**PySpark and Spark SQL Coding Challenge**

**Name : Podutur Lahari - DE126**

**Date : 26-11-2024**

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

### **ETL Process Using PySpark**

ETL stands for Extract, Transform, Load, a common process in data engineering and analytics to move data from various sources to a centralized data repository. Here's a detailed explanation of the ETL process with PySpark, a Python-based API for Apache Spark, which is widely used for handling large-scale data processing.

**1. Extract**

The Extract step involves fetching raw data from various sources such as relational databases, APIs, CSV files, or other external systems. PySpark provides DataFrame APIs to efficiently load data into Spark for further processing.

**Key Methods in PySpark for Extraction:**

* spark.read.format("format"): Used for reading data in formats like CSV, JSON, or Parquet.
* jdbc: Connects to relational databases.
* hdfs: Fetches data from Hadoop Distributed File System.

Example :

from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder.appName("ETL\_Example").getOrCreate()

# Extract data from CSV

data = spark.read.csv("path/to/data.csv", header=True, inferSchema=True)

### **2. Transform**

The Transform step involves cleaning, enriching, and structuring the data to make it ready for analysis. This may include operations like filtering, joining datasets, changing formats, or performing aggregations.

**Key PySpark Operations:**

* Filtering: .filter()
* Grouping/Aggregations: .groupBy()
* Joins: .join()
* Adding Columns: .withColumn()
* Data Cleansing: Using SQL queries or Spark DataFrame APIs.

Example :

# Filter out rows with null values

cleaned\_data = data.filter(data['column\_name'].isNotNull())

# Add a new column

transformed\_data = cleaned\_data.withColumn("new\_column", data['existing\_column'] \* 2)

**3. Load**

The Load step involves saving the processed data into a target system, such as a database, data warehouse, or file system, in a suitable format for further use.

Key Methods in PySpark for Loading:

* .write.format("format"): Writes data in formats like Parquet, JSON, or CSV.
* .jdbc: Writes data to relational databases.
* save: Stores data into storage systems like HDFS or Amazon S3.

Example :

# Write transformed data to a Parquet file

transformed\_data.write.parquet("path/to/processed\_data.parquet")

**ETL Diagram :**

Here’s a visual representation of the ETL process with PySpark:

1. **Extract**: Data sources (CSV, APIs, Databases).
2. **Transform**: PySpark DataFrame operations (Filter, GroupBy, Join).
3. **Load**: Save data to a target system.



**Illustrative Example of ETL Using PySpark :**

#### **1. Extract**

This is the first step where raw data is collected from different sources, such as databases, APIs, or file systems. In PySpark, the read function provides options to load data in various formats like CSV, JSON, and Parquet.

* Purpose: To fetch data for processing.
* PySpark Example:

sales\_data = spark.read.csv("sales\_data.csv", header=True, inferSchema=True)

Here, a CSV file containing sales data is loaded into a DataFrame. The header=True argument ensures the first row is treated as column headers, and inferSchema=True automatically detects data types.

**Real-World Example**: Extracting sales reports from a database or fetching user activity logs from an API.

#### **2. Transform**

Once data is extracted, it often needs cleaning and reformatting. This step prepares the data for analysis by:

* Removing null or invalid values.
* Creating new features (e.g., calculating tax).
* Joining datasets for enrichment.
* Aggregating or summarizing data for analysis.
* **Purpose**: To cleanse and reshape data for meaningful analysis.
* **PySpark Example**:

# Filter valid sales entries

valid\_sales = sales\_data.filter(sales\_data["sales\_amount"] > 0)

# Add a tax column

taxed\_sales = valid\_sales.withColumn("tax", valid\_sales["sales\_amount"] \* 0.1)

* filter: Removes rows where the sales\_amount is less than or equal to 0.
* withColumn: Creates a new column tax by applying a transformation to the sales\_amount column.

**Real-World Example**: Cleaning e-commerce transaction data to remove canceled orders and adding calculated fields like shipping cost or tax.

#### **3. Load**

The final step involves storing the processed data into a target location, such as:

* A database for further querying.
* A data warehouse for business intelligence.
* A file system like HDFS or Amazon S3.
* **Purpose**: To make data available for analysis or reporting.
* **PySpark Example**:

taxed\_sales.write.jdbc(

url="jdbc:mysql://localhost:3306/salesdb",

table="processed\_sales",

mode="overwrite",

properties={"user": "root", "password": "password"})

* This stores the processed DataFrame into a MySQL table named processed\_sales.
* **Real-World Example**: Saving cleaned sales data into a central database for use by dashboards or machine learning models.

**2. Using Spark SQL and PySpark - Transformations such as Filter, Join, Simple Aggregations, GroupBy on the case study dataset**

**Step 1: Initialize PySpark session and Spark session :**

**Code :**

#Initialize PySpark session

from pyspark .sql import SparkSession

from pyspark.sql.functions import col, regexp\_replace

#Initialize Spark session

spark = SparkSession.builder.appName("SparkSQL\_CreditCard").getOrCreate()

# Step 1: Load the datasets

credit\_card\_path = "dbfs:/FileStore/tables/credit\_card\_\_1\_-1.csv"

txn\_path = "dbfs:/FileStore/tables/txn\_\_1\_-1.csv"

# Load credit card dataset

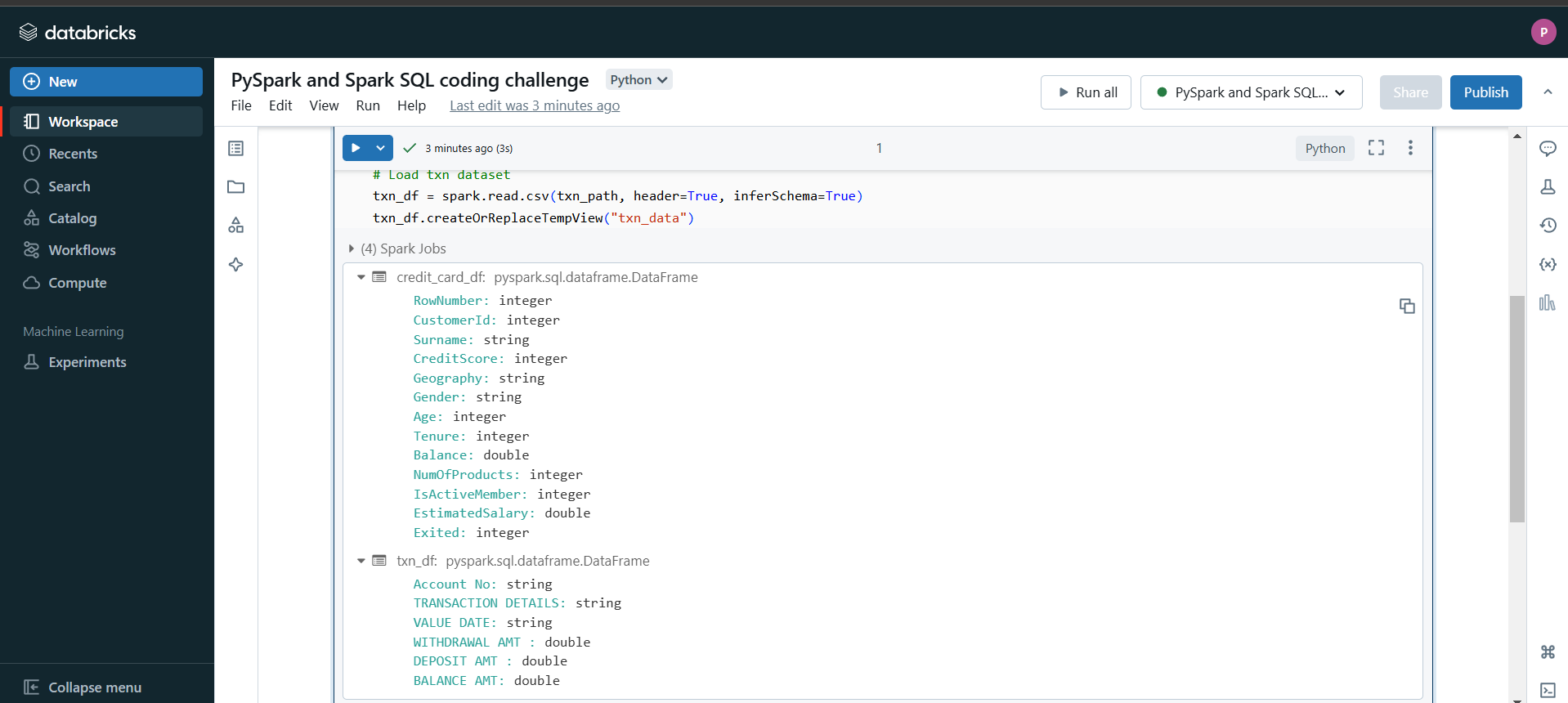
credit\_card\_df = spark.read.csv(credit\_card\_path, header=True, inferSchema=True)

credit\_card\_df.createOrReplaceTempView("credit\_card\_data")

# Load txn dataset

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

txn\_df.createOrReplaceTempView("txn\_data")

****

**Explanation :**

This code initializes a PySpark session and loads two datasets (credit\_card.csv and txn.csv) into PySpark DataFrames for analysis. Here's a concise explanation of the key steps:

1. Importing Libraries:  
   Essential PySpark modules (SparkSession, col, regexp\_replace) are imported to create a Spark session and process data.
2. Initialize Spark Session:  
   A PySpark session is started using SparkSession.builder, naming the application as "SparkSQL\_CreditCard".
3. Load Datasets:
   * Paths to the CSV files (credit\_card\_path and txn\_path) are defined.
   * credit\_card\_df and txn\_df are created by reading the respective CSV files.
     + header=True ensures the first row is treated as column headers.
     + inferSchema=True enables automatic detection of column data types.
4. Create Temporary SQL Views:
   * createOrReplaceTempView registers the DataFrames (credit\_card\_df and txn\_df) as temporary SQL views (credit\_card\_data and txn\_data), enabling SQL queries on them.

The setup is now ready for querying and further transformations using PySpark or SparkSQL.

**Step 2 : Inspect Datasets**

**Code :**

print("Credit Card Dataset Schema:")

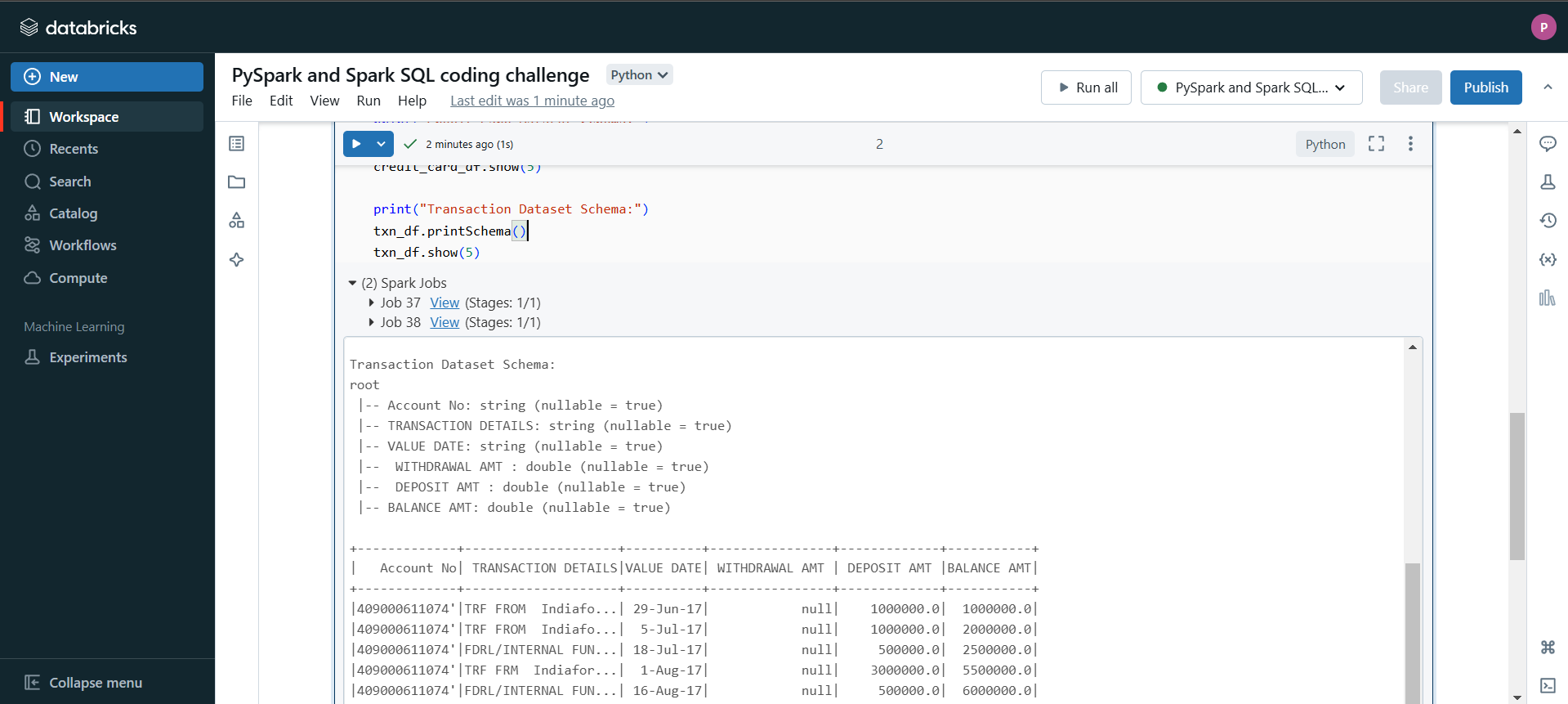
credit\_card\_df.printSchema()

credit\_card\_df.show(5)

print("Transaction Dataset Schema:")

txn\_df.printSchema()

txn\_df.show(5)



**Explanation :**

Credit Card Dataset:

* credit\_card\_df.printSchema() prints the schema of the credit\_card\_df DataFrame, displaying the column names and their data types.
* credit\_card\_df.show(5) displays the first 5 rows of the credit\_card\_df DataFrame.

Transaction Dataset:

* txn\_df.printSchema() prints the schema of the txn\_df DataFrame, showing the column names and data types.
* txn\_df.show(5) displays the first 5 rows of the txn\_df DataFrame.

**Step 3 : Filter Transformation**

**Code :**

# PySpark

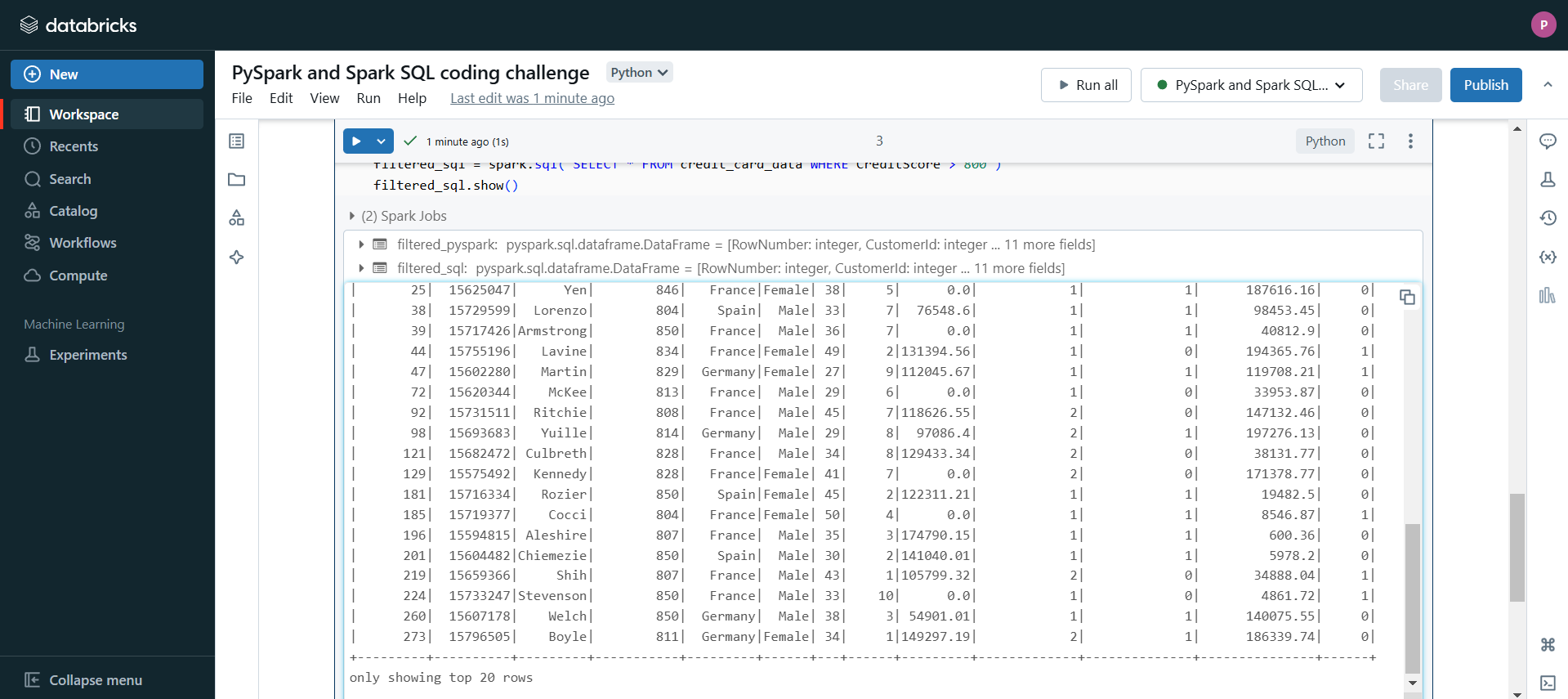
filtered\_pyspark = credit\_card\_df.filter(credit\_card\_df['CreditScore'] > 800)

filtered\_pyspark.show()

# Spark SQL

filtered\_sql = spark.sql("SELECT \* FROM credit\_card\_data WHERE CreditScore > 800")

filtered\_sql.show()

****

**Explanation :**

PySpark DataFrame API:

* credit\_card\_df.filter(credit\_card\_df['Amount'] > 1000) filters the credit\_card\_df DataFrame to include only rows where the Amount column is greater than 1000.
* filtered\_pyspark.show() displays the filtered result.

Spark SQL:

* spark.sql("SELECT \* FROM credit\_card\_data WHERE Amount > 1000") runs a SQL query on the registered credit\_card\_data view to select rows where the Amount is greater than 1000.
* filtered\_sql.show() displays the result of the SQL query.

**Step 4 : Aggregations**

**Code :**

# PySpark

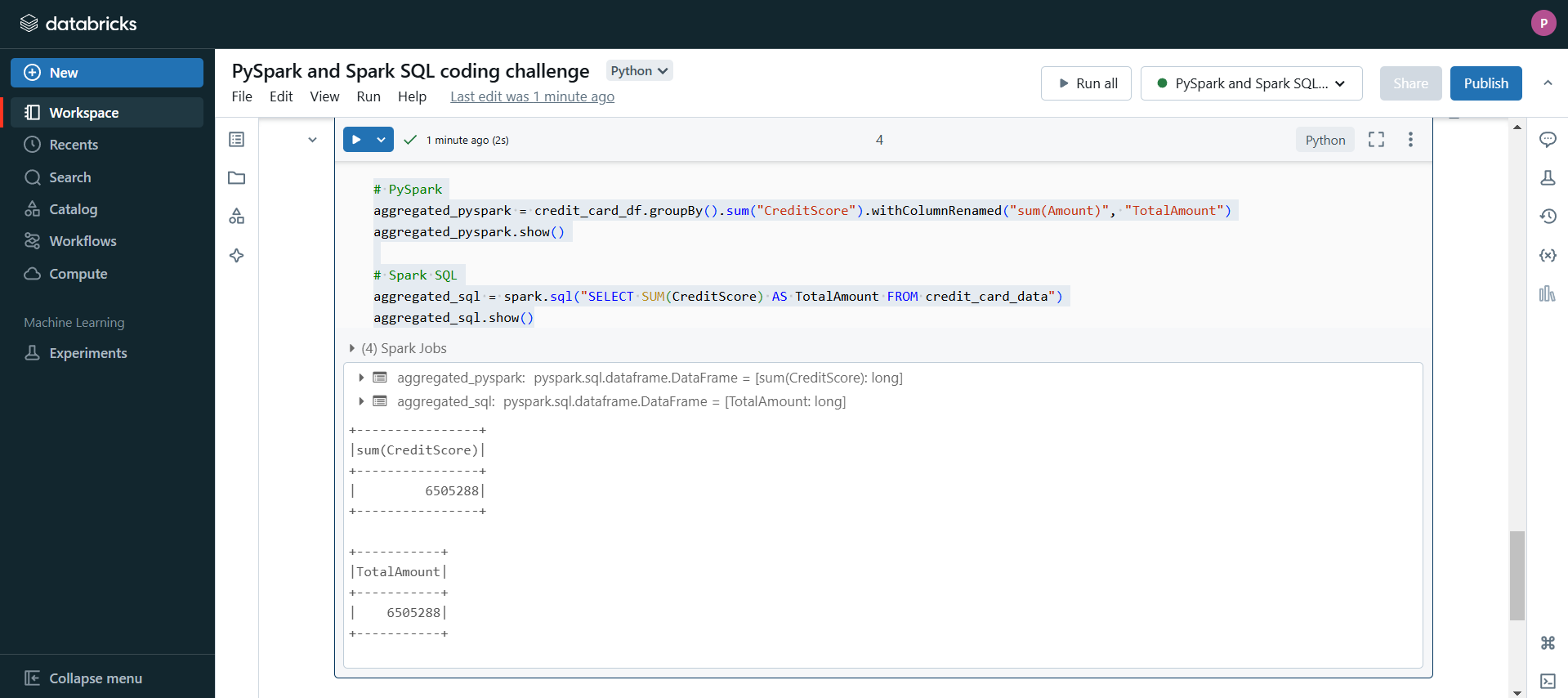
aggregated\_pyspark = credit\_card\_df.groupBy().sum("CreditScore").withColumnRenamed("sum(Amount)", "TotalAmount")

aggregated\_pyspark.show()

# Spark SQL

aggregated\_sql = spark.sql("SELECT SUM(CreditScore) AS TotalAmount FROM credit\_card\_data")

aggregated\_sql.show()



**Explanation :**

PySpark DataFrame API:

* credit\_card\_df.groupBy().sum("Amount") groups the data (without any specific grouping column) and calculates the sum of the Amount column.
* .withColumnRenamed("sum(Amount)", "TotalAmount") renames the resulting sum column to TotalAmount.
* aggregated\_pyspark.show() displays the result.

Spark SQL:

* spark.sql("SELECT SUM(Amount) AS TotalAmount FROM credit\_card\_data") executes a SQL query to calculate the sum of the Amount column and renames it to TotalAmount.
* aggregated\_sql.show() displays the result.

**Step 5 : GroupBy**

**Code :** # PySpark

grouped\_pyspark = credit\_card\_df.groupBy("CreditScore").sum("Balance").withColumnRenamed("sum(Amount)", "TotalAmount")

grouped\_pyspark.show()

# Spark SQL

grouped\_sql = spark.sql("""

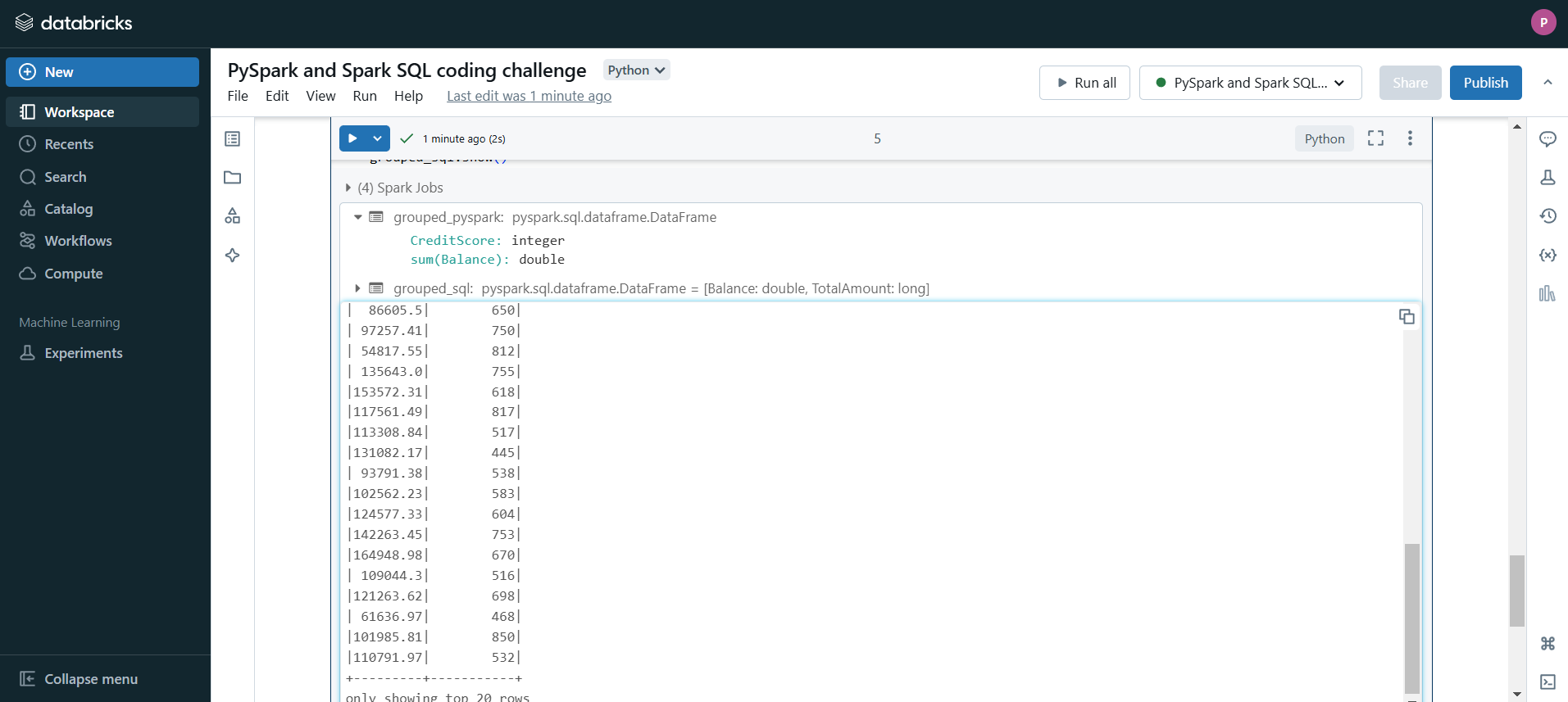
SELECT Balance, SUM(CreditScore) AS TotalAmount

FROM credit\_card\_data

GROUP BY Balance

""")

grouped\_sql.show()



**Explanation :**

PySpark DataFrame API:

* credit\_card\_df.groupBy("CreditScore").sum("Balance") groups the data by the CreditScore column and calculates the sum of the Balance column for each credit score.

Spark SQL:

Spark.sql : ("""SELECT Balance, SUM(CreditScore) AS TotalAmount FROM credit\_card\_data GROUP BY Balance""") executes a SQL query where the data is grouped by the Balance column and the sum of the CreditScore column is calculated for each balance.

**Step 6 : Joins**

**Code :**

# PySpark Inner Join

inner\_join\_pyspark = credit\_card\_df.join(txn\_df, credit\_card\_df["CreditScore"] == txn\_df["TRANSACTION DETAILS"], "inner")

inner\_join\_pyspark.show(5)

# Spark SQL Inner Join

credit\_card\_df.createOrReplaceTempView("credit\_card\_data")

txn\_df.createOrReplaceTempView("txn\_data")

# Perform the join using Spark SQL

inner\_join\_sql = spark.sql("""

SELECT ccd.CreditScore, txn.`TRANSACTION DETAILS`

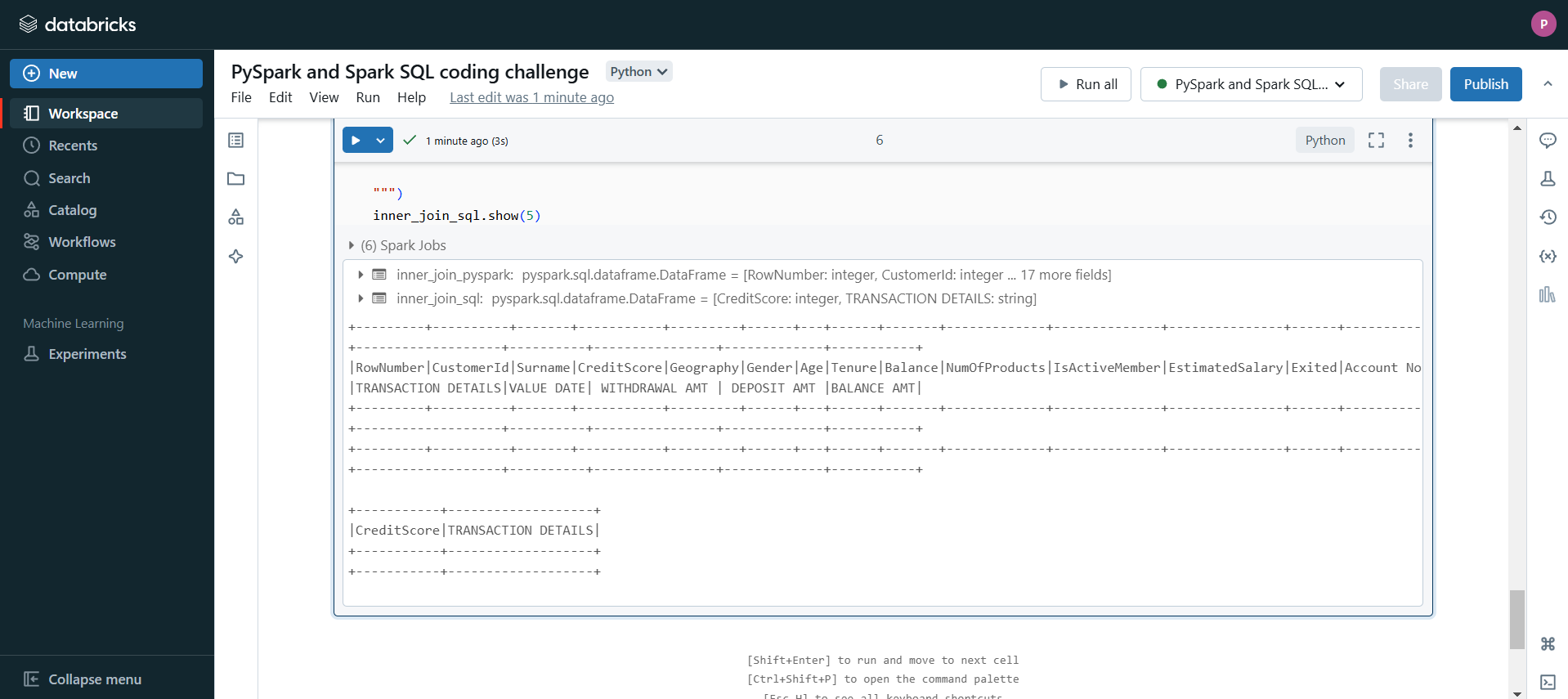
FROM credit\_card\_data ccd

INNER JOIN txn\_data txn

ON ccd.CreditScore = txn.`TRANSACTION DETAILS`

""")

inner\_join\_sql.show(5)



**Explanation :**

PySpark Inner Join:

* Joins credit\_card\_df and txn\_df on the condition where the CreditScore from credit\_card\_df matches the TRANSACTION DETAILS from txn\_df.
* Displays the first 5 rows of the result.

Spark SQL Inner Join:

* Registers the DataFrames as temporary views (credit\_card\_data and txn\_data).
* Executes a SQL query that joins the two tables on the same condition (CreditScore = TRANSACTION DETAILS).
* Displays the first 5 rows of the SQL query result.

**Step 7 : Left Join**

**Code :**

# Left join in PySpark

left\_join\_pyspark = credit\_card\_df.join(txn\_df, credit\_card\_df["CreditScore"] == txn\_df["TRANSACTION DETAILS"], "left")

left\_join\_pyspark.show(5)

# Left join in Spark SQL

left\_join\_sql = spark.sql("""

SELECT ccd.CreditScore, txn.`TRANSACTION DETAILS`

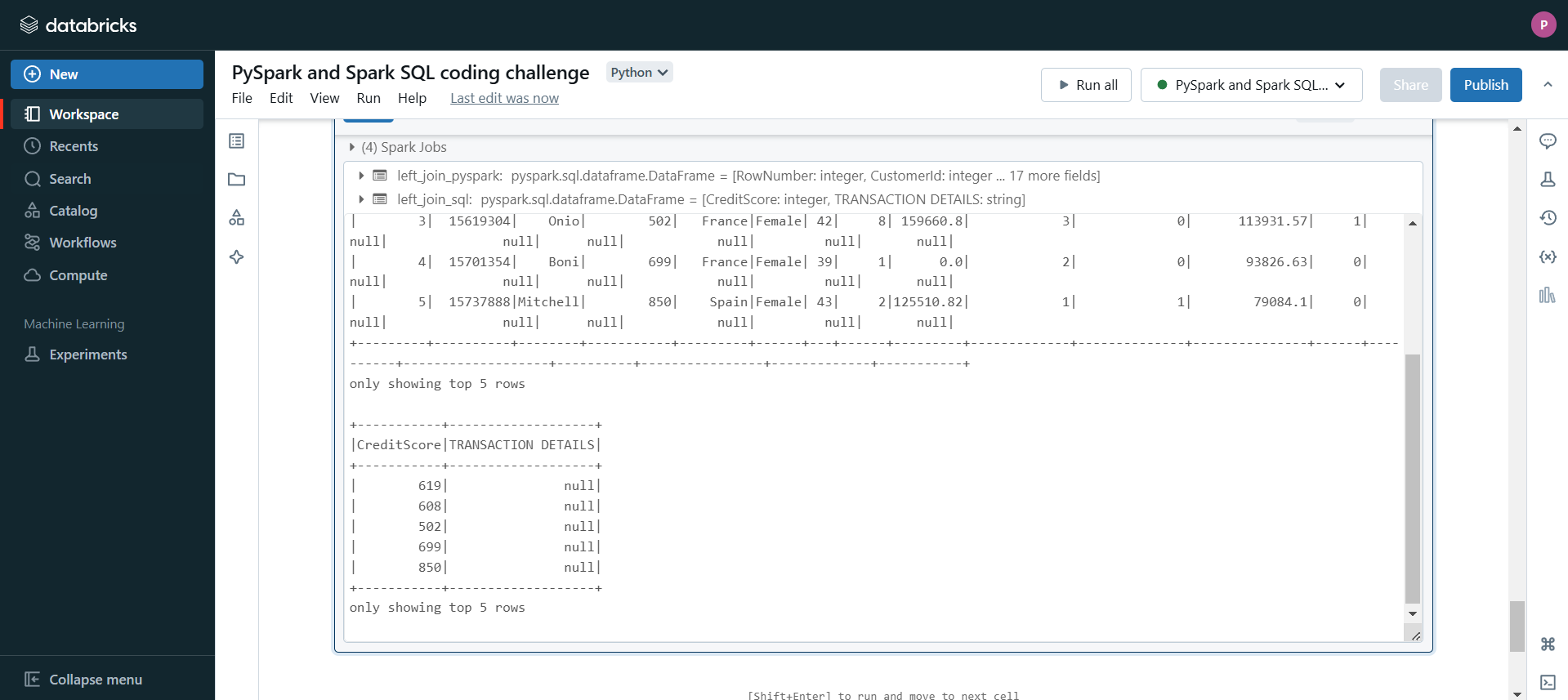
FROM credit\_card\_data ccd

LEFT JOIN txn\_data txn

ON ccd.CreditScore = txn.`TRANSACTION DETAILS`

""")

left\_join\_sql.show(5)



**Explanation :**

PySpark: Joins using CreditScore and TRANSACTION DETAILS, keeping all rows from credit\_card\_df.

Spark SQL: Executes a similar left join query using SQL syntax.

**Step 8 : Right Join**

**Code :**

# Right join in PySpark

right\_join\_pyspark = credit\_card\_df.join(txn\_df, credit\_card\_df["CreditScore"] == txn\_df["TRANSACTION DETAILS"], "right")

right\_join\_pyspark.show(5)

# Right join in Spark SQL

right\_join\_sql = spark.sql("""

SELECT ccd.CreditScore, txn.`TRANSACTION DETAILS`

FROM credit\_card\_data ccd

RIGHT JOIN txn\_data txn

ON ccd.CreditScore = txn.`TRANSACTION DETAILS`

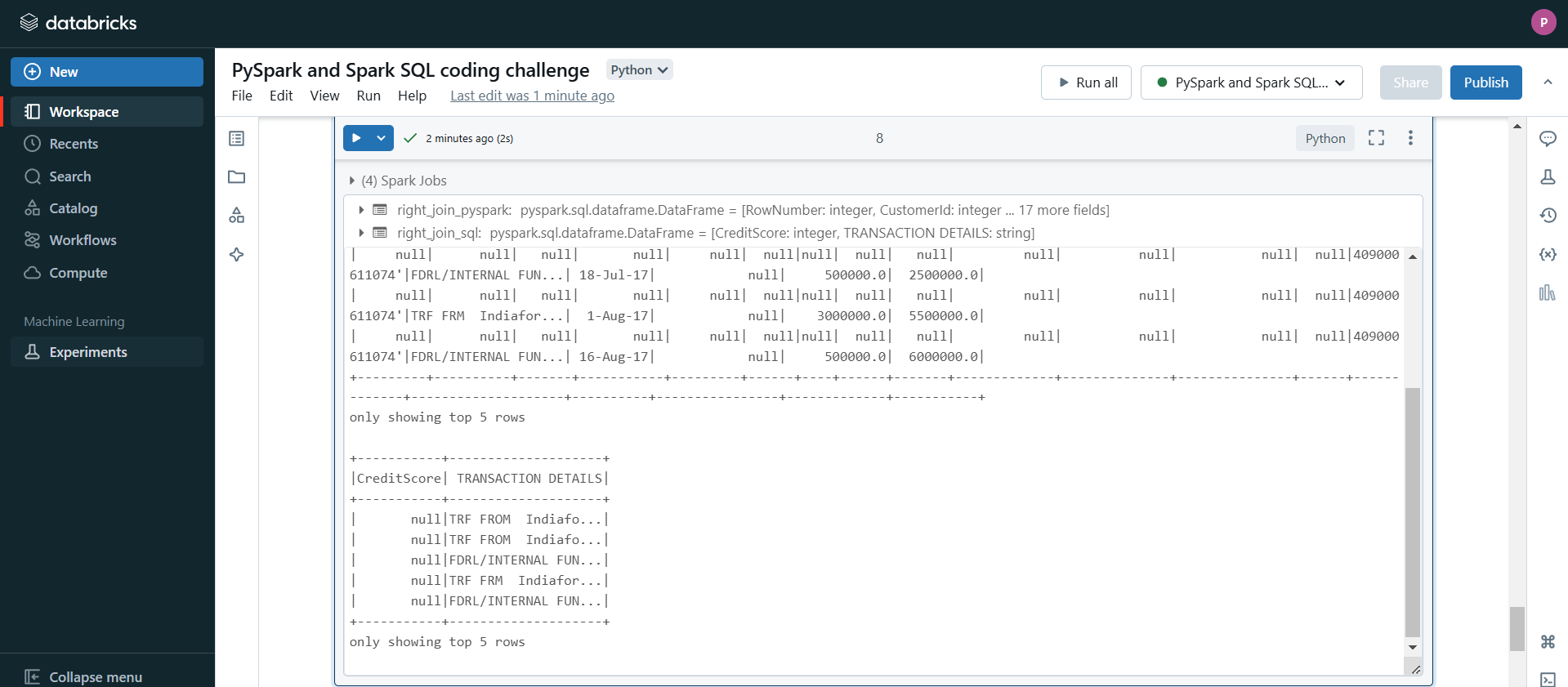
""")

right\_join\_sql.show(5)

**Explanation :**

PySpark: Joins using CreditScore and TRANSACTION DETAILS, keeping all rows from txn\_df.

Spark SQL: Executes a right join query using SQL syntax.



**Step 9 : Outer Join**

**Code :**

# Outer join in PySpark

outer\_join\_pyspark = credit\_card\_df.join(txn\_df, credit\_card\_df["CreditScore"] == txn\_df["TRANSACTION DETAILS"], "outer")

outer\_join\_pyspark.show(5)

# Outer join in Spark SQL

outer\_join\_sql = spark.sql("""

SELECT ccd.CreditScore, txn.`TRANSACTION DETAILS`

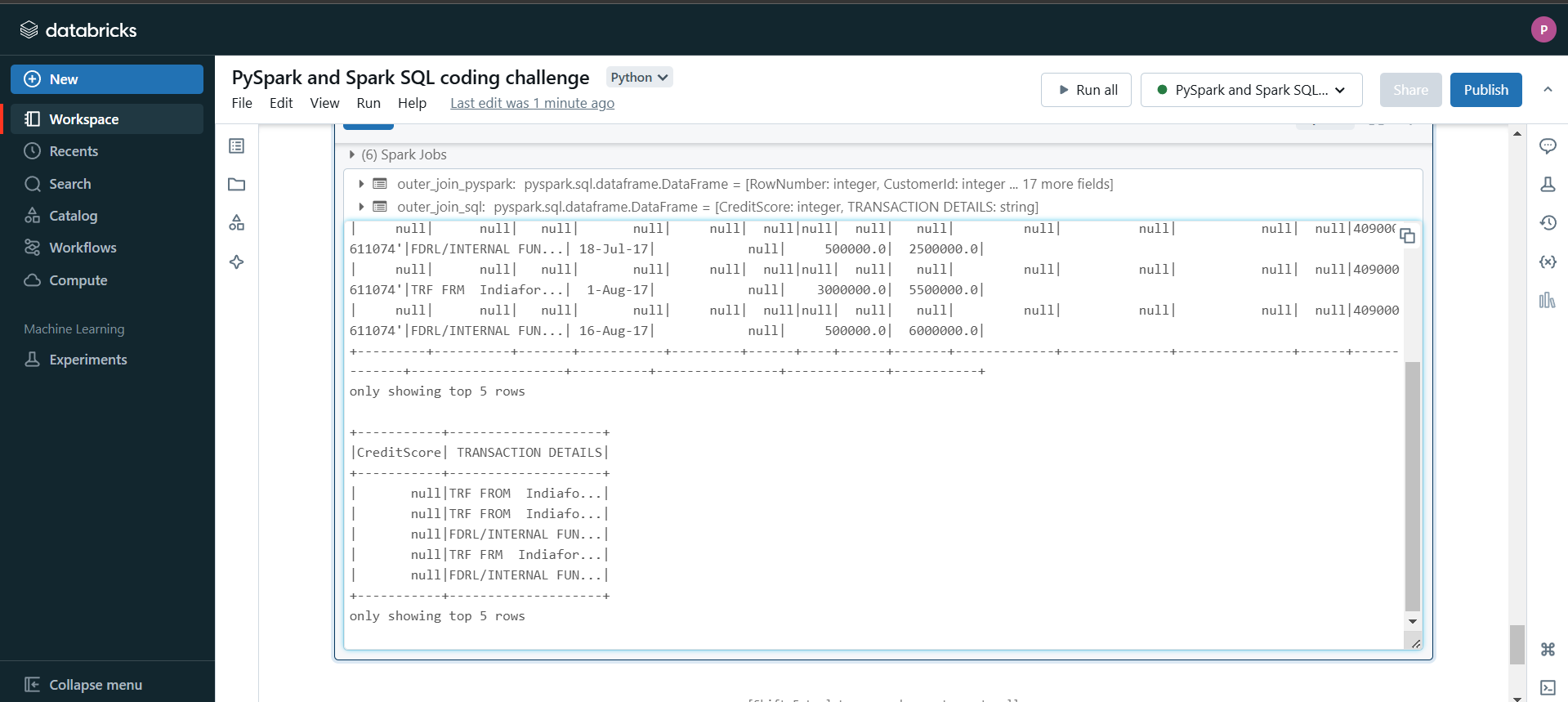
FROM credit\_card\_data ccd

FULL OUTER JOIN txn\_data txn

ON ccd.CreditScore = txn.`TRANSACTION DETAILS`

""")

outer\_join\_sql.show(5)



**Explanation :**

Outer Join in PySpark:

* The .join() method is used with the outer join type.
* This returns all rows from both credit\_card\_df and txn\_df. If there’s no match, the result will have null values in the columns where there is no corresponding data.
* Displays the first 5 rows of the result.

Outer Join in Spark SQL:

* The SQL query uses FULL OUTER JOIN to combine all rows from both tables (credit\_card\_data and txn\_data), with null values where there is no match.
* Displays the first 5 rows of the SQL query result.

**Step 10 : Cross Join**

**Code :**

# Perform a cross join in PySpark

cross\_join\_pyspark = credit\_card\_df.crossJoin(txn\_df)

cross\_join\_pyspark.show(5)

# Register the DataFrames as SQL temporary views

credit\_card\_df.createOrReplaceTempView("credit\_card\_data")

txn\_df.createOrReplaceTempView("txn\_data")

# Perform a cross join in Spark SQL

cross\_join\_sql = spark.sql("""

SELECT \*

FROM credit\_card\_data ccd

CROSS JOIN txn\_data txn

""")

cross\_join\_sql.show(5)

**Explanation :**

Cross Join in PySpark:

* The .crossJoin() method is used to perform a Cartesian join, meaning it returns the combination of every row from credit\_card\_df with every row from txn\_df.
* This can result in a large number of rows if both DataFrames have many records, as the number of rows in the result will be the product of the number of rows in each DataFrame.
* Displays the first 5 rows of the result.

Cross Join in Spark SQL:

* The SQL query uses CROSS JOIN to achieve the same Cartesian product between credit\_card\_data and txn\_data.
* Displays the first 5 rows of the SQL query result.