CSE 4262 Data Analytics Lab

Lab 2: Familiarization with Spark SQL and Spark SQL Programming

Outcomes: After this lab students will be able to:

- Explain the fundamentals of SparkSQL libraries and their impact on data analytics.
- Apply PySpark SQL functions and Spark SQL libraries to solve practical problems.
- Apply different PySpark functions to do Data Wrangling operations.

Datasets – a quick introduction

A Spark Dataset is a group of specified heterogeneous columns, akin to a spreadsheet or a relational database table. RDDs have always been the basic building blocks of Spark and they still are. But RDDs deal with objects; we might know what the objects are but the framework doesn't. So, things such as **type checking and semantic queries are not possible with RDDs**. Then came DataFrames, which added schemas; we can associate schemas with an RDD. **DataFrames also added SQL and SQL-like capabilities.**
In the previous lab, we experimented with different ways to create a dataframe. Now that we have created the dataframe containing our data, it is time to look at data manipulation frequently used in the analysis.

Task1. Apply some basic functions to manipulate the dataframe's data using PySpark SQL Use

the following command to read the Datacamp_Ecommerce.csv file and show the contents:

```
df =
spark.read.csv('/content/Datacamp_Ecommerce.csv',header=True,escape="\"")
df.show(5,0)
```

- Find out the number of rows, number of columns, datatypes of the columns, and schema using the following commands: **df.count(), len(df.columns), df.dtypes and df.schema/df.printSchema()**. **Place the results into the following blank box.**

```
        Number of rows:541909
Number of columns:8
[('InvoiceNo', 'string'), ('StockCode', 'string'), ('Description', 'string'), ('Quantity', 'string'), ('InvoiceDate',
'string'), ('UnitPrice', 'string'), ('CustomerID', 'string'), ('Country', 'string')]
StructType(List(StructField(InvoiceNo,StringType,true),StructField(StockCode,StringType,true),Struct
Field(Description,StringType,true),StructField(Quantity,StringType,true),StructField(InvoiceDate,Strin
gType,true),StructField(UnitPrice,StringType,true),StructField(CustomerID,StringType,true),StructFiel
d(Country,StringType,true)))
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: string (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- CustomerID: string (nullable = true)
 |-- Country: string (nullable = true)
None
```

- Remove the duplicates from the dataframe using df.**dropDuplicates() and count the rows now.**
- • Select the InvoiceNo and Description columns using pySpark sql select command **df.select(*['InvoiceNo', 'Description']).show().**

- Use the following command

```
df.select(*[list(set(df.columns)-{'InvoiceNo',
 'Description'})]).show()
```

and check the output. Describe the activities of the command into the following blank box.

The command `df.select(*[list(set(df.columns)-{'InvoiceNo', 'Description'})]).show()` picks all columns from the DataFrame `df` except for 'InvoiceNo' and 'Description'. Here's what each part does:

1. `set(df.columns)`: Turns the list of all column names in `df` into a set, which automatically removes any duplicates.
2. `{'InvoiceNo', 'Description'}`: Makes a set with just 'InvoiceNo' and 'Description'.
3. `set(df.columns) - {'InvoiceNo', 'Description'}`: Subtracts the set from step 2 from the set in step 1, leaving a set of all other column names.
4. `list(...)`: Changes the set from step 3 back into a list.
5. `df.select(*[list(set(df.columns)-{'InvoiceNo', 'Description'})])`: Uses this list to select columns from `df`.
6. `.show()`: Shows the DataFrame with the selected columns.

```
df3=df\
  .withColumnRenamed('InvoiceNo', 'IN')\
  .withColumnRenamed('StockCode', 'SC').show()
```

- PySpark SQL libraries contain many functions that we will be using. Note that col is one such function that returns column values. Use the following line to add a column and copy data using the col function. `df3=df.withColumn('IN', col('InvoiceNo')).show()` Use the lit command to linear transform the given value to all rows of the additional column `df3=df.withColumn('Inv', lit('IN')).show()`
- Use df.drop() command to drop a column from a dataframe

```
df3=df3.drop('Inv’).show()
df3=df.drop(*[‘Country’, ’Quantity’]).show()#drop multiple
columns
```

- Use df.na.replace() command to replace all values in dataframe

```
df4=df.na.replace('United Kingdom’,‘UK’).show()
```

- Use `df.sort(‘InvoiceNo’).show()` to sort the dataframe in ascending order and `df.sort(‘InvoiceNo’, ascending=False).show()` to sort in descending order

Show the outputs to your instructor and get a tick mark here

Task2. Apply some basic functions to manipulate the dataframe's data using SparkSQL libraries

SparkSQL is a powerful feature in Apache Spark that enables users to perform SQL-like operations on large datasets. By combining the strengths of Spark and SQL, SparkSQL offers a powerful tool for large-scale data processing, making it a popular choice for big data applications. In this work, we have covered the basics of Spark SQL and how it can be used in tandem with PySpark.

```
df.registerTempTable("temp_table")
result = spark.sql("SELECT * from temp_table WHERE Quantity = 6 ")
result.show()
print(result.count())

df.createOrReplaceTempView("temp_table2")
spark.sql("select * from temp_table2").show()

result = spark.sql("SELECT CustomerID,sum(UnitPrice) as S_UnitPrice from
temp_table group by CustomerID ")
result.show()
```

Explain the task of `registerTempTable()`

The `registerTempTable` method in PySpark registers a DataFrame as a temporary table that can be queried using SQL syntax within the Spark session. It is session-scoped, meaning the table exists only for the duration of the session and is useful for conducting SQL operations on DataFrame data. Note that `registerTempTable` has been deprecated in favor of `createOrReplaceTempView`, which offers the same functionality but can also replace an existing view.

Explain the difference between `registerTempTable()` and `createOrReplaceTempView()`

1. **Functionality**: `createOrReplaceTempView()` can create a new view or replace an existing one, whereas `registerTempTable()` only registers a new temporary table and does not handle existing tables with the same name.
2. **Usage**: `registerTempTable()` is deprecated and no longer recommended for use, while `createOrReplaceTempView()` is the current standard for creating temporary views in Spark applications.

Task3: Dataframe Joins

Joining data between DataFrames is one of the most common multi-DataFrame transformations. The standard SQL join types are all supported and can be specified as the joinType in df.join(otherDf, sqlCondition, joinType) when performing a join.

```
# List of employee data
data = [["1", "sravan", "company 1"],
 ["2", "ojaswi", "company 1"],
 ["3", "rohith", "company 2"],
 ["4", "sridevi", "company 1"],
 ["5", "bobby", "company 1"]]

# specify column names
columns = ['ID', 'NAME', 'Company']

# creating a dataframe from the lists of data
dataframe = spark.createDataFrame(data, columns)

# list of employee data
data1 = [["1", "45000", "IT"],
 ["2", "145000", "Manager"],
 ["6", "45000", "HR"],
 ["5", "34000", "Sales"]]

# specify column names
columns = ['ID', 'salary', 'department']

# creating a dataframe from the lists of data
dataframe1 = spark.createDataFrame(data1, columns)

# create a view for dataframe named student
dataframe.createOrReplaceTempView("student")

# create a view for dataframe1 named department
dataframe1.createOrReplaceTempView("department")

#use sql expression to select ID column
spark.sql("select * from student, department where student.ID ==
department.ID").show()

dataframe.join(dataframe1,dataframe.ID==dataframe1.ID, "inner").show()

# inner join on id column using sql expression
spark.sql("select * from student INNER JOIN department on student.ID ==
department.ID").show()
```

Explain the task of inner join.

An inner join is a type of join used in databases to combine rows from two or more tables based on a common field between them. It returns a new table with rows that have matching values in both joined tables. If a row in one table does not have a matching row in the other table, it will not appear in the result. This makes inner joins useful for filtering data by intersecting sets of rows from different tables based on shared attributes while filtering out unmatched rows.

Full Outer Join

This join joins the two dataframes with all matching and non-matching rows, we can perform this join in three ways

***Syntax**:*

- ***outer**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”outer”)*
- ***full**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”full”)*
- ***fullouter**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”fullouter”)*

Apply the above code snippets to the above program and write down their differences.

All three types of syntax for a full outer join achieve the same outcome: they combine all rows from both input DataFrames. If rows from one DataFrame don't match rows in the other, those places are filled with null values. The only difference between the three methods is the specificword used to indicate the full outer join.

Left Join

Here this join joins the dataframe by returning all rows from the first dataframe and only matched rows from the second dataframe concerning the first dataframe. We can perform this type of join using left and leftouter.

***Syntax**:*

- ***left**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”left”)*
- ***leftouter**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”leftouter”)*

Right Join

Here this join joins the dataframe by returning all rows from the second dataframe and only matched rows from the first dataframe concerning the second dataframe. We can perform this type of join using right and rightouter.

***Syntax**:*

- ***right**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”right”)*
- ***rightouter**: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”rightouter”)*

Task4: Data Wrangling: Clean, Transform, Merge, and Reshape

Data cleaning is an essential step in the data preparation process to ensure that the data is accurate, consistent, and ready for analysis or modeling.

Collect the data from the following datasets:

```
url =
"https://raw.githubusercontent.com/selva86/datasets/master/Churn_Modellin
g _m.csv"
spark.sparkContext.addFile(url)
```

Read the datasets into a dataframe

```
from pyspark import SparkFiles
df = spark.read.csv(SparkFiles.get("Churn_Modelling_m.csv"), header=True,
inferSchema=True)
df.show(2, truncate=False)
```

Handling Missing Values

- Dropping Missing Values
  - # Drop rows with any missing values

    ```
    cleaned_df = df.dropna()

    # Drop rows with missing values in specific columns
    cleaned_df = df.dropna(subset=['column1', 'column2'])
    ```

Write the row numbers before and after dropping rows.

Rows before dropping: 10000

Rows after dropping: 9908

- Imputing or Filling Missing Data
  - We can use PySpark's DataFrame API along with the Imputer class from the pyspark.ml.feature to fill the missing using Mean, Median, or Mode. Currently, Imputer supports only continuous variables, so before using Imputer class let's find out the continuous variables in the DataFrame.

```
from pyspark.sql.types import IntegerType, FloatType, DoubleType
numeric_column_names = [column.name for column in df.schema.fields
 if isinstance(column.dataType, (IntegerType, FloatType, DoubleType))]
```

  - Create an instance of the Imputer class by specifying the input and output, and the strategy for handling missing values.

```
from pyspark.ml.feature import Imputer

# Initialize the Imputer
imputer = Imputer(
 inputCols= numeric_column_names, #specifying the input column names
 outputCols=numeric_column_names, #specifying the output column names
 strategy="mean" # or "median" if you want to use the median value
)
```

  - Fit the Imputer instance on the dataset to compute the imputation statistics (mean, median, or most frequent value) for each specified column. Use the fitted Imputer model to transform the dataset and fill in the missing values.

```
from pyspark.ml.feature import Imputer
# Fit the Imputer
model = imputer.fit(df)
```

```
#Transform the dataset
imputed_df = model.transform(df)

imputed_df.show(5)
```

Write the total number of rows in dataframe. Is it changed? Why?

Total number of rows in original DataFrame: 10000
Total number of rows in imputed DataFrame: 10000

Both the original DataFrame and the imputed DataFrame have 10,000 rows each. If there were any gaps in the data, they were filled using the average (mean) method. This process did not change the overall number of rows, so both DataFrames still have the same total of 10,000 rows after the imputation.

- Handling Duplicates
  - # Remove duplicate rows

    ```
    cleaned_df = df.dropDuplicates()
    ```

- Removing Outliers and Anomalies

  Outliers can be identified and removed based on statistical measures such as z-score, standard deviation, or percentiles. For example, you can filter rows based on a condition:

  - # Remove rows where a column value is beyond a certain threshold

    ```
    cleaned_df = df.filter(df['column'] < threshold)
    ```

Data Transformation

Data transformation involves reshaping, converting, or enriching the data to prepare it for further analysis or modeling. Let's explore some common data transformation techniques in PySpark.

- Renaming Columns
  - # Rename columns

  ```
  cleaned_df = df.withColumnRenamed('old_column_name', 'new_column_name')
  ```
- Data Type Conversion
  - cleaned_df = df.withColumnRenamed('old_column_name', 'new_column_name')
- Creating New Columns and Derived Features
  - # Create new columns

    ```
    cleaned_df = df.withColumn('new_column', df['column1'] + df['column2'])

    # Create derived features using UDFs (User Defined Functions)
    from pyspark.sql.functions import udf
    from pyspark.sql.types import IntegerType

    # Define a Python function
    def squared(x):
     return x ** 2

    # Register the function as a UDF
    squared_udf = udf(squared, IntegerType())
    ```

```
# Apply the UDF to create a new column
cleaned_df = df.withColumn('squared_column', squared_udf(df['numeric_column']))
```

Advanced-Data Cleaning and Manipulation with PySpark

- Handling Complex Data Types

  Handling nested data structures such as arrays, structs, and maps is common in PySpark when dealing with complex datasets. Let's explore how to work with these data types along with examples:

  - Working with Nested Data Structures

    PySpark allows you to handle nested data structures efficiently, including arrays, structs, and maps. These data types can be nested within each other, providing flexibility in representing complex data.

    ```
    from pyspark.sql.functions import struct, array, map

    # Sample data
    data = [
     (1, [10, 20, 30], {"name": "Ram", "age": 30}),
     (2, [40, 50], {"name": "Shyam", "age": 25}),
     (3, [60, 70, 80], {"name": "Hari", "age": 35})
    ]

    # Define schema for the DataFrame
    schema = ["id", "numbers", "info"]

    # Create DataFrame
    df = spark.createDataFrame(data, schema=schema)
    df.show(truncate=False)
    ```

  - Exploding Arrays and Unnesting Nested Data

    PySpark provides functions like explode() to unnest arrays and selectExpr() to access nested data directly. These functions are useful for flattening nested structures and working with individual elements. # Explode array column

    ```
    exploded_df = df.withColumn("number", explode(df["numbers"]))
    exploded_df.show()

    # Access nested data using dot notation
    nested_df = df.selectExpr("id", "info.name", "info.age")
    nested_df.show()
    ```

  - Handling Timestamp and Date-Time Data

    PySpark provides built-in functions for handling timestamp and date-time data, such as to_timestamp() and to_date().

    ```
    from pyspark.sql.functions import to_timestamp, to_date
    # Convert string column to timestamp

    timestamp_df = df.withColumn("timestamp", to_timestamp(df["timestamp_string"], "yyyy-MM-dd
    HH:mm:ss"))

    # Extract date from timestamp

    date_df = timestamp_df.withColumn("date", to_date(timestamp_df["timestamp"]))
    ```

  - Handling JSON Data and Complex Schemas

    PySpark allows you to work with JSON data and handle complex schemas using functions like from_json() and explode().

    ```
    from pyspark.sql.functions import from_json, col
    # Convert JSON string column to structured data

    schema = "name STRING, age INT"
    ```

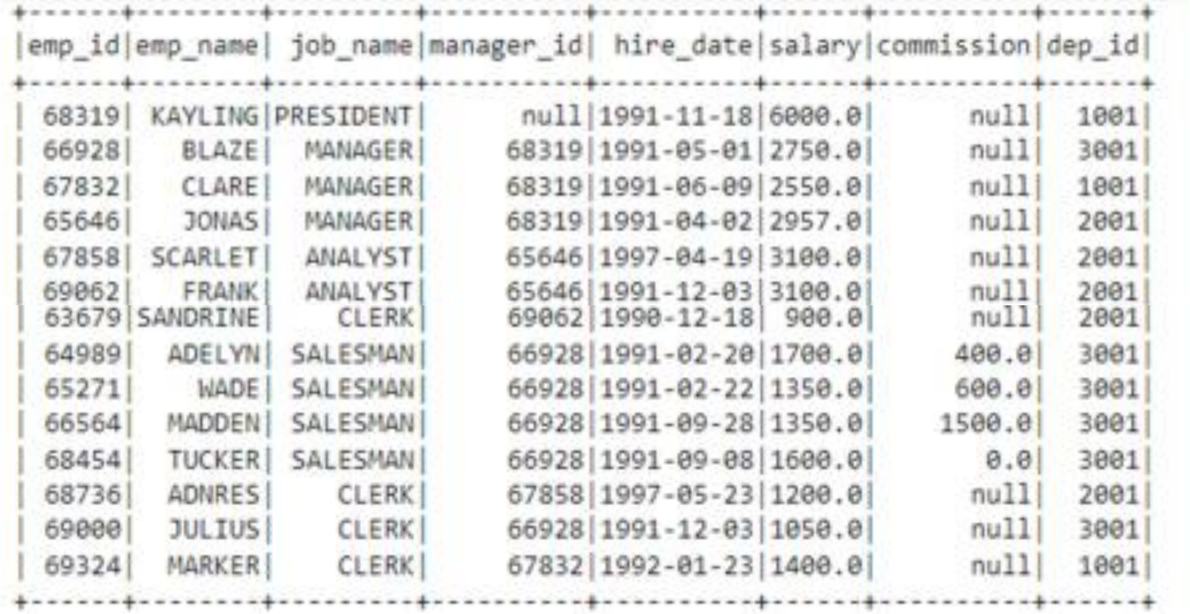
json_df = df.withColumn("json_struct", from_json(col("json_string"), schema))

# Explode nested data

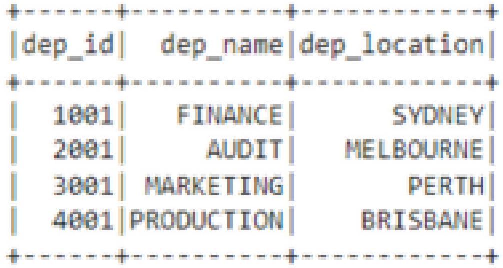
exploded_df = json_df.withColumn("exploded_data", explode(col("json_struct")))

Assignment 2

Consider the following dataframes.



```
+------+--------+---------+----------+----------+------+----------+------+
|emp_id|emp_name| job_name|manager_id| hire_date|salary|commission|dep_id|
+------+--------+---------+----------+----------+------+----------+------+
| 68319| KAYLING|PRESIDENT|      null|1991-11-18|6000.0|      null|  1001|
| 66928|   BLAZE|  MANAGER|     68319|1991-05-01|2750.0|      null|  3001|
| 67832|   CLARE|  MANAGER|     68319|1991-06-09|2550.0|      null|  1001|
| 65646|   JONAS|  MANAGER|     68319|1991-04-02|2957.0|      null|  2001|
| 67858| SCARLET|  ANALYST|     65646|1997-04-19|3100.0|      null|  2001|
| 69062|   FRANK|  ANALYST|     65646|1991-12-03|3100.0|      null|  2001|
| 63679|SANDRINE|    CLERK|     69062|1990-12-18| 900.0|      null|  2001|
| 64989|  ADELYN| SALESMAN|     66928|1991-02-20|1700.0|     400.0|  3001|
| 65271|    WADE| SALESMAN|     66928|1991-02-22|1350.0|     600.0|  3001|
| 66564|  MADDEN| SALESMAN|     66928|1991-09-28|1350.0|    1500.0|  3001|
| 68454|  TUCKER| SALESMAN|     66928|1991-09-08|1600.0|       0.0|  3001|
| 68736|  ADNRES|    CLERK|     67858|1997-05-23|1200.0|      null|  2001|
| 69000|  JULIUS|    CLERK|     66928|1991-12-03|1050.0|      null|  3001|
| 69324|  MARKER|    CLERK|     67832|1992-01-23|1400.0|      null|  1001|
+------+--------+---------+----------+----------+------+----------+------+
```



```
+------+----------+------------+
|dep_id|  dep_name|dep_location|
+------+----------+------------+
|  1001|   FINANCE|      SYDNEY|
|  2001|     AUDIT|   MELBOURNE|
|  3001| MARKETING|       PERTH|
|  4001|PRODUCTION|    BRISBANE|
+------+----------+------------+
```

The necessary code to create the dataframes can be found in this link. Now write PySpark SQL queries to perform the following tasks:

1. Retrieve employees' names along with their department name.

```
employee_dept_df = emp_df.join(dept_df, emp_df.dep_id ==
dep_df.dep_id, "left_outer")
employee_dept_df.select("emp_name", "dept_name").show()
```

2. Display the details of all employees who have managers, along with the names of their respective managers.

```
      emp_mgr_df = emp_df.alias("emp").join(emp_df.alias("mgr"),
col("emp.manager_id") == col("mgr.emp_id"), "inner")
emp_mgr_df.select("emp.emp_id", "emp.emp_name",
"emp.job_name","emp.hire_date","emp.salary","emp.commission","emp.dep_id","m
gr.emp_name").show()
```

3. Display the details of all employees, including those who don't have a manager, along with the name of their manager if they have one.

```
      emp_mgr_optional_df = emp_df.alias("emp").join(emp_df.alias("mgr"),
col("emp.manager_id") == col("mgr.emp_id"), "left_outer")

emp_mgr_optional_df.select("emp.emp_id", "emp.emp_name", "emp.job_name",
"emp.hire_date", "emp.salary", "emp.commission", "emp.dep_id",
"mgr.emp_name").show()
```

4. Display the details of all employees who do not have any manager

```
emp_no_mgr_df = emp_df.alias("emp").join(emp_df.alias("mgr"),
col("emp.manager_id") == col("mgr.emp_id"), "left_anti")
emp_no_mgr_df.show()
```

5. Show the details of the manager who has the most number of employees working under him/her.

```
manager_employee_count = emp_df.groupBy("manager_id").count()

manager_with_most_employees =
manager_employee_count.orderBy(desc("count")).first()["manager_id"]

manager_details_df = emp_df.filter(emp_df.emp_id ==
manager_with_most_employees)

manager_details_df.show()
```