**Hexaware Technologies**

PySpark & SparkSQL -coding

**Explain ETL (Extract, Transform, Load) with PySpark**

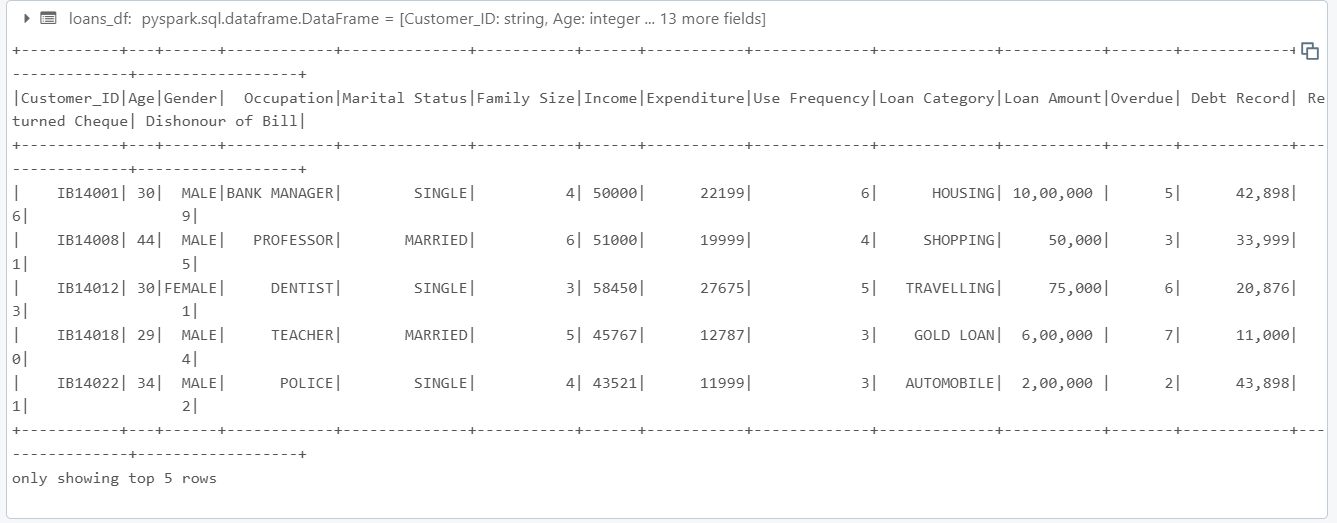
ETL stands for **Extract, Transform, Load**, which is a process used to move and manipulate data from one system to another, making it ready for analysis. PySpark is a powerful tool in the Apache Spark ecosystem, commonly used to handle big data, and it’s very effective for performing ETL tasks. Here's how each part of ETL works with PySpark:

1. **Extract**: The first step is to extract data from various sources, like databases, CSV files, or cloud storage. PySpark makes it easy to load this data into a DataFrame, which is a structured representation of the data. You can use **spark.read** to load data in different formats such as CSV, Parquet, or JSON. This is the stage where you gather raw data from different sources to start working with it.
2. **Transform**: Once the data is extracted, you need to clean and manipulate it to make it useful. This transformation process can involve filtering out unnecessary data, changing data formats, or adding new calculated fields. PySpark provides many built-in functions like **filter(), select(), groupBy(),** and **withColumn()** that help you adjust the data as needed.This step turns the raw data into something structured and ready for analysis.
3. **Load**:After transforming the data, the final step is to load it into a target storage system, such as a relational database, data warehouse, or cloud storage like Amazon S3 or Google Cloud Storage. In PySpark, you can use **DataFrame.write** methods to load the data into various formats like CSV, or directly into a database using JDBC connections. Once the data is loaded, it becomes available for analysis, reporting, or further processing.

**Step 1: Import Libraries and Create Spark Session**

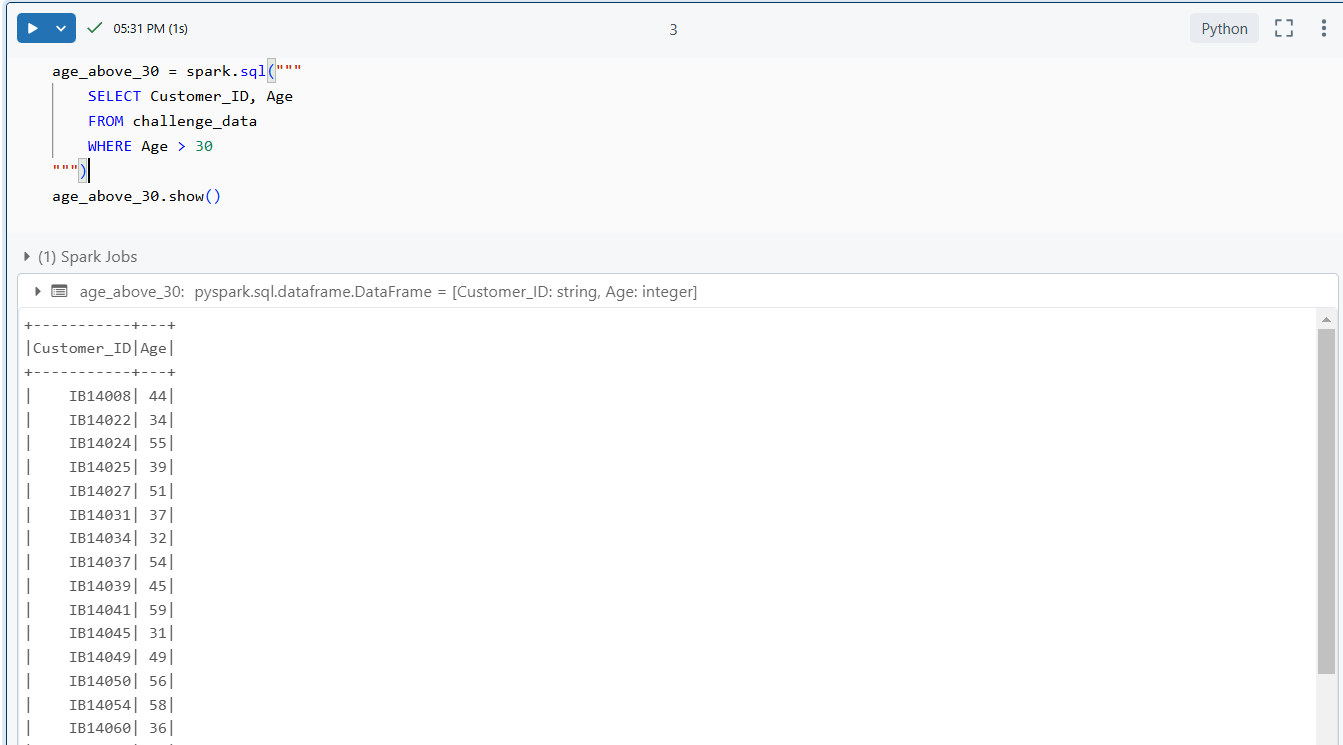
In this step, we imported the necessary libraries from PySpark, specifically **SparkSession** from **pyspark.sql**, which is essential for initializing a Spark session. A Spark session allows us to interact with Spark through its DataFrame API. We then created a Spark session using **SparkSession.builder.appName("Loan Data Processing").getOrCreate()** to initialize the Spark environment with the name "Loan Data Processing". After that, we defined the file path for the **loan-1.csv** dataset and loaded it into a DataFrame using the **spark.read.format("csv")** method, ensuring that the first row is treated as headers and the schema is inferred automatically





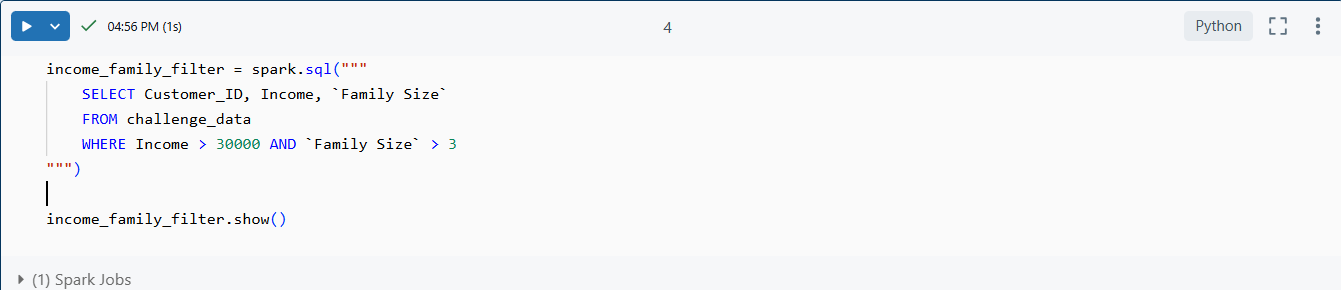
**1. Find all customers who are above 30 years of age.**

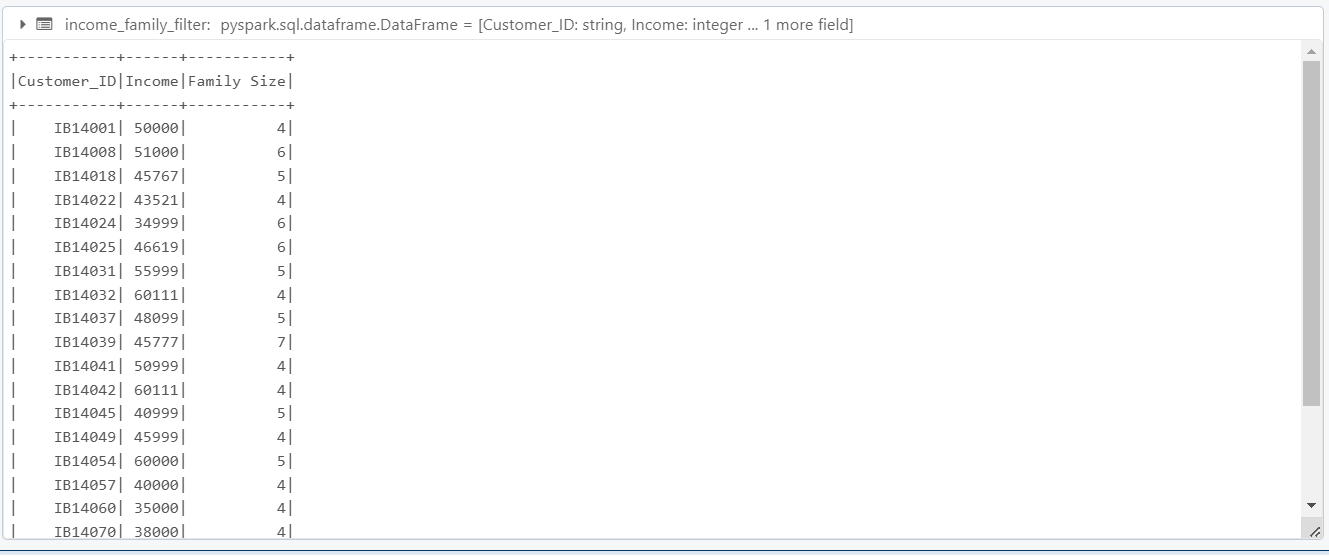
In this query, we used SparkSQL to filter and retrieve the details of customers who are older than 30. By selecting the **Customer\_ID** and **Age** columns from the dataset and applying a condition in the **WHERE** clause **(Age > 30),** we filtered out customers under the age of 30.



**2. Find all customers whose Income is greater than 30,000 and Family Size is greater than**

For this query, we used SparkSQL to filter customers based on both their income and family size. The condition **Income** > **30000** and **Family Size > 3** ensures that only customers with a high income and a larger family size are selected.



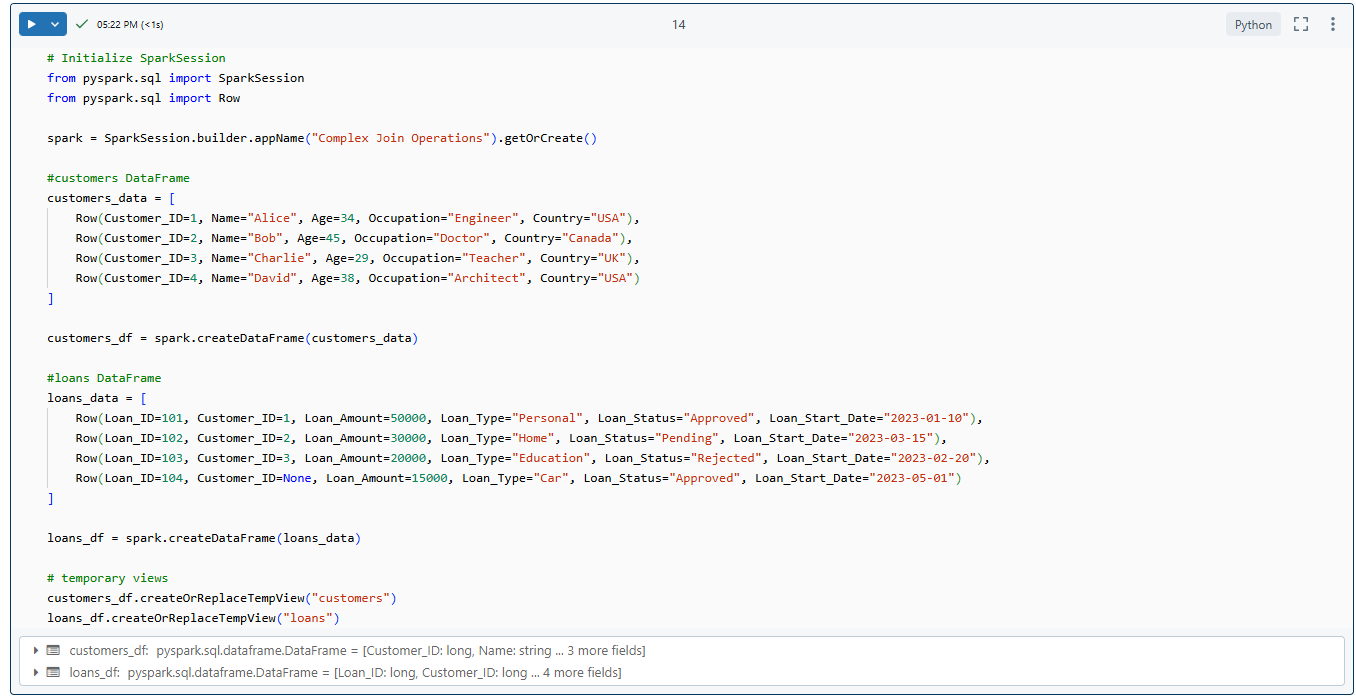


**Join Operations**

**Creating DataFrames and Views**

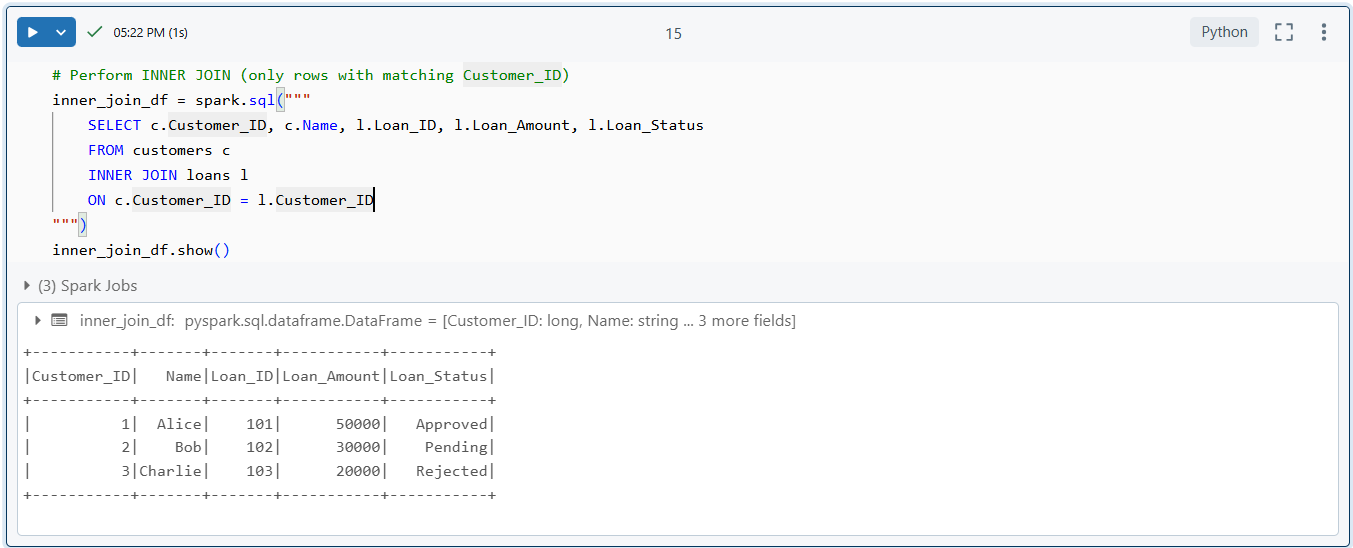
In this section, we created two DataFrames: **customers\_df** and **loans\_df**. These DataFrames represent customer information and loan details, respectively. We then registered them as temporary views using the **createOrReplaceTempView** method, which allows us to perform SQL queries on them within SparkSQL.

* **Customers Data:** Contains details like **Customer\_ID, Name, Age, Occupation**, and **Country.**
* **Loans Data:** Contains **Loan\_ID, Customer\_ID, Loan\_Amount, Loan\_Type, Loan\_Status**, and **Loan\_Start\_Date**.

****

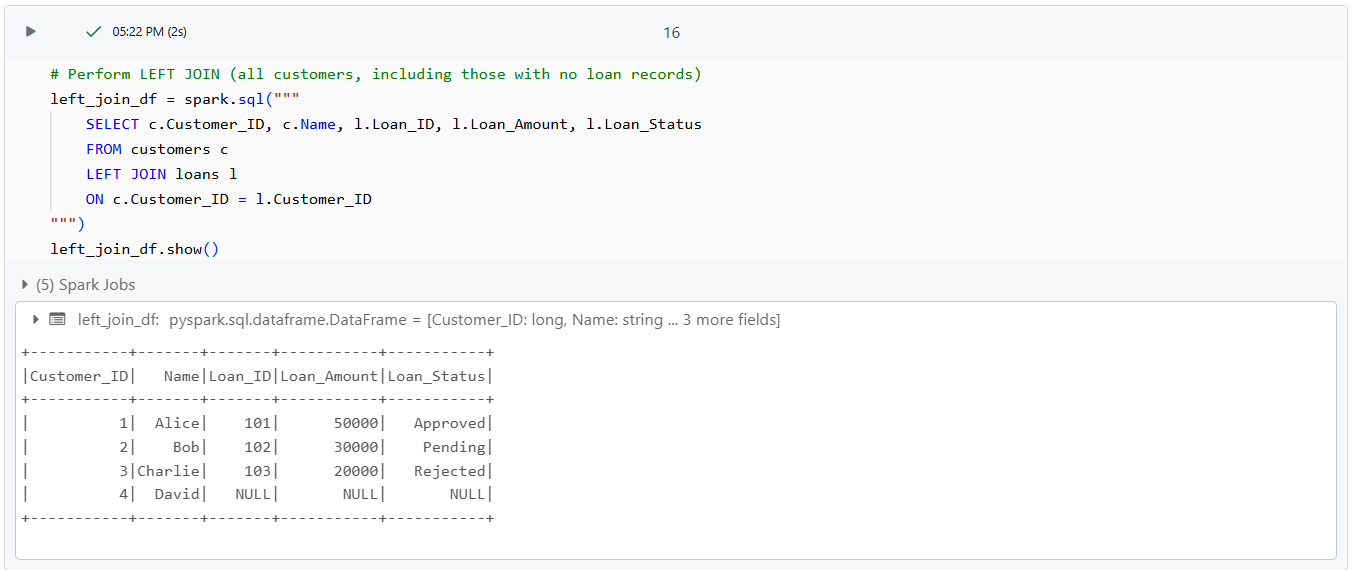
**1. INNER JOIN**

The **INNER JOIN** operation retrieves only the rows where there is a match between the **Customer\_ID** in the **customers** and **loans** DataFrames. In this case, customers who have a loan associated with them are selected. The query joins the two tables on **Customer\_ID** and selects the **Customer\_ID, Name, Loan\_ID, Loan\_Amount, and Loan\_Status** for each matching record.



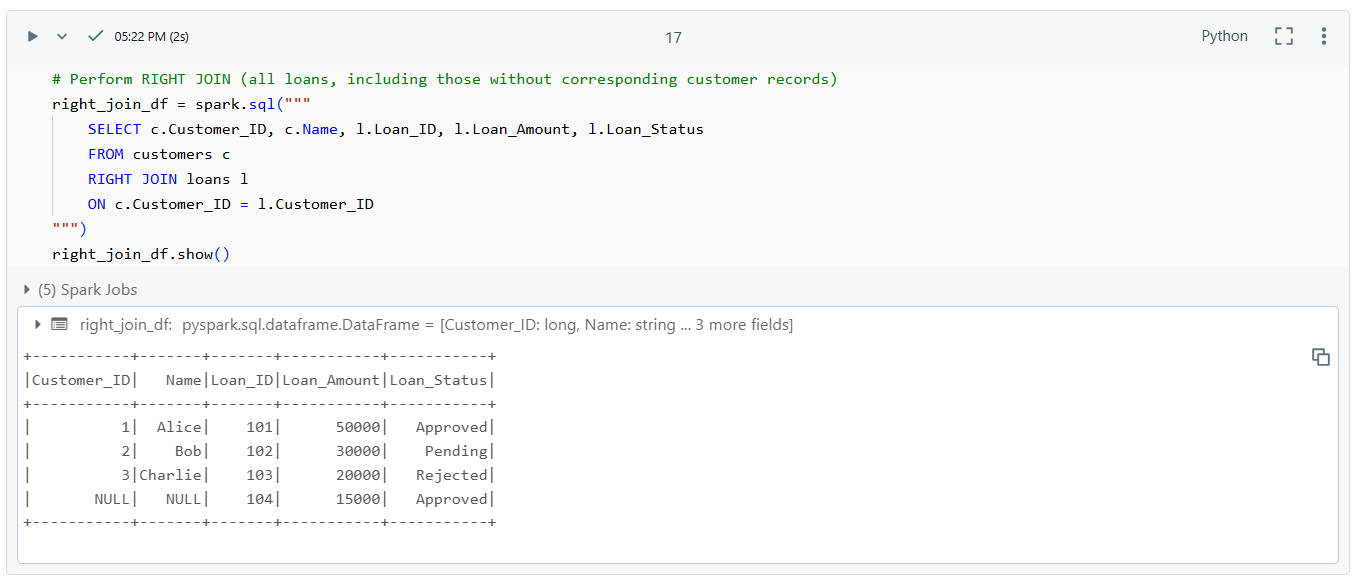
**2. LEFT JOIN**

The **LEFT JOIN** operation returns all records from the left table (**customers**) and the matching records from the right table (**loans**). If there is no match, **NULL** values are returned for the columns from the right table. In this case, all customers are selected, including those who do not have an associated loan record.



**3. RIGHT JOIN**

The **RIGHT JOIN** operation returns all records from the right table (**loans**) and the matching records from the left table (**customers**). If there is no match, **NULL** values are returned for the columns from the left table. In this case, all loans are selected, including those that do not have an associated customer record.



**Simple Aggregations**

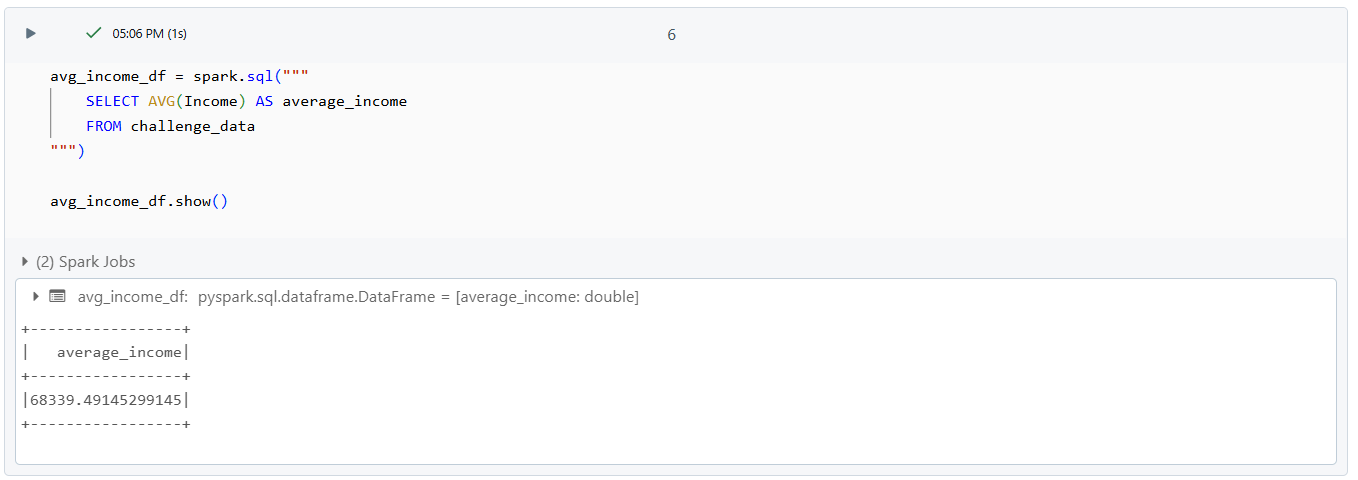
**1. Count Unique Occupations**

In this query, we used the **COUNT(DISTINCT Occupation)** function to find the number of unique occupations in the **challenge\_data** dataset. The **COUNT** function is commonly used to count the number of rows, and the **DISTINCT** keyword ensures that only unique occupation values are considered in the count.



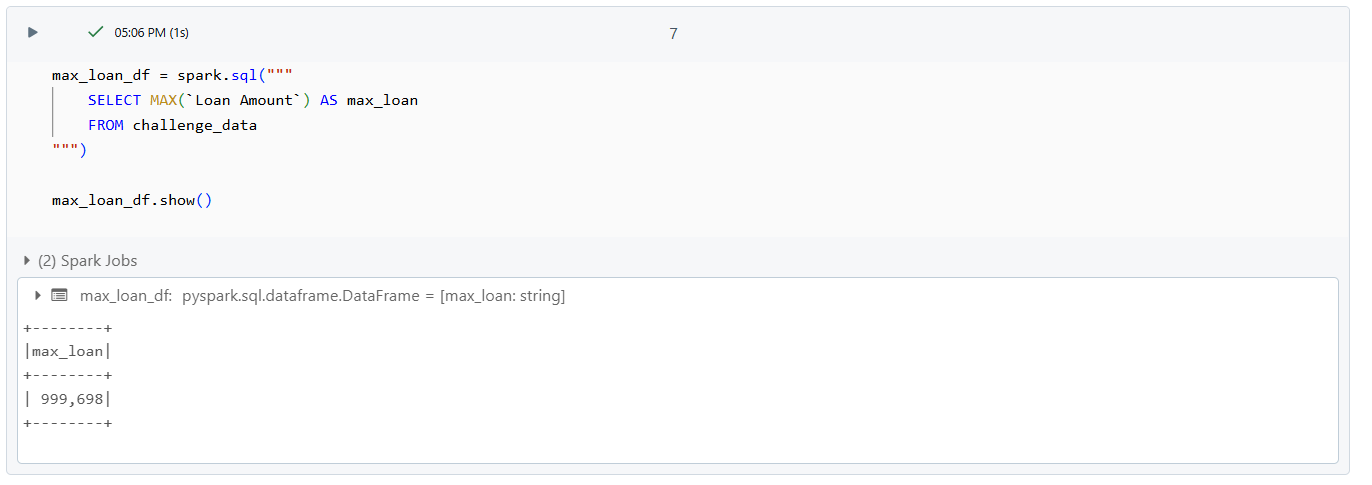
**2. Calculate the Average Income**

To calculate the **average income** of the customers, we used the **AVG(Income)** function. This function computes the mean value of the **Income** column across all records in the dataset.



**3. Find the Maximum and Minimum Loan Amounts**

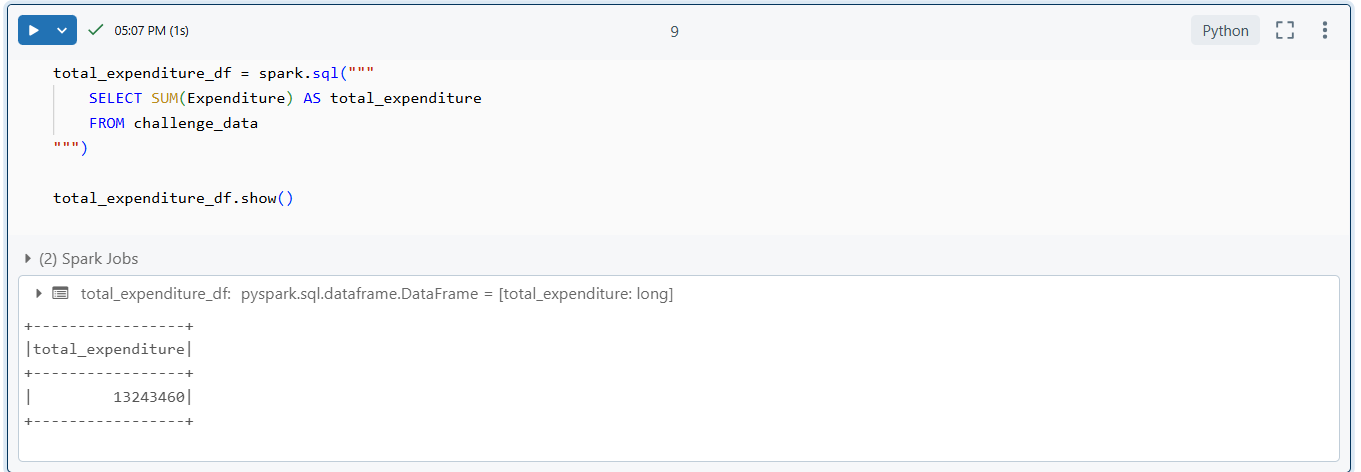
For the **maximum loan amount**, we used the **MAX(Loan Amount)** function, which returns the largest loan value from the Loan Amount column. Similarly, the **minimum loan amount** was calculated using the **MIN(Loan Amount)** function, which returns the smallest loan value.





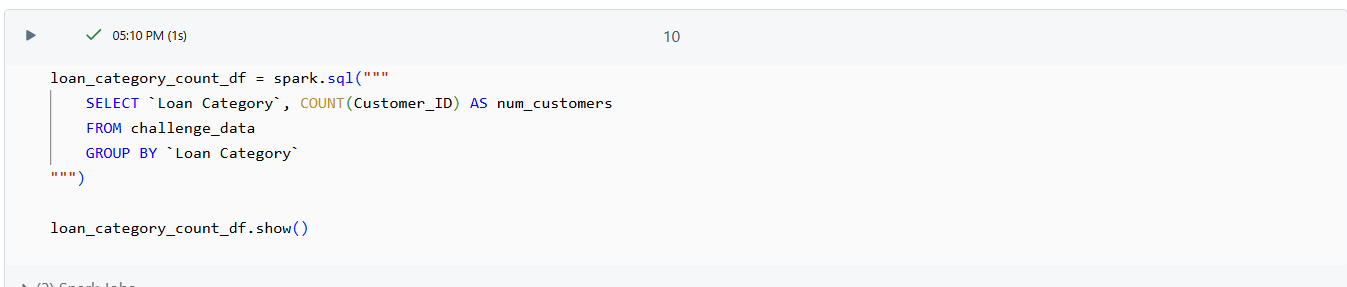
**4. Sum of Expenditure**

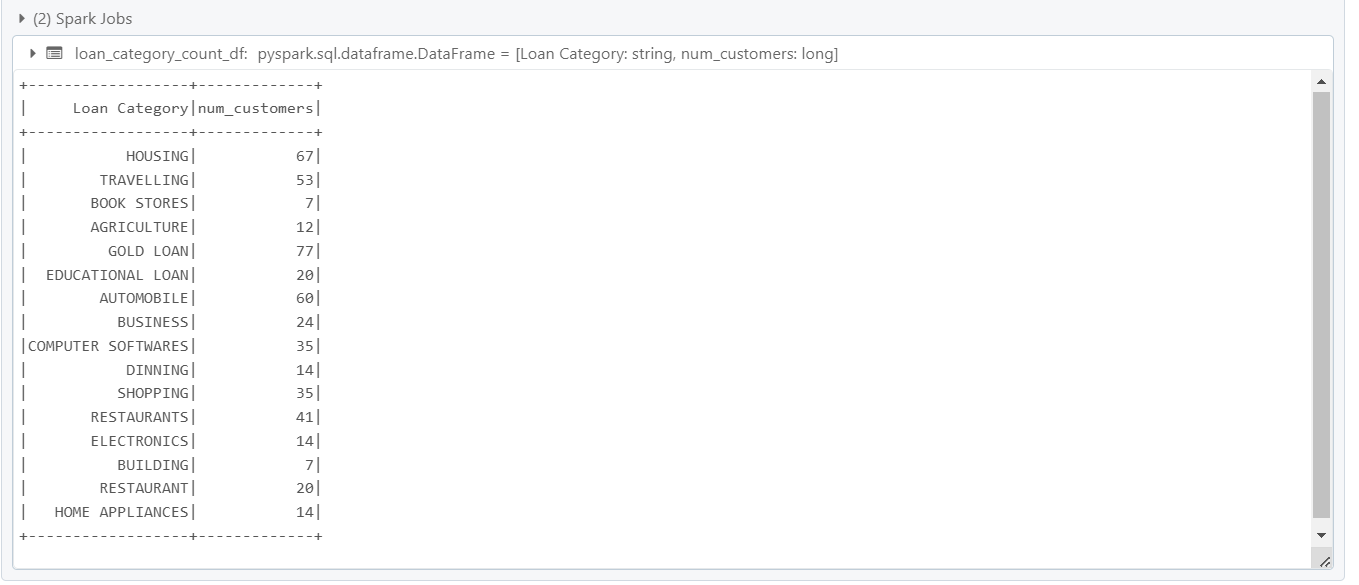
The **SUM(Expenditure)** function was used to calculate the total expenditure of all customers by summing the values in the Expenditure column.



**GroupBy Operations  
  
1. Count the Number of Customers in Each Loan Category**

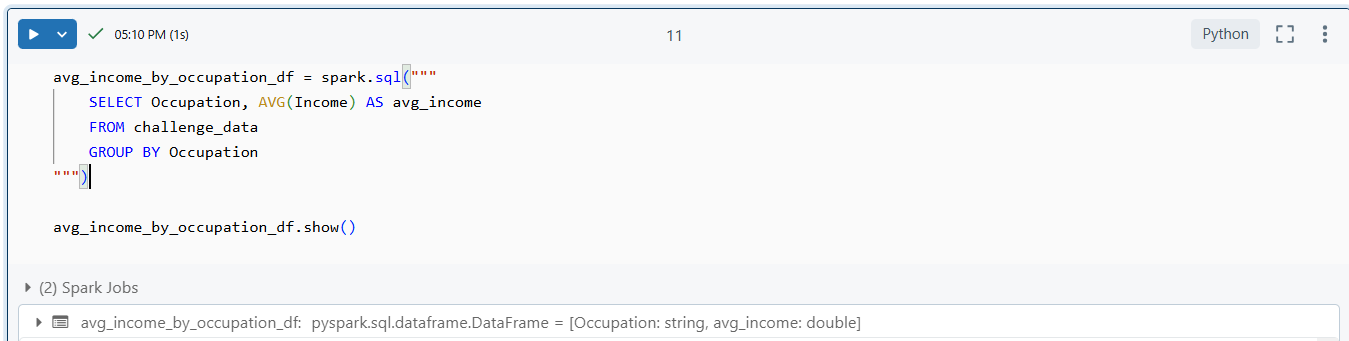
In this query, we used the **GROUP BY** clause to group the data by the **Loan Category** column. Then, we applied the **COUNT(Customer\_ID)** function to count the number of customers in each loan category.

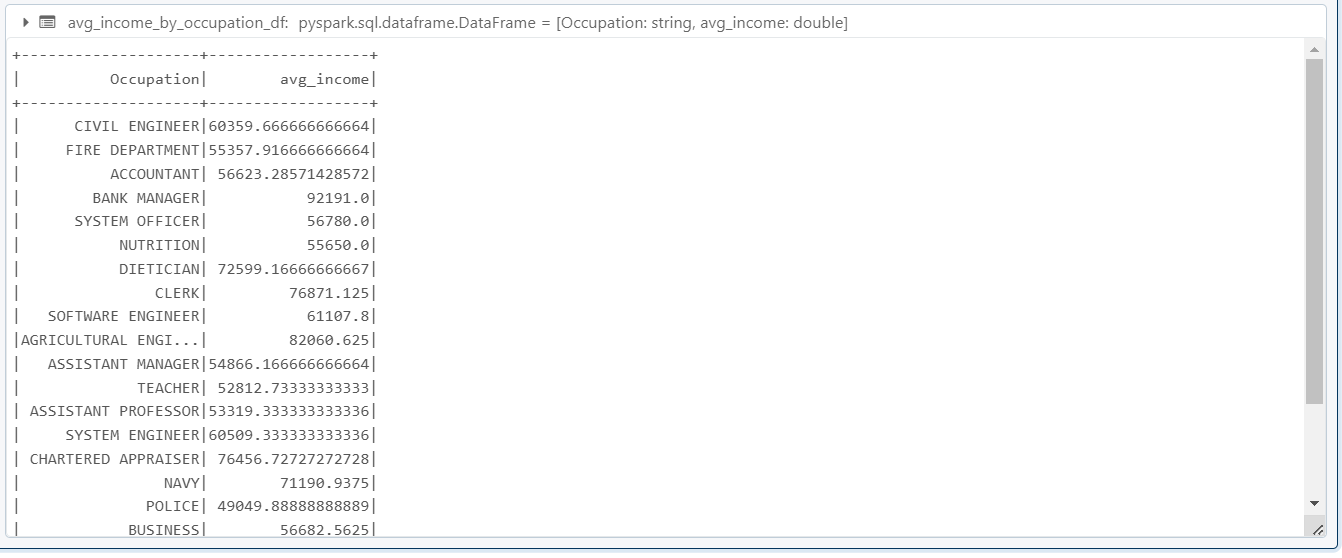




**2. Calculate the Average Income for Each Occupation**

For this query, we used the **GROUP BY** clause to group the dataset by the **Occupation** column. We then used the **AVG(Income)** function to calculate the average income for each occupation.





**3. Find the Maximum and Minimum Loan Amount for Each Marital Status**

In this query, we grouped the data by **Marital Status** and applied the **MAX(Loan Amount**) and **MIN(Loan Amount)** functions to find the highest and lowest loan amounts for each marital status group.

