**Ahsanullah University of Science & Technology**

**Department of Computer Science & Engineering**

# *Semester: Fall 2023*



# CSE 4262- Data Analytic Lab

**Familiarizaon with Spark SQL and Spark SQL Programming**

**Assignment On:**

**Submitted To**

Prof. Dr. Md. Shamim Akhter

Mr. Md. Zahid Hossain

CSE, AUST

**Submitted By**

Jalisha Jashim Era ID:20200104038

Date: 7 May 2024

**CSE 4262 Data Analycs Lab**

**Lab 2: Familiarizaon 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 analycs. • Apply PySpark SQL functions and Spark SQL libraries to solve practical problems. • Apply different PySpark functions to do Data Wrangling operaons.

**Datasets – a quick introducon**

A Spark Dataset is a group of specified heterogeneous columns, akin to a spreadsheet or a relaonal database table. RDDs have always been the basic building blocks of Spark and they sll 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 semanc 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 capabilies.**

**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 me to look at data manipulaon frequently used in the analysis.**

**Task1. Apply some basic funcons 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.**

541909

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),StructField(Description,StringType,true),StructField(Quantity,StringType,true),StructField(InvoiceDate,StringType,true),StructField(UnitPrice,StringType,true),StructField(CustomerID,StringType,true),StructField(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 Descripon columns using pySpark sql select command **df.select(\*['InvoiceNo', 'Descripon']).show().**

• Use the following command

df.select(\*[list(set(df.columns)-{'InvoiceNo',

'Description'})]).show()

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

The command `df.select(\*[list(set(df.columns)-{'InvoiceNo', 'Description'})]).show()` selects all columns from the DataFrame `df` except for 'InvoiceNo' and 'Description'

1. `set(df.columns)`: Converts the list of column names in the DataFrame `df` into a set, removing any duplicate column names.

2. `{'InvoiceNo', 'Description'}`: Creates a set containing the column names 'InvoiceNo' and 'Description'.

3. `set(df.columns) - {'InvoiceNo', 'Description'}`: Performs set difference, resulting in a set containing all column names except for 'InvoiceNo' and 'Description'.

4. `list(...)`: Converts the resulting set back into a list.

5. `df.select(\*[list(set(df.columns)-{'InvoiceNo', 'Description'})])`: Selects the columns specified in the list generated in step 4.

6. `.show()`: Displays the selected columns in the DataFrame.

• Use **withColumnRenamed** command to rename specific column names

df3=df\

.withColumnRenamed('InvoiceNo', 'IN')\

.withColumnRenamed('StockCode', 'SC').show()

• PySpark SQL libraries contain many funcons that we will be using. Note that col is one such funcon that returns column values. Use the following line to add a column and copy data using the col funcon. df3=df.withColumn('IN', col('InvoiceNo').show() Use the lit command to linear transform the given value to all rows of the addional 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 ck mark here**

**Task2. Apply some basic funcons to manipulate the dataframe’s data using SparkSQL libraries** SparkSQL is a powerful feature in Apache Spark that enables users to perform SQL-like operaons 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 applicaons. 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()

When `registerTempTable()` is called on a DataFrame, it generates a temporary table or view bearing the given name within the Spark SQL catalog. This temporary representation is linked to the DataFrame and is accessible for reference in SQL queries. Once the DataFrame is registered as a temporary table, SQL queries can be employed to manipulate and analyze the data within the DataFrame without affecting the original DataFrame.

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

Both registerTempTable() and createOrReplaceTempView() are used to register a DataFrame for SQL operations within a SparkSession’s lifespan. The key distinction is that registerTempTable() will generate an error if a table of the same name is already present, whereas createOrReplaceTempView() will overwrite any existing table with the new DataFrame.

**Task3: Dataframe Joins**

Joining data between DataFrames is one of the most common mul-DataFrame transformaons. The standard SQL join types are all supported and can be specified as the joinType in df.join(otherDf, sqlCondion, 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 merges rows from two DataFrames when they meet a specific condition, often related to a shared column. It filters and merges only the rows that have corresponding values in both DataFrames, creating a new DataFrame. Essentially, it extracts the overlapping data between the two DataFrames, allowing for the integration of related data from different sources and excluding rows that don’t have a match.

**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.**

Indeed, the three syntax variations for a full outer join all yield an identical outcome: a DataFrame that includes every row from both source DataFrames. Where matches are absent, null values are inserted. The sole distinction among them is the specific keyword utilized to denote the full outer join.Top of Form

**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 essenal step in the data preparaon 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\_Modelling \_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

o # 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

• Impung or Filling Missing Data

o 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 connuous variables, so before using Imputer class let’s find out the connuous 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))] |

o 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

# Inialize 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

)

o Fit the Imputer instance on the dataset to compute the imputaon stascs (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

The total number of rows in both the original DataFrame and the imputed DataFrame is 10,000. There were any missing values, they were filled using the mean strategy without affecting the total number of rows. Therefore, the total number of rows remains unchanged after imputation.

• Handling Duplicates

o # Remove duplicate rows

cleaned\_df = df.dropDuplicates()

• Removing Outliers and Anomalies

Outliers can be idenfied and removed based on stascal measures such as z-score, standard deviaon, or percenles. For example, you can filter rows based on a condion:

o # 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

o # Rename columns

cleaned\_df = df.withColumnRenamed('old\_column\_name', 'new\_column\_name')

• Data Type Conversion

o cleaned\_df = df.withColumnRenamed('old\_column\_name', 'new\_column\_name')

• Creating New Columns and Derived Features

o # 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:

o 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)

o 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()

o 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"]))

o 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"

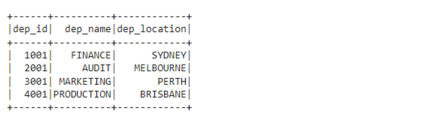
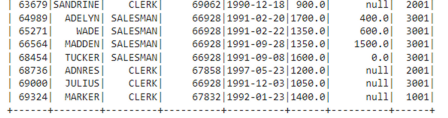
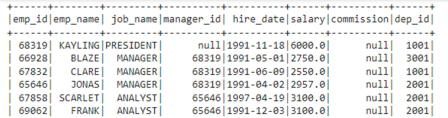
json\_df = df.withColumn("json\_struct", from\_json(col("json\_string"), schema))

# Explode nested data

exploded\_df = json\_df.withColumn("exploded\_data", explode(col("json\_struct")))

**Assignment 2**

Consider the following dataframes.



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.

emp\_dept\_df = emp\_df.join(dep\_df, emp\_df.dep\_id == dep\_df.dep\_id, "left\_outer")

emp\_dept\_df.select("emp\_name", "dep\_name").show()

1. 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","mgr.emp\_name").show()

1. 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()

1. 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()

1. 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()