**PYSPARK CASE STUDY**

**NAME : Podutur Lahari**

**Roll No : DE126**

**DATE : 23-11-2024**

**TOPIC : ONLINE BANKING ANALYSIS**

**Introduction**

Online Banking Analysis is our first project leveraging big data tools like Apache Spark, Hive, and HDFS to process and analyze large datasets. We sourced datasets for loans, credit cards, and transactions from Kaggle, cleaned and ingested them into Hive using Sqoop, and performed advanced analytics using PySpark. Key use cases included analyzing loan categories, customer credit card eligibility, transaction trends, and income-based metrics. This project showcased how big data technologies can handle real-world financial datasets efficiently and generate actionable insights, providing a strong foundation in distributed data processing and analysis.

## **1. Dataset Selection and Source**

For this analysis, we have Loan dataset, credit card dataset, transaction dataset

Links :

1. <https://drive.google.com/file/d/12qnb6p6v2JEIWHaczabeX-DhHrygm9B-/view?usp=sharing>
2. <https://drive.google.com/file/d/1B-OsGlcSMcIDBu23nzLa8MjDd-E64kSU/view?usp=sharing>
3. <https://drive.google.com/file/d/1M6pyN33ue9y0fcfucpMnr5_EJfwPN4kd/view?usp=sharing>

## **2. Data Preparation & Cleaning**

* **Loan Dataset operations :**

Load Dataset into Cluster notebook :

**Code :**

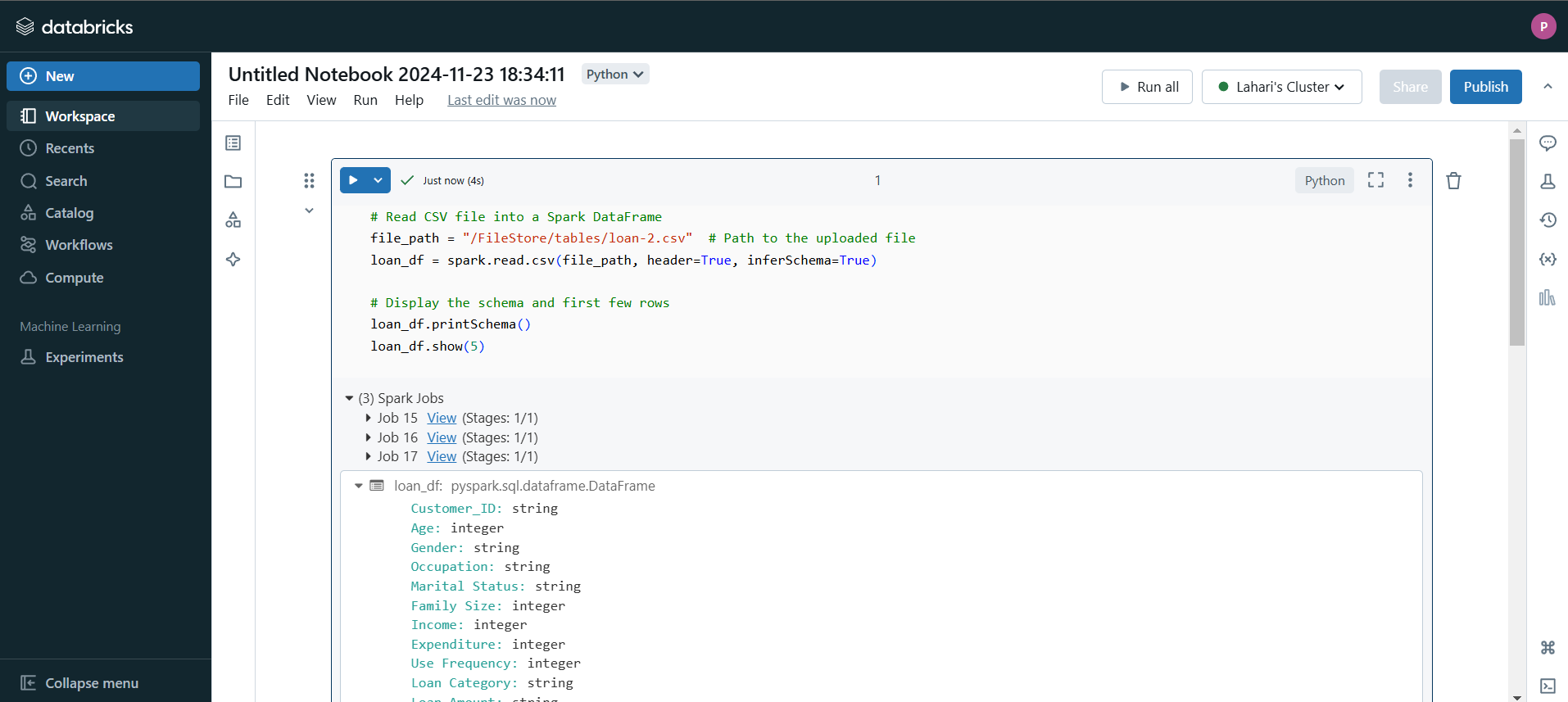
# Read CSV file into a Spark DataFrame

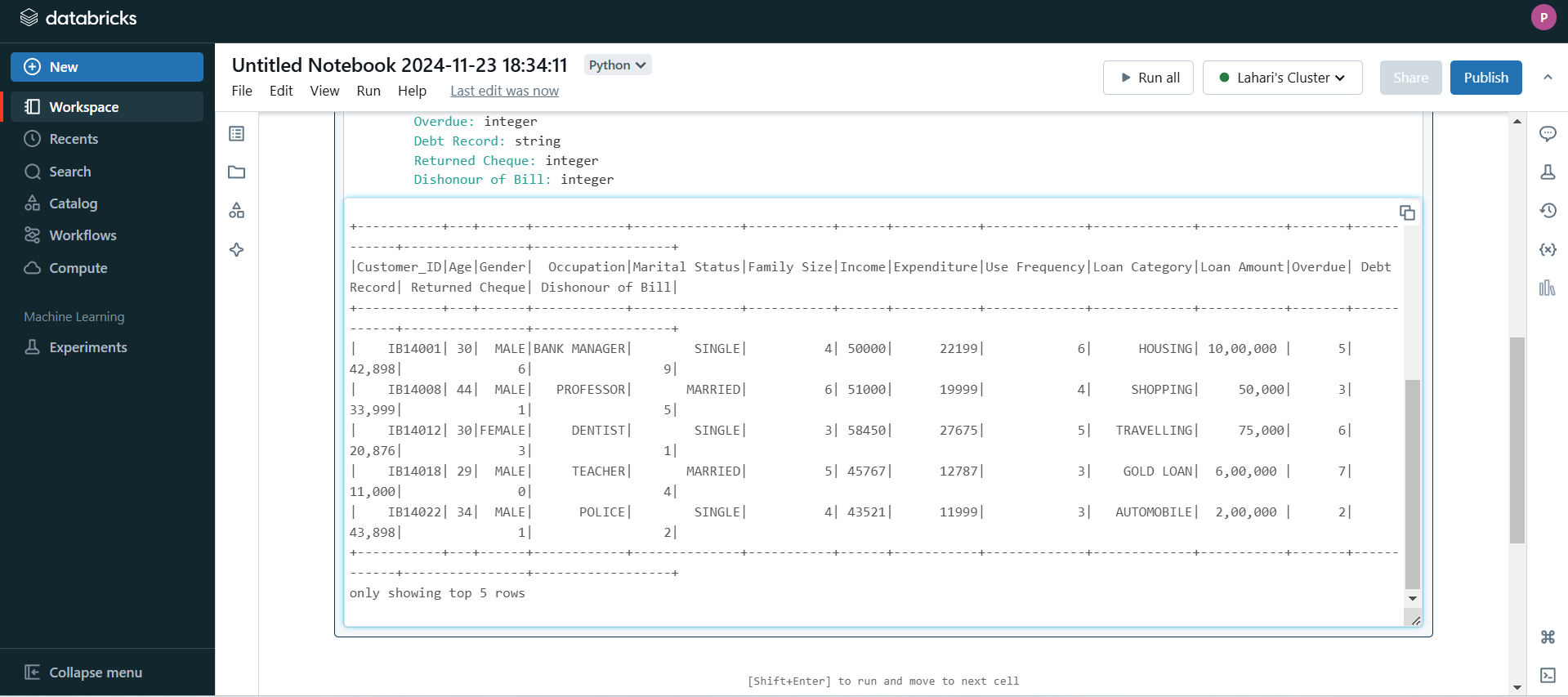
file\_path = "/FileStore/tables/loan-2.csv" # Path to the uploaded file

loan\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Display the schema and first few rows

loan\_df.printSchema()

loan\_df.show(5)

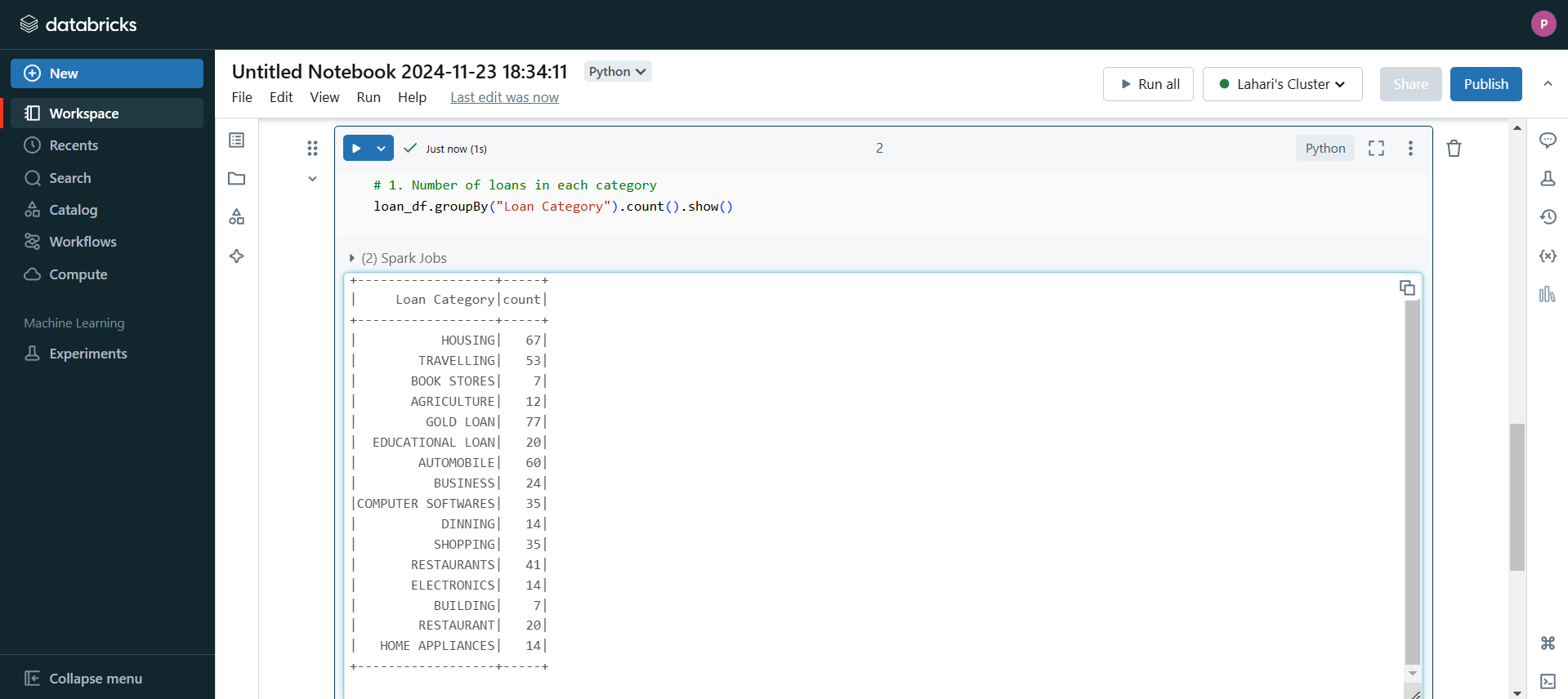


**Explanation:**

* header=True: Indicates that the first row contains column names.
* inferSchema=True: Automatically infers the data types of columns.

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

**Code :** loan\_df.groupBy("Loan Category").count().show()



**Explanation:**

* groupBy("Loan Category"): Groups the data based on the Loan Category column, which contains the types of loans.
* count(): Counts the number of records in each loan category.
* show(): Displays the result of the grouped count.

1. **Number of People Who Have Taken More Than 1 Lakh Loan :**

**Code :**

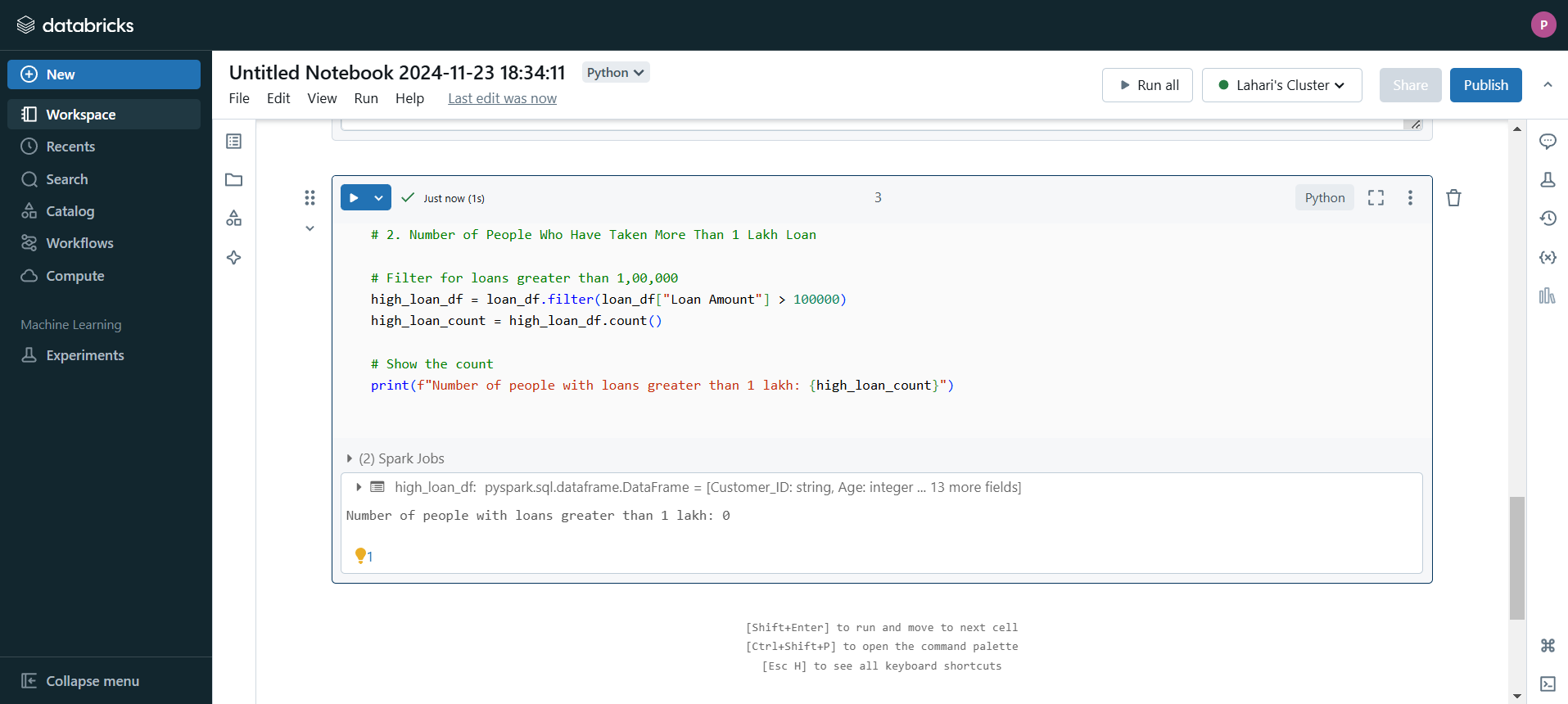
# Filter for loans greater than 1,00,000

high\_loan\_df = loan\_df.filter(loan\_df["Loan Amount"] > 100000)

high\_loan\_count = high\_loan\_df.count()

# Show the count

print(f"Number of people with loans greater than 1 lakh: {high\_loan\_count}")



**Explanation:**

* filter(loan\_df["Loan Amount"] > 100000): Filters rows where the Loan Amount is greater than 1 lakh.
* count(): Counts the number of records that satisfy the condition.
* print(): Displays the count of people who have taken loans exceeding 1 lakh.

1. **Number of People With Income Greater Than 60,000 Rupees :**

**Code :**

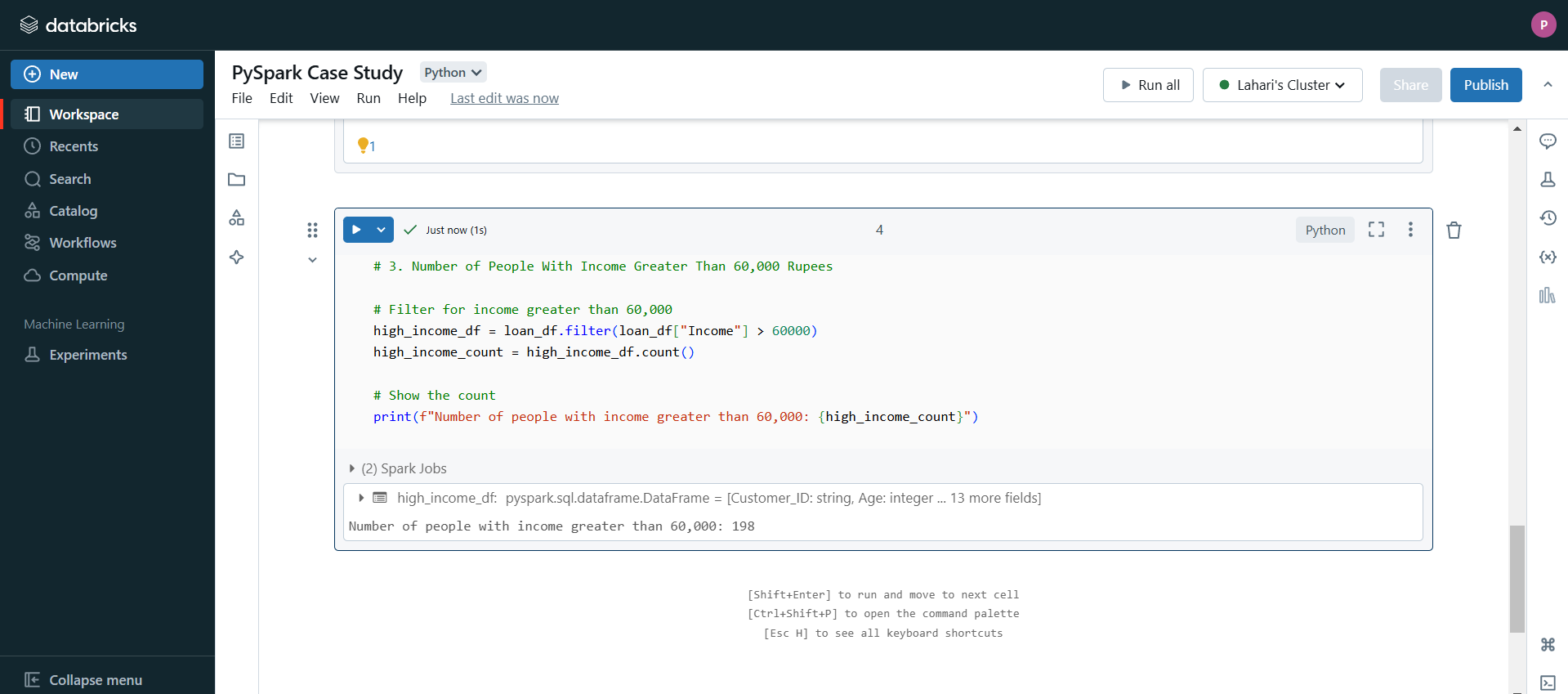
# Filter for income greater than 60,000

high\_income\_df = loan\_df.filter(loan\_df["Income"] > 60000)

high\_income\_count = high\_income\_df.count()

# Show the count

print(f"Number of people with income greater than 60,000: {high\_income\_count}")



**Explanation:**

* filter(loan\_df["Income"] > 60000): Filters rows where the Income column is greater than ₹60,000.
* count(): Counts how many records meet this condition.
* print(): Displays the count of people with income above ₹60,000.

1. **Number of People With 2+ Returned Cheques and Income Less Than 50,000**

**Code :**

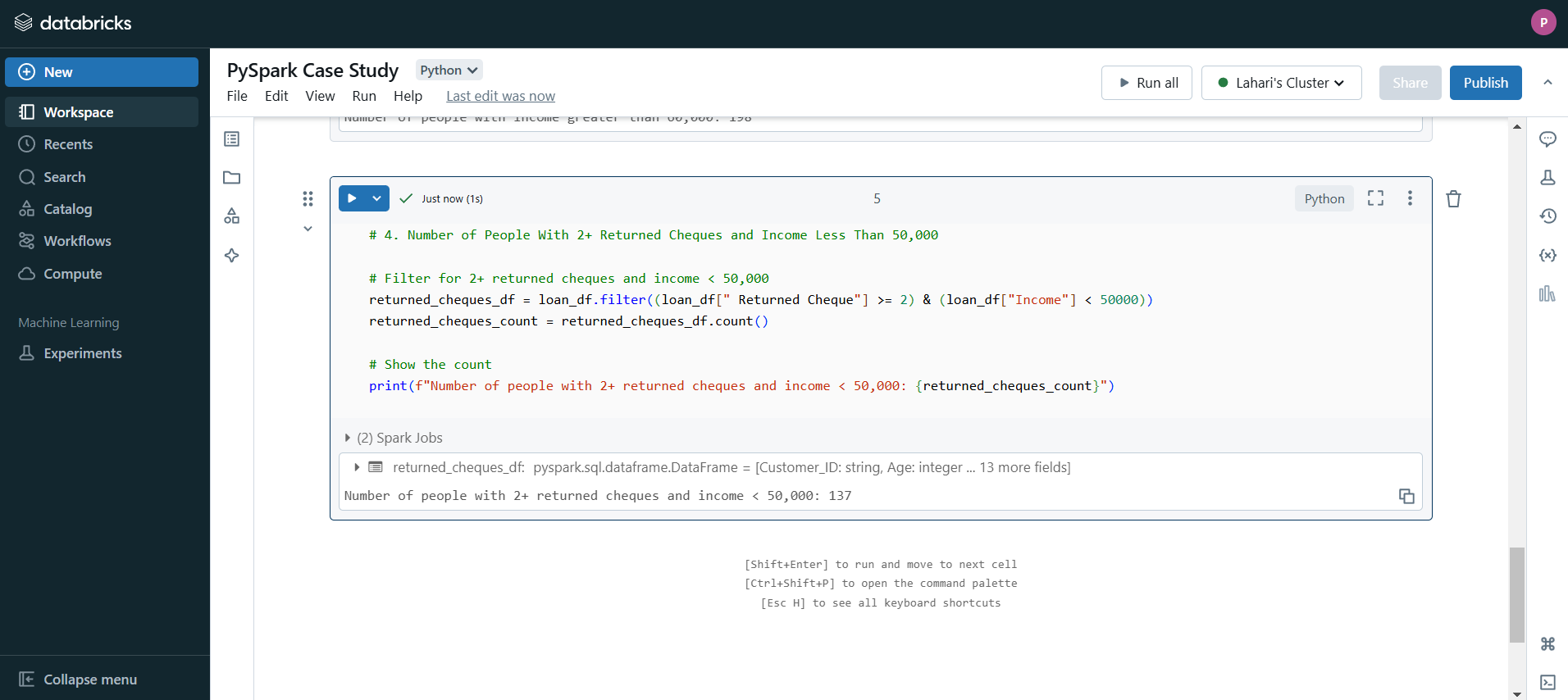
# Filter for 2+ returned cheques and income < 50,000

returned\_cheques\_df = loan\_df.filter((loan\_df[" Returned Cheque"] >= 2) & (loan\_df["Income"] < 50000))

returned\_cheques\_count = returned\_cheques\_df.count()

# Show the count

print(f"Number of people with 2+ returned cheques and income < 50,000: {returned\_cheques\_count}")



**Explanation:**

* filter(): Filters rows where the Returned Cheque column is greater than or equal to 2 and the Income is less than ₹50,000.
* count(): Counts how many records satisfy both conditions.
* print(): Displays the count of such individuals.

1. **Number of People With 2+ Returned Cheques and Are Single**

**Code :**

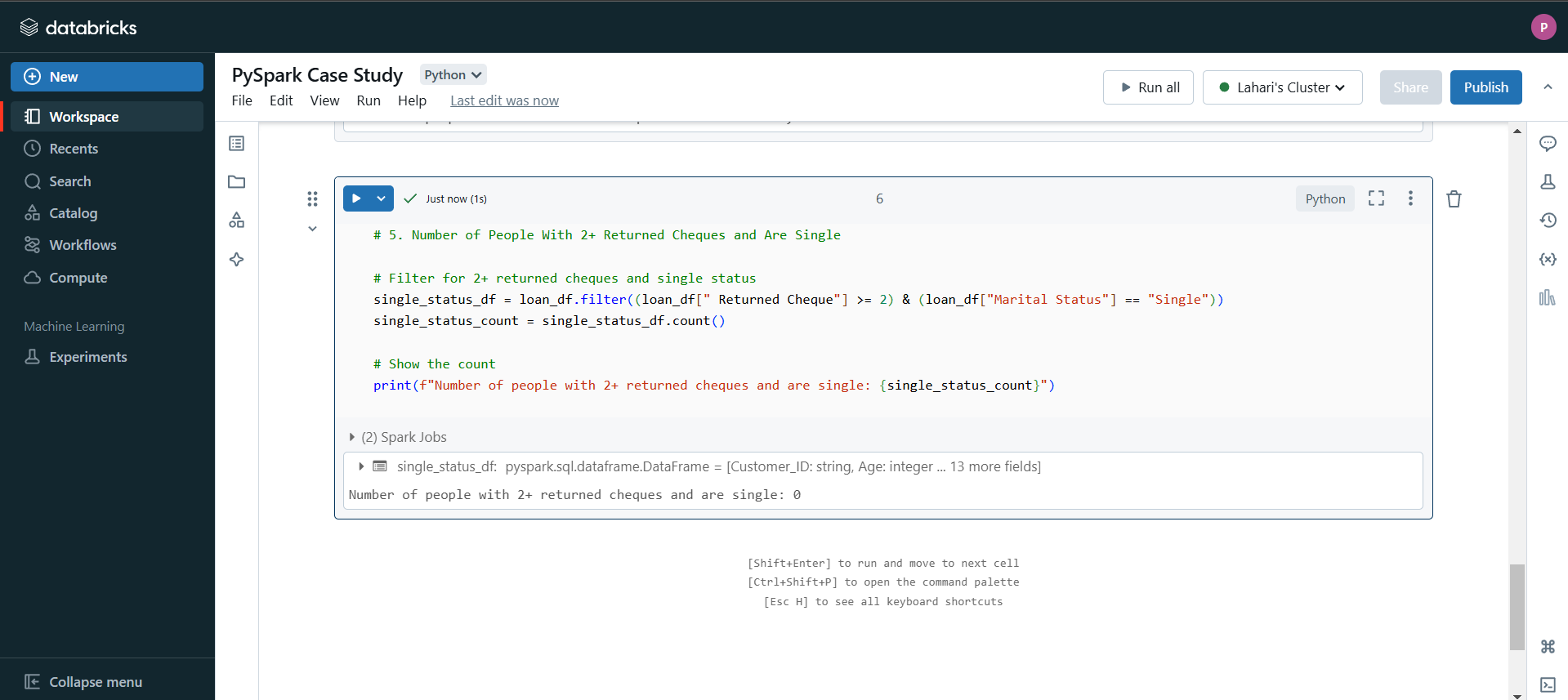
# Filter for 2+ returned cheques and single status

single\_status\_df = loan\_df.filter((loan\_df[" Returned Cheque"] >= 2) & (loan\_df["Marital Status"] == "Single"))

single\_status\_count = single\_status\_df.count()

# Show the count

print(f"Number of people with 2+ returned cheques and are single: {single\_status\_count}")

****

**Explanation:**

* **filter()**: Filters for rows where the Returned Cheque is greater than or equal to 2 and the Marital Status is "Single".
* **count()**: Counts the number of such records.
* **print()**: Displays the result.

1. **Number of People with Expenditure Over 50,000 a Month**

Code :

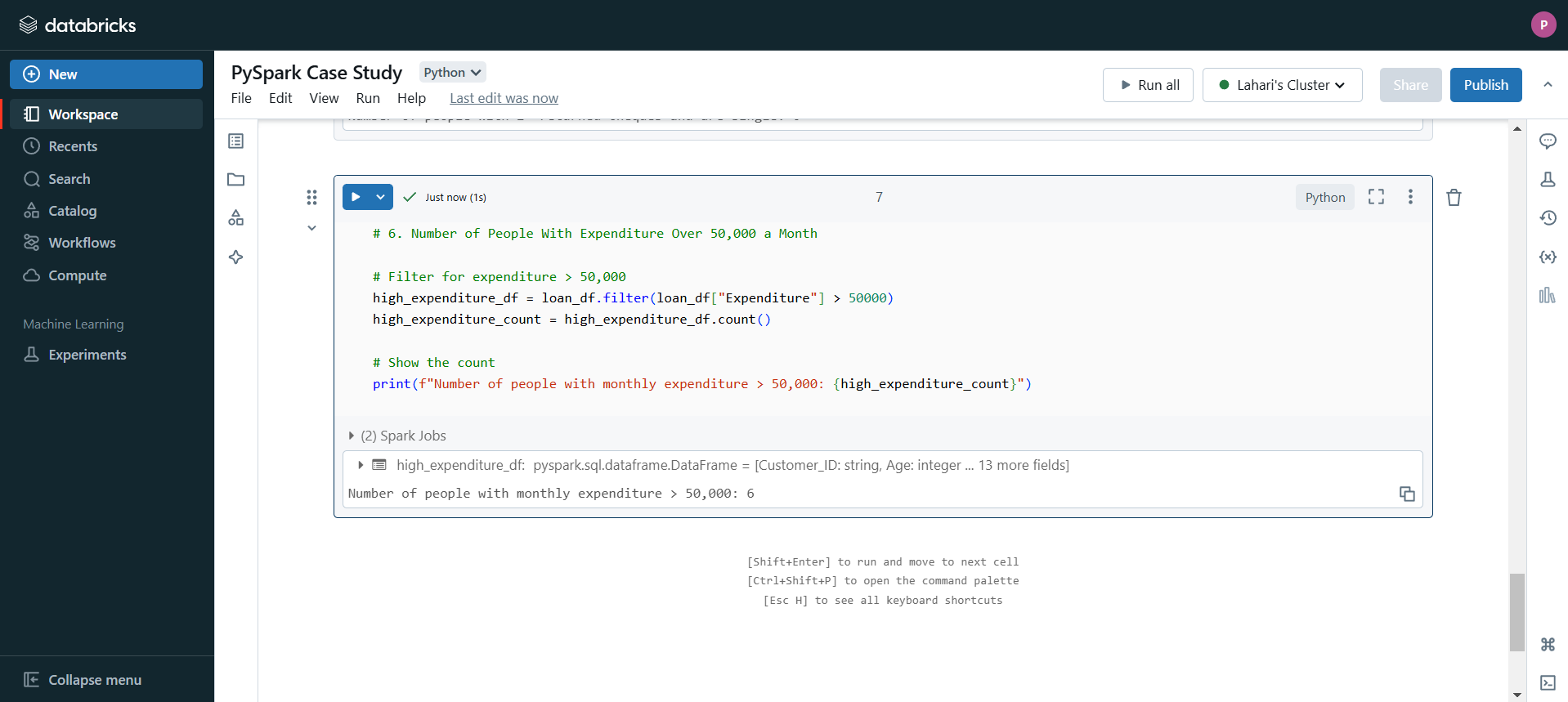
# Filter for expenditure > 50,000

high\_expenditure\_df = loan\_df.filter(loan\_df["Expenditure"] > 50000)

high\_expenditure\_count = high\_expenditure\_df.count()

# Show the count

print(f"Number of people with monthly expenditure > 50,000: {high\_expenditure\_count}")

****

**Explanation:**

* **filter(loan\_df["Expenditure"] > 50000)**: Filters the dataset for records where the Expenditure is greater than ₹50,000.
* **count()**: Counts how many records meet the condition.
* **print()**: Displays the number of people who spend more than ₹50,000 monthly

1. **Number of Members Who Are Eligible for Credit Card**

**Code :**

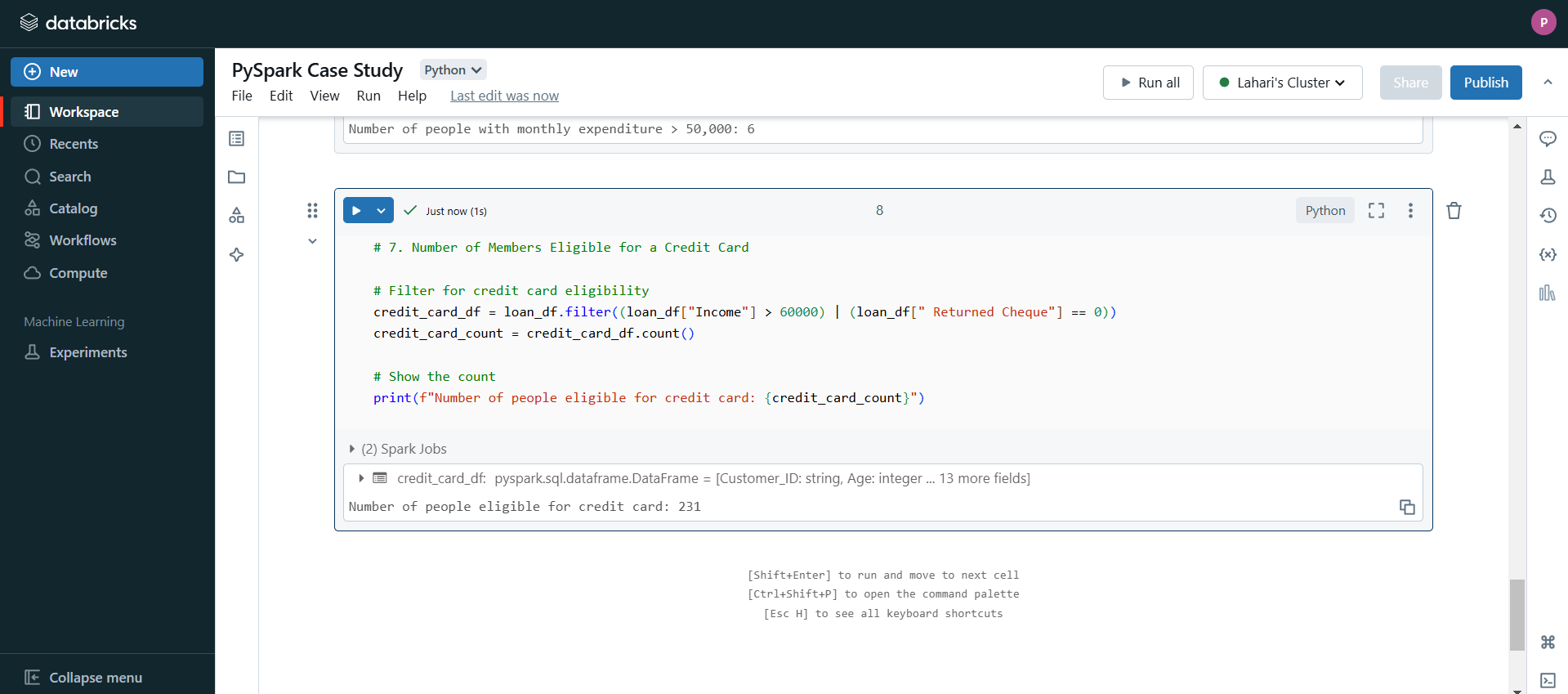
**# Filter for credit card eligibility**

**credit\_card\_df = loan\_df.filter((loan\_df["Income"] > 60000) | (loan\_df[" Returned Cheque"] == 0))**

**credit\_card\_count = credit\_card\_df.count()**

**# Show the count**

**print(f"Number of people eligible for credit card: {credit\_card\_count}")**

****

**Explanation:**

* **filter()**: Filters records where the Credit Score is above 650 and the Income exceeds ₹50,000, assuming these are the criteria for credit card eligibility.
* **count()**: Counts how many records meet both conditions.
* **print()**: Displays the count of eligible members.
* **Credit card Dataset operations :**

**Reading Dataset** :

**Code :**

# Read the uploaded CSV file into a Spark DataFrame

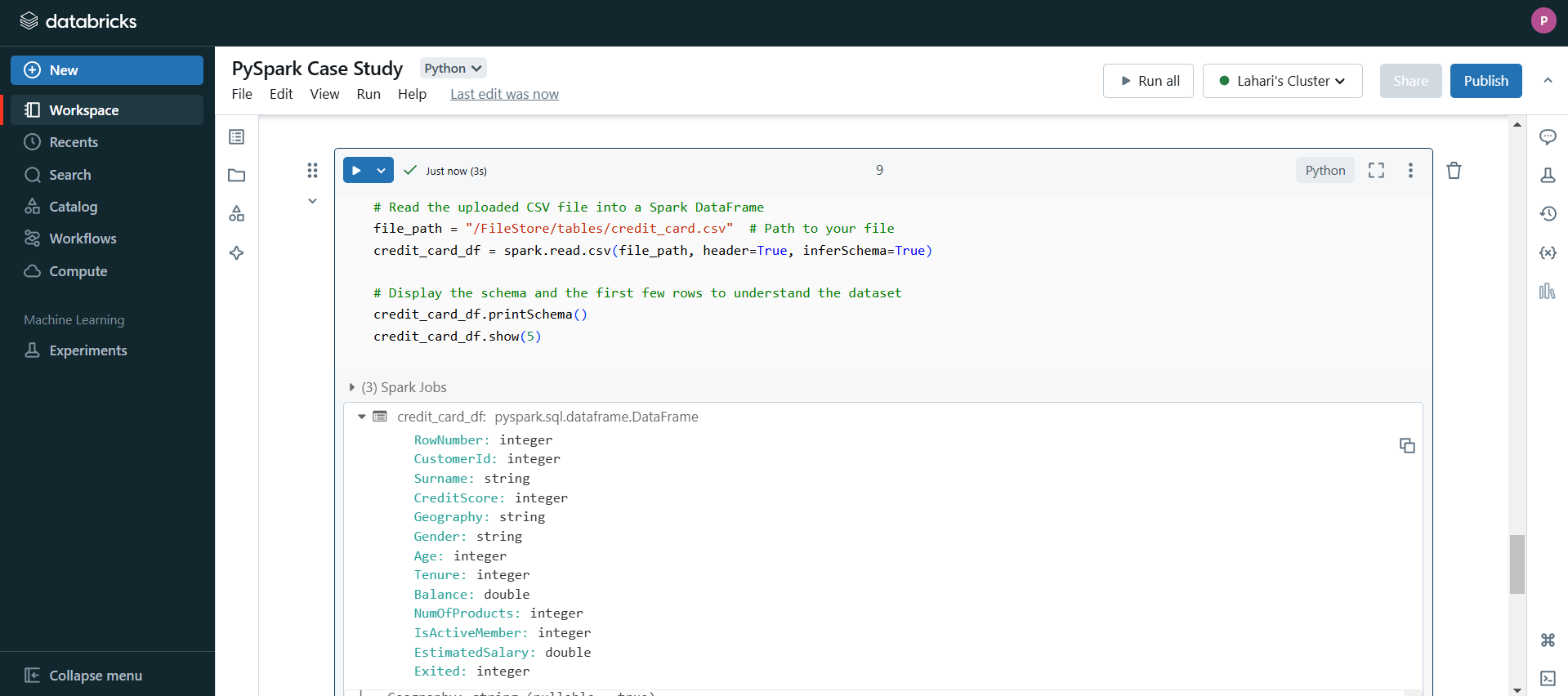
file\_path = "/FileStore/tables/credit\_card.csv" # Path to your file

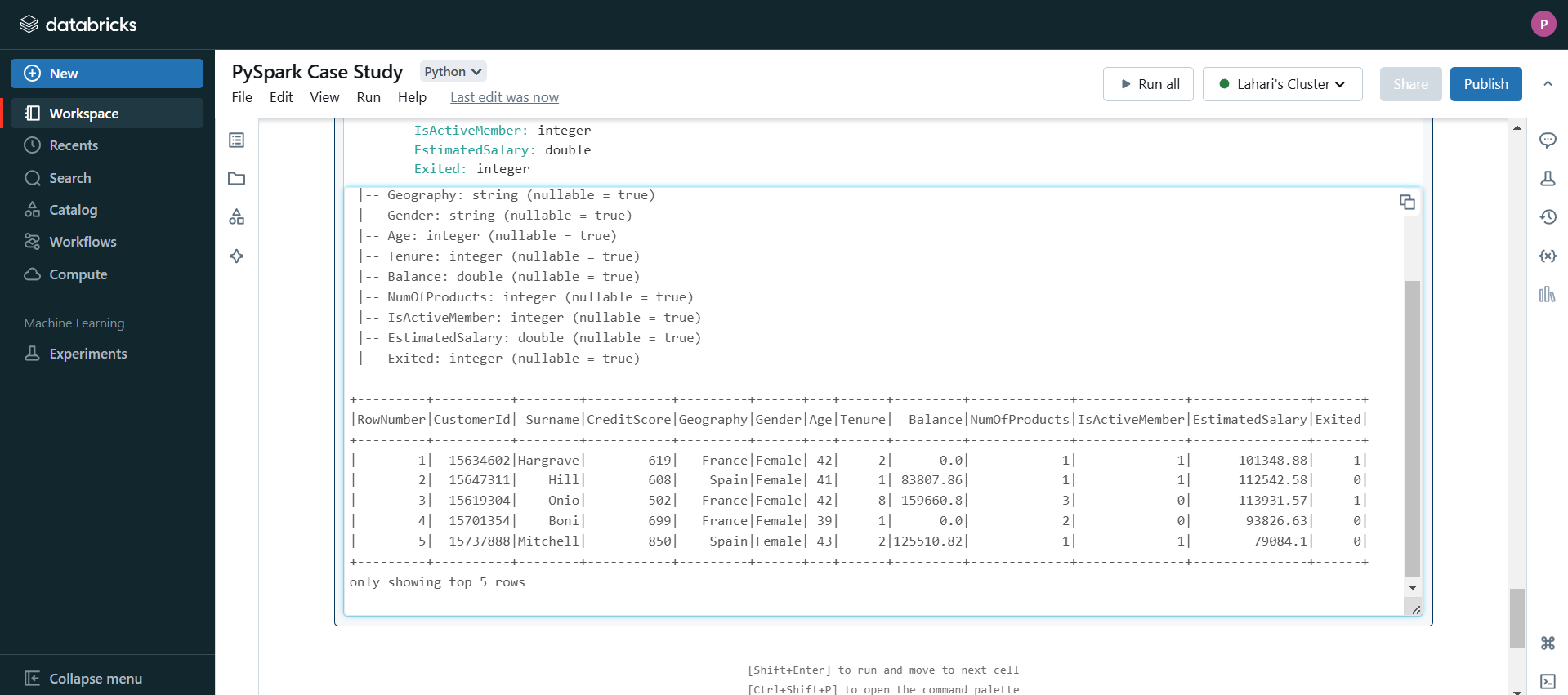
credit\_card\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Display the schema and the first few rows to understand the dataset

credit\_card\_df.printSchema()

credit\_card\_df.show(5)

****

****

**Explanation:**

* **filter(): Filters the rows where the Geography column is "Spain".**
* **count(): Counts the number of rows (credit card users) in Spain.**
* **print(): Displays the result.**

1. **credit card users in Spain**

**Code :**

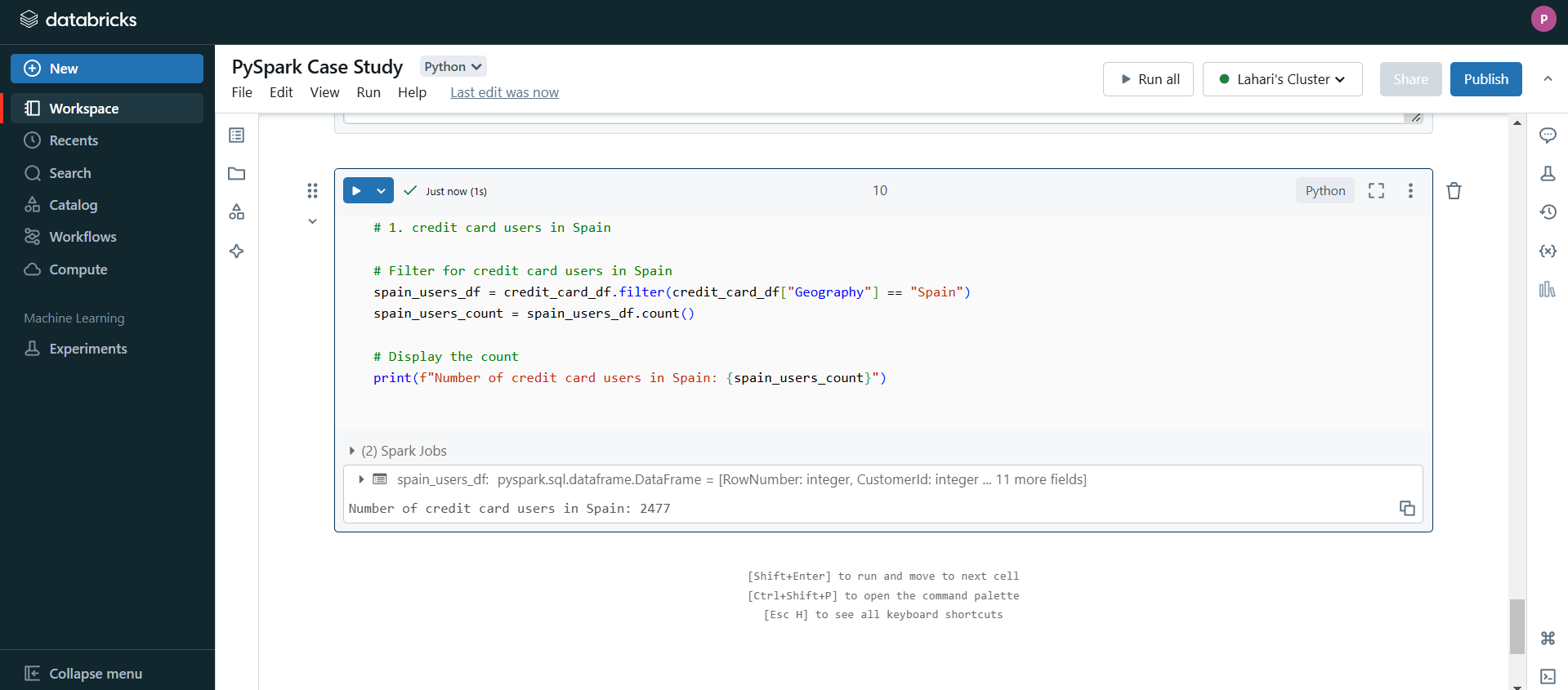
# Filter for credit card users in Spain

spain\_users\_df = credit\_card\_df.filter(credit\_card\_df["Geography"] == "Spain")

spain\_users\_count = spain\_users\_df.count()

# Display the count

print(f"Number of credit card users in Spain: {spain\_users\_count}")



**Explanation:**

* filter(): Filters the rows where the Geography column is "Spain".
* count(): Counts the number of rows (credit card users) in Spain.
* print(): Displays the result.

1. **Number of Eligible and Active Members**

**Code :**

# Filter for eligible and active members

eligible\_active\_df = credit\_card\_df.filter(

(credit\_card\_df["Exited"] == "Eligible") & # Replace with the actual eligibility column/condition

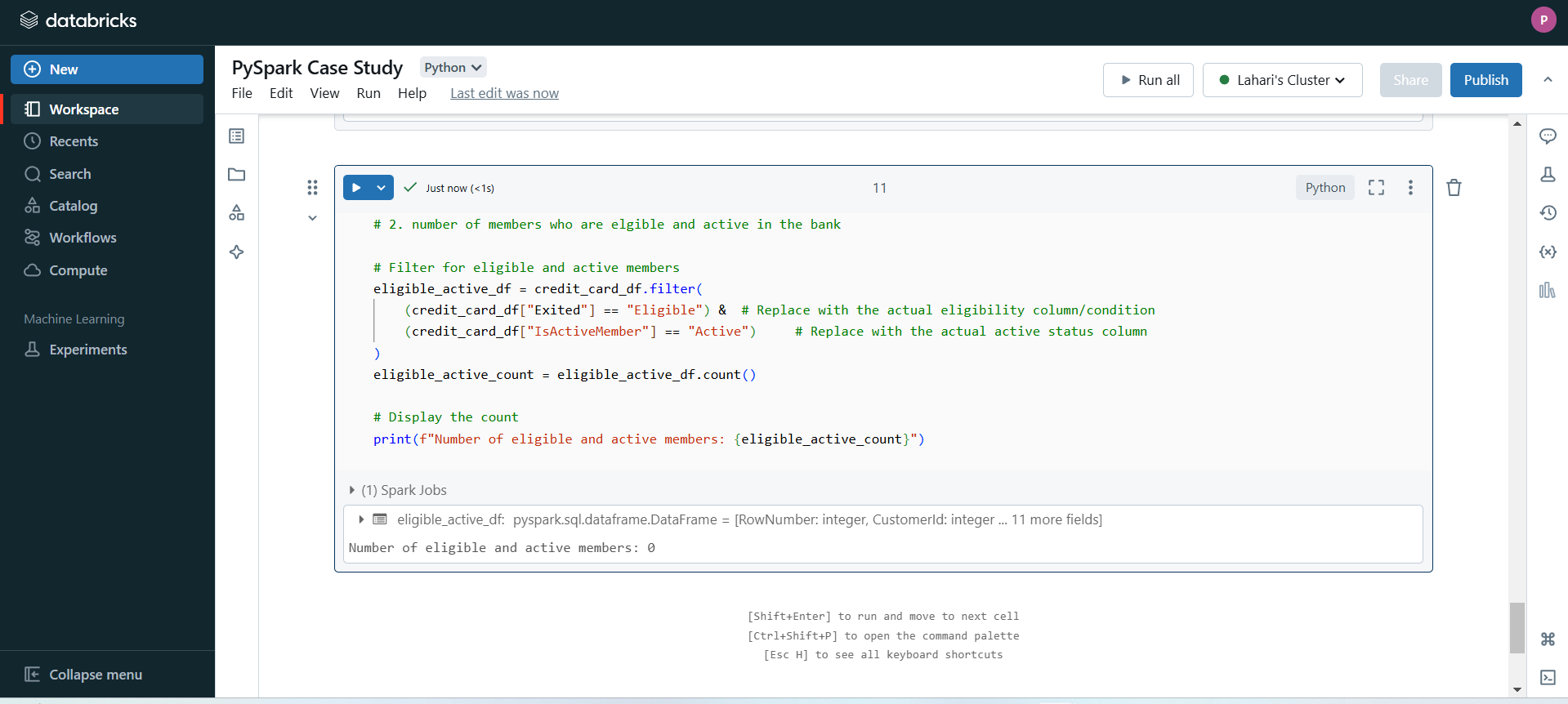
(credit\_card\_df["IsActiveMember"] == "Active") # Replace with the actual active status column

)

eligible\_active\_count = eligible\_active\_df.count()

# Display the count

print(f"Number of eligible and active members: {eligible\_active\_count}")



**Explanation:**

* **filter()**: Filters the rows based on two conditions: CreditScore > 650 and IsActiveMember == 1 (indicating the user is both eligible and active).
* **count()**: Counts the number of rows that meet both criteria.
* **print()**: Displays the result.
* **Transactions Dataset Operations :**

**Read and Load Dataset :**

**Code :**

# Load the file

file\_path = "/FileStore/tables/txn.csv"

df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Display the first few rows of the DataFrame

df.show()

#### 

#### 

#### **Explanation:**

* file\_path: The location of your uploaded CSV file.
* spark.read.csv: Reads the file into a PySpark DataFrame.
  + header=True: Indicates the first row of the CSV contains column names.
  + inferSchema=True: Automatically detects the data types of each column.
* df: The DataFrame that will hold the CSV data.

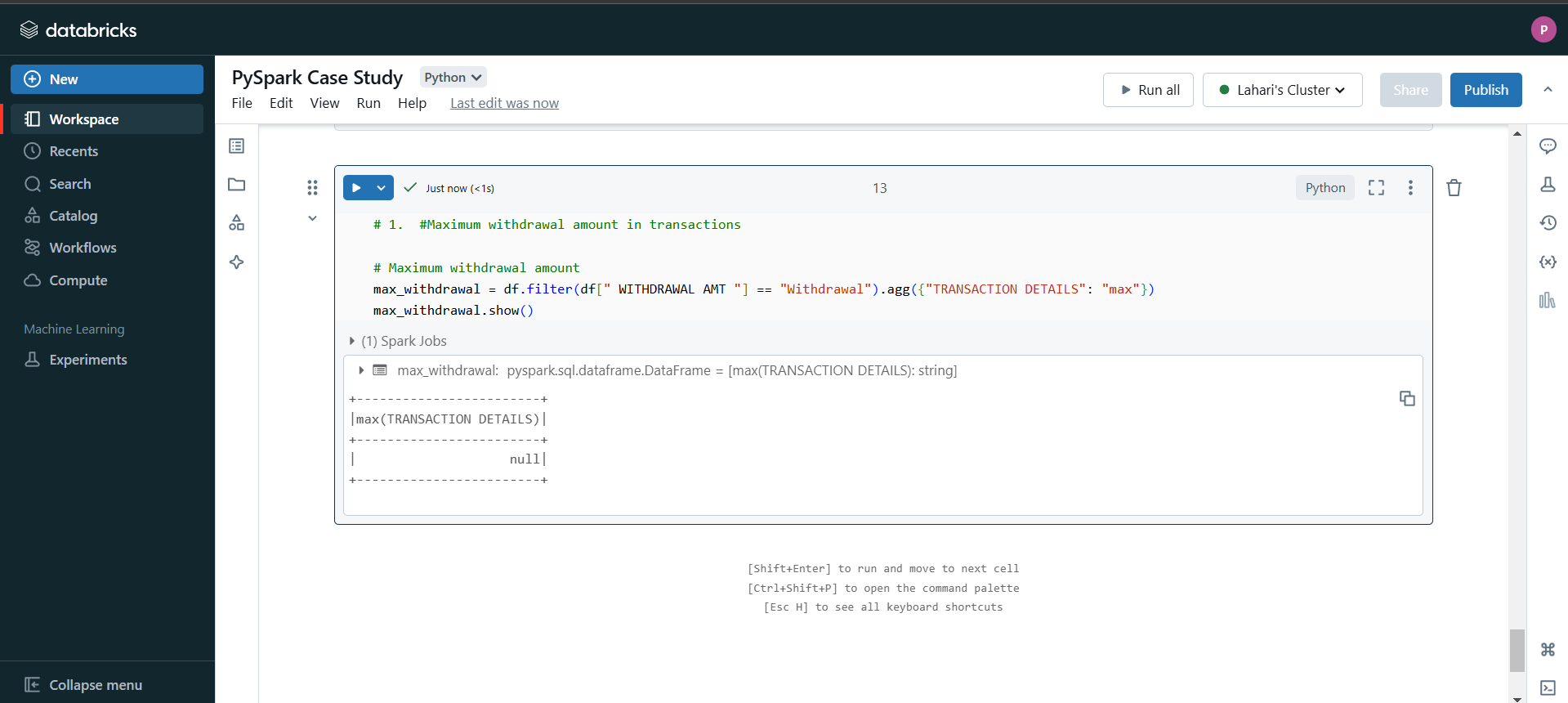
1. **Maximum withdrawal amount in transactions**

Code :

# Maximum withdrawal amount

max\_withdrawal = df.filter(df[" WITHDRAWAL AMT "] == "Withdrawal").agg({"TRANSACTION DETAILS": "max"})

max\_withdrawal.show()



**Explanation:**

* agg: Aggregates data using a specified function (max or min in this case).

"WITHDRAWAL AMT": "max": Finds the maximum value in the WITHDRAWAL AMT column.

1. **Minimum withdrawal amount :**

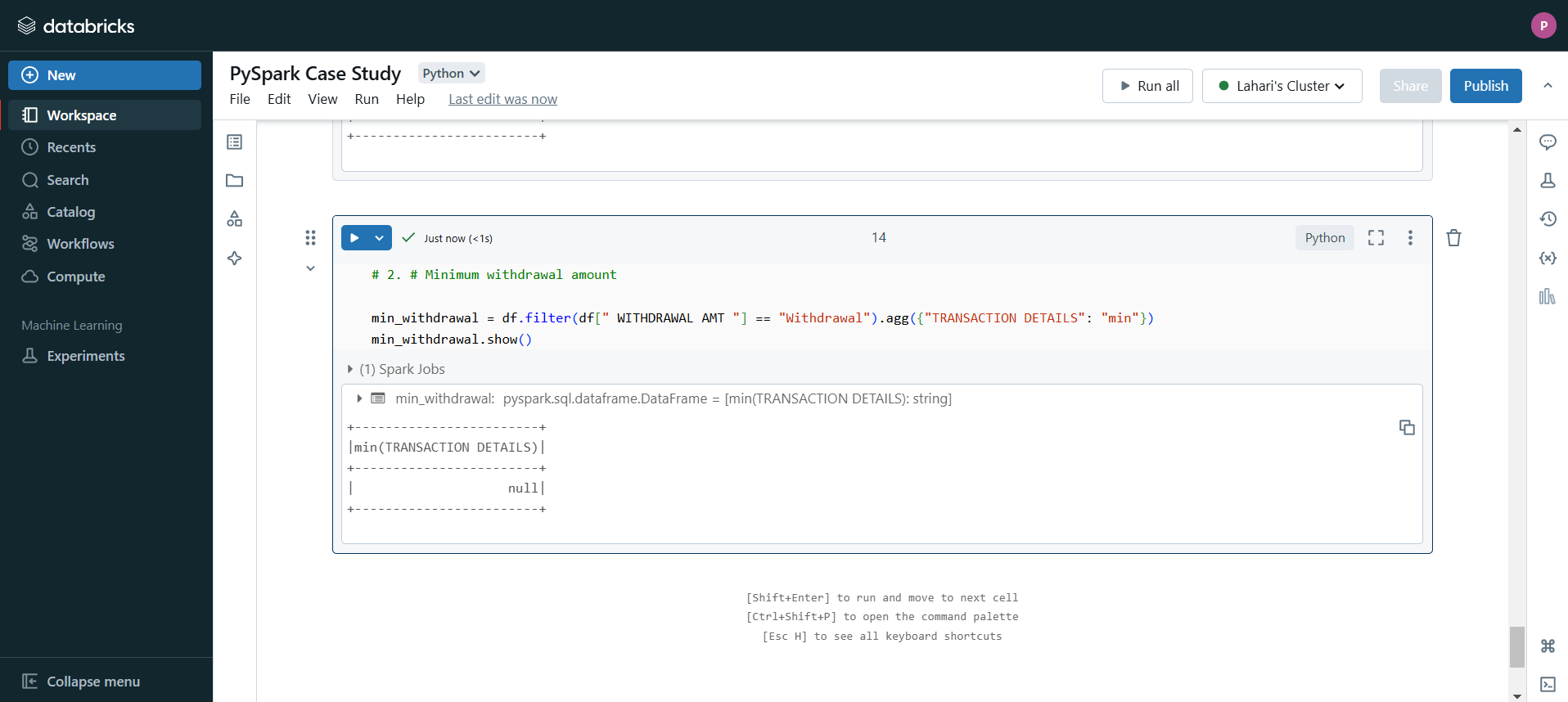
**Code :**

min\_withdrawal = df.filter(df[" WITHDRAWAL AMT "] == "Withdrawal").agg({"TRANSACTION DETAILS": "min"})

min\_withdrawal.show()

#### **Explanation:**

* agg: Aggregates data using a specified function (max or min in this case).
  + "WITHDRAWAL AMT": "min": Finds the minimum value in the WITHDRAWAL AMT column.



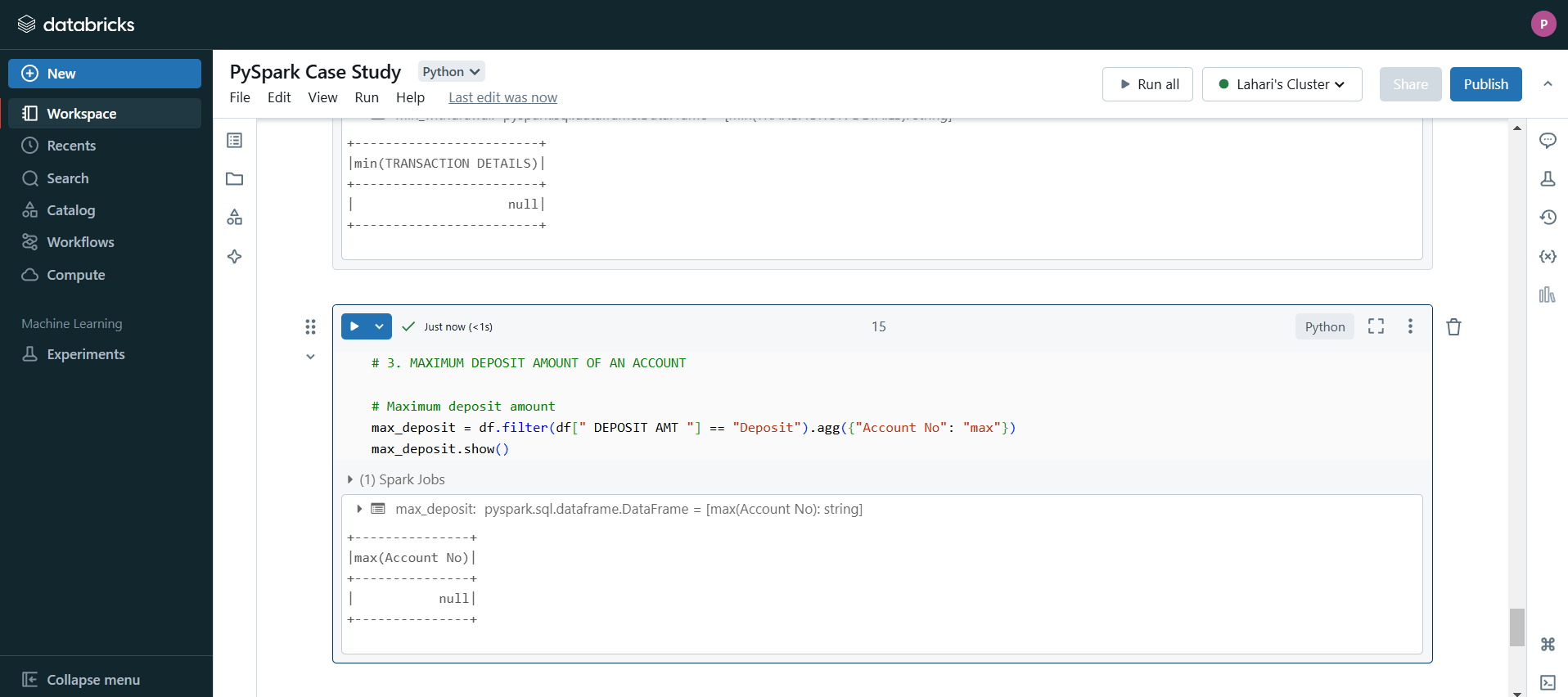
1. **MAXIMUM DEPOSIT AMOUNT OF AN ACCOUNT**

**Code :**

# Maximum deposit amount

max\_deposit = df.filter(df[" DEPOSIT AMT "] == "Deposit").agg({"Account No": "max"})

max\_deposit.show()



**Explanation:**

* Similar to withdrawals, this calculates the maximum and minimum values for the DEPOSIT AMT column using agg.

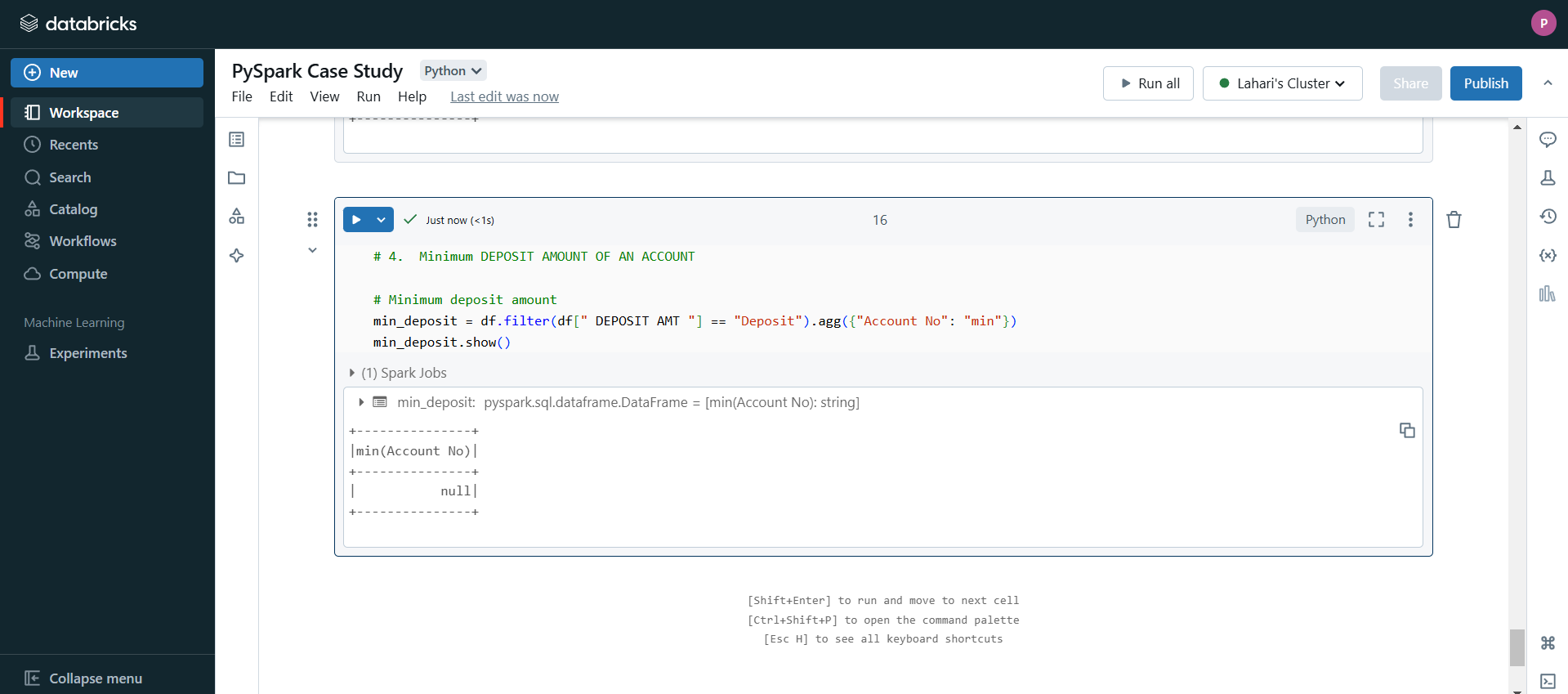
1. **Minimum DEPOSIT AMOUNT OF AN ACCOUNT**

**Code :**

# Minimum deposit amount

min\_deposit = df.filter(df[" DEPOSIT AMT "] == "Deposit").agg({"Account No": "min"})

min\_deposit.show()



#### **Explanation:**

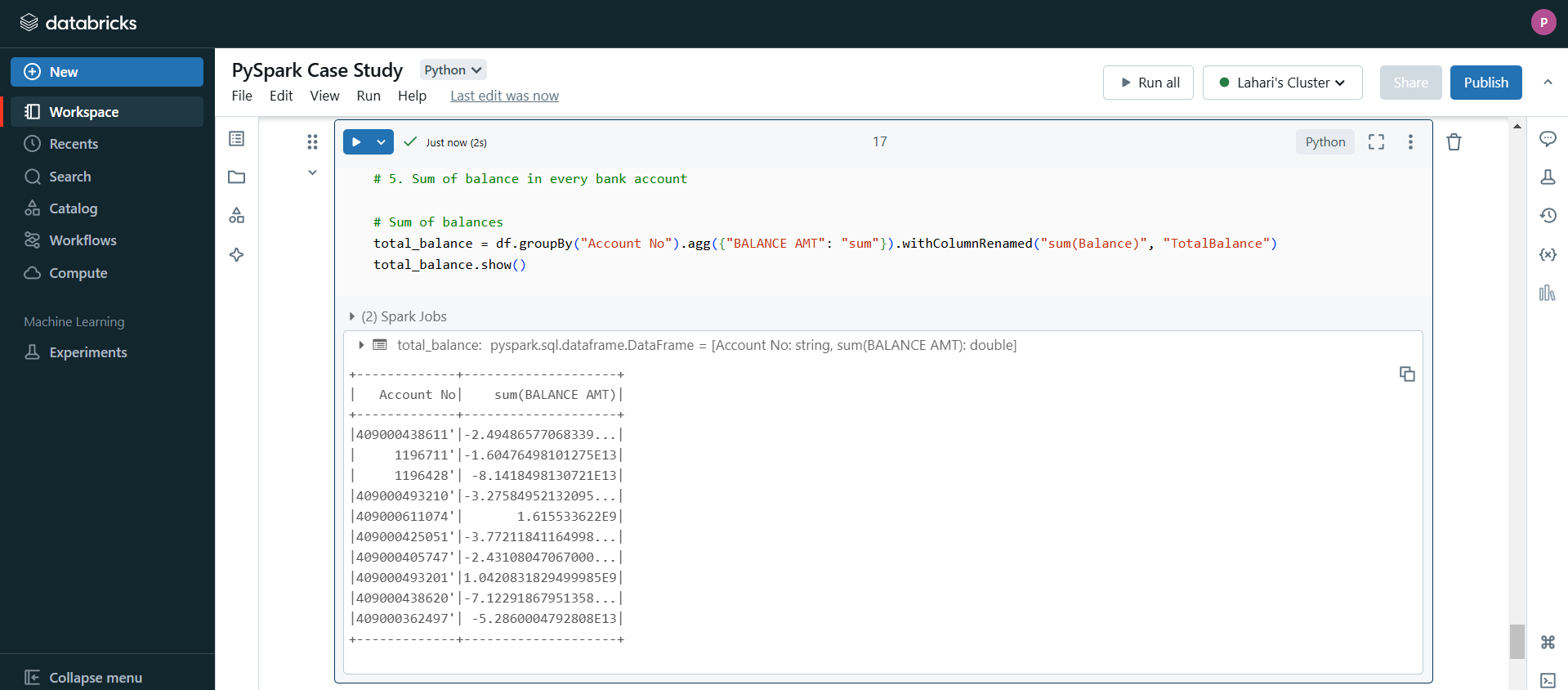
* Similar to withdrawals, this calculates the maximum and minimum values for the DEPOSIT AMT column using agg.

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

**Code :** # Sum of balances

total\_balance = df.groupBy("Account No").agg({"BALANCE AMT": "sum"}).withColumnRenamed("sum(Balance)", "TotalBalance")

total\_balance.show()

****

**Explanation:**

* groupBy("Account No"): Groups the data by the Account No column, treating each account as a unique group.
* agg({"BALANCE AMT": "sum"}): Computes the total (sum) of the BALANCE AMT column for each account group.
* withColumnRenamed: Renames the resulting column to TotalBalance.

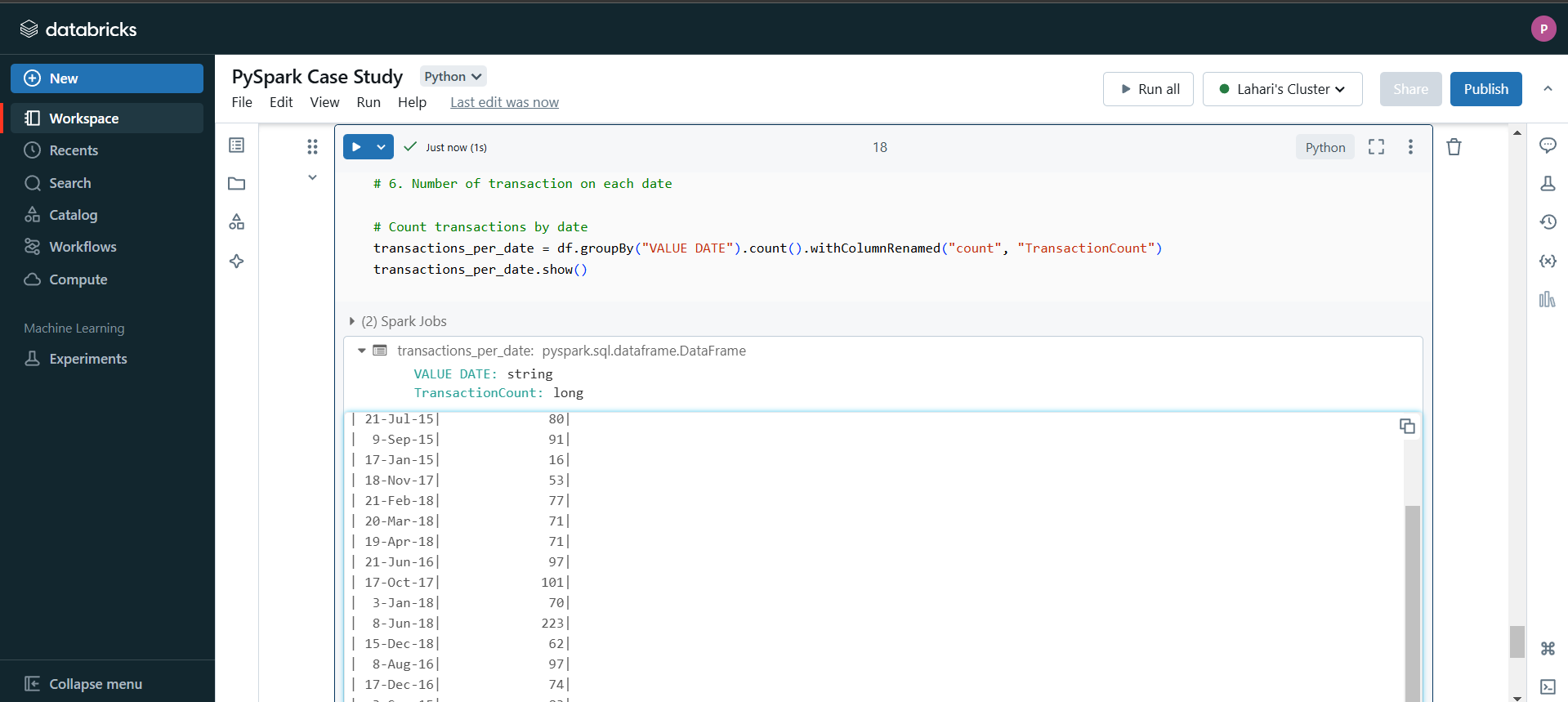
1. **Number of transaction on each date**

**Code :**

# Count transactions by date

transactions\_per\_date = df.groupBy("VALUE DATE").count().withColumnRenamed("count", "TransactionCount")

transactions\_per\_date.show()

****

#### **Explanation:**

* groupBy("VALUE DATE"): Groups the data by the VALUE DATE column, treating each date as a unique group.
* count(): Counts the number of transactions (rows) for each date.
* withColumnRenamed: Renames the resulting column to TransactionCount.

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

**Code :**

# Filter withdrawals greater than 1 lakh

high\_withdrawals = df.filter((df[" WITHDRAWAL AMT "] > 100000))

# Select relevant columns: Account No, WITHDRAWAL AMT, VALUE DATE

high\_withdrawals.select("Account No", " WITHDRAWAL AMT ", "VALUE DATE").show()

#### 

#### 

#### 

#### 

#### **Explanation:**

* .show(): Displays the results of each DataFrame. By default, it shows the first 20 rows of data in a tabular format.
* Each .show() statement corresponds to a specific result:
  + Maximum/Minimum Withdrawal: Displays the max and min withdrawal amounts.
  + Maximum/Minimum Deposit: Displays the max and min deposit amounts.
  + Total Balance per Account: Displays the sum of balances for each account.
  + Transactions per Date: Displays the count of transactions for each date.
  + High Withdrawals: Displays accounts with withdrawals greater than 1 lakh.