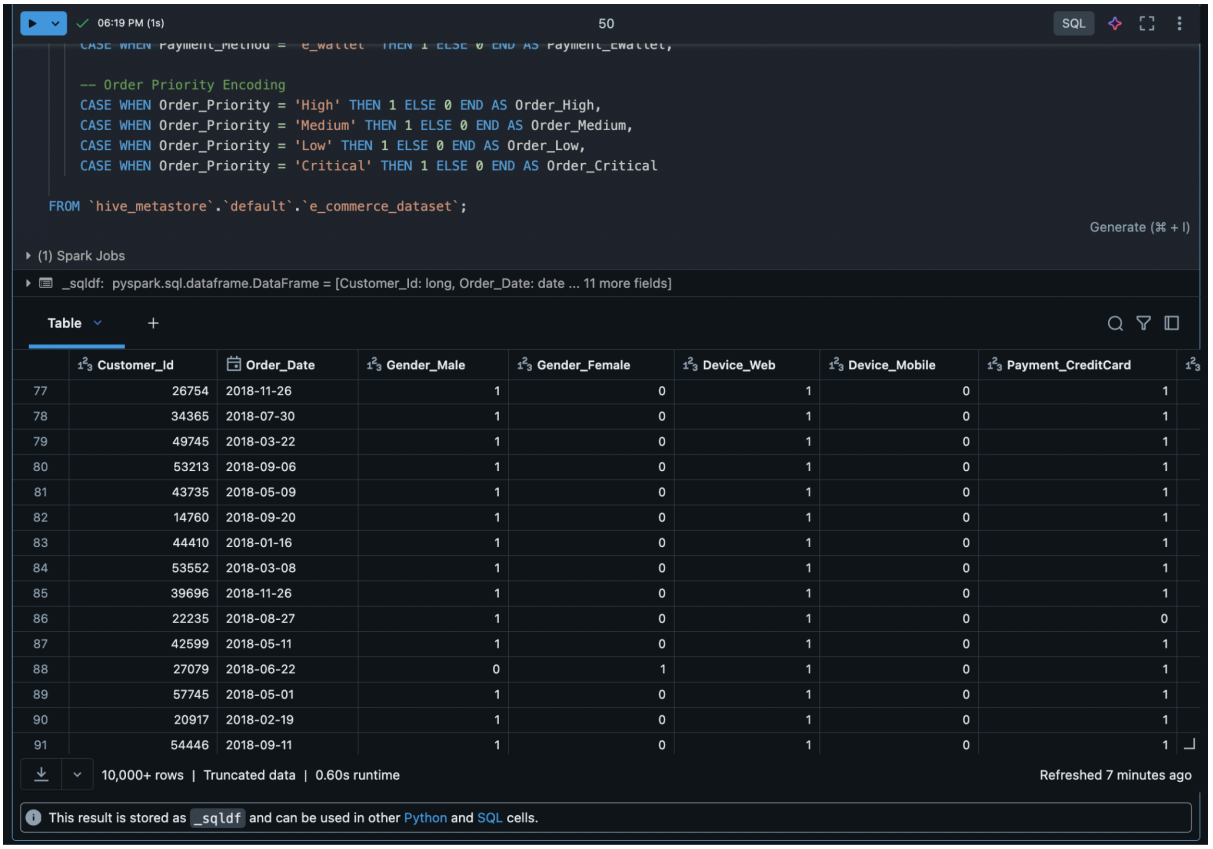
The goal is to convert categorical variables into numerical format using **One-Hot Encoding (OHE)** so that they can be used for **machine learning models, analytics, and statistical processing**.

The given SQL query converts categorical variables into **binary columns (0 or 1)** for each unique category. Below is the breakdown:

Categorical Columns Transformed

Original Column	Encoded Columns	Transformation
Gender	`Gender_Male`, `Gender_Female`	Male = 1, Female = 1 (Mutually exclusive)
Device_Type	`Device_Web`, `Device_Mobile`	Web = 1, Mobile = 1
Payment_Method	`Payment_CreditCard`, `Payment_DebitCard`, `Payment_EWallet`	Converts each payment method into separate columns

Order_Priority	`Order_High`, `Order_Medium`, `Order_Low`, `Order_Critical`	Encodes different priority levels



Task 15: Normalize numerical variables.

Normalization is essential to bring numerical values into a **consistent scale** (e.g., between 0 and 1 or standardizing to have a mean of 0 and a standard deviation of 1). This is important for **machine learning models** that are sensitive to scale differences.

The following numerical columns in our dataset require normalization:

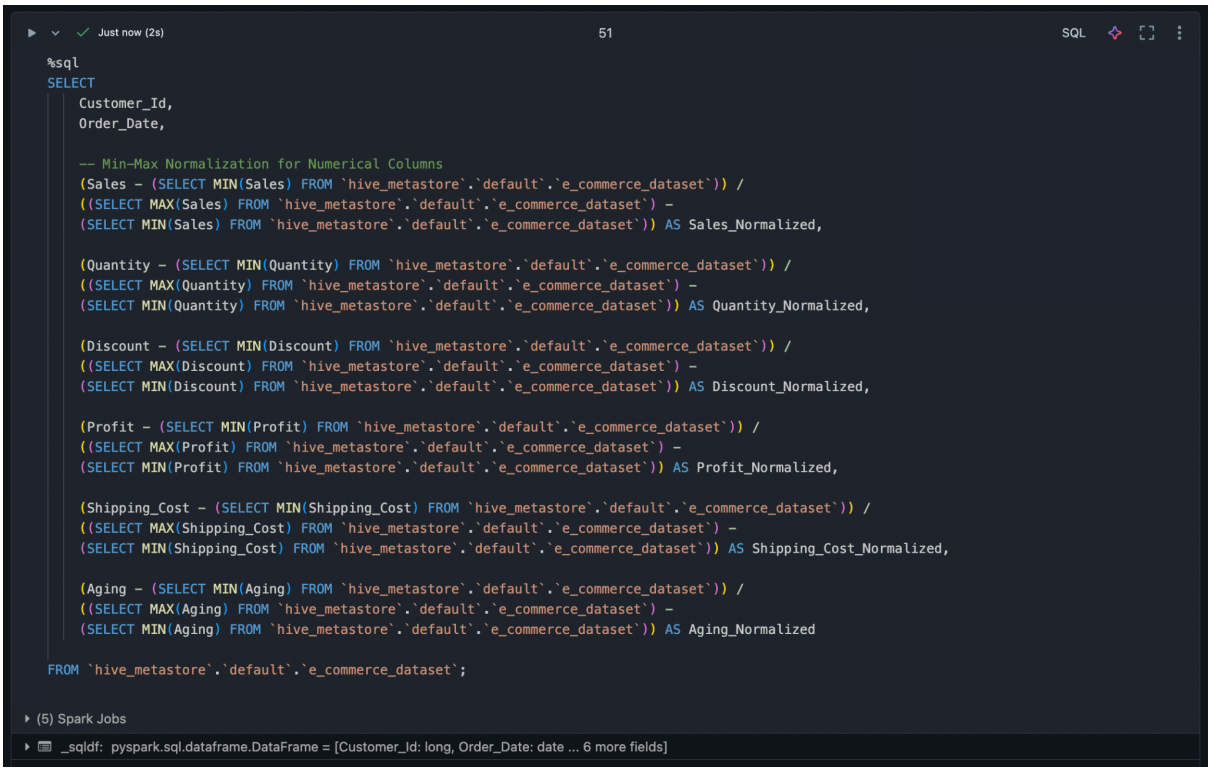
Column Name	Description
Sales	Total sales per transaction
Quantity	Number of items purchased
Discount	Discount percentage applied
Profit	Profit from the transaction
Shipping_Cost	Cost of shipping
Aging	Number of days taken for delivery

2. SQL Approach: Min-Max Normalization

Min-Max normalization scales values between **0 and 1** using the formula:

$$X\prime = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

3. Creating a Normalized Table



```
%sql
SELECT
    Customer_Id,
    Order_Date,

    -- Min-Max Normalization for Numerical Columns
    (Sales - (SELECT MIN(Sales) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Sales) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Sales) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Sales_Normalized,

    (Quantity - (SELECT MIN(Quantity) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Quantity) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Quantity) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Quantity_Normalized,

    (Discount - (SELECT MIN(Discount) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Discount) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Discount) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Discount_Normalized,

    (Profit - (SELECT MIN(Profit) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Profit) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Profit) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Profit_Normalized,

    (Shipping_Cost - (SELECT MIN(Shipping_Cost) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Shipping_Cost) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Shipping_Cost) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Shipping_Cost_Normalized,

    (Aging - (SELECT MIN(Aging) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) /
    ((SELECT MAX(Aging) FROM `hive_metastore`.`default`.`e_commerce_dataset`) -
    (SELECT MIN(Aging) FROM `hive_metastore`.`default`.`e_commerce_dataset`)) AS Aging_Normalized

FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

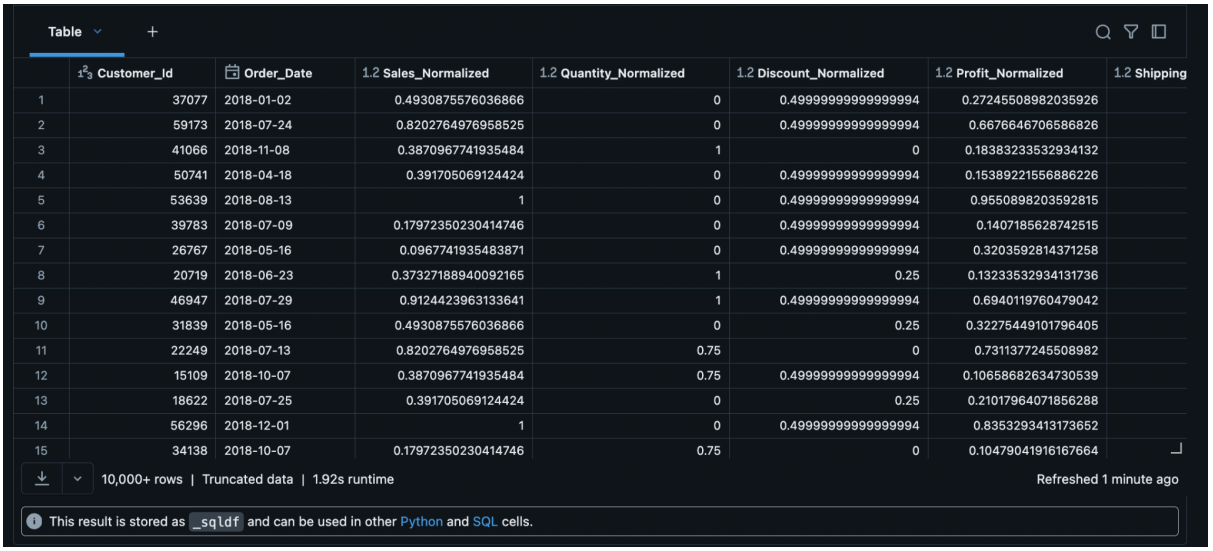


Table +

	Customer_Id	Order_Date	Sales_Normalized	Quantity_Normalized	Discount_Normalized	Profit_Normalized	Shipping
1	37077	2018-01-02	0.4930875576036866	0	0.49999999999999994	0.27245508982035926	
2	59173	2018-07-24	0.8202764976958525	0	0.49999999999999994	0.6676646706586826	
3	41066	2018-11-08	0.3870967741935484	1	0	0.18383233532934132	
4	50741	2018-04-18	0.391705069124424	0	0.49999999999999994	0.15389221556886226	
5	53639	2018-08-13	1	0	0.49999999999999994	0.9550898203592815	
6	39783	2018-07-09	0.17972350230414746	0	0.49999999999999994	0.1407185628742515	
7	26767	2018-05-16	0.0967741935483871	0	0.49999999999999994	0.3203592814371258	
8	20719	2018-06-23	0.37327188940092165	1	0.25	0.13233532934131736	
9	46947	2018-07-29	0.9124423963133641	1	0.49999999999999994	0.6940119760479042	
10	31839	2018-05-16	0.4930875576036866	0	0.25	0.32275449101796405	
11	22249	2018-07-13	0.8202764976958525	0.75	0	0.7311377245508982	
12	15109	2018-10-07	0.3870967741935484	0.75	0.49999999999999994	0.10658682634730539	
13	18622	2018-07-25	0.391705069124424	0	0.25	0.21017964071856288	
14	56296	2018-12-01	1	0	0.49999999999999994	0.8353293413173652	
15	34138	2018-10-07	0.17972350230414746	0.75	0	0.10479041916167664	

10,000+ rows | Truncated data | 1.92s runtime

Refreshed 1 minute ago

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