**Hexaware Technologies**

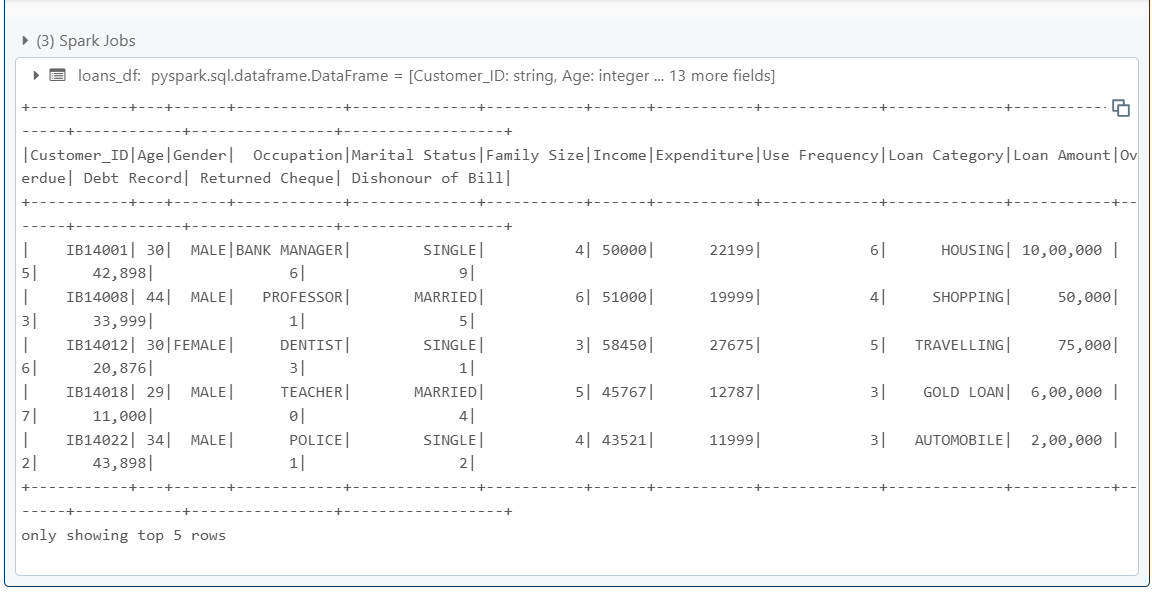
PySpark-Case Study

**loandata.csv file**

**Step 1: Importing Required Libraries**

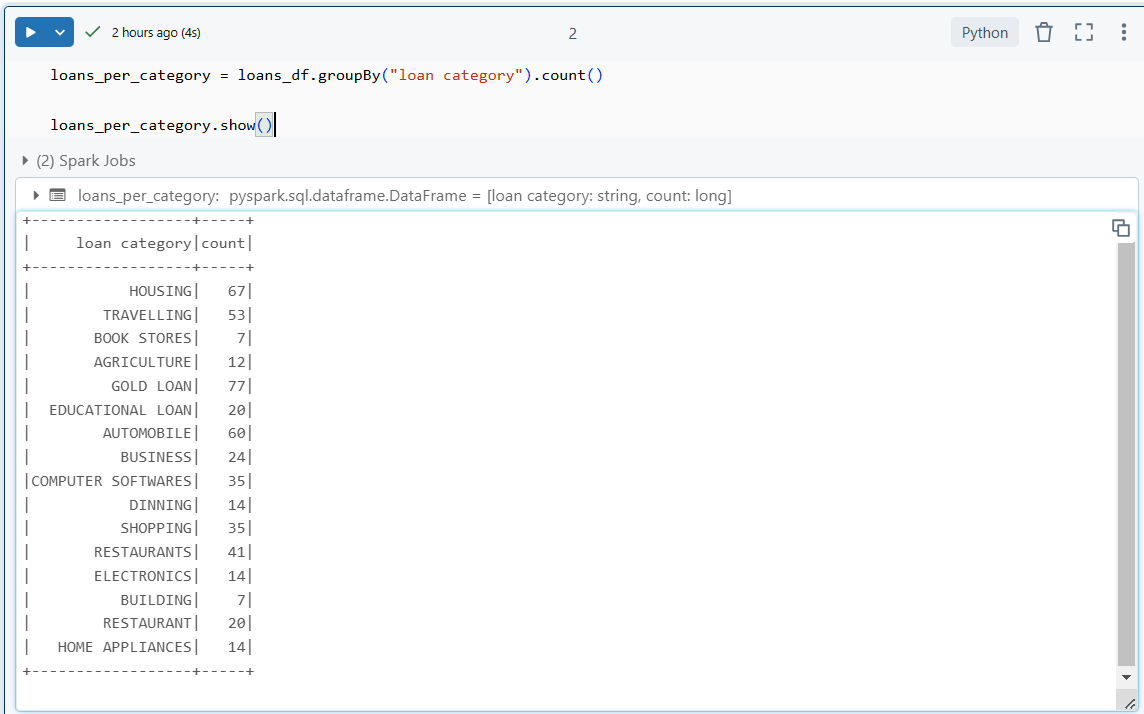
In this step, we imported the **SparkSession** from the **pyspark.sql** module, which is essential for working with Spark. We initialized a **SparkSession** using **.builder** to configure and create the session. The **appName()** method was used to set a name for the application, and **getOrCreate()** ensures that we either create a new session or retrieve the existing one. After setting up the session, we defined the file path for the dataset and used the **spark.read.format("csv")** method to load the CSV file into a DataFrame. We specified options to treat the first row as column headers **(option("header", "true"))** and automatically infer data types **(option("inferSchema", "true"))**. Finally, we used **.show(5)** to display the first 5 rows of the DataFrame, verifying that the data was loaded correctly and ready for analysis.





1. **Number of loans in each category**

We used the DataFrame API to calculate the number of loans in each category. We grouped the data by the **Loan Category** column using the **groupBy()** method and then applied the **count()** function to count the number of rows (loans) in each category.



1. **Number of people who have taken more than 1 lack loan**

We first cleaned the **Loan Amount** column by removing any non-numeric characters using **regexp\_replace()** and then converted the column to an integer using the **.cast("int")** method. This step ensures that the **Loan Amount** is in a proper numeric format for comparison. Next, we filtered the rows where the Loan Amount is greater than 1 Lakh (100,000) using the **filter()** function. Finally, we used the **distinct()** method on the **Customer\_ID** column to count the number of unique customers who have taken loans above this threshold.



1. **Number of people with income greater than 60000 rupees**

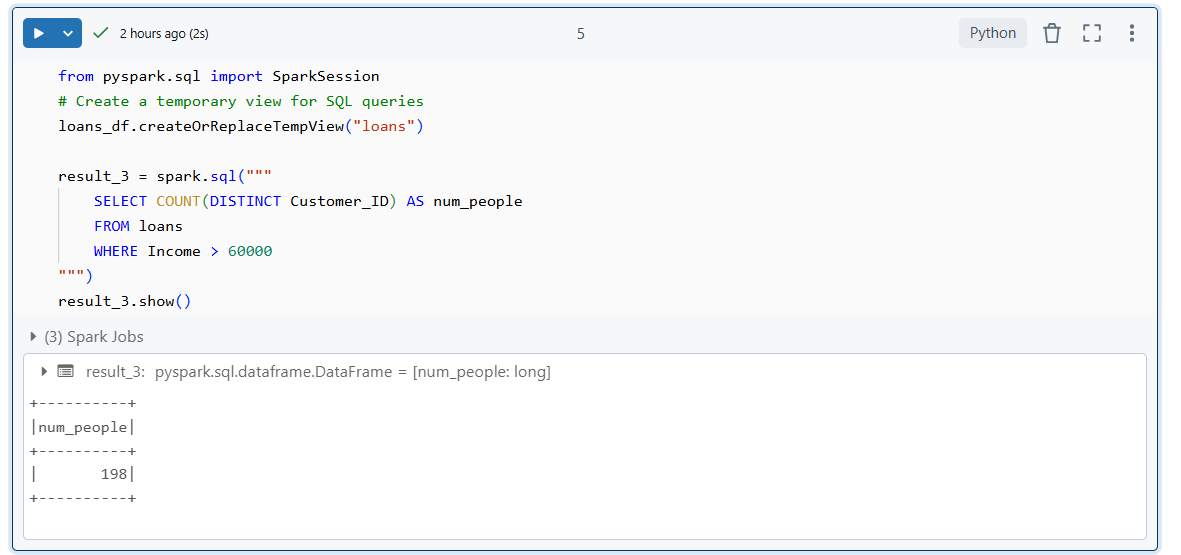
**Without SQL:**

We filter the DataFrame for rows where the **Income** column is greater than 60,000 using the **filter()** function. After filtering, we count the number of distinct **Customer\_ID** values using **select("Customer\_ID").distinct().count()**. This gives the number of unique customers whose income is above the specified threshold



**With SQL:**

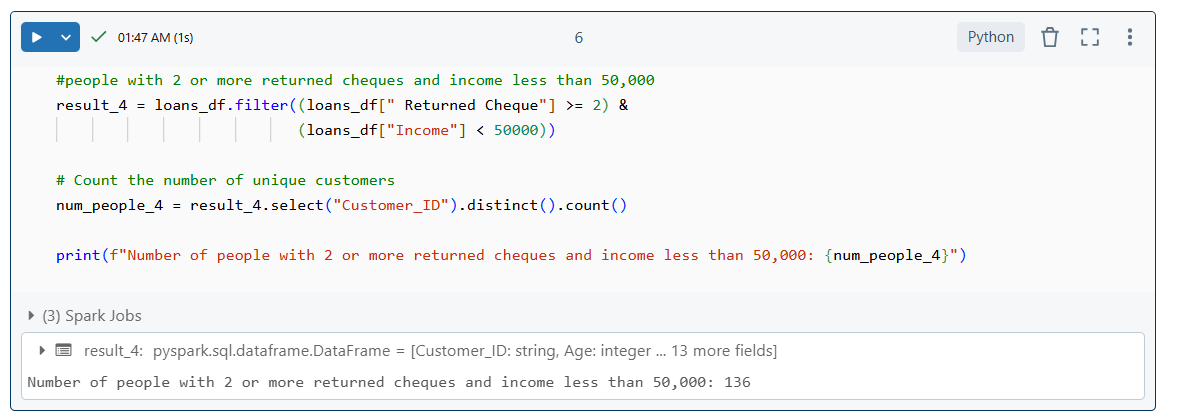
In SQL-based approach, we first create a temporary view of the DataFrame using **createOrReplaceTempView()**, which makes it accessible for SQL queries. We then write a SQL query that counts the distinct **Customer\_ID** values from the **loans** table where the **Income** is greater than 60,000. The **SELECT COUNT(DISTINCT Customer\_ID)** statement performs the counting operation.



1. **Number of people with 2 or more returned cheques and income less than 50000**

**Without SQL:**

We apply the **filter()** function to select rows where the **Returned Cheque** column is greater than or equal to 2 and the **Income** column is less than 50,000. We combine these conditions using the **&** operator for an AND condition. After filtering the data, we use **select("Customer\_ID").distinct().count()** to count the unique customers that meet these criteria.



**With SQL:**

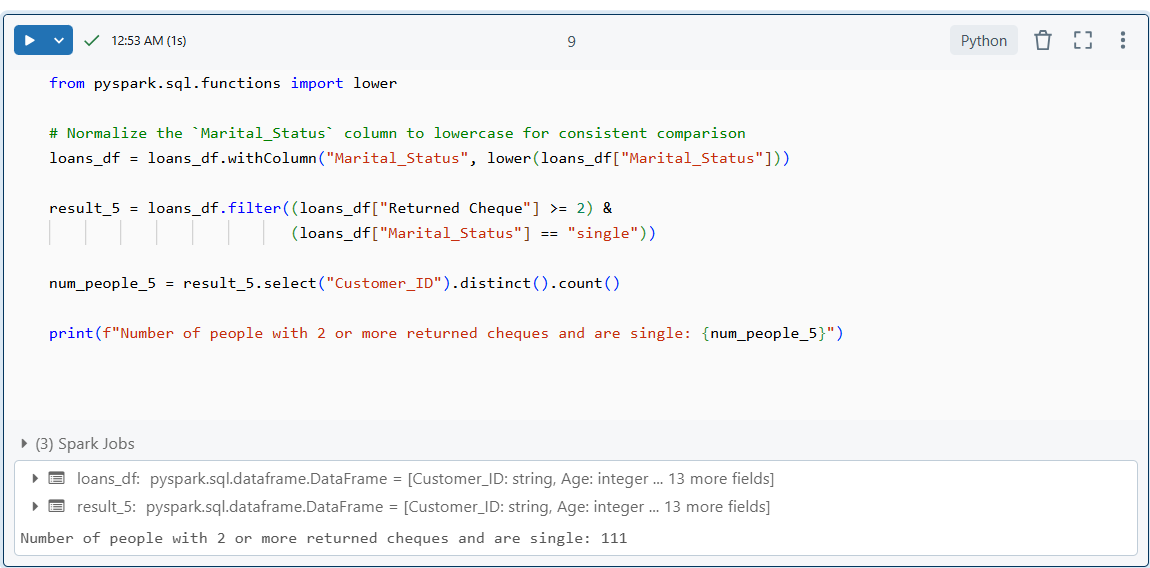
In SQL-based approach, we use the **spark.sql()** method to write a query that selects the count of distinct **Customer\_ID** values from the loans table. The **WHERE** clause filters for customers who have 2 or more returned cheques (using **Returned Cheque >= 2**) and whose income is less than 50,000.



1. Number of people with 2 or more returned cheques and are single

**Without SQL:**

we first handled the column name issues by renaming the columns with extra spaces using **.withColumnRenamed().** We then normalized the **Marital\_Status** column to lowercase using **lower()** from **pyspark.sql.functions**. This step ensures consistency in the comparison (since **single** might appear as **Single** in some rows). After that, we applied a filter to select records where the number of returned cheques is greater than or equal to 2 and the **Marital\_Status** is single. Finally, we counted the distinct **Customer\_ID** values that meet these conditions.



**With SQL:**

In SQL method, we created a temporary view of the DataFrame using **createOrReplaceTempView().** Then, we used the **spark.sql()** method to write a query that counts distinct **Customer\_ID** values. The **WHERE** clause filters for records where **Returned Cheque >= 2** and **Marital\_Status** equals **'single'**. To handle case-insensitivity, we used the **LOWER()** function within the query to ensure the comparison is consistent.



1. **Number of people with expenditure over 50000 a month**

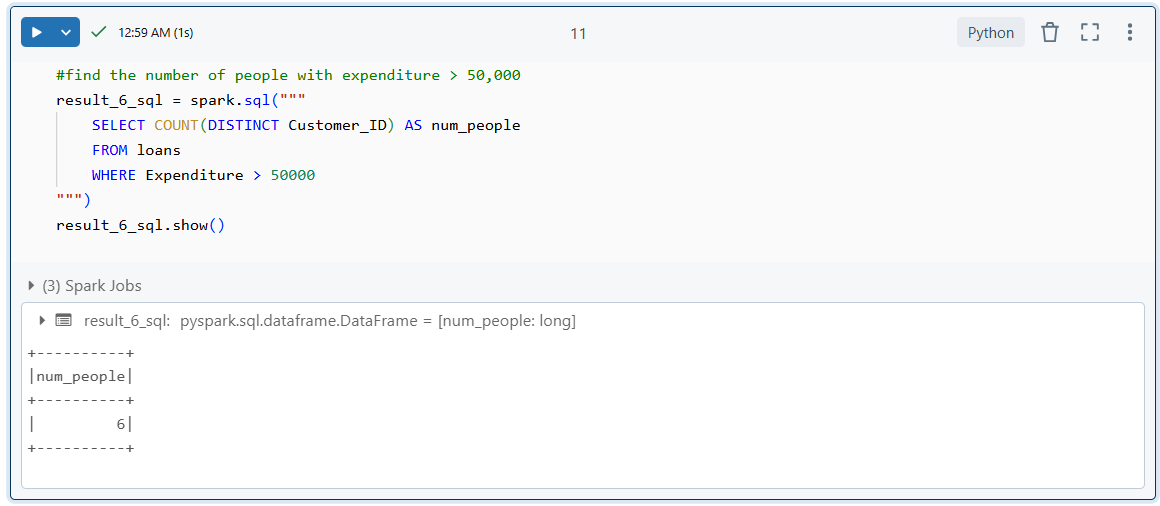
**Without SQL:**

We used the DataFrame API to filter the records where the **Expenditure** column is greater than 50,000 using the **filter()** function. After filtering, we counted the distinct **Customer\_ID** values using **select("Customer\_ID").distinct().count().**



**With SQL:**

In SQL approach, we created a temporary view of the DataFrame with **createOrReplaceTempView().** We then used **spark.sql()** to run a SQL query that counts the distinct **Customer\_ID** values from the **loans** table, where the **Expenditure** is greater than 50,000.



1. **Number of members who are elgible for credit card**

***Note:*** *Since there are no records with Overdue = 0, we are using Overdue = 1 as the condition to identify eligible members.*

**Without SQL:**

We filtered the DataFrame to find members who meet the eligibility criteria for a credit card. Specifically, the **Income** must be greater than 30,000 and **Overdue** must be 1. After applying the filter, we counted the distinct **Customer\_ID** values using **select("Customer\_ID").distinct().count()** to get the number of eligible members.



**With SQL:**

In SQL-based approach, we created a temporary view of the DataFrame and used **spark.sql()** to write a query that counts the distinct **Customer\_ID** values from the **loans** table where the Income is greater than 30,000 and the **Overdue** is 1. The SQL query allows for a straightforward, declarative approach to filtering and aggregation, and using SQL makes it easier to express conditions like **Overdue = 1** clearly.

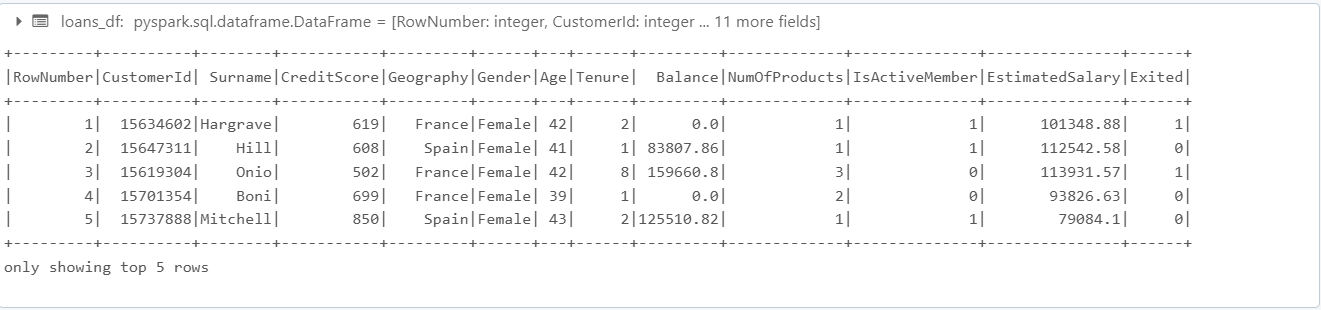


**credit.csv file**

**Step 1: Loading the CSV File into a DataFrame**

Before we analyze the data, we first need to load it into a PySpark DataFrame. We use the **spark.read.csv()** function to load the CSV file into a DataFrame. The **header=True** option ensures that the first row of the file is used as the column headers, and **inferSchema=True** automatically infers the data types for each column. This allows us to correctly handle numerical and categorical data for further analysis.

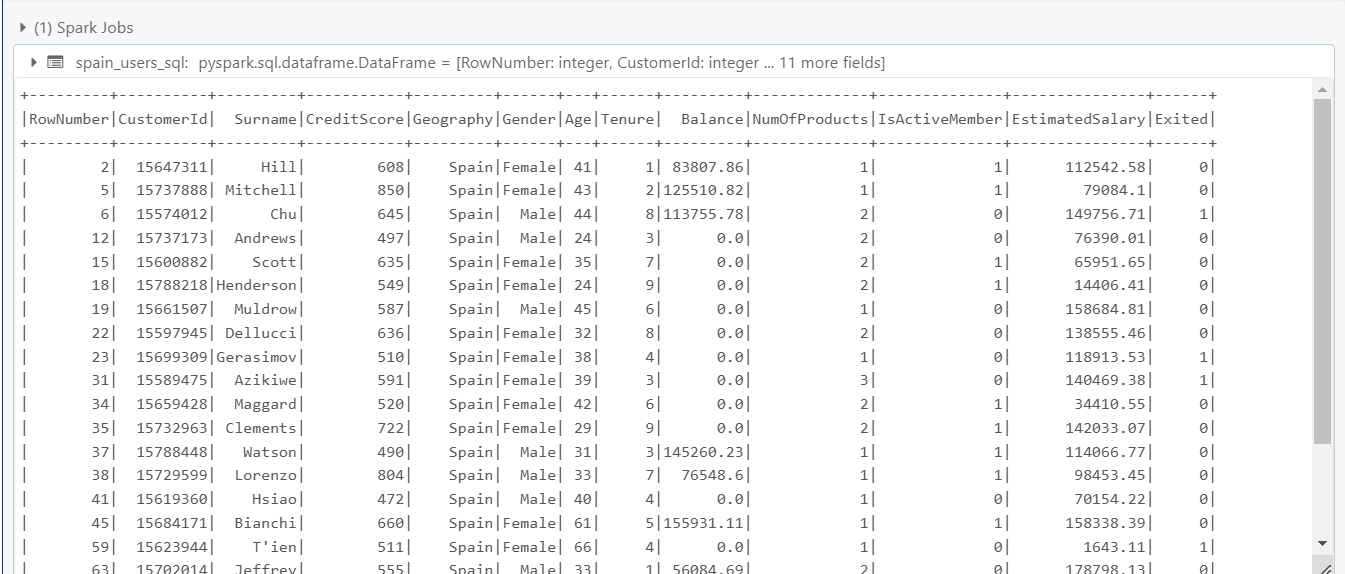
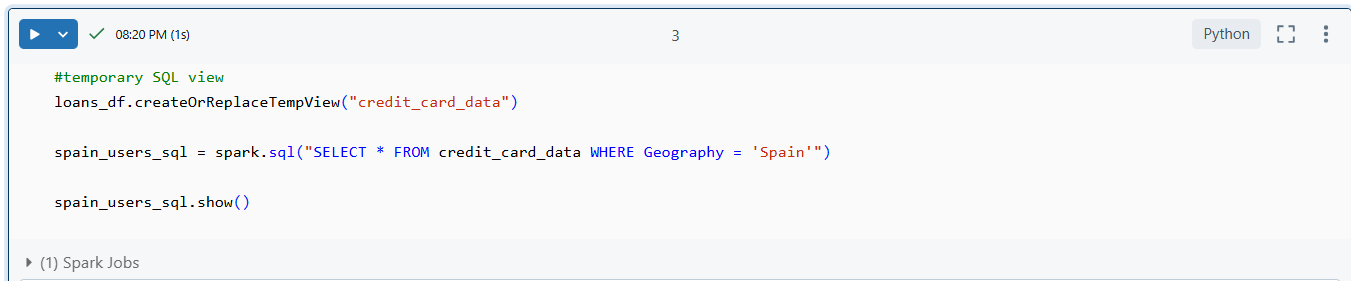




1. **credit card users in Spain**

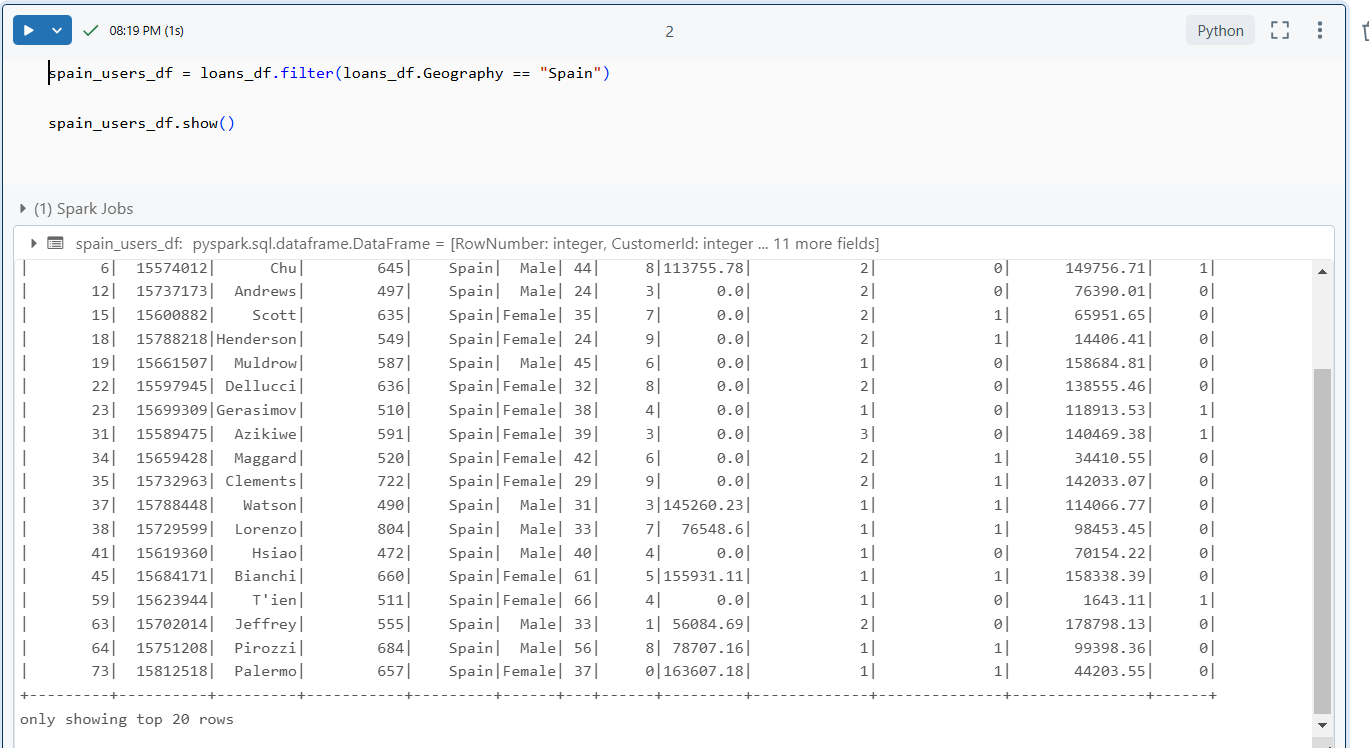
**with SQL:**

To find the credit card users in Spain using SQL, we first create a temporary SQL view of the DataFrame using the **createOrReplaceTempView()** function. This allows us to interact with the DataFrame using SQL queries. Once the view is created, we write a simple SQL query: **SELECT \* FROM credit\_card\_data WHERE Geography = 'Spain'.** This query filters the dataset to only include users whose **Geography** column is set to "Spain."



**without SQL:**

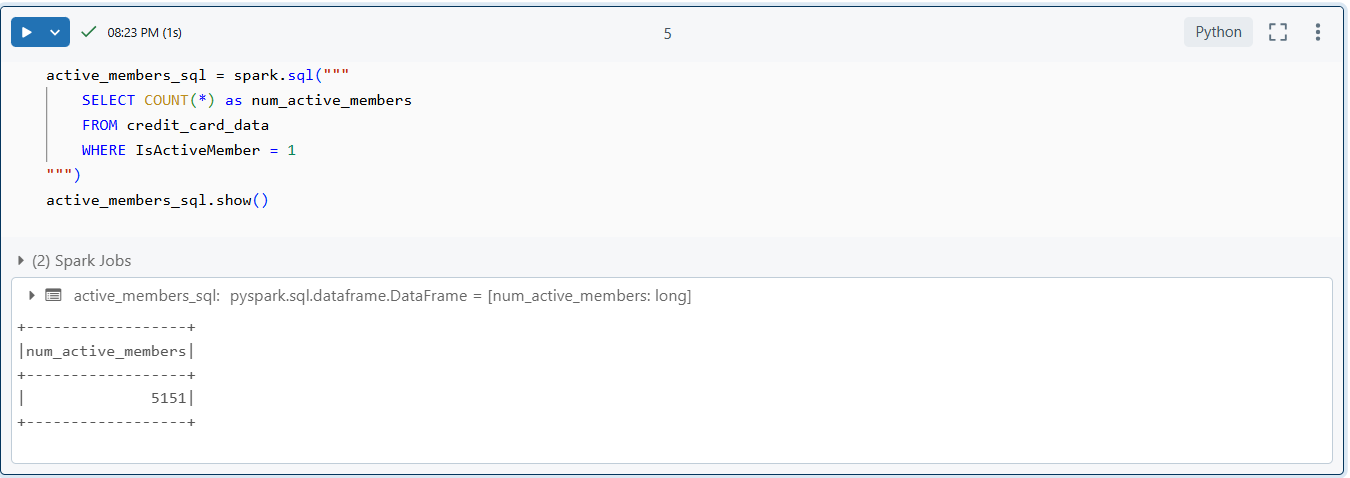
In the PySpark DataFrame API, we use the **filter()** function to directly filter the rows where the **Geography** column equals "Spain." This approach is straightforward and utilizes PySpark’s built-in DataFrame functions to apply the filter condition.



1. **number of members who are elgible and active in the bank**

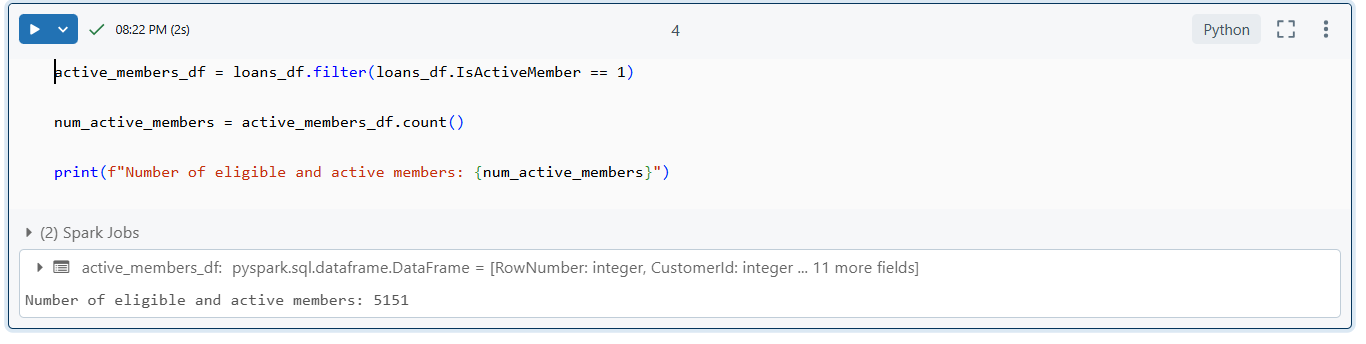
**with SQL:**

To count the number of eligible and active members using Spark SQL, we first create a temporary SQL view of the DataFrame using **createOrReplaceTempView()**. This allows us to run SQL queries against the DataFrame. We then write a SQL query that counts the rows where the **IsActiveMember** column is equal to **1** (indicating an active member). The query **SELECT COUNT(\*) as num\_active\_members FROM credit\_card\_data WHERE IsActiveMember = 1** counts all the active members in the dataset.



**without SQL:**

Using PySpark’s DataFrame API, we achieve the same result by applying the **filter()** function to isolate rows where the **IsActiveMember** column equals 1. After filtering the active members, we use the **count()** method to calculate the total number of active members in the dataset.

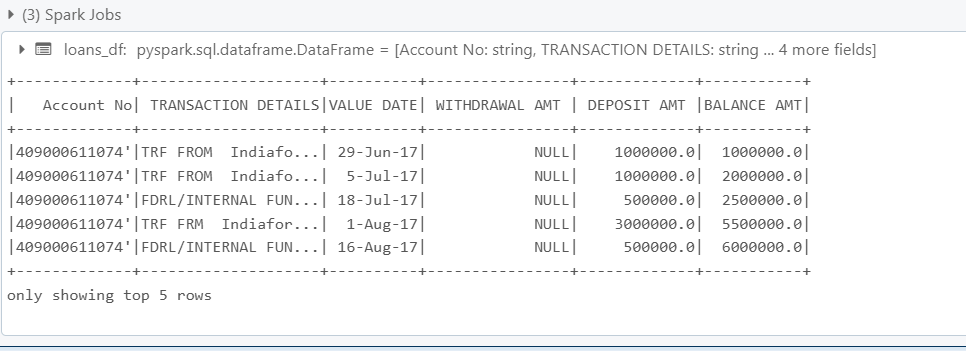


**Transactions file**

**Step 1: Loading the CSV File into a DataFrame**

In this step, we initialize a **SparkSession** and use **spark.read.format("csv")** to load the **txn.csv** file into a PySpark DataFrame. The **header="true"** option ensures the first row is used as column names, while **inferSchema="true"** automatically infers the data types of each column. This allows us to handle both numerical and categorical data appropriately. Finally, we use the **show(5)** function to display the first five rows of the DataFrame for verification.



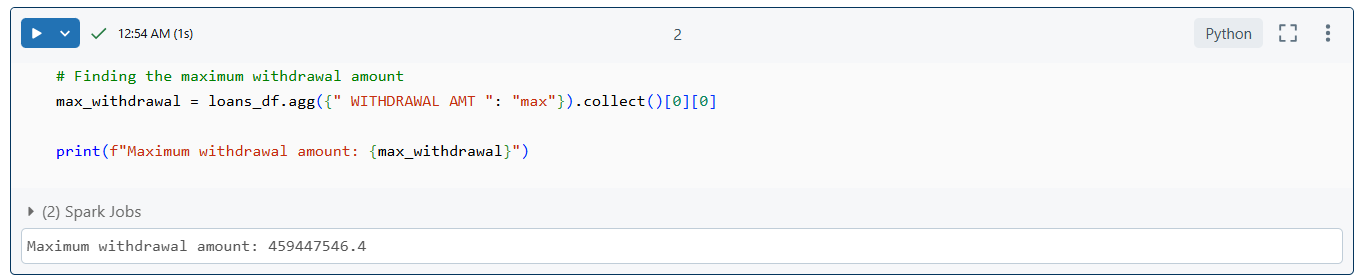


1. **Maximum withdrawal amount in transactions**

**With SQL**:  
Using SparkSQL, we wrote a query with the **MAX()** function to find the maximum value in the **WITHDRAWAL AMT** column. To ensure precision in the result, we cast the output to **DECIMAL(20, 2).**



**Without SQL**:  
In PySpark, we used the **agg()** function to perform an aggregation on the **WITHDRAWAL AMT** column, applying the **max** operation. The result was collected using **.collect()** to fetch the value from the distributed DataFrame.



1. **minimum withdrawal amount of an account in txn.csv**

**With SQL**:  
Using SparkSQL, we wrote a query with the **MIN()** function to calculate the minimum withdrawal amount for each account. The **GROUP BY** clause was used to group the data **by Account No**.

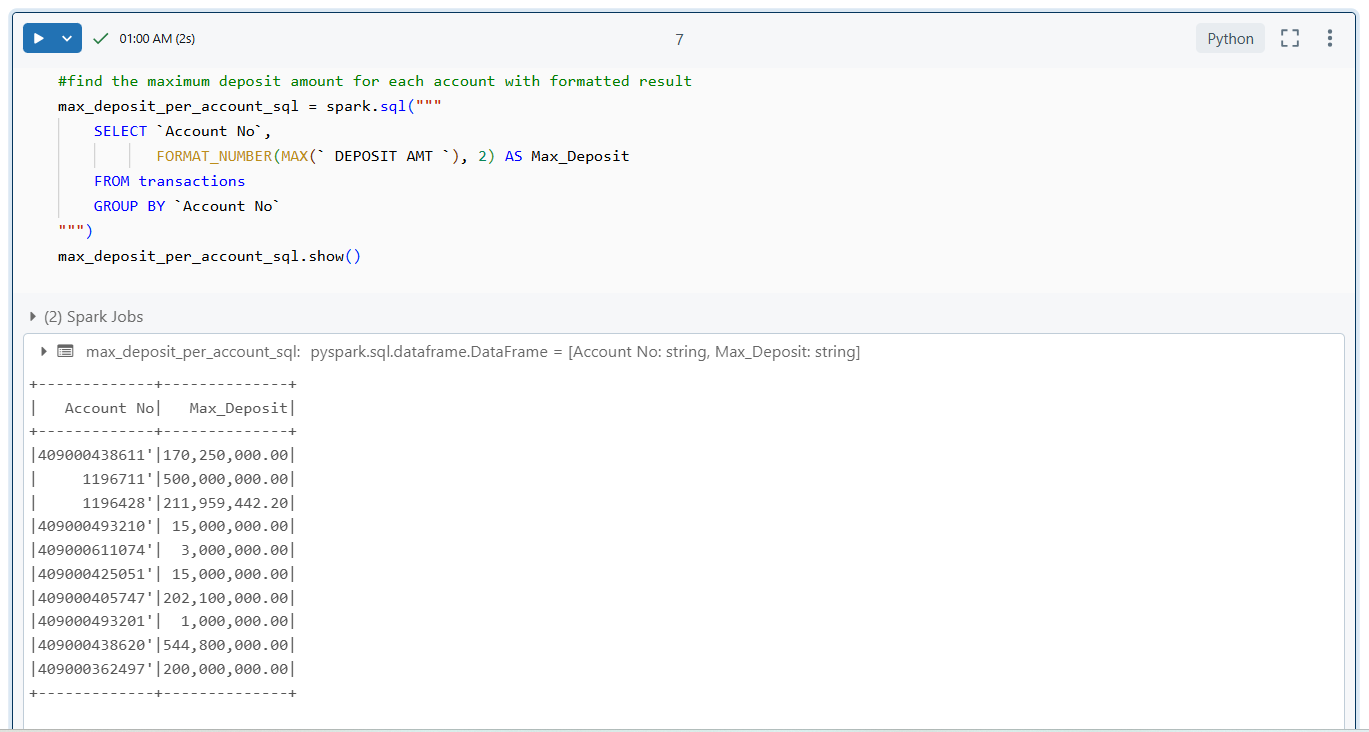


**Without SQL**:  
In PySpark, we used the **groupBy()** function to group the DataFrame by **Account No**, followed by the **agg()** method to compute the minimum value of the **WITHDRAWAL AMT** column. The result was then refined by renaming the generated column **(min(...))** to **Min\_Withdrawal** for clarity.

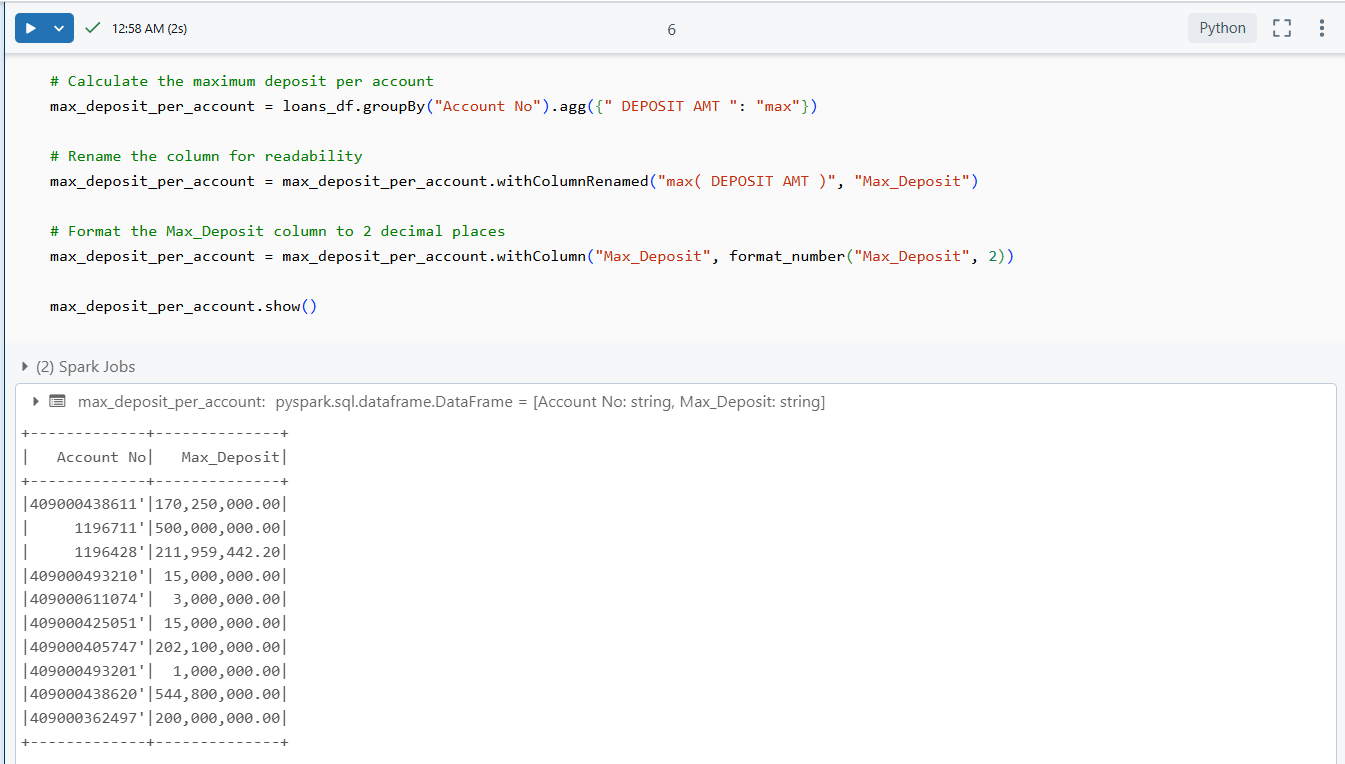


1. **maximum deposit amount of an account**

**With SQL**:  
We utilized a SQL query with the **MAX()** function to find the highest deposit amount for each account. The **GROUP BY** clause grouped the data by **Account No**, and the result was formatted using **FORMAT\_NUMBER()** to display the maximum deposit with two decimal places.

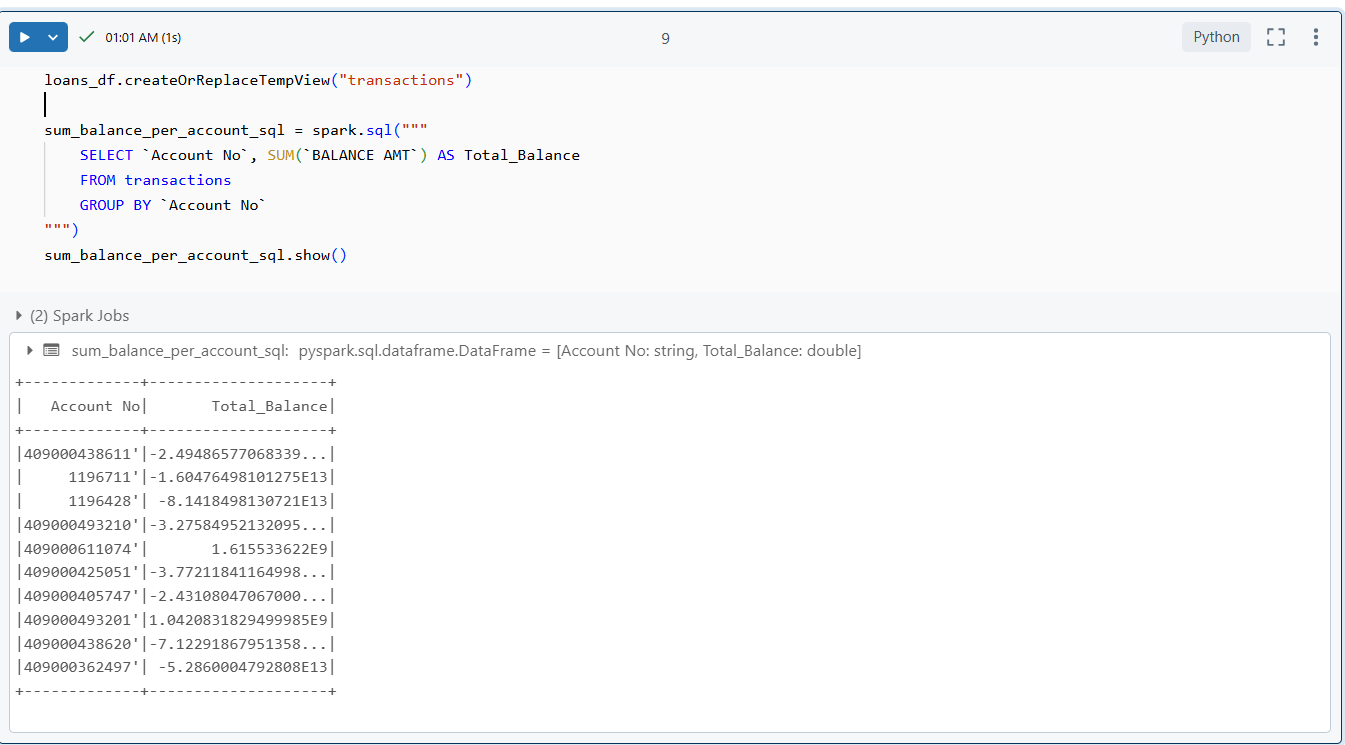


**Without SQL**:  
In PySpark, the **groupBy()** method was used to group the DataFrame by Account No, followed by the **agg()** function to compute the maximum deposit for each group. The resulting column was renamed for clarity using **withColumnRenamed(),** and the **format\_number()** function was applied to format the values to two decimal places.

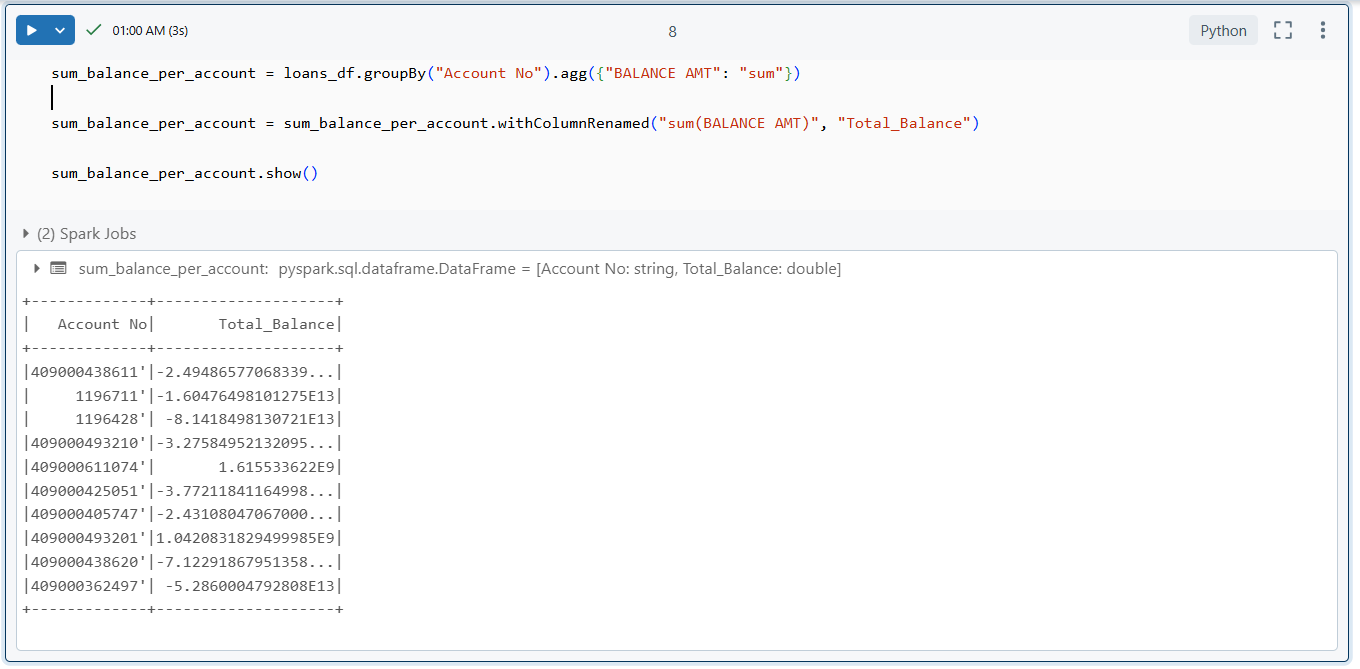


1. **sum of balance in every bank account**

**With SQL**:  
A SQL query was written using the **SUM()** function to compute the total **BALANCE AMT** for each **Account No**. The **GROUP BY** clause grouped the transactions by account number, ensuring that the sum was calculated individually for each account.

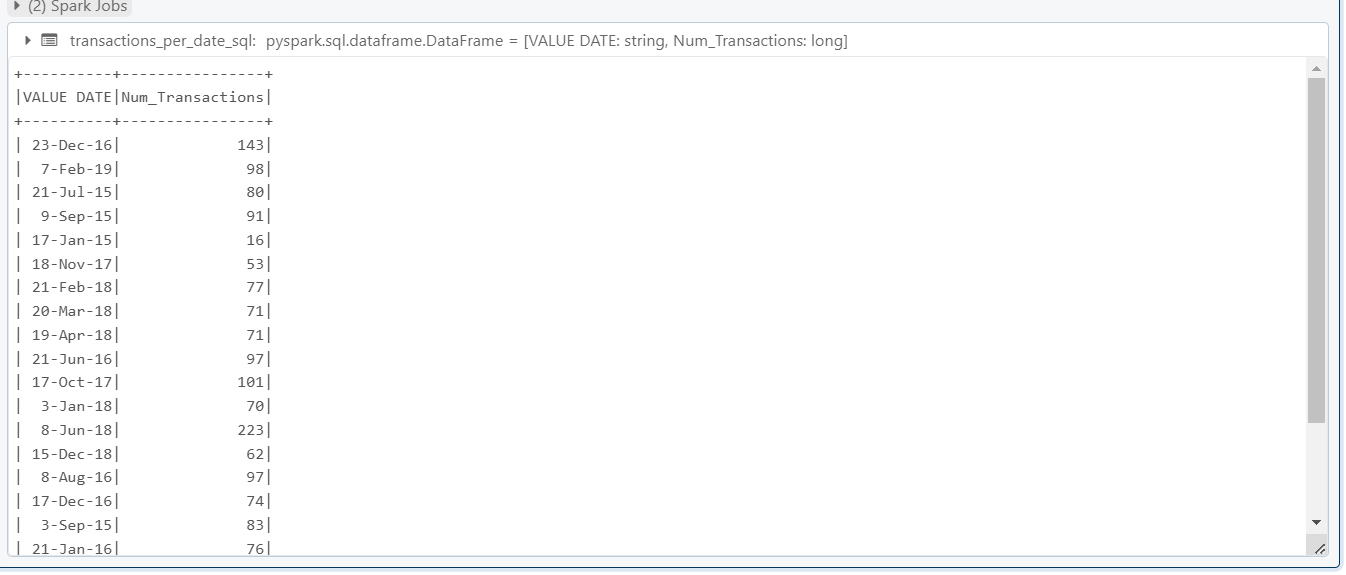
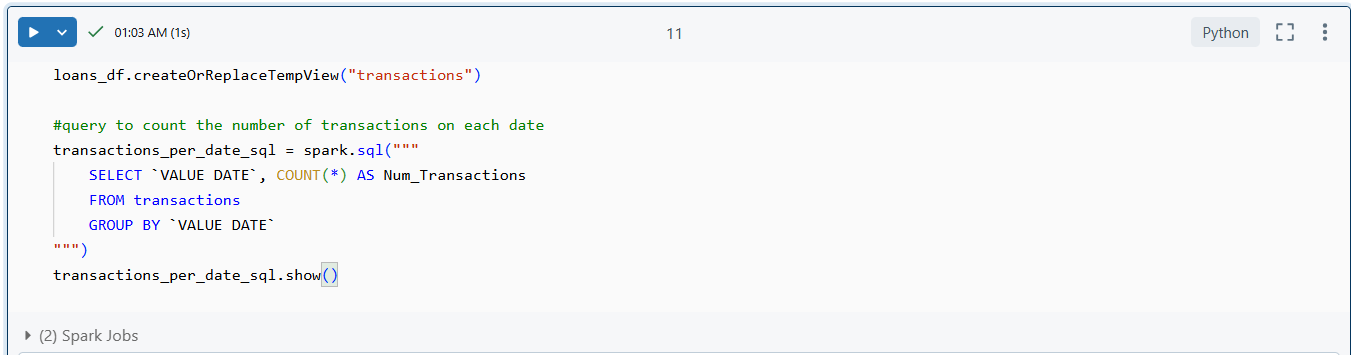


**Without SQL**:  
In PySpark, we used the **groupBy()** function to group the DataFrame by Account No, followed by the **agg()** function to apply the **sum()** operation on the **BALANCE AMT** column. The resulting column was renamed using **withColumnRenamed()** for better readability.

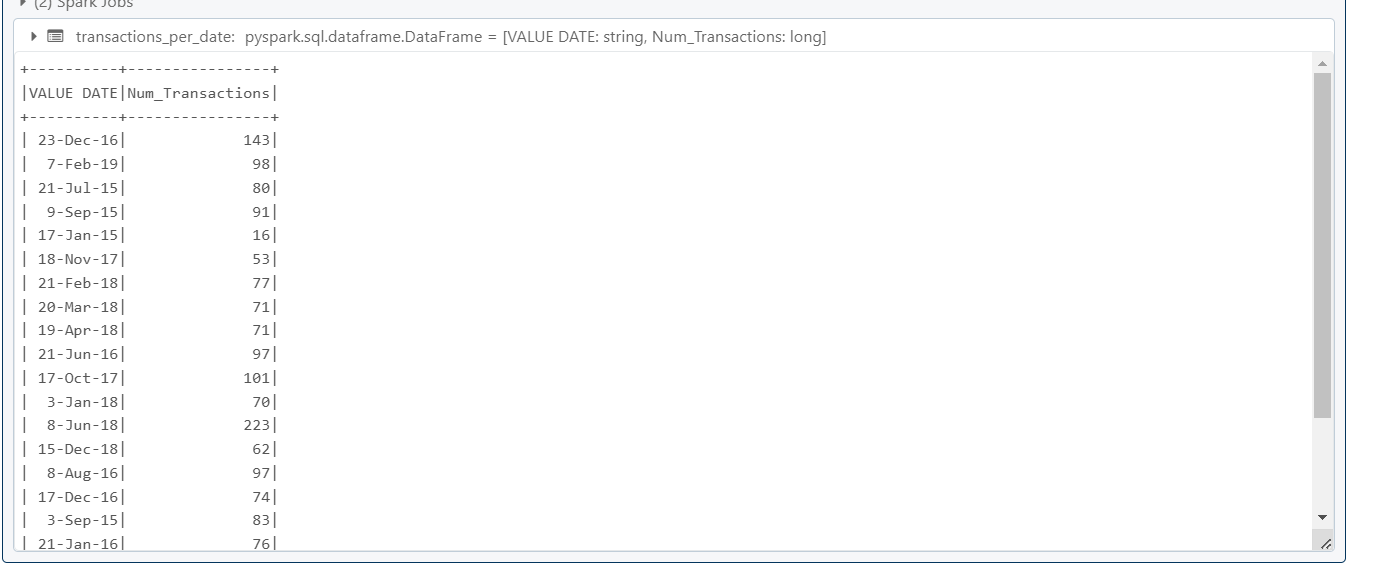
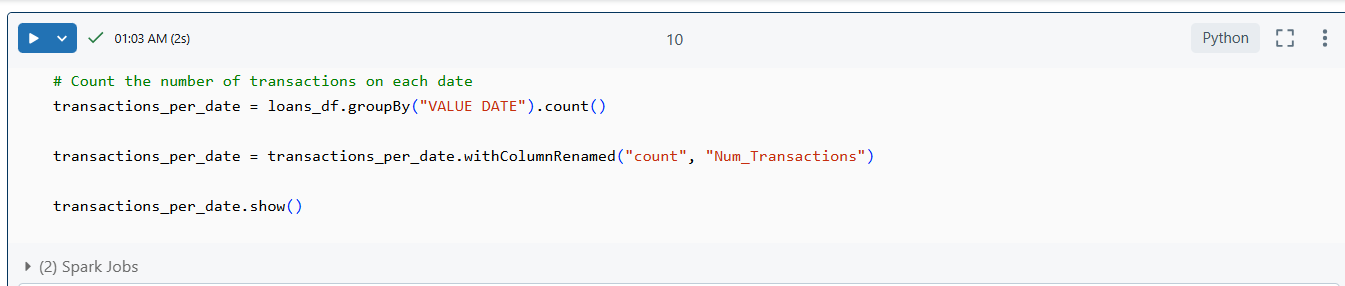


1. **Number of transaction on each date**

**With SQL**:  
We wrote a SQL query using the **COUNT(\*)** function to calculate the total number of transactions for each **VALUE DATE**. The **GROUP BY** clause grouped the data by transaction date, ensuring the counts were aggregated per date.

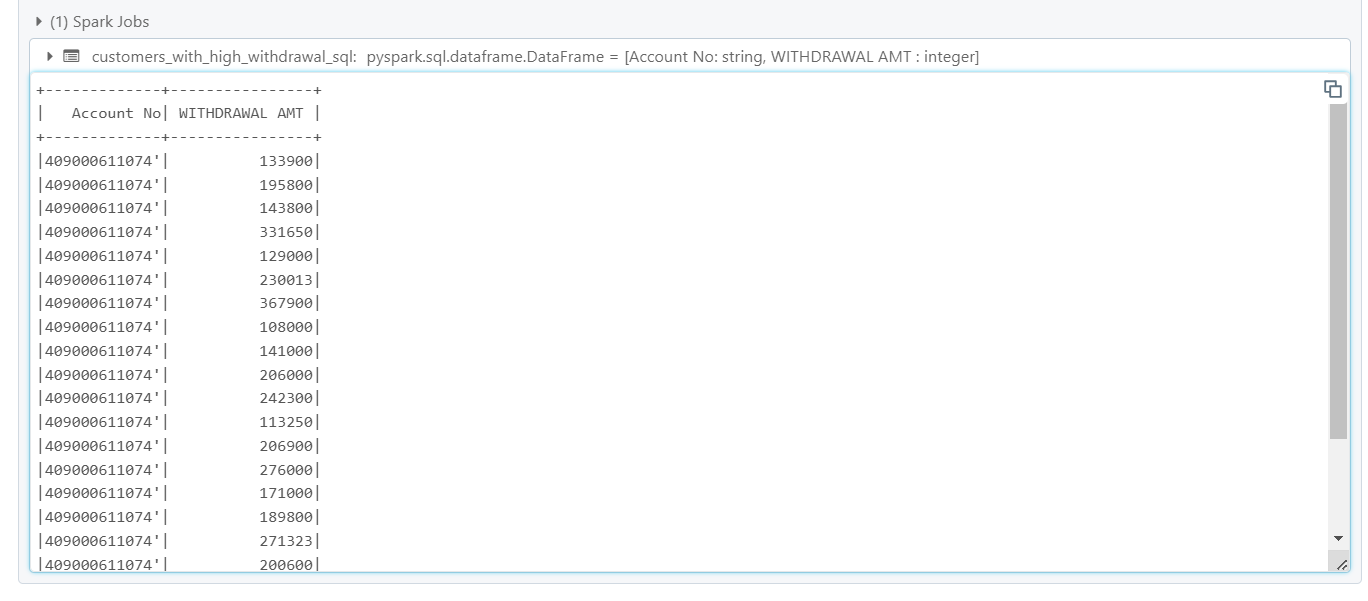
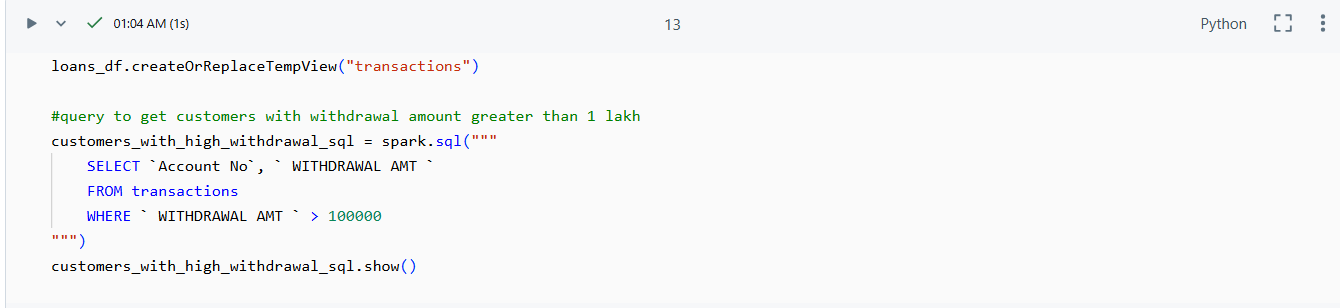


**Without SQL**:  
In PySpark, the **groupBy()** function was used to group the DataFrame by the **VALUE DATE** column, followed by the **count()** function to compute the number of transactions for each date. The resulting column was renamed using **withColumnRenamed()** for clarity.



1. **List of customers with withdrawal amount more than 1 lakh**

**With SQL**:  
A SQL query was written to filter transactions where the **WITHDRAWAL AMT** is greater than **1,00,000**. The **SELECT** clause retrieved the **Account No** and **WITHDRAWAL AMT** columns, and the **WHERE** condition ensured only relevant records were included.



**Without SQL**:  
In PySpark, the **filter()** function was applied on the DataFrame to select rows where the **WITHDRAWAL AMT** exceeded **1,00,000**. The **select()** method was then used to retrieve only the **Account No** and **WITHDRAWAL AMT** columns for the output.

