

# 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

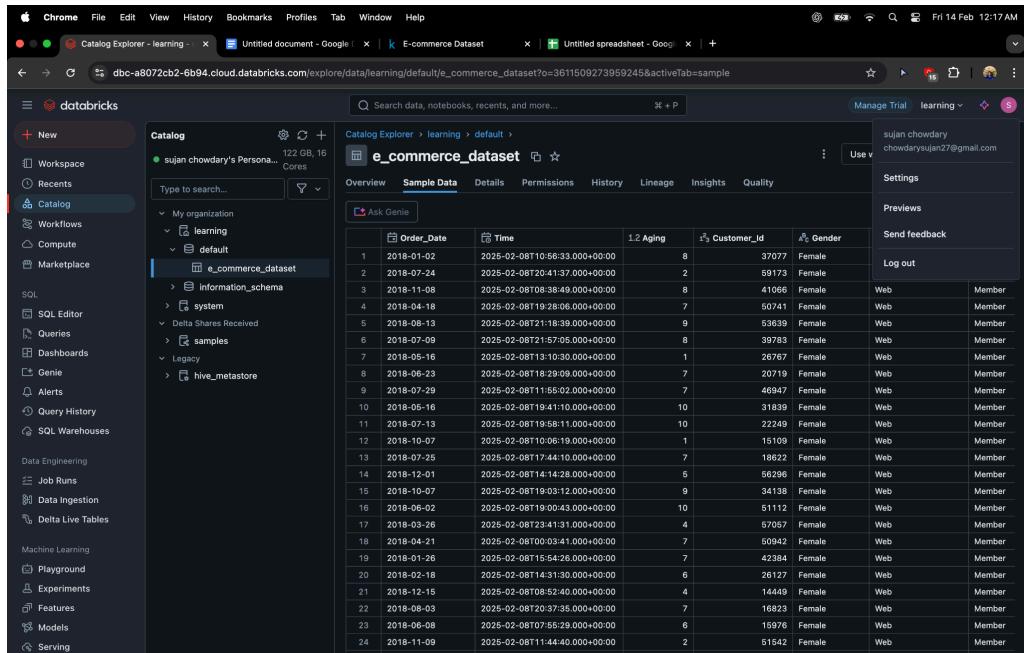
This dataset will be used for **data ingestion, exploratory data analysis (EDA), and transformation** to derive meaningful business insights and optimize data processing in Databricks.

## Tasks:

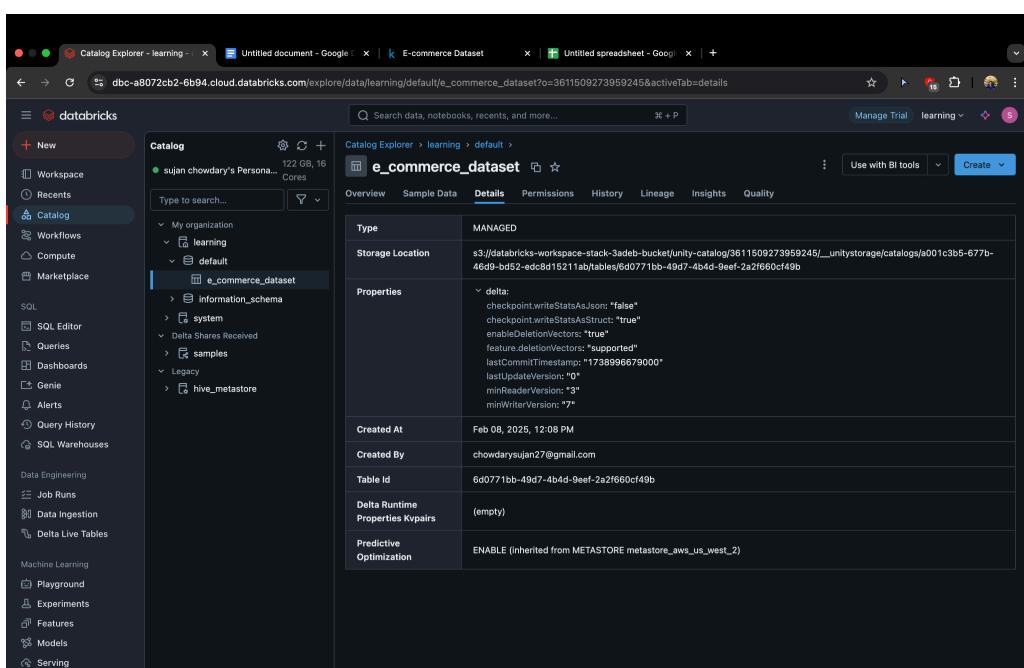
**Task 1: Use Databricks to upload your dataset (CSV format) into a table.**

### 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”



The screenshot shows the Databricks Catalog Explorer interface. On the left sidebar, under the Catalog section, there is a tree view showing 'My organization' and 'learning'. Under 'learning', there is a 'default' folder which contains an 'e\_commerce\_dataset'. This dataset is highlighted in blue. To the right of the tree view, there is a search bar and a 'Sample Data' button. Below the search bar, there are tabs for Overview, Sample Data, Details, Permissions, History, Lineage, Insights, and Quality. The 'Sample Data' tab is active, displaying a table with 25 rows of data. The columns are Order\_Date, Time, Aging, Customer\_Id, and Gender. The data spans from January 2018 to October 2018. The 'Details' tab is also visible, showing the table's properties, storage location (s3://databricks-workspace-stack-3deb-bucket/unity-catalog/3611509273959245/\_unitystorage/catalogs/a001c3b5-677b-46db-bd52-edc8d15211ab/tables/6d0771bb-49d7-4b4d-9eef-2a2f660cf49b), and other metadata like type (MANAGED), created at (Feb 08, 2025, 12:08 PM), and created by (chowdarysujan27@gmail.com).



The screenshot shows the 'Details' tab for the 'e\_commerce\_dataset' table. It displays various properties of the table, such as Type (MANAGED), Storage Location (s3://databricks-workspace-stack-3deb-bucket/unity-catalog/3611509273959245/\_unitystorage/catalogs/a001c3b5-677b-46db-bd52-edc8d15211ab/tables/6d0771bb-49d7-4b4d-9eef-2a2f660cf49b), Properties (delta: checkpoint.writeStatusAsJson: "false", checkpoint.writeStatusAsStruct: "true", enableDeletionVectors: "true", feature.deletionVectors: "supported", lastCommitTimestamp: "1738996679000", lastUpdateVersion: "0\*", minReaderVersion: "3", minWriterVersion: "7"), and Predictive Optimization (ENABLE). The sidebar on the left remains the same as the previous screenshot.

## 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.

The screenshot shows a Databricks notebook interface. At the top, there's a header bar with 'Explore hive\_metastore.default.e\_commerce\_dataset 2025-02-08 22:25:...' and various navigation options like File, Edit, View, Run, Help, and a 'Last edit was 4 minutes ago' message. Below the header is a search bar and a toolbar with icons for Python, SQL, and other operations. The main area contains a code cell with the following content:

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

Below the code cell, it says '(2) Spark Jobs'. Under the 'Table' tab, there's a preview of the data with the following columns: Order\_Date, Time, Aging, Customer\_Id, Gender, Device\_Type, and Customer\_Login\_. The data consists of 15 rows of customer transaction details. At the bottom of the table preview, it says '10,000+ rows | Truncated data | 9.82s runtime' and 'Refreshed 5 days ago'. A note at the bottom states: 'This result is stored as \_sqlldf and can be used in other Python and SQL cells.'

### **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.

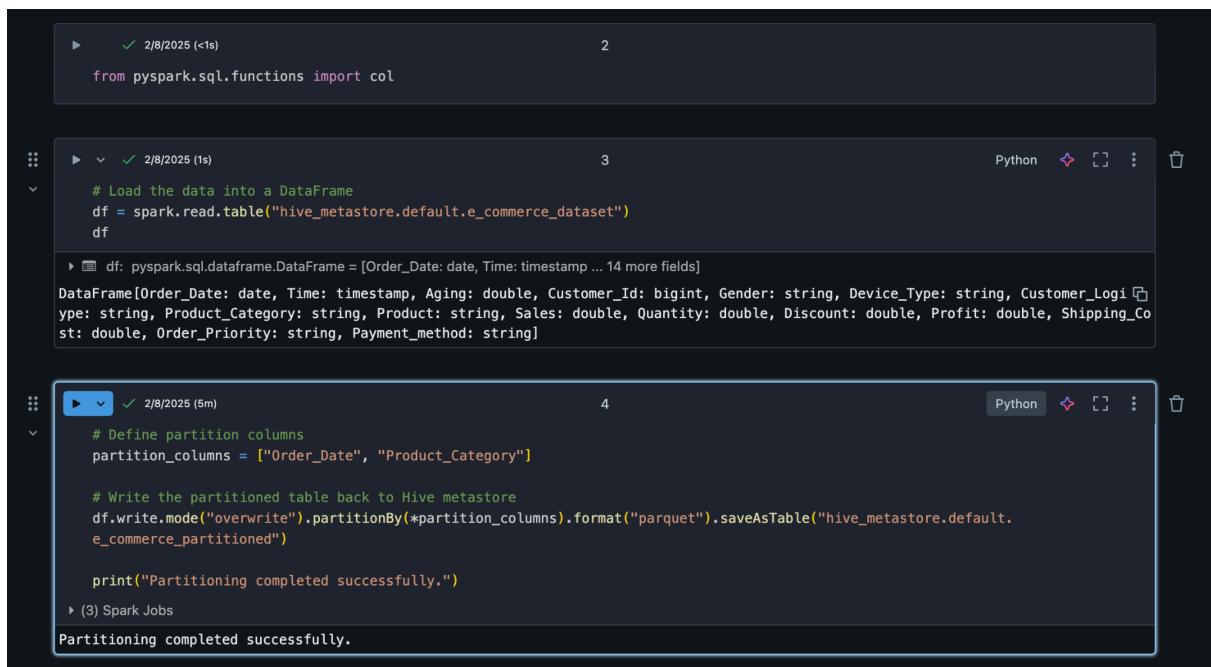
#### **Write the partitioned table back to Hive metastore**

```
`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.



```

1  ▶   ✓  2/8/2025 (<1s) 2
from pyspark.sql.functions import col

2  ▶   ✓  2/8/2025 (1s) 3
# Load the data into a DataFrame
df = spark.read.table("hive_metastore.default.e_commerce_dataset")
df

3  ▶ df: pyspark.sql.DataFrame = [Order_Date: date, Time: timestamp ... 14 more fields]
DataFrame[Order_Date: date, Time: timestamp, Aging: double, Customer_Id: bigint, Gender: string, Device_Type: string, Customer_Logi
ype: string, Product_Category: string, Product: string, Sales: double, Quantity: double, Discount: double, Profit: double, Shipping_Co
st: double, Order_Priority: string, Payment_method: string]

4  ▶   ✓  2/8/2025 (6m) 4
# 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.")

(3) Spark Jobs
Partitioning completed successfully.

```

The screenshot shows a Jupyter Notebook interface with three code cells. Cell 1 imports the necessary functions. Cell 2 loads the data from a Hive metastore table into a DataFrame named 'df'. Cell 3 displays the schema of the DataFrame. Cell 4 defines partition columns ('Order\_Date' and 'Product\_Category'), writes the DataFrame back to the metastore using the 'parquet' format, and prints a success message. The notebook is set to Python and has three active Spark jobs.

2/8/2025 (<1s) 5 SQL

```
%sql
SHOW PARTITIONS `hive_metastore`.`default`.`e_commerce_partitioned`;
```

\_sqldf: pyspark.sql.dataframe.DataFrame = [partition: string]

partition
1 Order_Date=2018-01-01/Product_Category=Auto & Accessories
2 Order_Date=2018-01-01/Product_Category=Electronic
3 Order_Date=2018-01-01/Product_Category=Fashion
4 Order_Date=2018-01-01/Product_Category=Home & Furniture
5 Order_Date=2018-01-02/Product_Category=Auto & Accessories
6 Order_Date=2018-01-02/Product_Category=Electronic
7 Order_Date=2018-01-02/Product_Category=Fashion
8 Order_Date=2018-01-02/Product_Category=Home & Furniture
9 Order_Date=2018-01-03/Product_Category=Auto & Accessories
10 Order_Date=2018-01-03/Product_Category=Electronic
11 Order_Date=2018-01-03/Product_Category=Fashion
12 Order_Date=2018-01-03/Product_Category=Home & Furniture
13 Order_Date=2018-01-04/Product_Category=Auto & Accessories
14 Order_Date=2018-01-04/Product_Category=Electronic
15 Order_Date=2018-01-04/Product_Category=Fashion

↓ 1,424 rows | 0.41s runtime Refreshed 5 days ago

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

#### Task 4: A screenshot of the table schema.

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

Just now (<1s) 6

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

```
root
|-- Time: timestamp (nullable = true)
|-- Aging: double (nullable = true)
|-- Customer_Id: long (nullable = true)
|-- Gender: string (nullable = true)
|-- Device_Type: string (nullable = true)
|-- Customer_Login_type: string (nullable = true)
|-- Product: string (nullable = true)
|-- Sales: double (nullable = true)
|-- Quantity: double (nullable = true)
|-- Discount: double (nullable = true)
|-- Profit: double (nullable = true)
|-- Shipping_Cost: double (nullable = true)
|-- Order_Priority: string (nullable = true)
|-- Payment_method: string (nullable = true)
|-- Order_Date: date (nullable = true)
|-- Product_Category: string (nullable = true)
```

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

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

(2) Spark Jobs

```
_sqldf: pyspark.sql.dataframe.DataFrame = [col_name: string, data_type: string ... 1 more field]
```

	col_name	data_type	commer
1	Order_Date	date	null
2	Time	timestamp	null
3	Aging	double	null
4	Customer_Id	bigint	null
5	Gender	string	null
6	Device_Type	string	null
7	Customer_Login_type	string	null
8	Product_Category	string	null
9	Product	string	null
10	Sales	double	null
11	Quantity	double	null
12	Discount	double	null
13	Profit	double	null
14	Shipping_Cost	double	null
15	Order_Priority	string	null

35 rows | 1.94s runtime      Refreshed 5 days ago

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

## 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.

```
%sql
SELECT COUNT(*) AS total_records FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

(2) Spark Jobs

```
_sqldf: pyspark.sql.dataframe.DataFrame = [total_records: long]
```

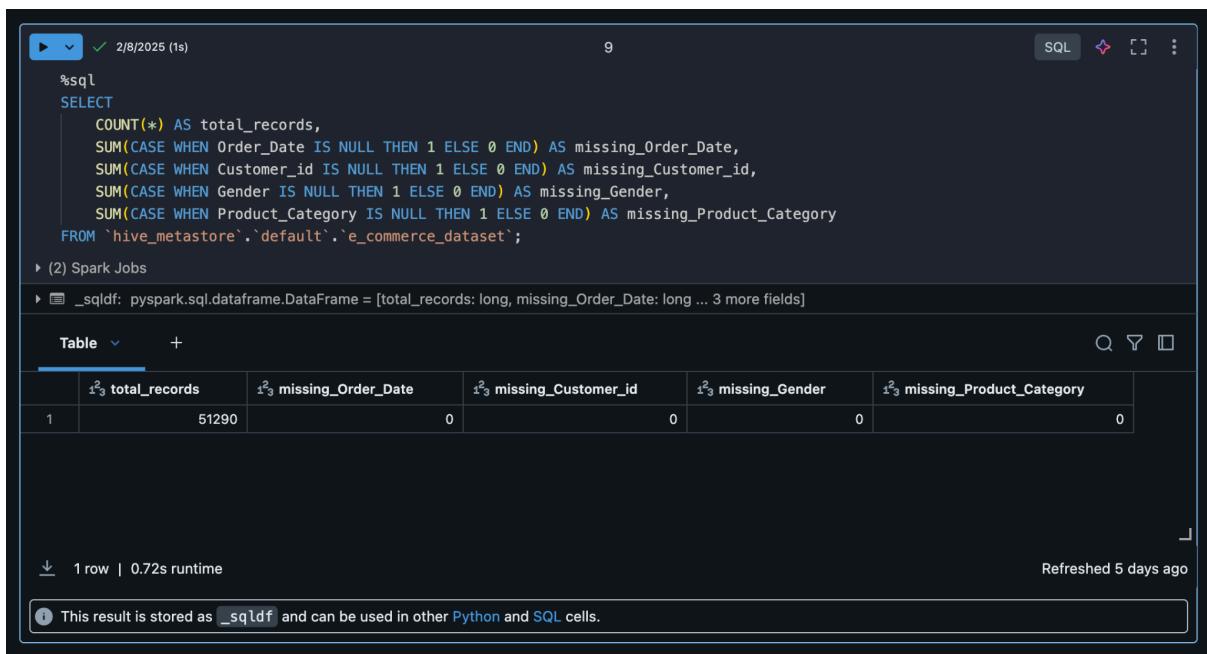
	total_records
1	51290

1 row | 0.63s runtime      Refreshed 5 days ago

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

- 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



```
%sql
SELECT
    COUNT(*) AS total_records,
    SUM(CASE WHEN Order_Date IS NULL THEN 1 ELSE 0 END) AS missing_Order_Date,
    SUM(CASE WHEN Customer_id IS NULL THEN 1 ELSE 0 END) AS missing_Customer_id,
    SUM(CASE WHEN Gender IS NULL THEN 1 ELSE 0 END) AS missing_Gender,
    SUM(CASE WHEN Product_Category IS NULL THEN 1 ELSE 0 END) AS missing_Product_Category
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Table +

	total_records	missing_Order_Date	missing_Customer_id	missing_Gender	missing_Product_Category
1	51290	0	0	0	0

↓ 1 row | 0.72s runtime      Refreshed 5 days ago

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

## 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

▶ 2/8/2025 (1s) 10

```
%sql
SELECT
    MIN(Sales) AS min_sales,
    MAX(Sales) AS max_sales,
    AVG(Sales) AS avg_sales,
    MIN(Profit) AS min_profit,
    MAX(Profit) AS max_profit,
    AVG(Profit) AS avg_profit
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

▶ (2) Spark Jobs

▶ \_sqldf: pyspark.sql.DataFrame = [min\_sales: double, max\_sales: double ... 4 more fields]

	1.2 min_sales	1.2 max_sales	1.2 avg_sales	1.2 min_profit	1.2 max_profit	1.2 avg_profit
1	33	250	152.34087231180175	0.5	167.5	70.40722558003488

↓ 1 row | 0.51s runtime Refreshed 5 days ago

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

### 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**

▶ 2/8/2025 (1s) 11

SQL

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

▶ (2) Spark Jobs

▶ \_sqldf: pyspark.sql.DataFrame = [Device\_Type: string, order\_count: long]

	Device_Type	order_count
1	Web	47632
2	Mobile	3658

↓ 2 rows | 0.79s runtime Refreshed 5 days ago

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

## 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

The screenshot shows a Jupyter Notebook cell with the following content:

```
%sql
SELECT Product_Category, SUM(Quantity) AS total_quantity_sold
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Product_Category
ORDER BY total_quantity_sold DESC
LIMIT 5;
```

Below the code, it says `_sqldf: pyspark.sql.dataframe.DataFrame = [Product_Category: string, total_quantity_sold: double]`.

A table is displayed with the following data:

	Product_Category	total_quantity_sold
1	Fashion	66639
2	Home & Furniture	38190
3	Auto & Accessories	17593
4	Electronic	5951

At the bottom, it says "4 rows | 0.86s runtime" and "Refreshed 5 days ago". A note at the bottom left says "This result is stored as \_sqldf and can be used in other Python and SQL cells."

## 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.

```

▶ ✓ 2/8/2025 (1s) 13
%sql
SELECT Order_Date, SUM(Sales) AS total_sales
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Order_Date
ORDER BY Order_Date ASC;
▶ (2) Spark Jobs
▶ _sqldf: pyspark.sql.dataframe.DataFrame = [Order_Date: date, total_sales: double]

Table + 

```

	Order_Date	total_sales
04	2018-03-06	19200
65	2018-03-07	19933
66	2018-03-08	16061
67	2018-03-09	12673
68	2018-03-10	15631
69	2018-03-11	21445
70	2018-03-12	18402
71	2018-03-13	17836
72	2018-03-14	16252
73	2018-03-15	16007
74	2018-03-16	9976
75	2018-03-17	10344
76	2018-03-18	15879
77	2018-03-19	12405
78	2018-03-20	11833

↓ 356 rows | 0.65s runtime Refreshed 5 days ago

## 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

```

▶ ✓ 2/8/2025 (<1s) 14
%sql
SELECT Product_Category, SUM(Profit) AS total_profit
FROM `hive_metastore`.`default`.`e_commerce_dataset`
GROUP BY Product_Category
ORDER BY total_profit DESC;
▶ (2) Spark Jobs
▶ _sqldf: pyspark.sql.dataframe.DataFrame = [Product_Category: string, total_profit: double]

Table + 

```

	Product_Category	total_profit
1	Fashion	2072623.9000000951
2	Home & Furniture	880058.9000000106
3	Auto & Accessories	48431.1999999963
4	Electronic	174190.60000000143

↓ 4 rows | 0.46s runtime Refreshed 5 days ago

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

## 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

The screenshot shows a Jupyter Notebook cell with the following content:

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

Below the code, the results are displayed in a table:

Order_Priority	order_count
Medium	29433
High	15499
Critical	3932
Low	2424
null	2

Information at the bottom of the cell includes:

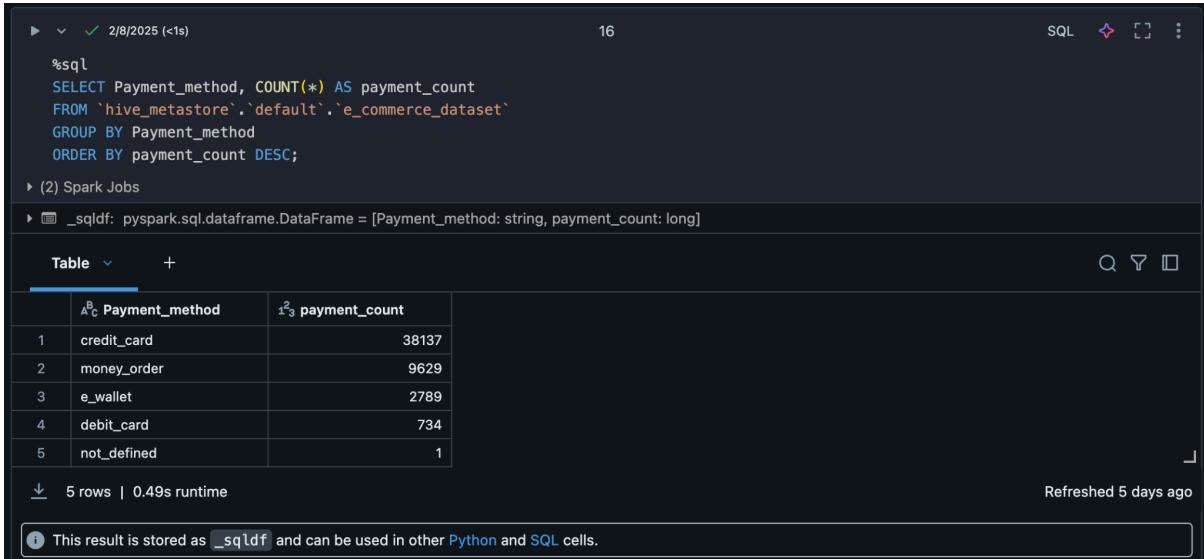
- 5 rows | 0.70s runtime
- Refreshed 5 days ago
- This result is stored as `_sqlldf` and can be used in other Python and SQL cells.

## 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



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

(2) Spark Jobs

\_sqlpdf: pyspark.sql.dataframe.DataFrame = [Payment\_method: string, payment\_count: long]

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

↓ 5 rows | 0.49s runtime      Refreshed 5 days ago

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

## 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.

```

# Load the dataset
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()

```

(2) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Order\_Date: date, Time: timestamp ... 14 more fields]

summary	Sales	Quantity	Discount	Profit	Shipping_cost	Aging
count	51289	51288	51289	51290	51289	51289
mean	152.34087231180175	2.5029831539541414	0.3038214821891464	70.40722558003488	7.041556669071595	5.255035192731385
stddev	66.49541941735221	1.5118586336411253	0.13102661107663202	48.72948829951265	4.871744672835287	2.959948361375533
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

## 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

```

# Load dataset
df = spark.read.table("hive_metastore.default.e_commerce_dataset")

# List of categorical columns
categorical_columns = ["Gender", "Device_Type", "Customer_Login_Type", "Product_Category", "Order_Priority", "Payment_method"]

# Display unique value counts for each categorical column
for col_name in categorical_columns:
    print(f"Distribution for {col_name}:")
    df.groupBy(col_name).count().orderBy("count", ascending=False).show()

```

Distribution for Order\_Priority:

Order_Priority	count
Medium	29433
High	15499
Critical	3932
Low	2424
NULL	2

Distribution for Payment\_method:

Payment_method	count
credit_card	38137
money_order	9629
e_wallet	2789
debit_card	734
not_defined	1

### 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;
```

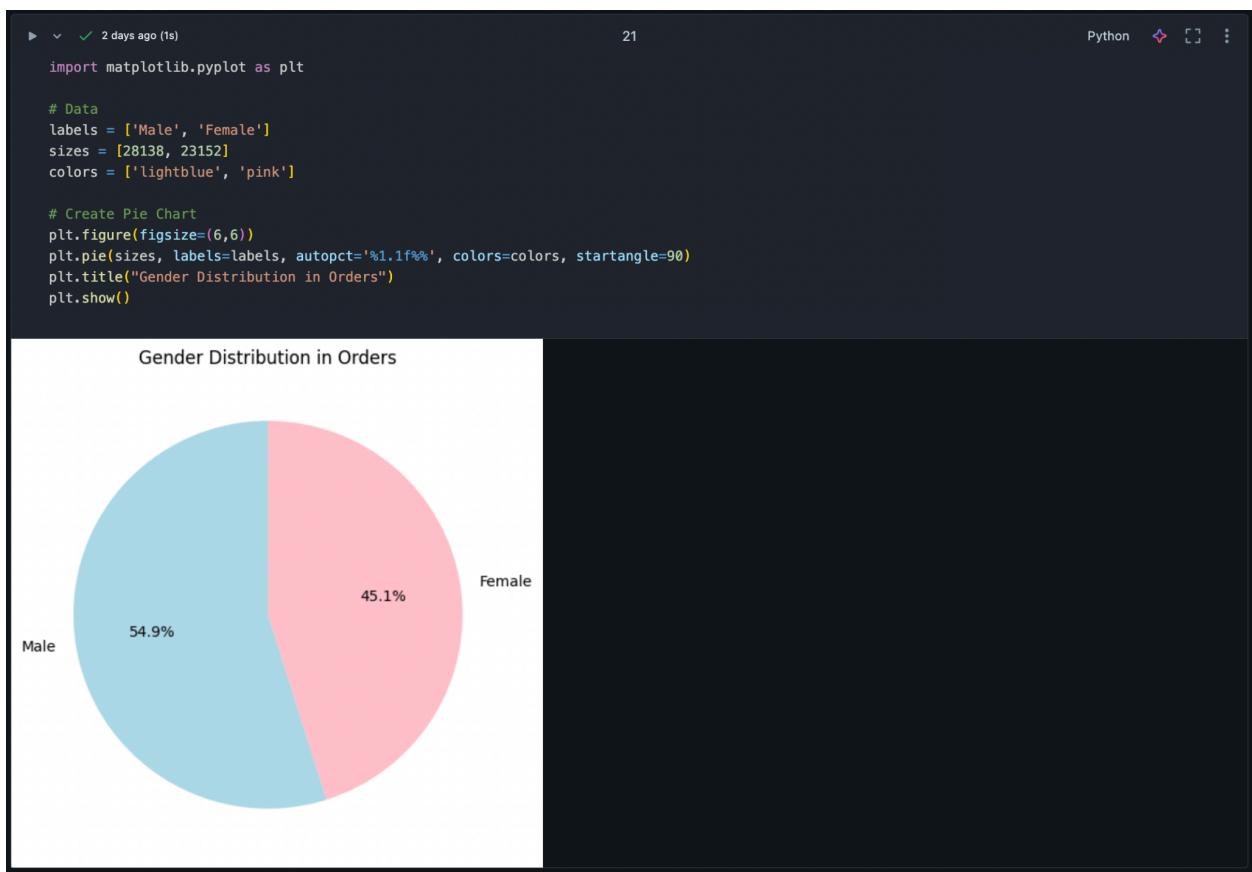
(2) Spark Jobs

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

	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.



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

Device_Type	count
Web	47632
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.



24

```
%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]

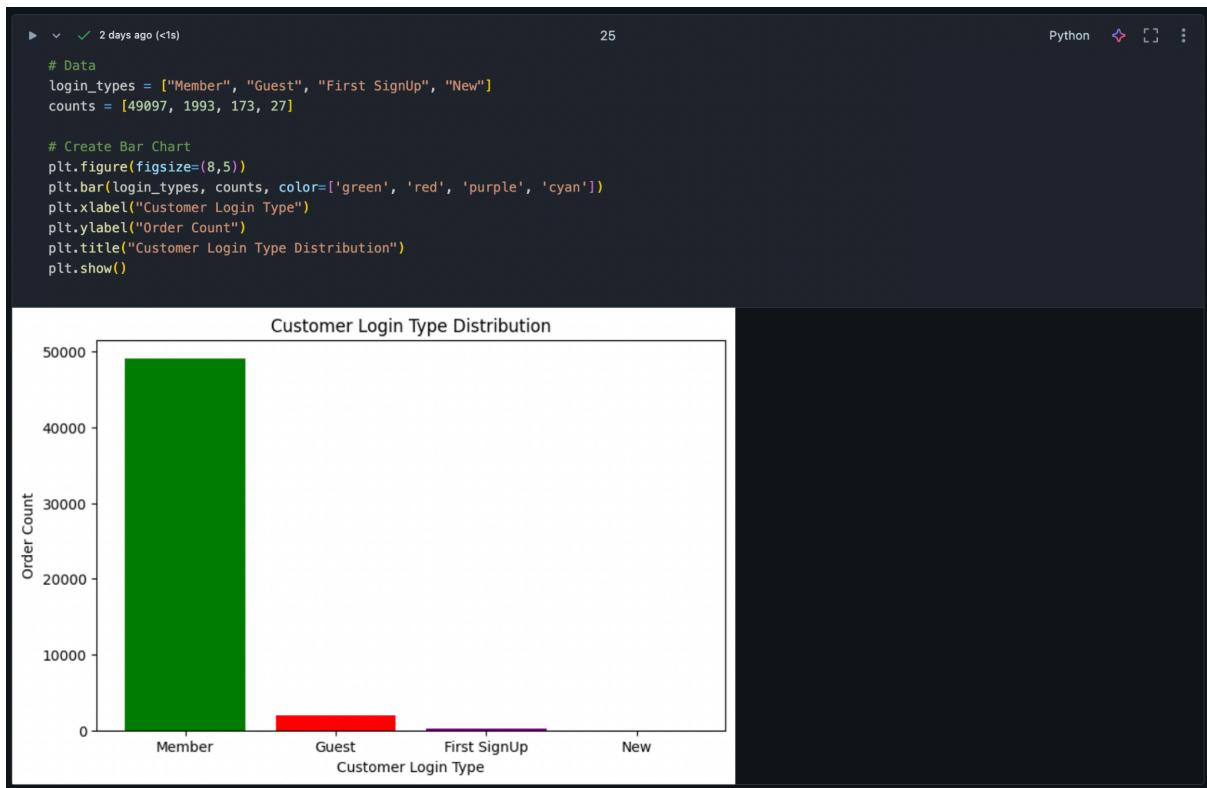
Table +

	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.



26

```
%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
Fashion	25646
Home & Furniture	15438
Auto & Accessories	7505
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

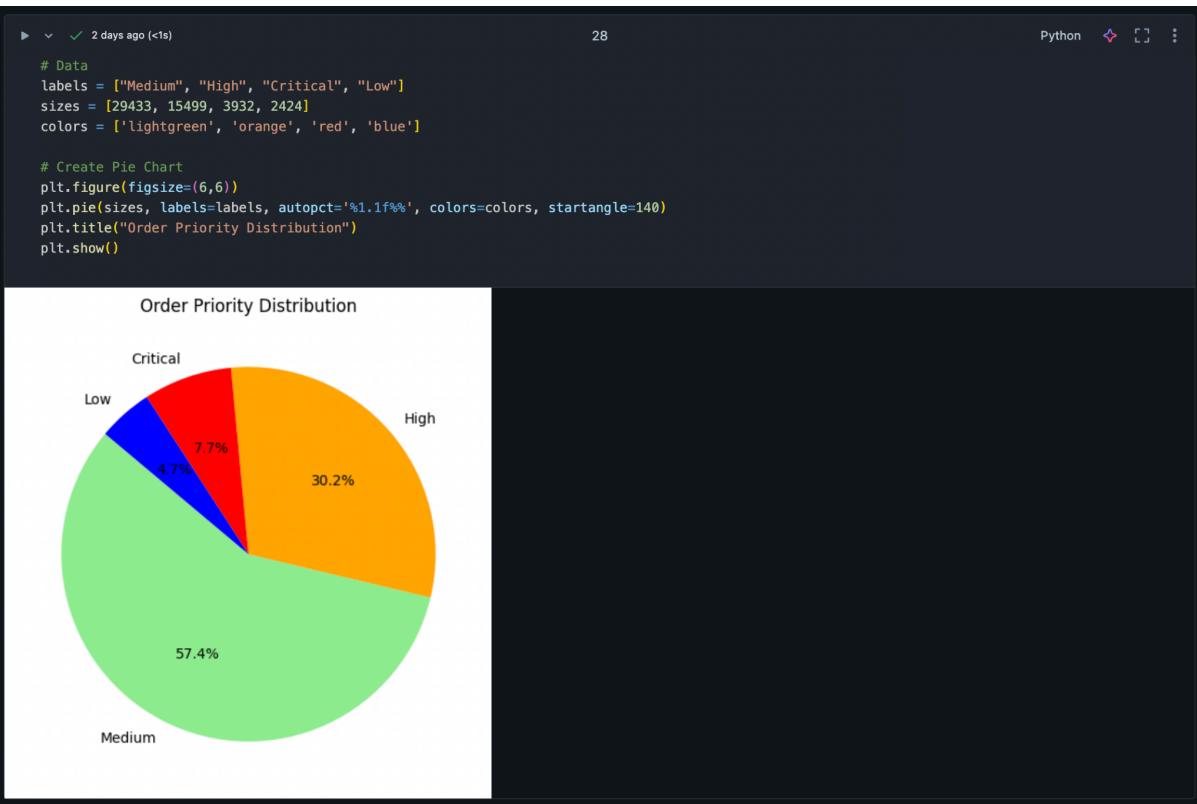
\_sqldf: pyspark.sql.DataFrame = [Order\_Priority: string, count: long]

Table +

Order_Priority	count
Medium	29433
High	15499
Critical	3932
Low	2424
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.



2/8/2025 (<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 = [Payment\_method: string, count: long]

Table +

Payment_method	count
credit_card	38137
money_order	9629
e_wallet	2789
debit_card	734
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.

The screenshot shows a Jupyter Notebook cell with the following content:

```
%sql
SELECT
    SUM(CASE WHEN Order_Date IS NULL THEN 1 ELSE 0 END) AS missing_Order_Date,
    SUM(CASE WHEN Customer_id IS NULL THEN 1 ELSE 0 END) AS missing_Customer_id,
    SUM(CASE WHEN Gender IS NULL THEN 1 ELSE 0 END) AS missing_Gender,
    SUM(CASE WHEN Product_Category IS NULL THEN 1 ELSE 0 END) AS missing_Product_Category,
    SUM(CASE WHEN Sales IS NULL THEN 1 ELSE 0 END) AS missing_Sales
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

Below the code, the result is displayed as a pandas DataFrame:

	missing_Order_Date	missing_Customer_id	missing_Gender	missing_Product_Category	missing_Sales
1	0	0	0	0	1

Information at the bottom of the cell includes:

- 1 row | 0.42s runtime
- Refreshed 7 days ago
- This result is stored as `_sqldf` and can be used in other Python and SQL cells.

## 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| Web| Member|Auto & Accessories| Car & Bike Care|118.0| 1.0|
|0.2| 35.6| 3.6| Critical| credit_card| Web| Member|Auto & Accessories| Tyre|250.0| 1.0|
|2018-12-01|2025-02-08 14:14:28| 5.0| 56296|Female| Web| Member|Auto & Accessories| Bike Tyres| 72.0| 4.0|
|0.3| 140.0| 14.0| High| credit_card| Web| Member|Auto & Accessories| Car Mat| 54.0| 1.0|
|0.1| 18.0| 1.8| Medium| credit_card| Web| Member|Auto & Accessories| Car Seat Covers|114.0| 1.0|
|0.1| 27.0| 2.7| Critical| credit_card| Web| Member|Auto & Accessories| Car Pillow & Neck...|231.0| 5.0|
|2018-03-26|2025-02-08 23:41:31| 4.0| 57057|Female| Web| Member|Auto & Accessories| Car Media Players|140.0| 1.0|
|0.3| 22.6| 2.3| High| credit_card| Web| Member|Auto & Accessories| Car Speakers|211.0| 1.0|
|0.3| 93.3| 9.3| High| credit_card| Web| Member|Auto & Accessories| 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 |
+-----+-----+-----+-----+
| Before Cleaning | 51290 | 1 | 0 |
| 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	1.2 Quantity	1.2 Discount	1.2 Profit	1.2 Shipping_Cost	A <sub>c</sub> Order_Priority	A <sub>c</sub> Payment_Method	Order_Date_Formatted
1	140		1	0.3	46	4.6	Medium	2018-01-02
2	211		1	0.3	112	11.2	Medium	2018-07-24
3	117		5	0.1	31.2	3.1	Critical	2018-11-08
4	118		1	0.3	26.2	2.6	High	2018-04-18
5	250		1	0.3	160	16	Critical	2018-08-13
6	72		1	0.3	24	2.4	Critical	2018-07-09
7	54		1	0.3	54	5.4	High	2018-05-16
8	114		5	0.2	22.6	2.3	Critical	2018-06-23
9	231		5	0.3	116.4	11.6	Critical	2018-07-29
10	140		1	0.2	54.4	5.4	Critical	2018-05-16
11	211		4	0.1	122.6	12.3	Critical	2018-07-13
12	117		4	0.3	18.3	1.8	High	2018-10-07
13	118		1	0.2	35.6	3.6	Critical	2018-07-25
14	250		1	0.3	140	14	High	2018-12-01
15	72		4	0.1	18	1.8	Medium	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]

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	2

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**.

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

(1) Spark Jobs

\_sqlpdf: pyspark.sql.dataframe.DataFrame = [Order\_Date: date, Time: timestamp ... 15 more fields]

	1.2 Sales	1.2 Quantity	1.2 Discount	1.2 Profit	1.2 Shipping_Cost	A <sub>c</sub> Order_Priority	A <sub>c</sub> Payment_method	1.2 Profit_Margin	
1	140	1	0.3	46		4.6	Medium	credit_card	32.857142857142854
2	211	1	0.3	112		11.2	Medium	credit_card	53.08056872037915
3	117	5	0.1	31.2		3.1	Critical	credit_card	26.666666666666668
4	118	1	0.3	26.2		2.6	High	credit_card	22.203389830508474
5	250	1	0.3	160		16	Critical	credit_card	64
6	72	1	0.3	24		2.4	Critical	credit_card	33.333333333333333
7	54	1	0.3	54		5.4	High	credit_card	100
8	114	5	0.2	22.6		2.3	Critical	credit_card	19.824561403508774
9	Rest	231	5	116.4		11.6	Critical	credit_card	50.3896103896104
10	140	1	0.2	54.4		5.4	Critical	money_order	38.857142857142854
11	211	4	0.1	122.6		12.3	Critical	credit_card	58.1042654028436
12	117	4	0.3	18.3		1.8	High	credit_card	15.64102564102564
13	118	1	0.2	35.6		3.6	Critical	credit_card	30.16949152542373
14	250	1	0.3	140		14	High	credit_card	56.000000000000001
15	72	4	0.1	18		1.8	Medium	credit_card	25

↓ ▾ 10,000+ rows | Truncated data | 0.78s runtime

Refreshed 7 days ago

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

## 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**.

```
%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`;
```

(1) Spark Jobs

\_sqldf: pyspark.sql.dataframe.DataFrame = [Order\_Date: date, Time: timestamp ... 15 more fields]

	1.2 Sales	1.2 Quantity	1.2 Discount	1.2 Profit	1.2 Shipping_Cost	A <sup>B</sup> Order_Priority	A <sup>B</sup> Payment_method	A <sup>B</sup> Sales_Category
1	140	1	0.3	46		4.6	Medium	Medium
2	211	1	0.3	112		11.2	Medium	High
3	117	5	0.1	31.2		3.1	Critical	Medium
4	118	1	0.3	26.2		2.6	High	Medium
5	250	1	0.3	160		16	Critical	High
6	72	1	0.3	24		2.4	Critical	Medium
7	54	1	0.3	54		5.4	High	Medium
8	114	5	0.2	22.6		2.3	Critical	Medium
9	231	5	0.3	116.4		11.6	Critical	High
10	140	1	0.2	54.4		5.4	Critical	Medium
11	211	4	0.1	122.6		12.3	Critical	High
12	117	4	0.3	18.3		1.8	High	Medium
13	118	1	0.2	35.6		3.6	Critical	Medium
14	250	1	0.3	140		14	High	High
15	72	4	0.1	18		1.8	Medium	Medium

10,000+ rows | Truncated data | 0.59s runtime

Refreshed 7 days ago

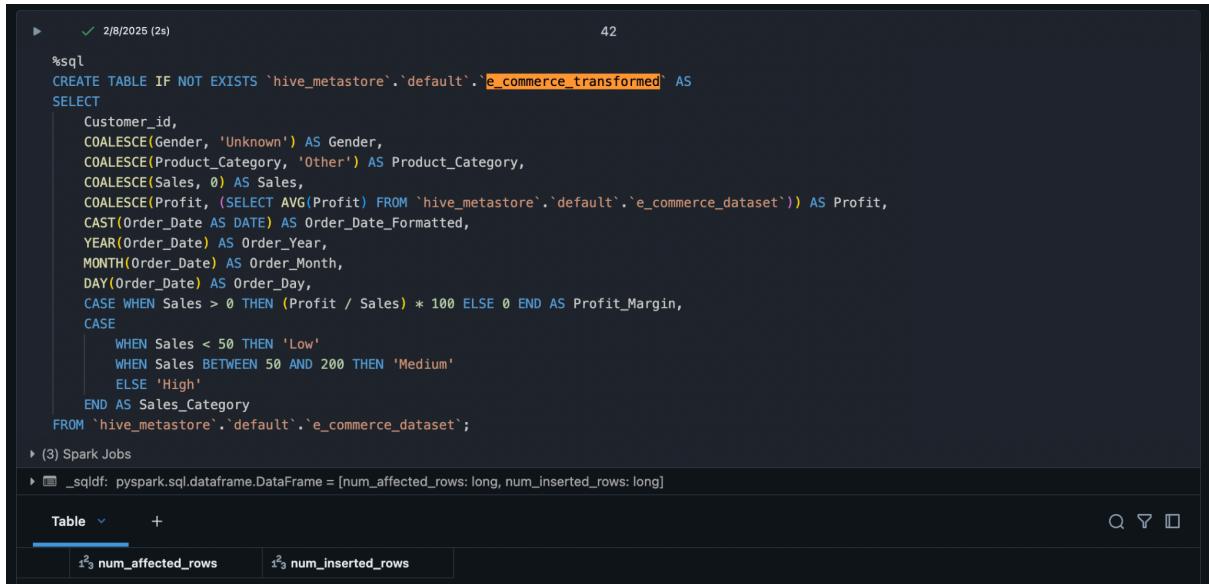
This result is stored as \_sqldf and can be used in other Python and SQL cells.

## 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)**.



```
%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_Catgeory
FROM `hive_metastore`.`default`.`e_commerce_dataset`;
```

(3) Spark Jobs

\_sqldf: pyspark.sql.dataframe.DataFrame = [num\_affected\_rows: long, num\_inserted\_rows: long]

Table	+ _sqldf	Q ▾ □
num_affected_rows	num_inserted_rows	

## 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 = [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 = [State: string, Total\_Records: long]

Table +

A <sup>b</sup> c State	A <sup>b</sup> c 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.

The screenshot shows a Jupyter Notebook cell with the following content:

```
%python
from pyspark.sql.functions import col, to_date

df = df.withColumn("Order_Date_Standardized", to_date(col("Order_Date"), "yyyy-MM-dd"))

# Verify
df.select("Order_Date", "Order_Date_Standardized").show()
```

Output:

▶ (2) Spark Jobs

▶ df: pyspark.sql.dataframe.DataFrame = [Time: timestamp, Aging: double ... 15 more fields]

Order_Date	Order_Date_Standardized
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24
2018-04-24	2018-04-24

### 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

<b>Order_Priority</b>	Order_High, Order_Medium, Order_Low, Order_Critical	Encodes different priority levels
-----------------------	--	-----------------------------------

```

06:19 PM (1s) 50
CASE WHEN Payment_Method = 'e_Wallet' THEN 1 ELSE 0 END AS Payment_eWallet,
-- Order Priority Encoding
CASE WHEN Order_Priority = 'High' THEN 1 ELSE 0 END AS Order_High,
CASE WHEN Order_Priority = 'Medium' THEN 1 ELSE 0 END AS Order_Medium,
CASE WHEN Order_Priority = 'Low' THEN 1 ELSE 0 END AS Order_Low,
CASE WHEN Order_Priority = 'Critical' THEN 1 ELSE 0 END AS Order_Critical
FROM `hive_metastore`.`default`.`e_commerce_dataset`;

```

Generate (36 + 1)

(1) Spark Jobs

\_sqldf: pyspark.sql.dataframe.DataFrame = [Customer\_Id: long, Order\_Date: date ... 11 more fields]

Table +

Customer_Id	Order_Date	Gender_Male	Gender_Female	Device_Web	Device_Mobile	Payment_CreditCard	...
26754	2018-11-26	1	0	1	0	1	...
34365	2018-07-30	1	0	1	0	1	...
49745	2018-03-22	1	0	1	0	1	...
53213	2018-09-06	1	0	1	0	1	...
43735	2018-05-09	1	0	1	0	1	...
14760	2018-09-20	1	0	1	0	1	...
44410	2018-01-16	1	0	1	0	1	...
53552	2018-03-08	1	0	1	0	1	...
39696	2018-11-26	1	0	1	0	1	...
22235	2018-08-27	1	0	1	0	0	...
42599	2018-05-11	1	0	1	0	1	...
27079	2018-06-22	0	1	1	0	1	...
57745	2018-05-01	1	0	1	0	1	...
20917	2018-02-19	1	0	1	0	1	...
54446	2018-09-11	1	0	1	0	1	...

10,000+ rows | Truncated data | 0.60s runtime

Refreshed 7 minutes ago

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

### 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:

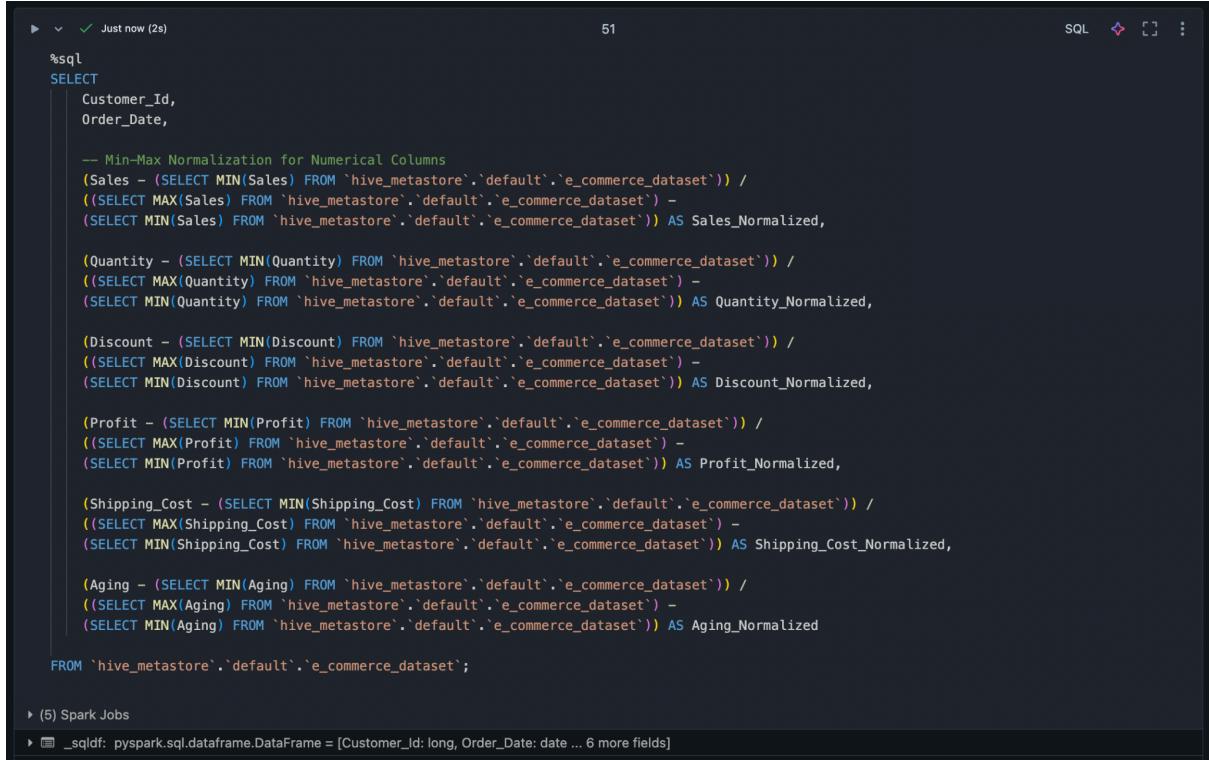
Column Name	Description
Sales	Total sales per transaction
Quantity	Number of items purchased
Discount	Discount percentage applied
Profit	Profit from the transaction
Shipping_Cost	Cost of shipping
Aging	Number of days taken for delivery

## 2. SQL Approach: Min-Max Normalization

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

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

## 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`;
```

The screenshot shows a Jupyter Notebook cell with a SQL cell magic (%sql). The code performs min-max normalization on various numerical columns (Sales, Quantity, Discount, Profit, Shipping\_Cost, Aging) from the e\_commerce\_dataset table in the default database of the hive metastore. The normalized values are stored as new columns (Sales\_Normalized, Quantity\_Normalized, etc.). The cell output shows the creation of a DataFrame named \_sqlldf with 7 columns: Customer\_Id, Order\_Date, Sales\_Normalized, Quantity\_Normalized, Discount\_Normalized, Profit\_Normalized, and Shipping\_Cost\_Normalized.

Table +

	<code>i3 Customer_Id</code>	<code>Order_Date</code>	<code>1.2 Sales_Normalized</code>	<code>1.2 Quantity_Normalized</code>	<code>1.2 Discount_Normalized</code>	<code>1.2 Profit_Normalized</code>	<code>1.2 Shipping</code>
1	37077	2018-01-02	0.4930875576036866	0	0.4999999999999994	0.27245508982035926	
2	59173	2018-07-24	0.820276497695825	0	0.4999999999999994	0.6676646706586826	
3	41066	2018-11-08	0.3870967741935484	1	0	0.18385233532934132	
4	50741	2018-04-18	0.391705069124424	0	0.4999999999999994	0.15389221556886226	
5	53639	2018-08-13	1	0	0.4999999999999994	0.9550898203592815	
6	39783	2018-07-09	0.17972350230414746	0	0.4999999999999994	0.1407185628742515	
7	26767	2018-05-16	0.0967741935483871	0	0.4999999999999994	0.3203592814371258	
8	20719	2018-06-23	0.37327188940092165	1	0.25	0.13233532934131736	
9	46947	2018-07-29	0.9124423963133641	1	0.4999999999999994	0.6940119760479042	
10	31839	2018-05-16	0.4930875576036866	0	0.25	0.32275449101796405	
11	22249	2018-07-13	0.820276497695825	0.75	0	0.7311377245508982	
12	15109	2018-10-07	0.3870967741935484	0.75	0.4999999999999994	0.10658682634730539	
13	18622	2018-07-25	0.391705069124424	0	0.25	0.21017964071856288	
14	56296	2018-12-01	1	0	0.4999999999999994	0.8353293413173652	
15	34138	2018-10-07	0.17972350230414746	0.75	0	0.10479041916167664	

↓ ▾ 10,000+ rows | Truncated data | 192s runtime Refreshed 1 minute ago

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