**ASSIGNMENT**

**Name:Tejaswini Gokanakonda**

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

## **Date:21-11-2024**

# **MANIPULATING, DROPING, SORTING, AGGREGATIONS JOINING,GROUPBY DATAFRAMES**

**1. PySpark DataFrame Creation and Inspection**

from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder \

.appName("example") \

.getOrCreate()

# Data

simpleData = [("James","Sales","NY",90000,34,10000),

("Michael","Sales","NY",86000,56,20000),

("Robert","Sales","CA",81000,30,23000),

("Maria","Finance","CA",90000,24,23000),

("Raman","Finance","CA",99000,40,24000),

("Scott","Finance","NY",83000,36,19000),

("Jen","Finance","NY",79000,53,15000),

("Jeff","Marketing","CA",80000,25,18000),

("Kumar","Marketing","NY",91000,50,21000)

]

# Create DataFrame

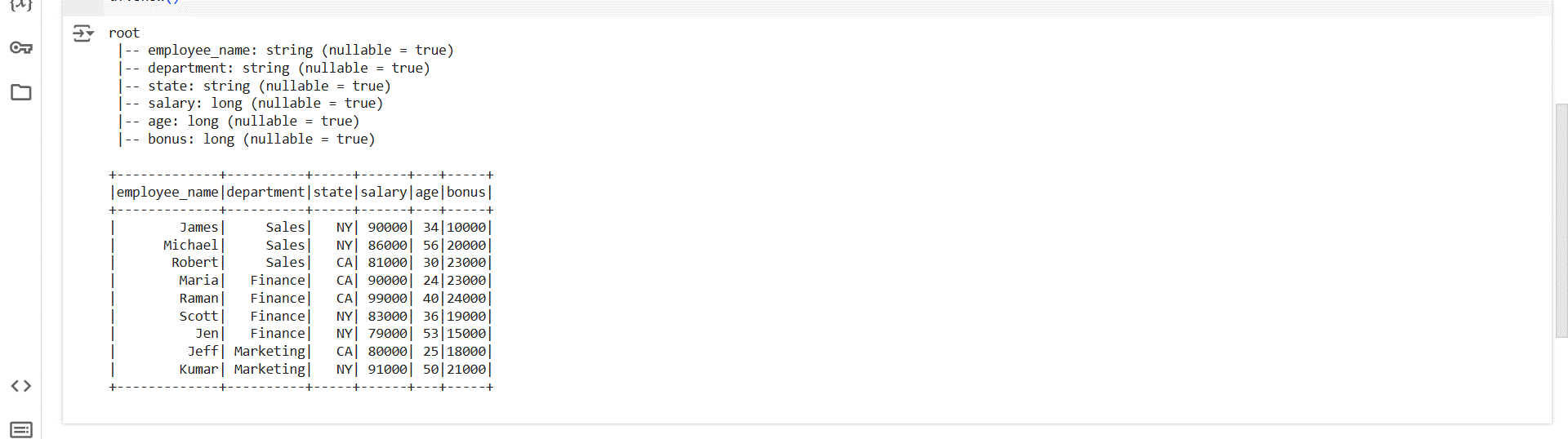
schema = ["employee\_name","department","state","salary","age","bonus"]

df = spark.createDataFrame(data=simpleData, schema = schema)

df.printSchema()

df.show()

**Output Screenshot:**



**Summary:**

The code initializes a SparkSession to enable PySpark operations and creates a DataFrame from a list of tuples representing employee data, including attributes like employee\_name, department, state, salary, age, and bonus. A schema is defined to assign column names to the data. The printSchema() method is used to display the structure of the DataFrame, including column names and data types, while show() outputs the data content in tabular form. This setup demonstrates PySpark's capabilities for processing and inspecting structured data efficiently.

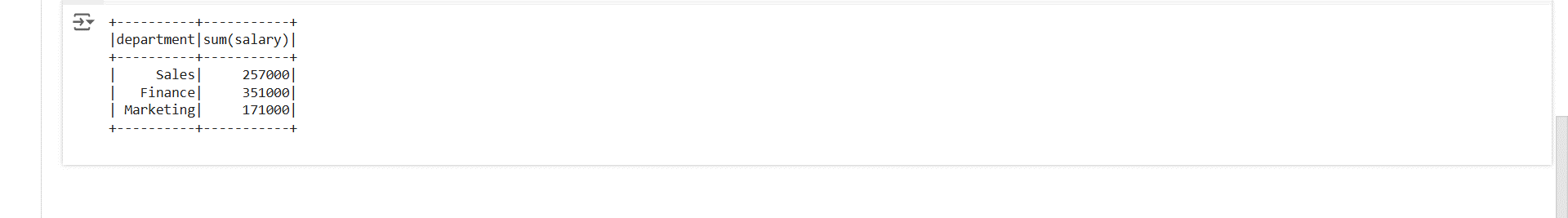
**2. PySpark GroupBy with Sum Operation**

# groupby with sum of salaries

sumdata = df.groupBy("department").sum("salary")

sumdata.show()

**Output Screenshot:**



**Summary:**

The code uses the groupBy() method to group rows based on the department column and applies the sum() function to compute the total salary for each department. This aggregation creates a new DataFrame containing the department names and their corresponding salary sums, which is then displayed using the show() method. This approach efficiently summarizes data by categories in PySpark.

**3. GroupBy with Aggregation Functions in PySpark**

df.groupBy("department").min("salary").show()

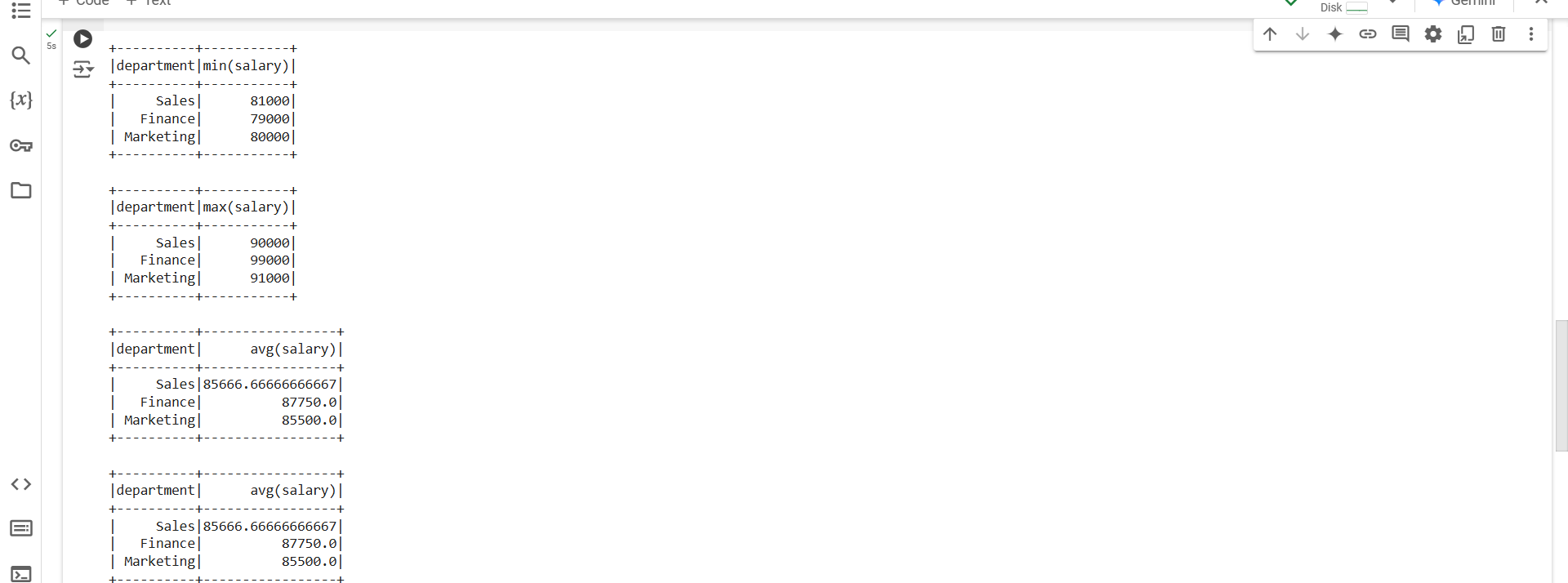
df.groupBy("department").max("salary").show()

df.groupBy("department").avg("salary").show()

df.groupBy("department").mean("salary").show()

df.groupBy("department").count().show()

**Output Screenshot:**





**Summary:**

The code showcases various aggregation functions applied to a PySpark DataFrame using the groupBy() method. It calculates the minimum (min("salary")), maximum (max("salary")), average (avg("salary")), and mean (mean("salary")) salaries for each department, as well as the count of employees (count()) in each department. These operations group the data by the department column and summarize it efficiently, providing insights into salary distribution and employee counts within each department. The results are displayed using the show() method.

**4. GroupBy with Multiple Columns in PySpark**

#grouping by multiple columns

df.groupBy("employee\_name","department").min("salary").show()

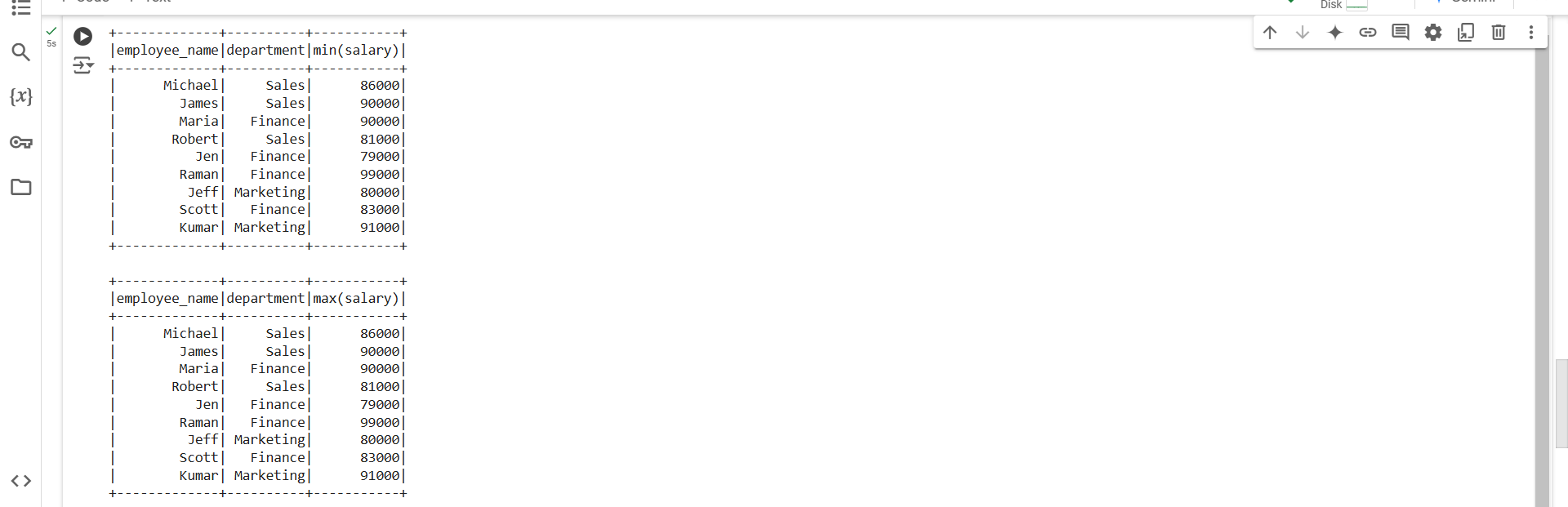
df.groupBy("employee\_name","department").max("salary").show()

df.groupBy("employee\_name","department").avg("salary").show()

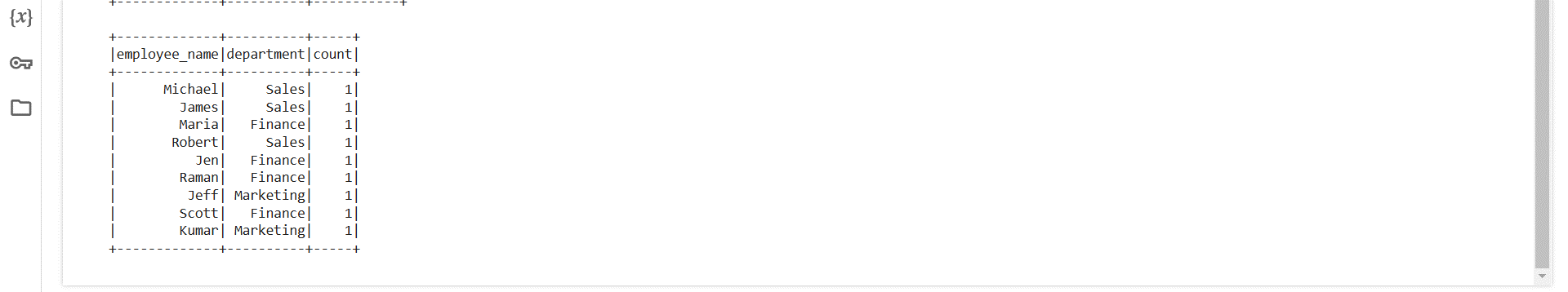
df.groupBy("employee\_name","department").mean("salary").show()

df.groupBy("employee\_name","department").count().show()

**Output Screenshot:**







**Summary:**

The code demonstrates grouping a PySpark DataFrame by multiple columns (employee\_name and department) to perform aggregation operations. It calculates the minimum (min("salary")), maximum (max("salary")), average (avg("salary")), and mean (mean("salary")) salaries, as well as the count of rows (count()) for each combination of employee\_name and department. This approach provides detailed insights into salary and employee statistics at a more granular level, and the results are displayed using the show() method.

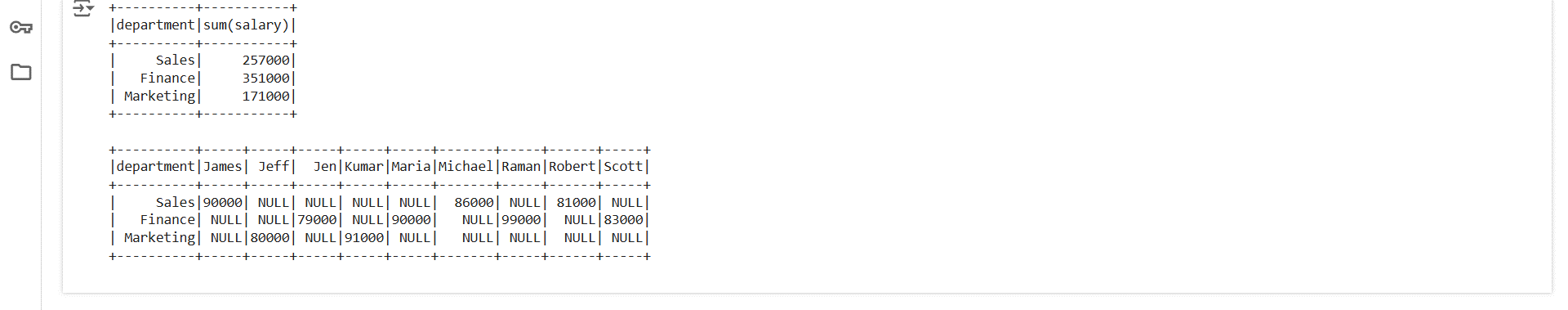
**5.** **Using Pivot Function in PySpark**

#using pivot function

df.groupBy("department").sum("salary").show()

df.groupBy("department").pivot("employee\_name").sum("salary").show()

**Output Screenshot:**



**Summary:**

The code demonstrates the use of the pivot() function for reshaping data during aggregation. First, it groups the DataFrame by the department column and computes the total salary using .sum("salary"). Then, it uses the pivot() function to transform the employee\_name column into new columns, summarizing the salary for each employee within each department. The result of the pivot operation creates a tabular format where the department remains as rows, and each employee\_name becomes a column with corresponding salary values. This method is useful for creating a dynamic summary table.

**6. Handling Missing Data in PySpark**

from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder \

.appName("example") \

.getOrCreate()

# Data

simpleData = [("James","Sales","NY",90000,34,10000),

("Michael","Sales","NY",86000,56,20000),

("Robert","Sales","CA",81000,None,23000),

("Maria","Finance","CA",90000,24,23000),

("Raman","Finance","CA",99000,40,None),

("Scott","Finance","NY",None,36,44000),

("Jen","Finance","NY",55000,53,15000),

("Jeff",None,"CA",80000,25,18000),

("null","Marketing","NY",91000,50,21000)

]

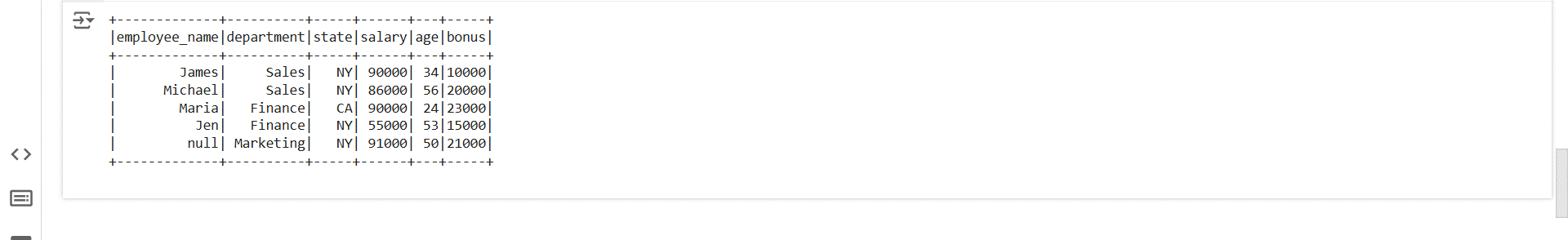
# Create DataFrame

schema = ["employee\_name","department","state","salary","age","bonus"]

dfa = spark.createDataFrame(data=simpleData, schema = schema)

dfa.na.drop().show()

**Output Screenshot:**



**Summary:**

The code demonstrates the use of the na.drop() function in PySpark to handle missing values in a DataFrame. The simpleData dataset contains some null or None values in columns like salary, age, bonus, department, and employee\_name. The na.drop() method removes rows containing any null or None values from the DataFrame, resulting in a cleaned DataFrame. Finally, the show() method displays the filtered DataFrame. This approach is useful for preprocessing data by removing incomplete records before analysis or further processing.

**7. Creating and Displaying a DataFrame with Missing Data in PySpark**

from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder \

.appName("example") \

.getOrCreate()

# Data

simpleData = [("James","Sales","NY",90000,34,10000),

("Michael","Sales","NY",86000,56,20000),

("Robert","Sales","CA",81000,None,23000),

("Maria","Finance","CA",90000,24,23000),

("Raman","Finance","CA",99000,40,None),

("Scott","Finance","NY",None,36,44000),

("Jen","Finance","NY",55000,53,15000),

("Jeff",None,"CA",80000,25,18000),

("null","Marketing","NY",91000,50,21000)

]

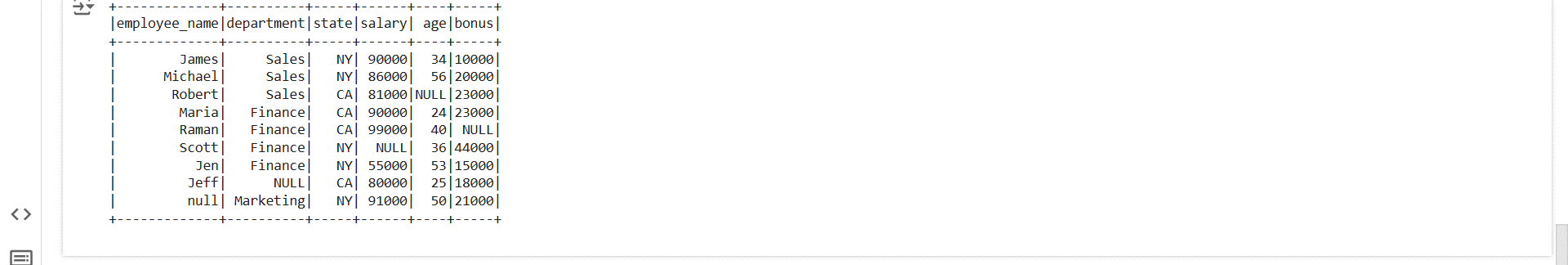
# Create DataFrame

schema = ["employee\_name","department","state","salary","age","bonus"]

dfa = spark.createDataFrame(data=simpleData, schema = schema)

dfa.show()

**Output Screenshot:**



**Summary:**

This code creates a PySpark DataFrame from a dataset (simpleData) that includes some rows with missing values (None or "null"). The schema is explicitly defined with column names: employee\_name, department, state, salary, age, and bonus. The DataFrame is created using the createDataFrame() method and displayed using the show() method. This setup highlights how PySpark handles data containing null or missing values, which can later be addressed with techniques like dropping rows (na.drop) or imputing values.

**8. Aggregation Using agg() in PySpark**

df.groupBy("department").agg(({"salary":"sum"})).show()

df.agg(({"salary":"sum"})).show()  # Without group using agg on salary colums

**Output Screenshot:**



**Summary:**

The code demonstrates using the agg() function in PySpark for performing aggregate operations. The first operation, groupBy("department").agg({"salary": "sum"}), groups the DataFrame by the department column and calculates the total salary for each department. The second operation, agg({"salary": "sum"}), computes the total salary across all rows without any grouping. Both approaches showcase efficient methods for deriving aggregate statistics, either at the group level or globally, and the results are displayed using the show() method.

### 9. **Displaying DataFrame Contents using** show() **in PySpark**

df.show()

**Output Screenshot:**



**Summary:**

The df.show() function in PySpark is used to display the contents of a DataFrame in a tabular format. It shows a snapshot of the data, typically the first 20 rows by default, along with the column headers. If the dataset is large, you can pass arguments like n to specify the number of rows to display and truncate to control the truncation of long strings.

**10. Sorting Data in PySpark**

df.sort("bonus").show()

df.sort(df["salary"].desc()).show()

df.orderBy("salary").show()

**Output Screenshot:**





**Summary:**

The code demonstrates various ways to sort data in a PySpark DataFrame. The sort("bonus") method sorts the DataFrame by the bonus column in ascending order. The sort(df["salary"].desc()) sorts the DataFrame by the salary column in descending order. Lastly, orderBy("salary") sorts the DataFrame by salary in ascending order, similar to sort. These methods are used to organize the data based on specific columns, either in ascending or descending order, providing flexibility in how data is displayed and analyzed.

**SPARK SQL EXAMPLES**

**1. Loading and Displaying CSV Files in PySpark**

# File location and type

file\_location = "/FileStore/tables/simple\_zipcodes-1.csv"

file\_type = "csv"

# CSV options

infer\_schema = "false"

first\_row\_is\_header = "false"

delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.

df = spark.read.format(file\_type) \

  .option("inferSchema", infer\_schema) \

  .option("header", first\_row\_is\_header) \

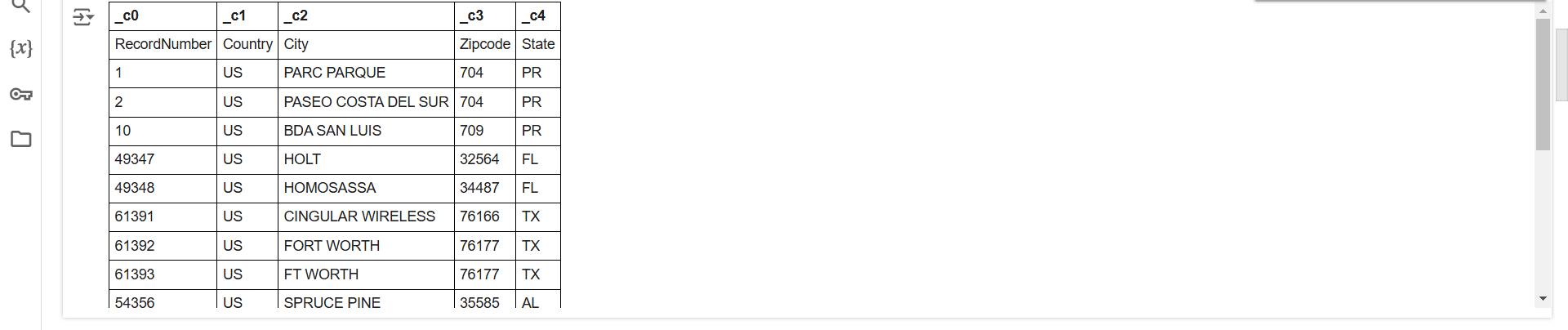
  .option("sep", delimiter) \

  .load(file\_location)

display(df)

df.createOrReplaceTempView("tempdata")

**Output Screenshot:**





**Summary:**

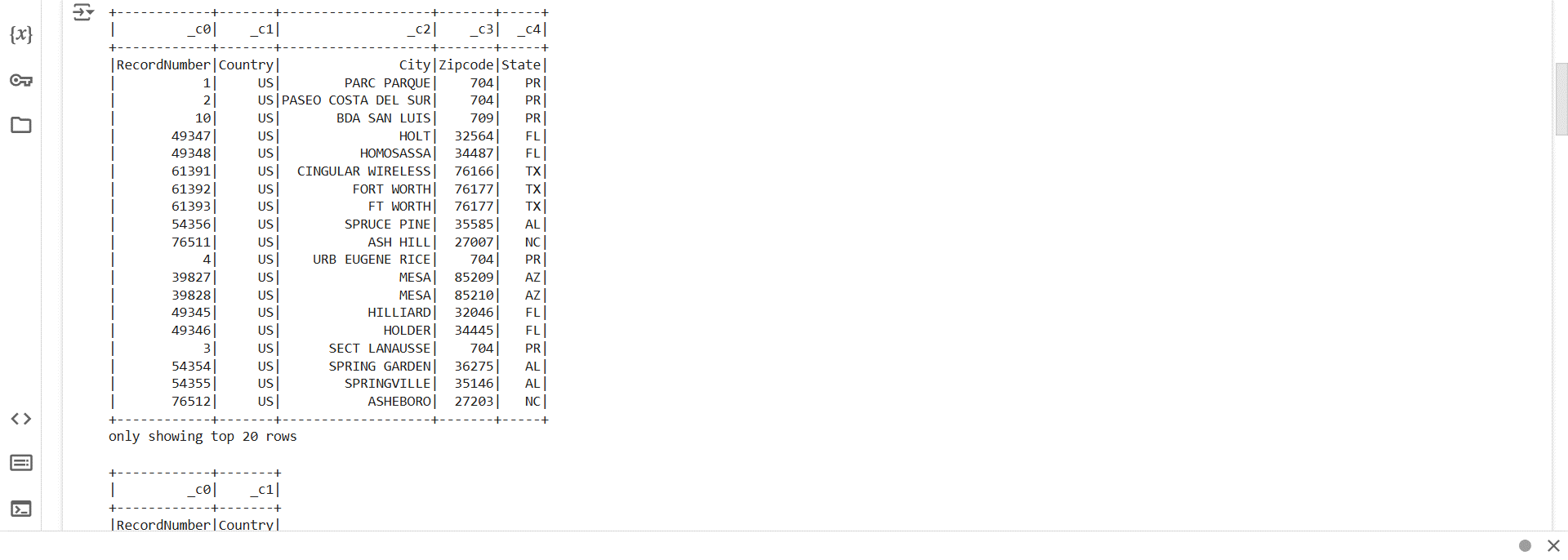
The code reads a CSV file located at /FileStore/tables/simple\_zipcodes-1.csv into a PySpark DataFrame using the spark.read.format("csv") method with specified options: inferSchema set to "false" to disable automatic schema inference, header set to "false" indicating no column headers, and sep set to "," as the delimiter. The DataFrame is displayed using display(df) for inspection, and a temporary SQL view named "tempdata" is created using createOrReplaceTempView for executing SQL queries on the data. This setup is efficient for loading and preparing CSV data for analysis in PySpark.

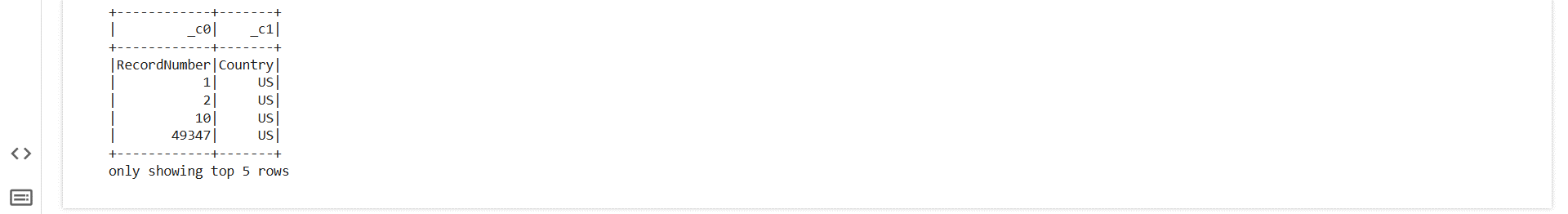
**2. Querying Data from a DataFrame in PySpark**

spark.sql("select \* from tempdata").show()

df.select("\_c0","\_c1").show(5)

**Output Screenshot:**





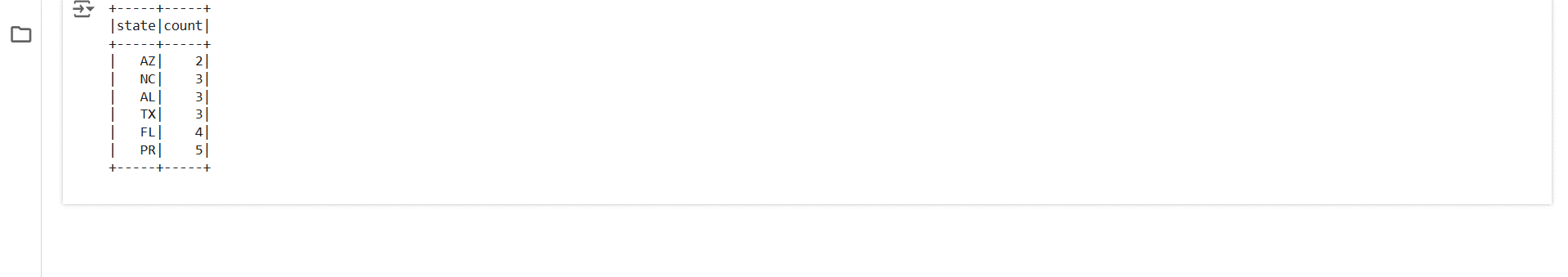
**Summary:**

The code demonstrates two methods for querying data in PySpark: the first uses spark.sql("select \* from tempdata").show() to execute an SQL query on a temporary view (tempdata), retrieving all columns from the DataFrame and displaying the result. The second method, df.select("\_c0", "\_c1").show(5), selects specific columns (\_c0 and \_c1) from the DataFrame and displays the first 5 rows. These approaches allow flexibility in querying and inspecting data either through SQL-style queries or programmatic column selection.

**3. SQL Query with GroupBy in PySpark**

spark.sql("""SELECT state,count(\*) as count FROM customer GROUP BY state""").show()

**Output Screenshot:**



**Summary:**

The code executes an SQL query using spark.sql() to group the customer DataFrame by the state column and count the number of entries (rows) for each state. The query SELECT state, count(\*) as count FROM customer GROUP BY state groups the data by state and counts the occurrences of each state, labeling the result as count. The .show() method displays the grouped data along with the count for each state. This demonstrates the use of SQL in PySpark for aggregation and grouping operations.

**Joins In Pyspark**

**1. Left Join in PySpark**

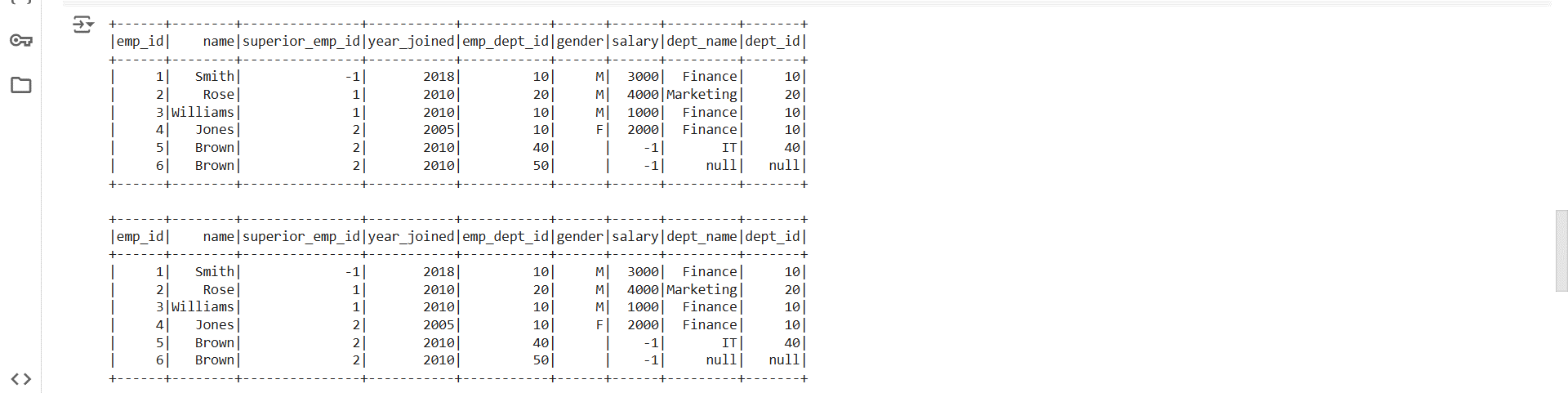
#Left join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "left").show()

#Left join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "leftouter").show()

**Output Screenshot:**



**Summary:**

The code demonstrates performing a left join between two DataFrames (empDF and deptDF) in PySpark using two methods. The first method, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "left").show(), performs a left join, keeping all rows from the left DataFrame (empDF) and adding matching rows from the right DataFrame (deptDF). If no match is found, the right-side columns are filled with null. The second method, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "leftouter").show(), is functionally equivalent to the first, as "left" and "leftouter" both represent a left outer join in PySpark.

**2. Right Join in PySpark**

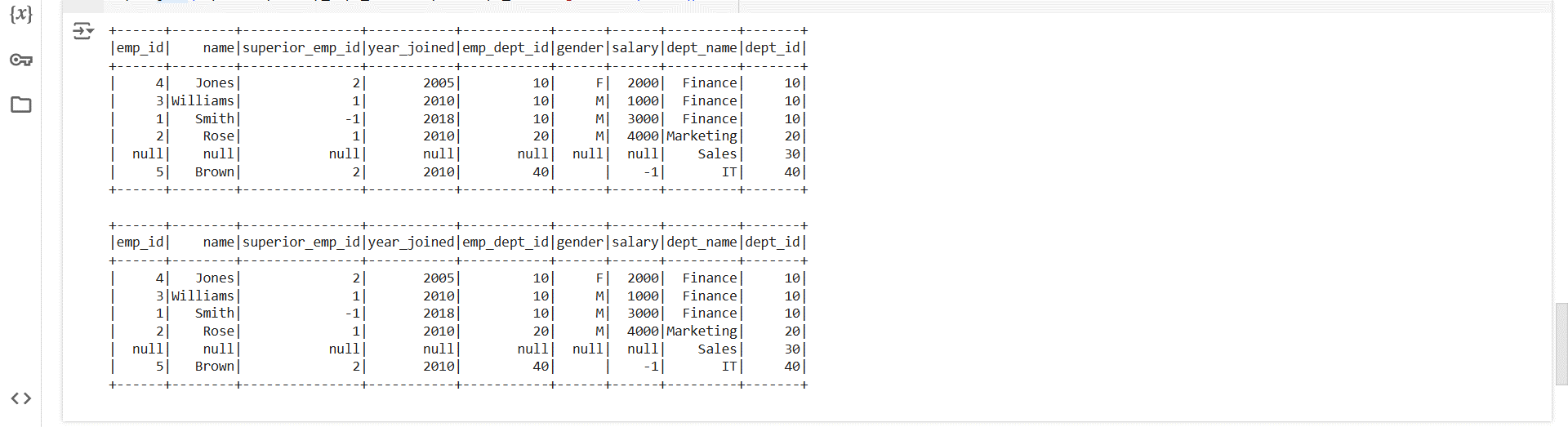
#right join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "right").show()

#right outer join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "rightouter").show()

**Output Screenshot:**



**Summary:**

The code demonstrates performing a right join between two DataFrames (empDF and deptDF) using two methods. The first method, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "right").show(), performs a right join, keeping all rows from the right DataFrame (deptDF) and adding matching rows from the left DataFrame (empDF), filling unmatched rows from the left with null. The second method, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "rightouter").show(), is functionally equivalent to the first, as "right" and "rightouter" both represent a right outer join in PySpark.

**3. Left Semi Join and Left Anti Join in PySpark**

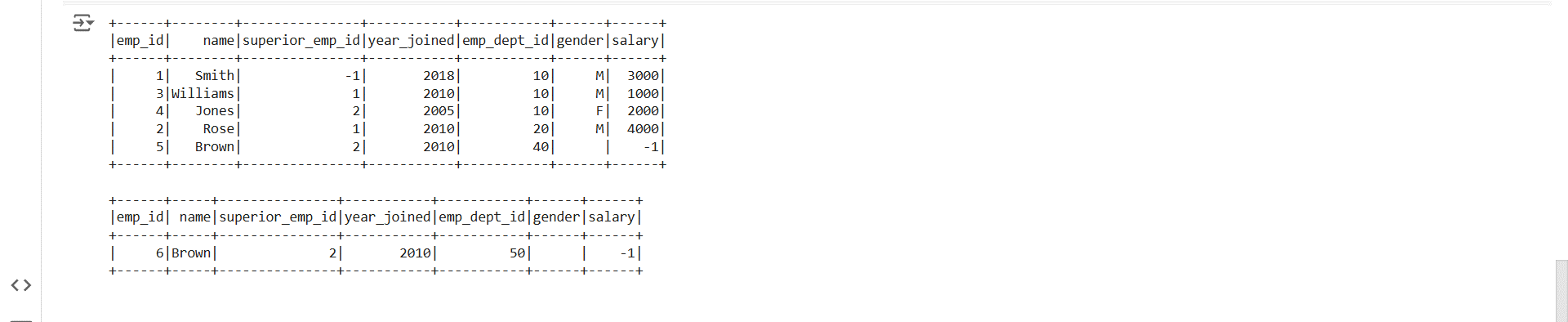
#leftsemijoin

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "leftsemi").show()

#leftanti

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id, "leftanti").show()

**Output Screenshot:**



**Summary:**

The code demonstrates two types of joins in PySpark: **left semi join** and **left anti join**. The left semi join, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "leftsemi").show(), returns all rows from the left DataFrame (empDF) that have matching rows in the right DataFrame (deptDF), but without including any columns from the right DataFrame. The left anti join, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "leftanti").show(), returns all rows from the left DataFrame (empDF) that do not have matching rows in the right DataFrame (deptDF), effectively filtering out matching rows. These joins are useful for extracting rows based on the presence or absence of matches between two DataFrames.