**PYSPARK CODING CHALLENGE**

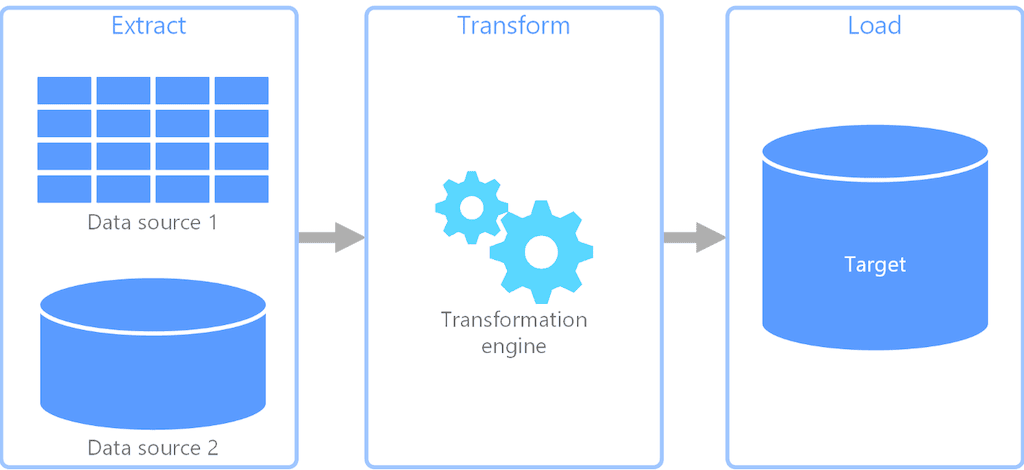
**Name:Tejaswini Gokanakonda**

**Roll no**:**DE142**

**Date: 26-11-2024**

**Q1. Explain ETL (Extract, Transform, Load) with PySpark(in your own words)**

ETL (Extract, Transform, Load) is a core process in data engineering, used to prepare data for analysis and decision-making. With PySpark, a powerful tool for big data processing, ETL becomes highly efficient and scalable.



**Extract :**

The first step involves collecting data from various sources, such as relational databases, cloud storage, APIs, or flat files. PySpark provides flexible APIs to read data in formats like CSV, JSON, Parquet, Avro, and more. It can connect to distributed systems like Hadoop, AWS S3, or databases using JDBC. Since PySpark works in a distributed environment, it can handle very large datasets during the extraction process without performance bottlenecks.

### **Transform :**

Once data is extracted, it often needs cleaning, formatting, and enrichment to become useful. This step may involve handling missing values, converting data types, filtering rows, joining datasets, or aggregating data. PySpark’s DataFrame API offers a wide range of transformations that are intuitive and optimized for parallel processing. Operations like filtering (filter()), grouping (groupBy()), and adding computed columns (withColumn()) allow you to shape the data effectively. PySpark also supports advanced transformations like working with nested data structures or applying custom functions using UDFs (User-Defined Functions).

**Load :**

The final step is loading the transformed data into a target system, such as a data warehouse, relational database, or another file system. PySpark supports writing data in various formats (CSV, JSON, Parquet) and integrates seamlessly with storage systems like HDFS, AWS S3, and Azure Blob Storage.

PySpark’s distributed nature ensures that ETL pipelines are fast and scalable, handling gigabytes to terabytes of data effortlessly. By leveraging PySpark for ETL, organizations can build reliable and efficient data workflows that process and deliver insights from massive datasets in real-time or batch processing modes.

In addition to its core functionality, PySpark provides significant advantages for modern ETL pipelines by integrating with the broader Apache Spark ecosystem.

### **Benefits of PySpark for ETL :**

1. **Scalability:** PySpark operates on distributed clusters, enabling it to handle massive datasets across multiple nodes. This makes it ideal for organizations working with large-scale data.
2. **Flexibility:** PySpark supports structured, semi-structured, and unstructured data, allowing it to adapt to diverse data sources and formats.
3. **Performance:** Spark’s in-memory computation optimizes ETL tasks, making them faster compared to traditional disk-based processing systems.
4. **Fault Tolerance:** PySpark provides automatic recovery mechanisms, ensuring the ETL process is resilient to hardware or network failures.
5. **Ease of Use:** PySpark combines the simplicity of Python with Spark’s robust backend, making it accessible to data engineers and scientists.

### **PySpark for Advanced ETL Scenarios :**

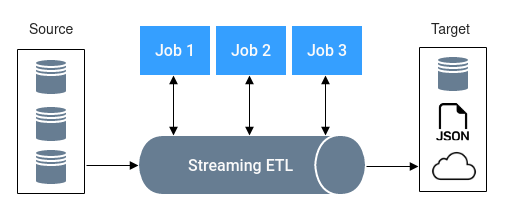
PySpark is not limited to simple ETL workflows; it can handle advanced scenarios like:

* **Streaming Data:** With Spark Streaming, PySpark can process real-time data for use cases like event monitoring or live dashboards.
* **Machine Learning:** PySpark integrates with Spark MLlib, enabling the inclusion of data modeling and predictions within ETL pipelines.
* **Graph Processing:** For use cases like social network analysis, PySpark can leverage Spark’s GraphX library.

**Advanced ETL Topics with PySpark :**

**Working with Streaming Data**  
PySpark’s **Structured Streaming** API allows you to process real-time data, making it ideal for live data pipelines such as monitoring, event processing, or financial transactions.

* **Real-Time ETL Pipelines:** Streaming data can be ingested from Kafka, file systems, or sockets and processed using similar transformations as batch jobs.
* **Windowed Operations:** You can perform aggregations over time windows to handle real-time data streams.



**Incremental ETL**  
In many ETL pipelines, it’s not feasible to process the entire dataset repeatedly, especially when new data is being added constantly. In such cases, **incremental ETL** is used, where only new or updated records are processed.

* **Change Data Capture (CDC):** You can implement CDC techniques to track changes in source data and apply transformations only to changed records.
* **Timestamp-Based Processing:** Use a timestamp column to identify new or modified records since the last run.

### **Conclusion :**

PySpark is an essential tool for modern ETL processes, combining speed, scalability, and flexibility. Whether dealing with batch jobs or real-time pipelines, PySpark simplifies the entire ETL lifecycle, helping organizations transform raw data into actionable insights.

**Q2. Using Spark SQL - Transformations such as Filter, Join, Simple Aggregations, GroupBy on the case study dataset.**

### **1. Basic Setup**

from pyspark.sql.session import SparkSession

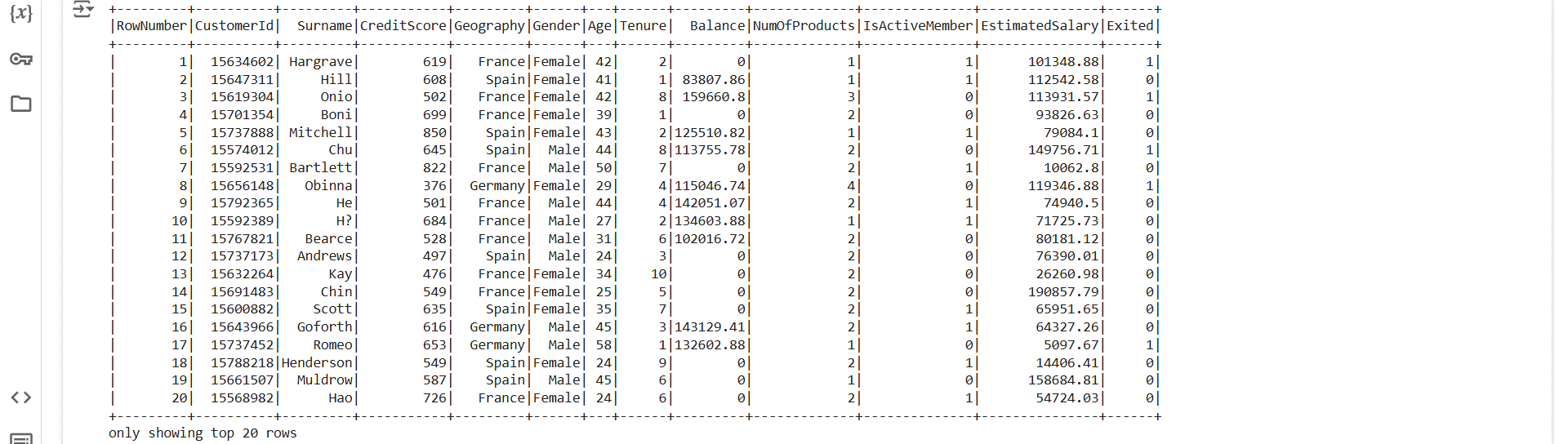
spark = SparkSession.builder.getOrCreate()

credit\_card\_data = spark.read.csv(path='credit card.csv', header=True)

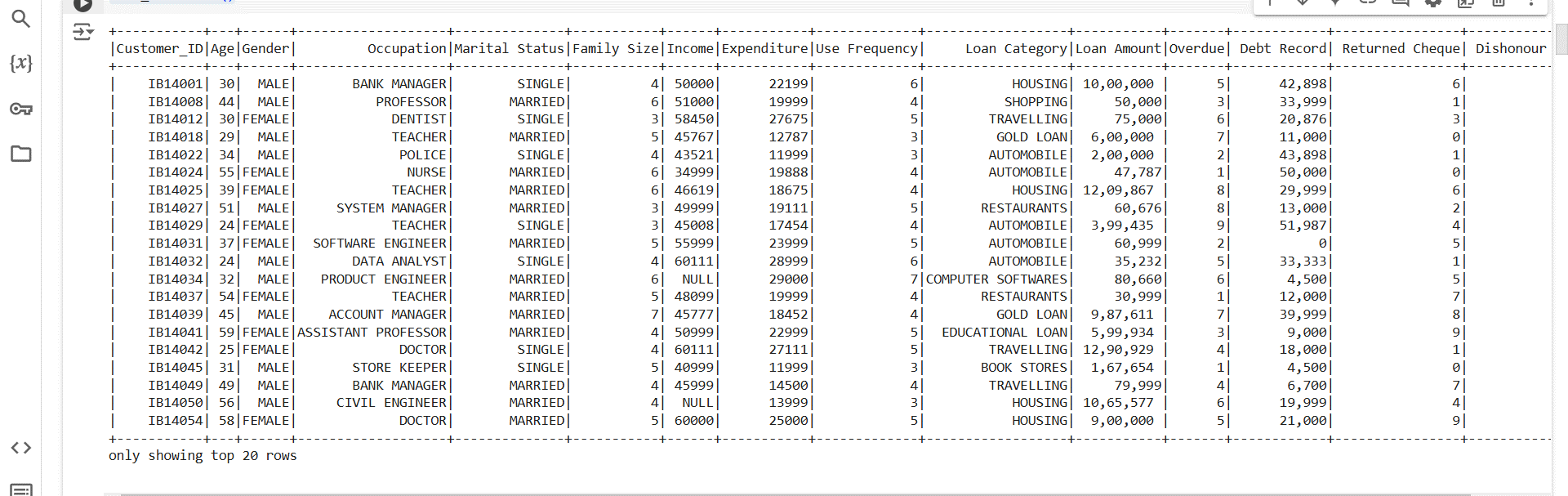
loan\_data = spark.read.csv(path='loan.csv', header=True)

txn\_data = spark.read.csv(path='txn.csv', header=True)

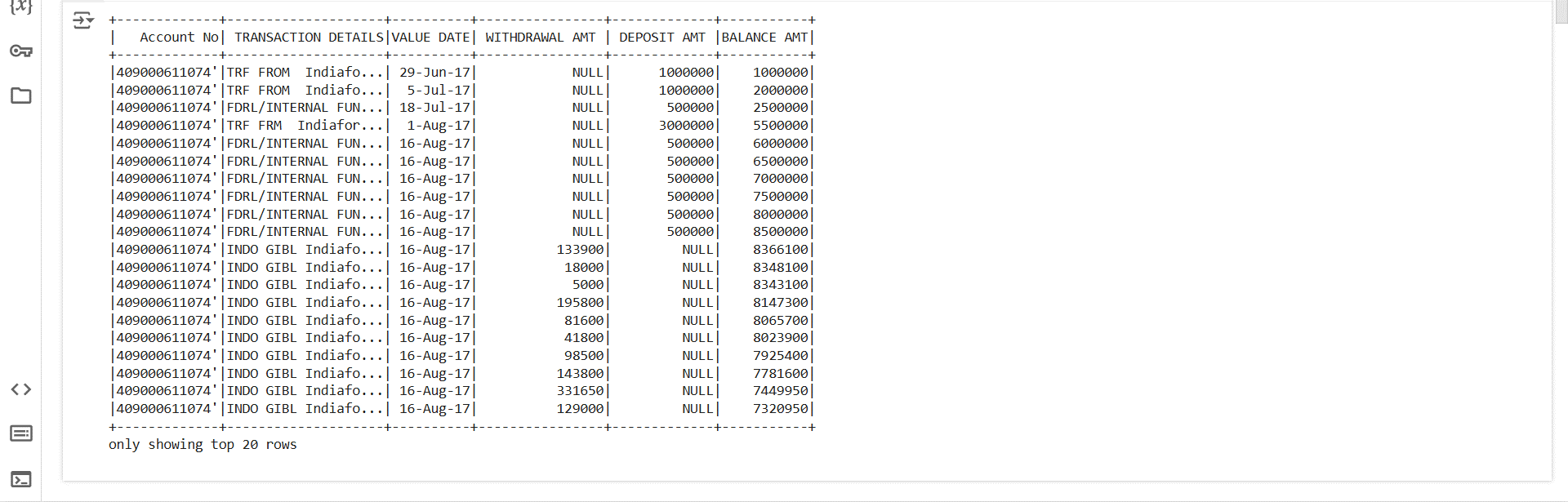
credit\_card\_data.show()



loan\_data.show()



txn\_data.show()



* **SparkSession**:
  + A SparkSession is initialized using SparkSession.builder.getOrCreate(), which either creates a new Spark session or gets an existing one.
* **Reading CSV Files**:
  + The spark.read.csv method is used to read CSV files into DataFrames. The header=True option means that the first row is used as column headers.
* **show()**:
  + This function displays the first 20 rows of the DataFrame, giving a preview of the data.

### **2. Registering DataFrames as Temporary Views**

# Registering DataFrames as temporary views

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

loan\_data.createOrReplaceTempView("loan")

txn\_data.createOrReplaceTempView("transaction")

* **CreateOrReplaceTempView**:

This function registers the DataFrames as temporary views in Spark SQL, allowing you to run SQL queries against them. These views exist only for the duration of the session.

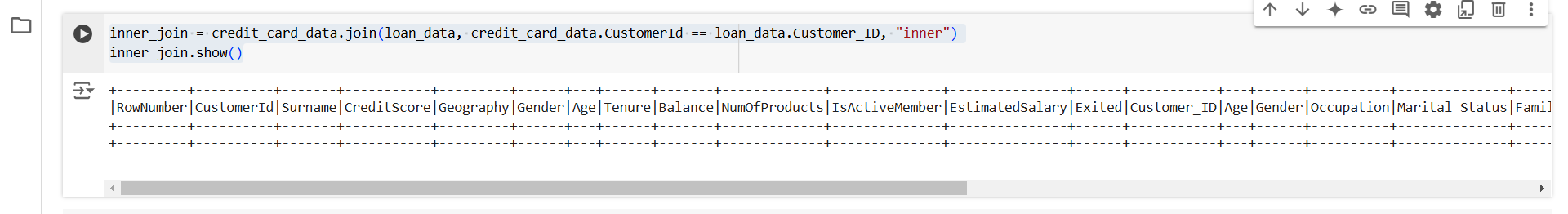
### **3. Joins**

#### **a) Inner Join**

-**Inner Join Join credit\_card\_data with loan\_data using CustomerId and Customer\_ID as the key.**

inner\_join = credit\_card\_data.join(loan\_data, credit\_card\_data.CustomerId == loan\_data.Customer\_ID, "inner")

inner\_join.show()



query = """

SELECT \*

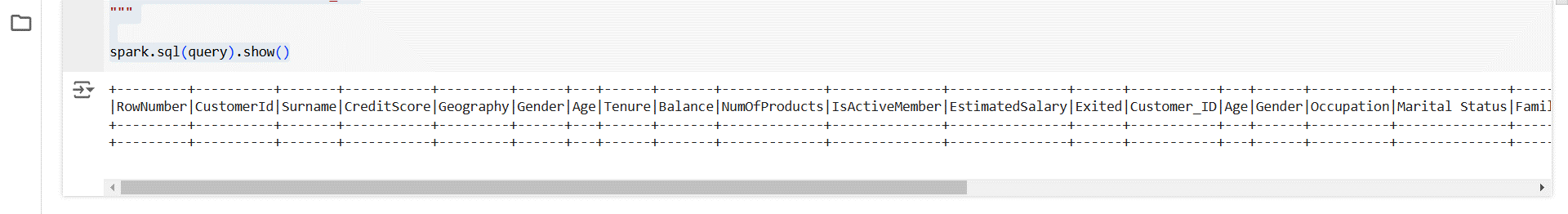
FROM credit\_card c

INNER JOIN loan l

ON c.CustomerId = l.Customer\_ID

"""

spark.sql(query).show()



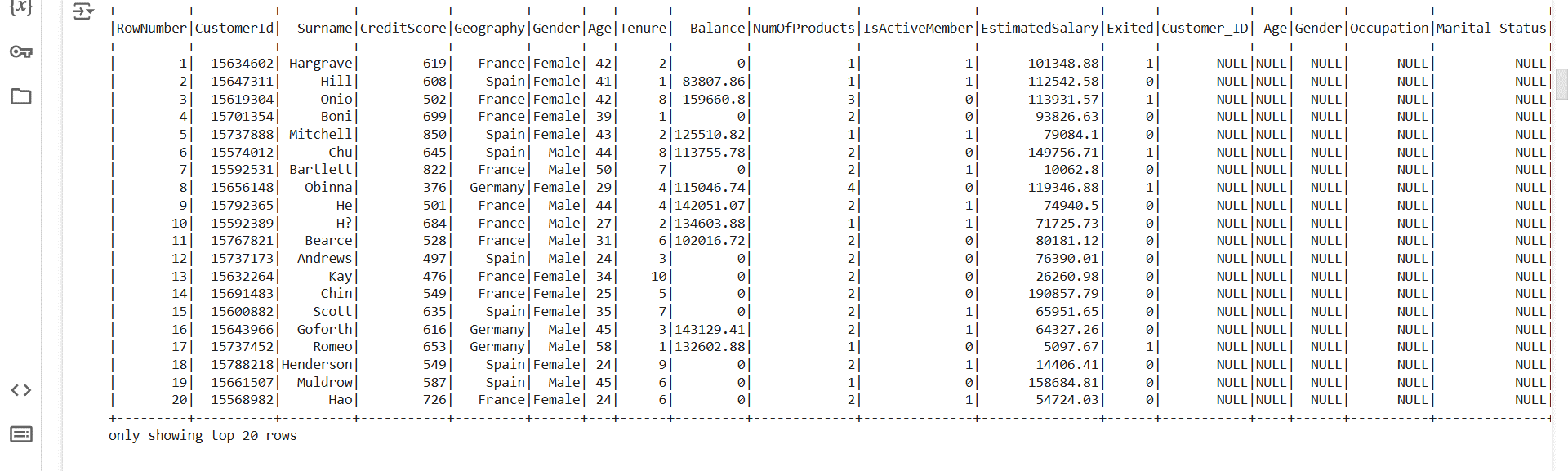
* **join()**:
  + This method performs a join operation between credit\_card\_data and loan\_data. The CustomerId column in credit\_card\_data is joined with the Customer\_ID column in loan\_data. The join type is specified as "inner," meaning only matching rows are included.
* **SQL Query**:
  + The SQL query achieves the same result as the join() method but via SQL syntax. The INNER JOIN ensures only records with matching CustomerId values in both DataFrames are included.

#### **b) Left Join**

**- Left Join Perform a left join between credit\_card\_data and loan\_data.**

left\_join = credit\_card\_data.join(loan\_data, credit\_card\_data.CustomerId == loan\_data.Customer\_ID, "left")

left\_join.show()



query = """

SELECT \*

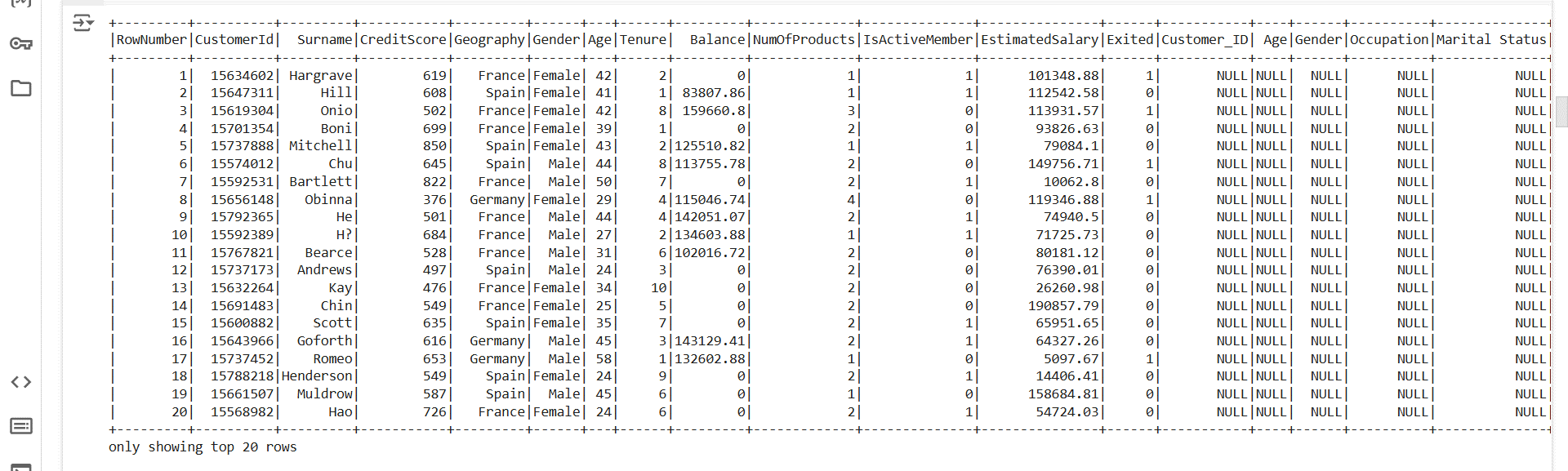
FROM credit\_card c

LEFT JOIN loan l

ON c.CustomerId = l.Customer\_ID

"""

spark.sql(query).show()



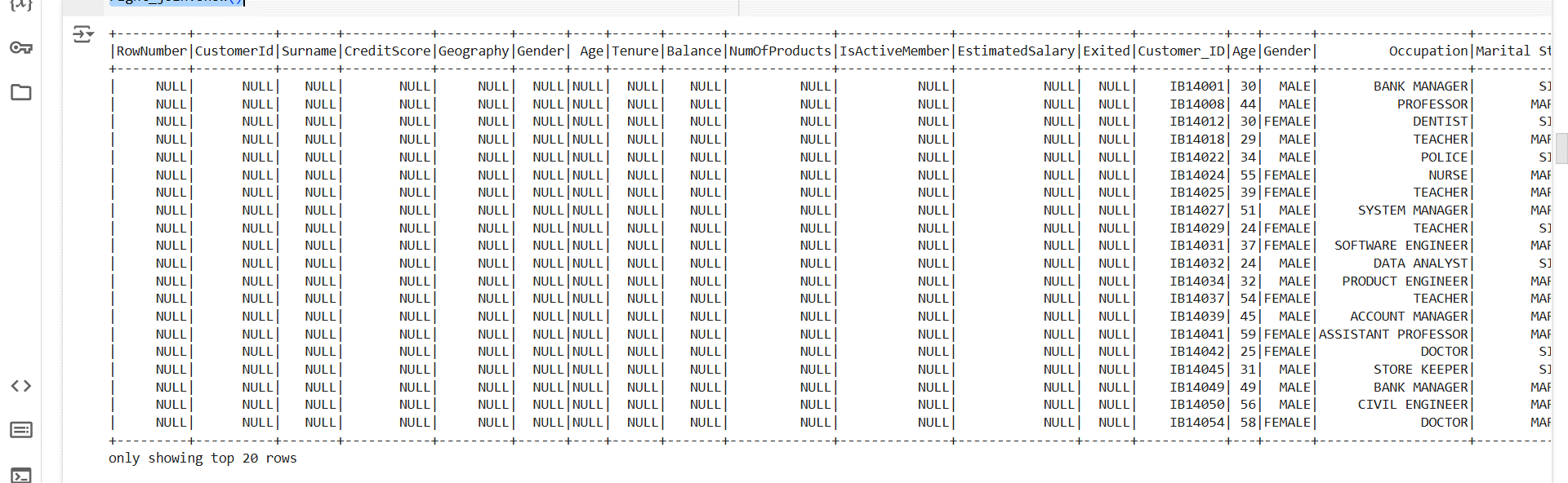
* **left join:**
  + The left join returns all rows from the credit\_card\_data DataFrame and the matching rows from the loan\_data DataFrame. If no match is found, null values are returned for the columns from loan\_data.

#### **c) Right Join**

**-Right Join Perform a right join between credit\_card\_data and loan\_data.**

right\_join = credit\_card\_data.join(loan\_data, credit\_card\_data.CustomerId == loan\_data.Customer\_ID, "right")

right\_join.show()



query = """

SELECT \*

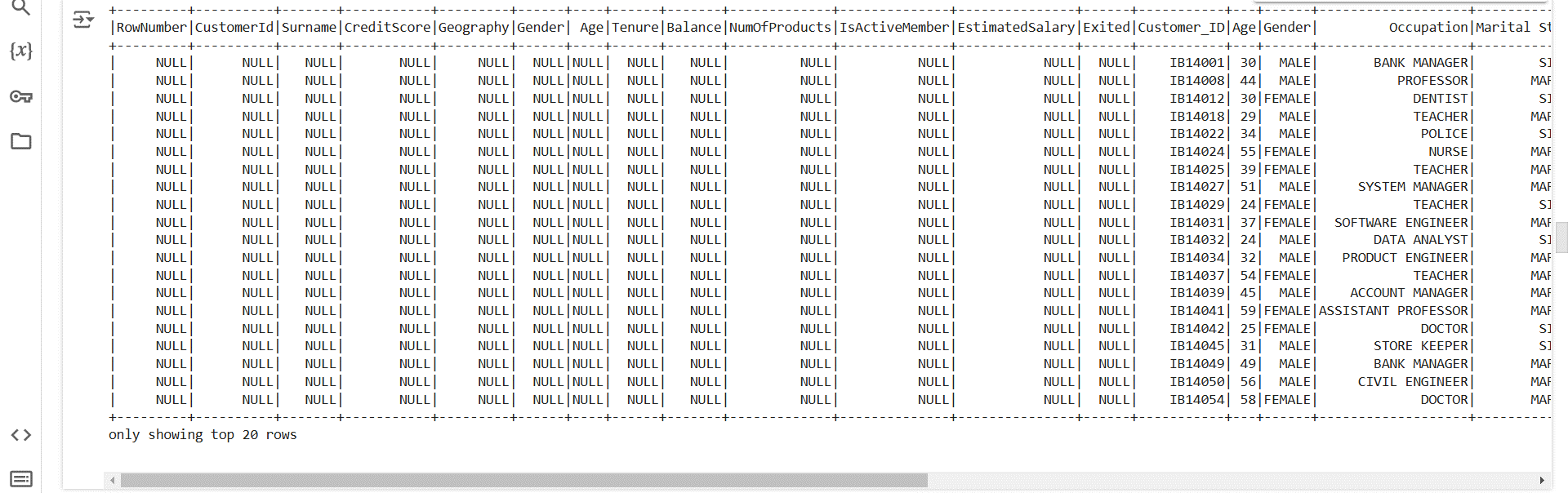
FROM credit\_card c

RIGHT JOIN loan l

ON c.CustomerId = l.Customer\_ID

"""

spark.sql(query).show()



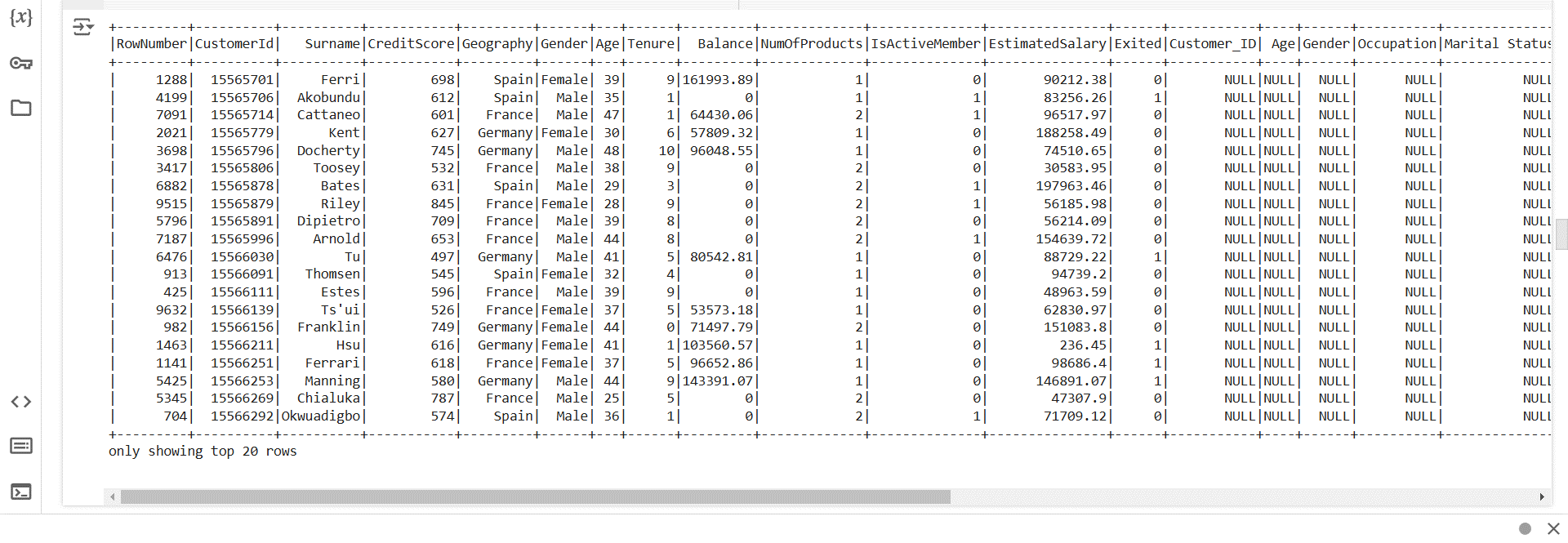
* **right join**:
  + A right join returns all rows from loan\_data and the matching rows from credit\_card\_data. If no match is found, null values are returned for the columns from credit\_card\_data.

#### **d) Full Outer Join**

#### **-** **Full Outer Join Perform a full outer join between credit\_card\_data and loan\_data.**

full\_outer\_join = credit\_card\_data.join(loan\_data, credit\_card\_data.CustomerId == loan\_data.Customer\_ID, "outer")

full\_outer\_join.show()



query = """

SELECT \*

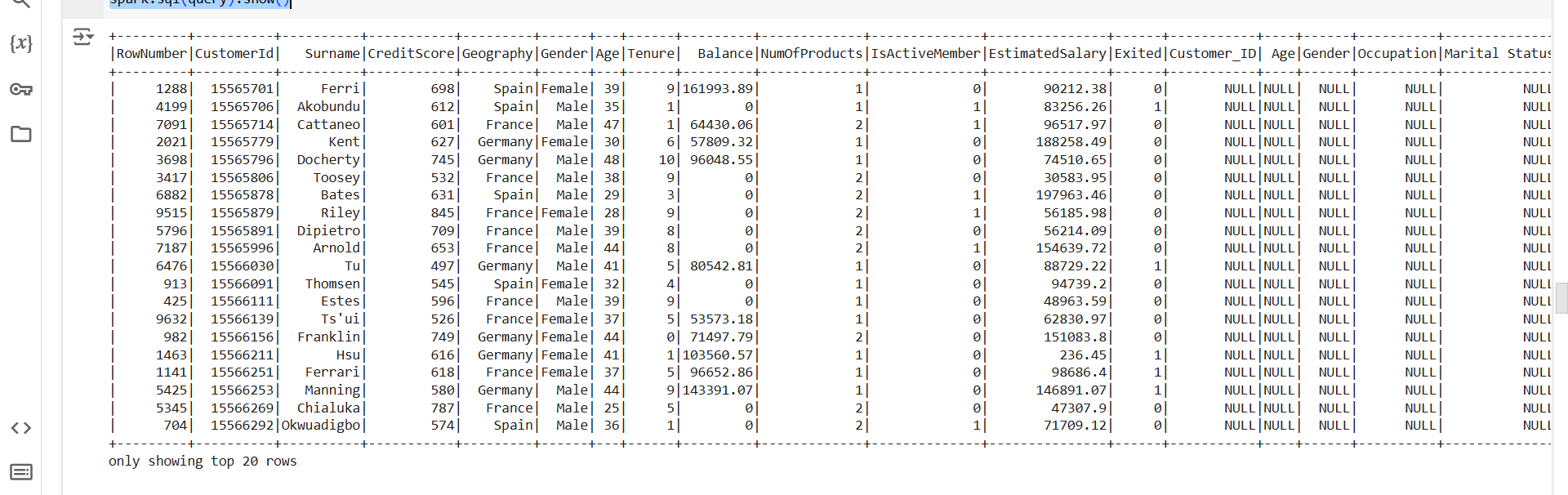
FROM credit\_card c

FULL OUTER JOIN loan l

ON c.CustomerId = l.Customer\_ID

"""

spark.sql(query).show()



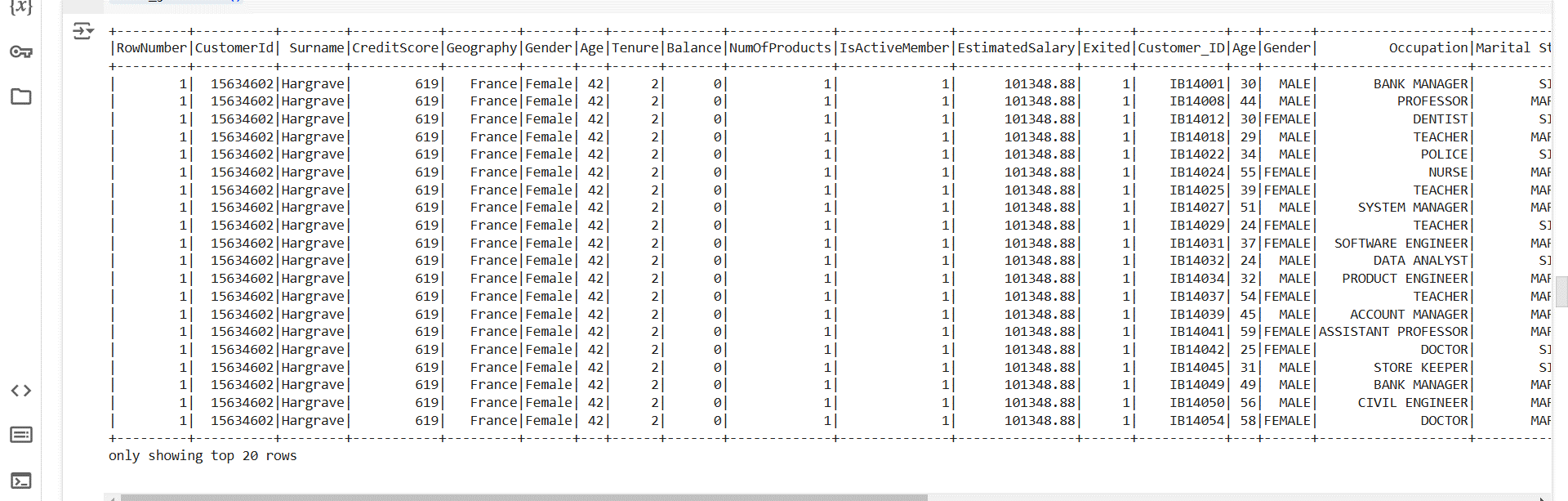
* **outer join**:
  + A full outer join returns all rows when there is a match in one of the DataFrames. If there is no match, null values are returned for the missing side's columns.

#### **e) Cross Join**

**- Cross Join Perform a cross join between credit\_card\_data and loan\_data.**

cross\_join = credit\_card\_data.crossJoin(loan\_data)

cross\_join.show()



query = """

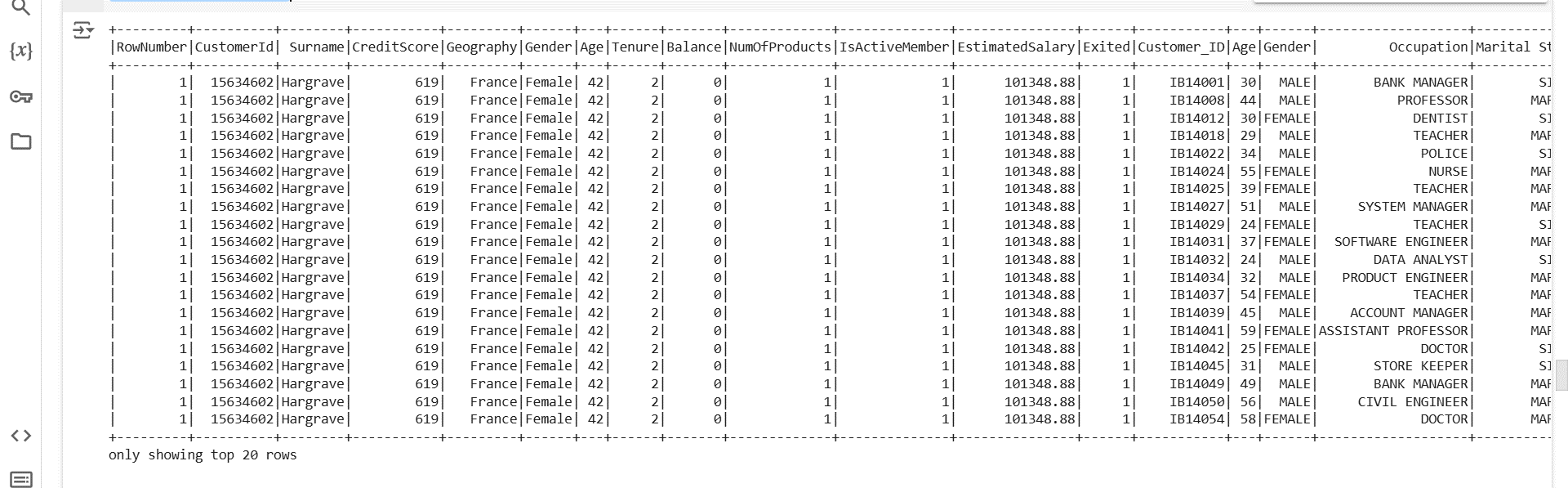
SELECT \*

FROM credit\_card c

CROSS JOIN loan l

"""

spark.sql(query).show()



* **crossJoin()**:
  + A cross join returns the Cartesian product of both DataFrames. This means every row from credit\_card\_data will be joined with every row from loan\_data, leading to a potentially large result set.

**- Filters**

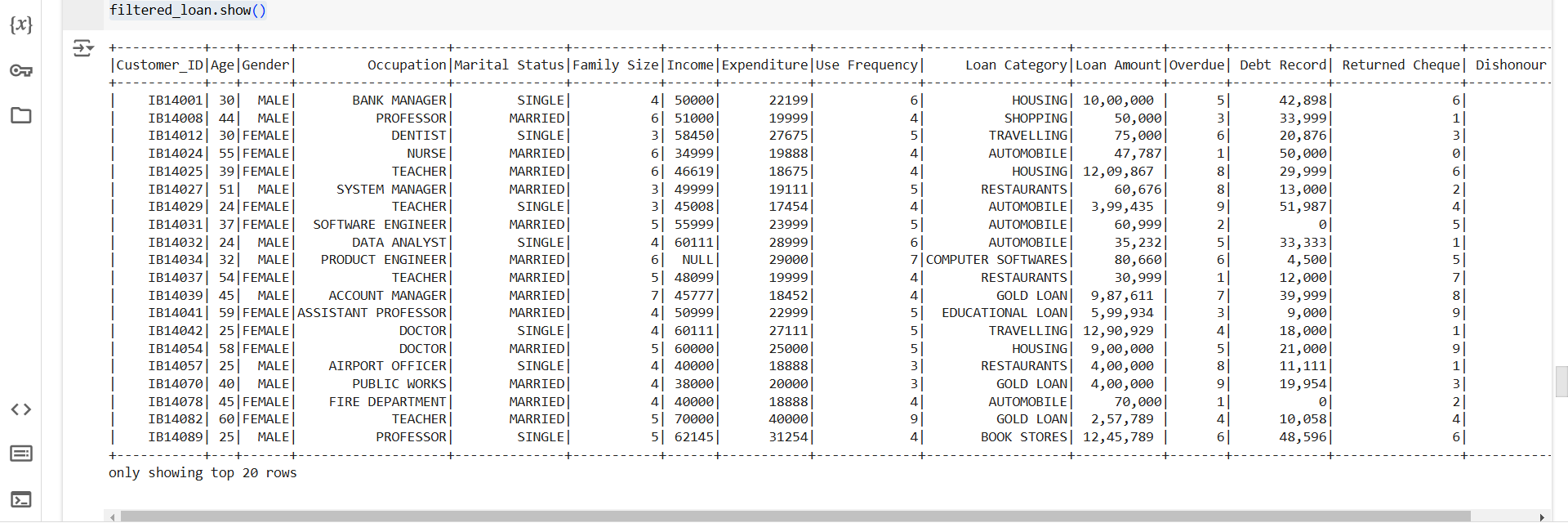
#### **a) Filter Loan Data by Expenditure**

filtered\_loan = loan\_data.filter(

    (loan\_data['Expenditure'] > 15000)

)

filtered\_loan.show()



query = """

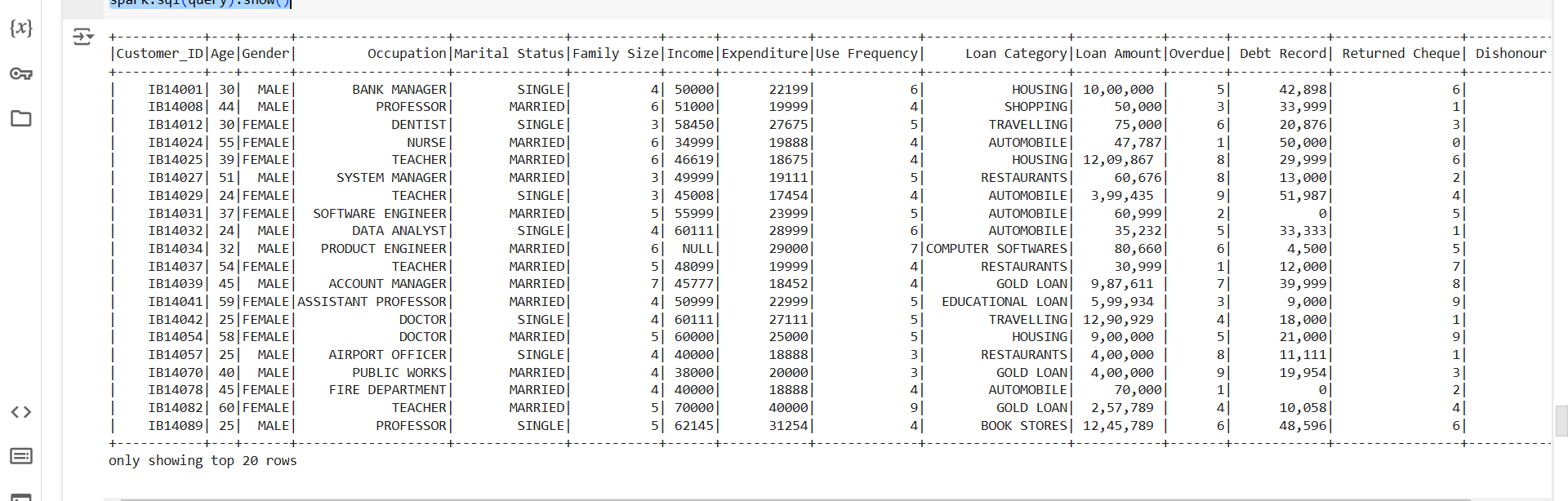
SELECT \*

FROM loan

WHERE Expenditure > 15000

"""

spark.sql(query).show()



* **filter()**:
  + The filter() method is used to apply a condition to the DataFrame. Here, it filters out rows where the Expenditure is greater than 15,000.
* **SQL Query**:
  + The same filter is applied using SQL syntax, selecting rows where Expenditure exceeds 15,000.

**b) Filter Customers with Balance Greater than 5000 and Active Member**

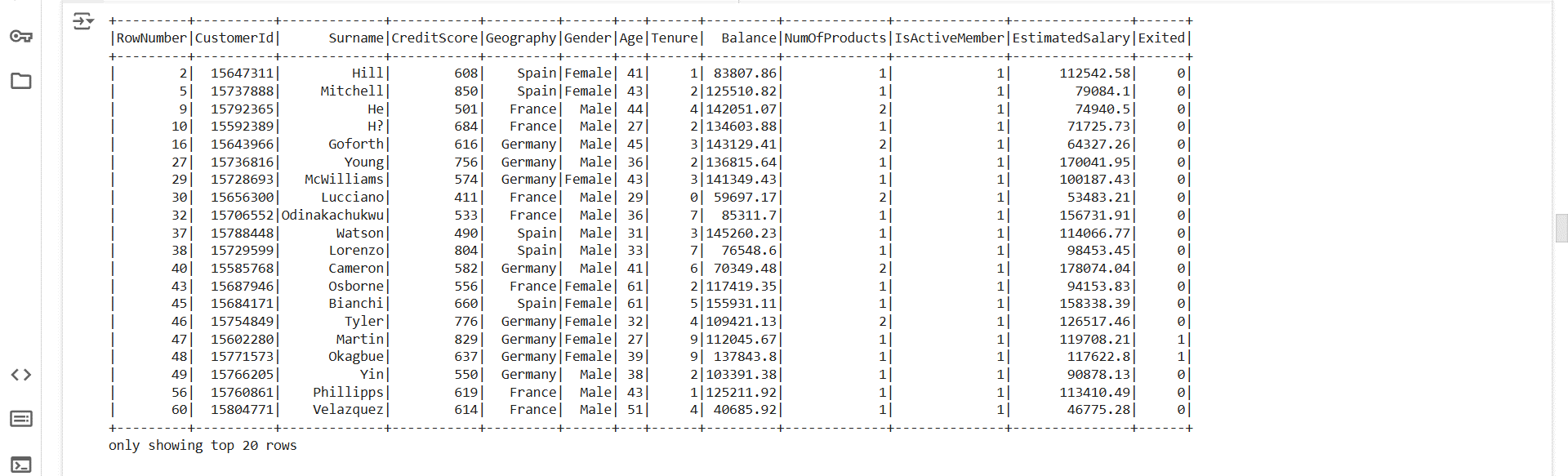
# Filter customers with Balance > 5000 and IsActiveMember = 1 (active member)

filtered\_data = credit\_card\_data.filter(

    (credit\_card\_data['Balance'] > 5000) & (credit\_card\_data['IsActiveMember'] == 1)

)

filtered\_data.show()



query = """

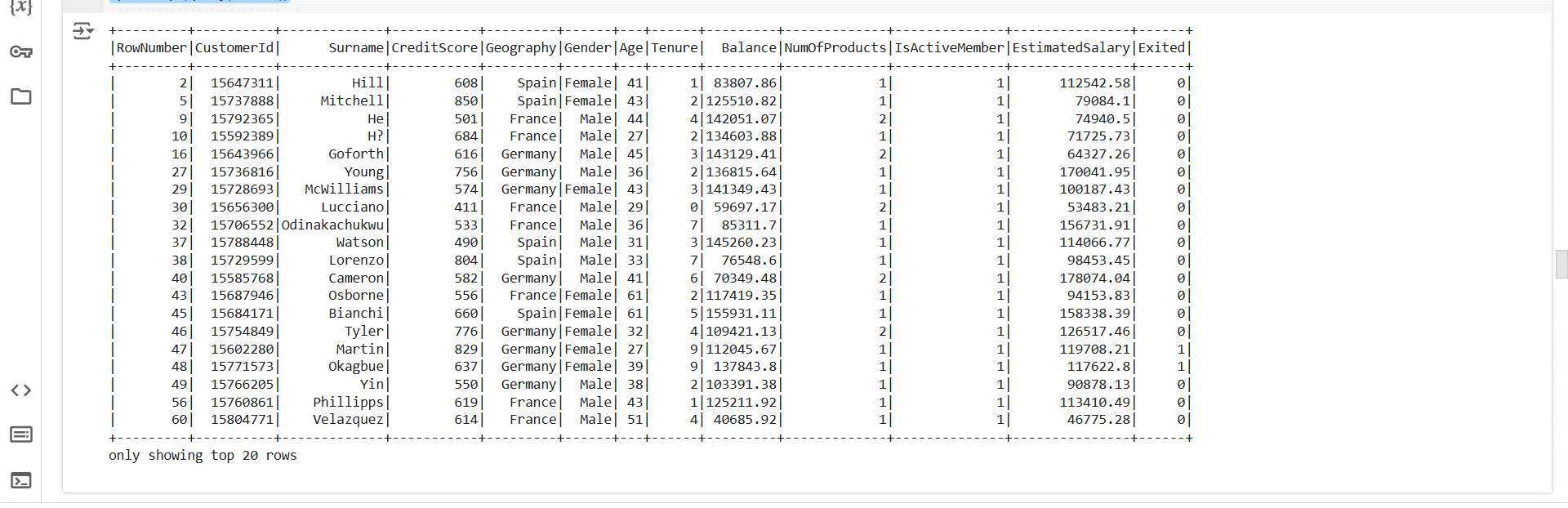
SELECT \*

FROM credit\_card

WHERE Balance > 5000 AND IsActiveMember = 1

"""

spark.sql(query).show()



* filter():
  + Filters customers who have a Balance greater than 5000 and are marked as active members (IsActiveMember = 1).

# **4. GroupBy and Aggregations**

**a)Simple Aggregation-Find the total loan amount and average income by loan category.**

grouped\_loan = loan\_data.groupBy(['Loan Category']).agg(

    {'Loan Amount': "sum", "Income": "avg"}

)

grouped\_loan.show()



query = """

    SELECT

        `Loan Category`,

        SUM(`Loan Amount`) AS Total\_Loan\_Amount,

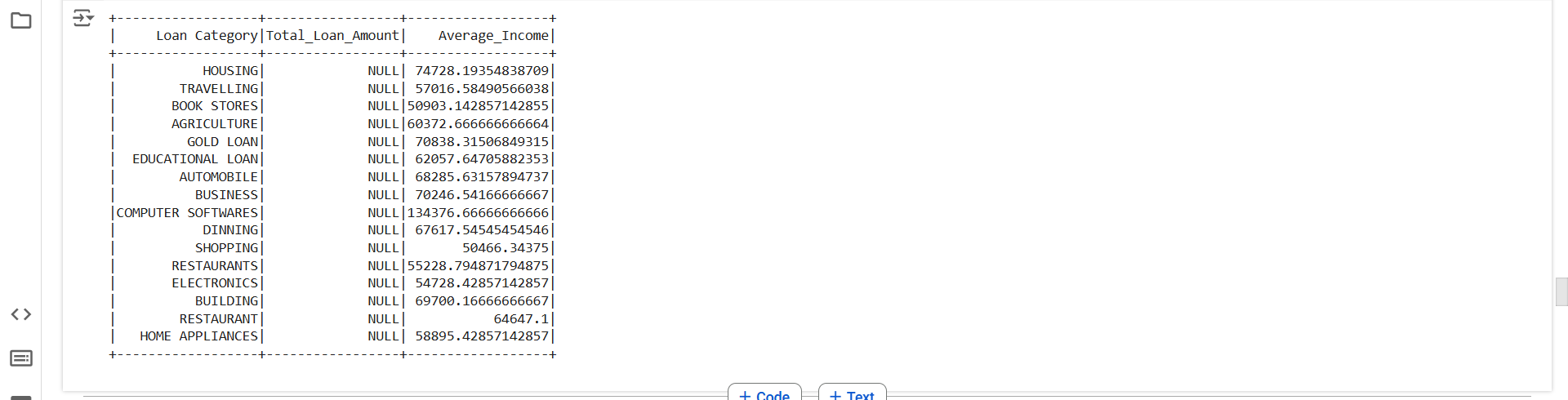
        AVG(`Income`) AS Average\_Income

    FROM loan

    GROUP BY `Loan Category`

"""

spark.sql(query).show()



* **groupBy()** and **agg()**:
  + The groupBy() method groups the data by Loan Category, and agg() is used to compute the sum of Loan Amount and average of Income for each category.
* **filter()**:
  + Filters customers who have a Balance greater than 5000 and are marked as active members (IsActiveMember = 1).

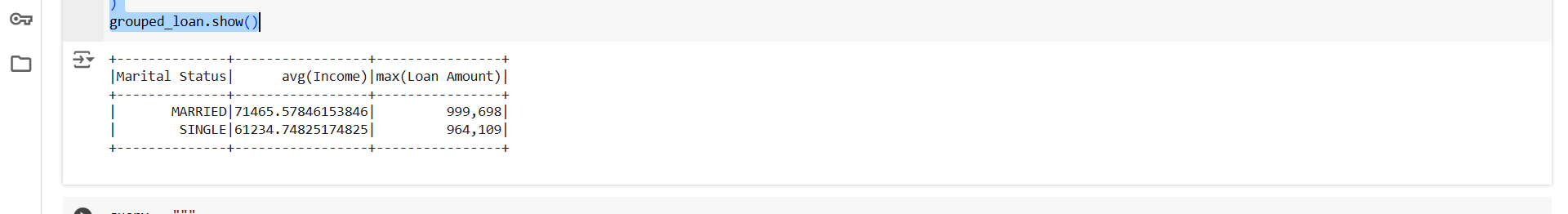
#### **b) Group By Aggregation-Marital Status, Calculate Average Income and Max Loan Amount**

grouped\_loan = loan\_data.groupBy('Marital Status').agg(

    {'Income': 'avg', 'Loan Amount': 'max'}

)

grouped\_loan.show()



query = """

SELECT `Marital Status`,

       AVG(Income) AS Avg\_Income,

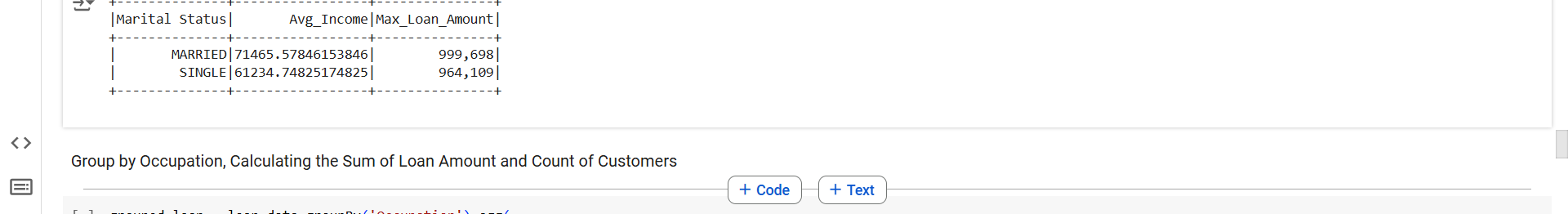
       MAX(`Loan Amount`) AS Max\_Loan\_Amount

FROM loan

GROUP BY `Marital Status`

"""

spark.sql(query).show()



* **Grouping by Marital Status**:
  + This groups the data by marital status and calculates the average Income and maximum Loan Amount for each marital status