**PYSPARK SQL MANUPULATION**

**AND**

**JOINS ASSIGNMENT**

**Name: S.Nikhil Sai**

**Date: 22-11-2024**

**1)Working with Spark SQL commands:**

from pyspark.sql import SparkSession

spark =SparkSession.builder.appName('sql example program').getOrCreate()

df = spark.read.csv("/FileStore/tables/simple\_zipcodes-1.csv", header=True, inferSchema=False, sep=",")

display(df)

df.createOrReplaceTempView("tempdata")

A screenshot of a computer

Description automatically generated

**Explanation:** The code initializes a SparkSession, which is the entry point forusing Spark functionality. The appName is set to "sql example program." It then reads a CSV file (simple\_zipcodes-1.csv) into a DataFrame using the read.csv method, specifying that the file has a header row (header=True) and not inferring the schema (inferSchema=False). The sep="," ensures the file is correctly parsed as comma-separated values. The display(df) function is used to visually display the DataFrame in an interactive notebook environment. Finally, the DataFrame is registered as a temporary SQL table named tempdata using createOrReplaceTempView, allowing SQL queries to be performed on it.

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

df.select("RecordNumber","Country").show(5)

A screenshot of a computer

Description automatically generated

Explanation: The first command, spark.sql("select \* from tempdata").show(), uses SQL to retrieve and display all rows and columns from the tempdata temporary view. The second command, df.select("RecordNumber", "Country").show(5), selects and displays only the RecordNumber and Country columns from the DataFrame df, showing the first 5 rows. These demonstrate querying data using both SQL and DataFrame APIs in Spark.

spark.sql("""SELECT \* From tempdata WHERE State='AZ'""").show(5)

A screen shot of a computer

Description automatically generated

**Explanation:** The first command, spark.sql("select \* from tempdata").show(), uses SQL to retrieve and display all rows and columns from the tempdata temporary view. The second command, df.select("RecordNumber", "Country").show(5), selects and displays only the RecordNumber and Country columns from the DataFrame df, showing the first 5 rows. These demonstrate querying data using both SQL and DataFrame APIs in Spark**.**

df.createOrReplaceTempView("tempdata")

df.display()

A screenshot of a computer

Description automatically generated

**Explanation:**The df.createOrReplaceTempView("tempdata") command registers the DataFrame df as a temporary SQL table named tempdata, allowing SQL queries to be performed on it using the spark.sql method. The df.display() command visually displays the content of the DataFrame in a tabular format, typically in interactive environments like Databricks, making it easy to explore and understand the data. These commands together facilitate SQL-based querying and visual inspection of the data.

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

df.select("RecordNumber","Country").show(5)

A screenshot of a computer

Description automatically generated

**Explanation:** The command spark.sql("select \* from customer").show() retrieves all columns (\*) and rows from the SQL table named customer and displays the result in tabular format in the console. The df.select("RecordNumber", "Country").show(5) command selects only the RecordNumber and Country columns from the DataFrame df and displays the first 5 rows in a tabular format. Together, these demonstrate querying data using both Spark SQL and DataFrame operations.

spark.sql("""SELECT \* From customer WHERE state='PR'""").show(5)

A screenshot of a computer program

Description automatically generated

**Explanation:** The command spark.sql("""SELECT \* From customer WHERE state='PR'""").show(5) performs a SQL query on the customer table, selecting all columns (\*) from rows where the state column has the value 'PR'. The .show(5) method limits the displayed output to the first 5 rows of the query result. This demonstrates how to filter data using a SQL query in Spark.

spark.sql("""select \* FROM customer WHERE state in ('PR','AZ','FL')order by state """).show(10)

A screenshot of a computer

Description automatically generated

**Explanation:** The command spark.sql("""select \* FROM customer WHERE state in ('PR','AZ','FL') order by state""").show(10) retrieves all columns from the customer table where the state is either 'PR', 'AZ', or 'FL'. It sorts the results by the state column in ascending order. The .show(10) method limits the output to display the first 10 rows. This query filters, sorts, and displays the data based on the specified conditions. It demonstrates the use of SQL in Spark for data selection and ordering.

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

A screen shot of a computer

Description automatically generated

**Expalnation:** The command spark.sql("""SELECT state, count(\*) as count FROM customer GROUP BY state""").show() groups the customer table by the state column. For each unique state, it counts the number of records using count(\*) and renames the count column as count. The .show() method displays the results of this aggregation. This query provides a summary of the number of customers in each state. It demonstrates the use of grouping and aggregation in Spark SQL.

**2) Manipulating data with spark sql:**

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("example").getOrCreate()

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)

]

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

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

df.printSchema()

df.show()

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Explanation:** The code initializes a Spark session named "example." It creates a list simpleData containing employee details such as name, department, state, salary, age, and bonus. A schema is defined with column names, and the spark.createDataFrame() method is used to convert simpleData into a DataFrame df. The df.printSchema() method displays the structure of the DataFrame, while df.show() displays the first few rows of the data. This demonstrates DataFrame creation and inspection in PySpark.

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

sumdata.show()

A screenshot of a computer

Description automatically generated

**Explanation:** The code groups the DataFrame df by the department column using groupBy("department"). It then calculates the sum of the salary column for each department using .sum("salary"). The result is stored in the sumdata DataFrame. Finally, sumdata.show() displays the summed salary values for each department. This demonstrates aggregation using groupBy 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()

A screenshot of a computer

Description automatically generated

**Explanation:** The code groups the df DataFrame by the department column and performs various aggregation operations. It calculates the minimum salary with min("salary"), the maximum salary with max("salary"), and the average salary with both avg("salary") and mean("salary"). Additionally, count() is used to find the number of employees in each department. Each result is displayed using .show(). These demonstrate different aggregation functions in PySpark.

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),

(None,"Marketing","NY",91000,50,21000)

]

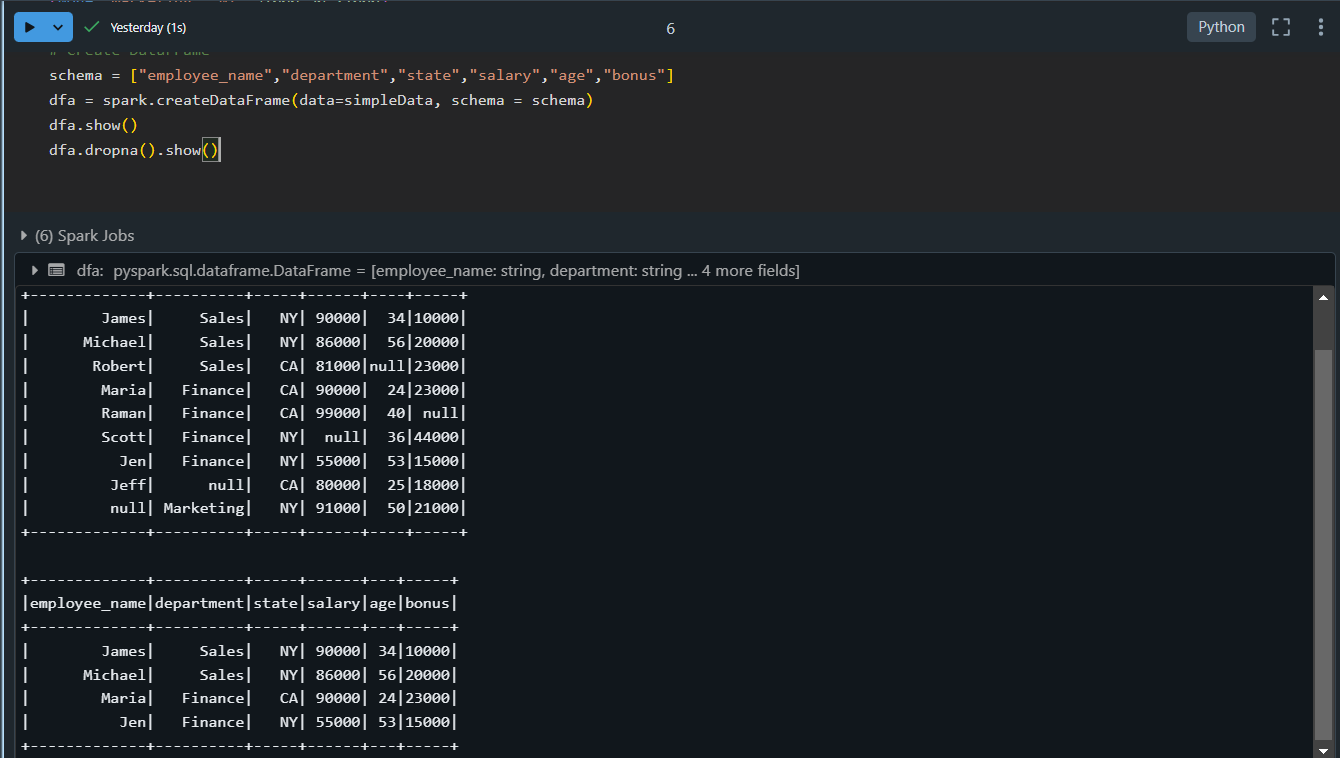
# Create DataFrame

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

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

dfa.show()

dfa.dropna().show()



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

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

A screenshot of a computer

Description automatically generated

**Explanation:** The code performs two aggregation operations on the df DataFrame.The first query df.groupBy("department").agg(({"salary":"sum"})).show(), groups the data by department and calculates the sum of salary for each department. The second query, df.agg(({"salary":"sum"})).show(), calculates the total sum of the salary column across the entire DataFrame without grouping. Both results are displayed using .show(). The first query provides the sum per department, while the second shows the overall sum of salaries.

df.sort("salary").show()           #sorted in ascending order by default

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

A screenshot of a computer

Description automatically generated

**Explanation:** The code sorts the df DataFrame by the salary column in two different ways. The first query, df.sort("salary").show(), sorts the data in ascending order by default. The second query, df.sort(df["salary"].desc()).show(), sorts the data in descending order using the .desc() method. Both results are displayed using .show(). The first query gives the employees sorted by salary in increasing order, and the second sorts them in decreasing order.

**3) Spark SQL Joins:**

from pyspark.sql import SparkSession

import pandas as pd

spark = SparkSession.builder \

.appName("example") \

.getOrCreate()

emp = [(1,"Smith",-1,"2018","10","M",3000),(2, "Rose",1 , "2010", "20","M", 4000),(3,"Williams",1,"2010","10","M",1000),(4, "Jones",2 ,"2005","10","F",2000),(5,"Brown",2,"2010","40","",-1),(6, "Brown", 2, "2010","50","",-1)]

empColumns = ["emp\_id","name","superior\_emp\_id","year\_joined", "emp\_dept\_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)

empDF.printSchema()

empDF.show()

dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]

deptColumns = ["dept\_name","dept\_id"]

deptDF = spark.createDataFrame(data=dept, schema = deptColumns)

deptDF.printSchema()

deptDF.show()

A computer screen shot of text

Description automatically generated

A screenshot of a computer

Description automatically generated

**Explanation:** The code initializes a SparkSession and creates two DataFrames. The first DataFrame, empDF, contains employee data with columns such as emp\_id, name, superior\_emp\_id, year\_joined, emp\_dept\_id, gender, and salary. The second DataFrame, deptDF, contains department data with columns dept\_name and dept\_id. Both DataFrames are created using sample data and corresponding schemas. The printSchema() and show() methods are used to display the structure and data of each DataFrame.

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

#outer join

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

#full join

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

A screenshot of a computer

Description automatically generated

**Explanation:** The code demonstrates three types of joins between the empDF and deptDF DataFrames. The first query performs an inner join, returning only the rows with matching emp\_dept\_id and dept\_id. The second query performs an outer join, which returns all rows from both DataFrames, filling in null where there is no match. The third query performs a full join, similar to the outer join, returning all rows from both DataFrames with null for unmatched rows. These join operations combine the two DataFrames in different ways based on matching and non-matching values.

#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()

A screenshot of a computer program

Description automatically generated

**Explanation:** The code demonstrates two variations of the left join between the empDF and deptDF DataFrames. The first query performs a left join, returning all rows from the empDF DataFrame and the matching rows from the deptDF DataFrame. If there is no match, it fills the missing values from deptDF with null. The second query performs a left outer join, which is essentially the same as a left join in PySpark, returning all rows from empDF and matching rows from deptDF, with null for non-matching rows from deptDF.

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()

A screenshot of a computer

Description automatically generated

**Explanation:** The code demonstrates two types of joins: **leftsemi** and **leftanti**. The first query, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "leftsemi").show(), performs a **left semi join**, which returns only the rows from empDF that have matching rows in deptDF based on the emp\_dept\_id and dept\_id columns, without including any columns from deptDF. The second query, empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "leftanti").show(), performs a **left anti join**, which returns the rows from empDF that do not have matching rows in deptDF, essentially filtering out the matches.