Slip 1

1

1. Create 2 text files. Read the contents in a single RDD. b) Create 2 CSV files. Read the contents in a single RDD.

**import** pyspark

**import** findspark

findspark**.**init()

**from** pyspark.sql **import** SparkSession

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("FileExample")**.**getOrCreate()

*# a) Create 2 text files and read them into a single RDD*

*# Assuming the files 'file1.txt' and 'file2.txt' exist in the specified path*

text\_rdd **=** spark**.**sparkContext**.**textFile(r"C:\Users\HomePC\Desktop\pyspark test\txt1.txt")

*# Show the contents of the RDD*

print("Text File RDD:")

**for** line **in** text\_rdd**.**collect():

print(line)

*# b) Create 2 CSV files and read them into a single RDD*

*# Assuming the files 'file1.csv' and 'file2.csv' exist in the specified path*

csv\_rdd **=** spark**.**sparkContext**.**textFile(r"C:\Users\HomePC\Desktop\pyspark test\csv1.csv")

*# Show the contents of the CSV RDD*

print("\nCSV File RDD:")

**for** line **in** csv\_rdd**.**collect():

print(line)

2.

Create two dataframes one for employee and other for dept. Perform a) Left outer join b) Full outer join c) Inner join

**from** pyspark.sql **import** SparkSession

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("JoinExample")**.**getOrCreate()

*# Create Employee DataFrame*

employee\_data **=** [

(1, "John", 1000),

(2, "Jane", 1200),

(3, "Doe", 1100),

(4, "Alice", 1300)

]

employee\_columns **=** ["emp\_id", "name", "salary"]

employee\_df **=** spark**.**createDataFrame(employee\_data, employee\_columns)

*# Create Department DataFrame*

dept\_data **=** [

(1, "HR"),

(2, "Finance"),

(3, "IT"),

(5, "Marketing")

]

dept\_columns **=** ["emp\_id", "dept"]

dept\_df **=** spark**.**createDataFrame(dept\_data, dept\_columns)

*# a) Left Outer Join*

left\_outer\_join\_df **=** employee\_df**.**join(dept\_df, on**=**"emp\_id", how**=**"left")

print("Left Outer Join:")

left\_outer\_join\_df**.**show()

*# b) Full Outer Join*

full\_outer\_join\_df **=** employee\_df**.**join(dept\_df, on**=**"emp\_id", how**=**"outer")

print("Full Outer Join:")

full\_outer\_join\_df**.**show()

*# c) Inner Join*

inner\_join\_df **=** employee\_df**.**join(dept\_df, on**=**"emp\_id", how**=**"inner")

print("Inner Join:")

inner\_join\_df**.**show()

Slip 2

1.

For the following data and schema create a dataframe and perform the given operations Data: Seq(Row(Row("James;","","Smith"),"36636","M","20000"), Row(Row("Michael","Rose",""),"40288","M","40000"), Row(Row("Robert","","Williams"),"42114","M","10000"), Row(Row("Maria","Anne","Jones"),"39192","F","45000"), Row(Row("Jen","Mary","Brown"),"","F","-1") ) Schema should have the columns as: firstname, middlename, lastname, dob, gender, expenses All columns will be of type String Perform the following operations: a) Change the data type of expenses to Integer b) Rename dob to DateOfBirth c) Create a column that has value expense\*5

**from** pyspark.sql **import** SparkSession, Row

**from** pyspark.sql.types **import** StructType, StructField, StringType

**from** pyspark.sql.functions **import** col

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("DataOperationsExample")**.**getOrCreate()

*# Create the Data*

data **=** [

Row(Row("James", "", "Smith"), "36636", "M", "20000"),

Row(Row("Michael", "Rose", ""), "40288", "M", "40000"),

Row(Row("Robert", "", "Williams"), "42114", "M", "10000"),

Row(Row("Maria", "Anne", "Jones"), "39192", "F", "45000"),

Row(Row("Jen", "Mary", "Brown"), "", "F", "-1")

]

*# Define the schema*

schema **=** StructType([

StructField("name", StructType([

StructField("firstname", StringType(), **True**),

StructField("middlename", StringType(), **True**),

StructField("lastname", StringType(), **True**)

])),

StructField("dob", StringType(), **True**),

StructField("gender", StringType(), **True**),

StructField("expenses", StringType(), **True**)

])

*# Create DataFrame*

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

*# a) Change the data type of 'expenses' to Integer*

df **=** df**.**withColumn("expenses", col("expenses")**.**cast("Integer"))

*# b) Rename 'dob' to 'DateOfBirth'*

df **=** df**.**withColumnRenamed("dob", "DateOfBirth")

*# c) Create a new column that is 'expenses \* 5'*

df **=** df**.**withColumn("expenses\_multiplied", col("expenses") **\*** 5)

*# Show the result*

df**.**select("name.firstname", "name.middlename", "name.lastname", "DateOfBirth", "gender", "expenses", "expenses\_multiplied")**.**show(truncate**=False**)

2.

Create a data frame with a nested array column. Perform the following operations: a) Flatten nested array b) Explode nested array c) Convert array of string to string column

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** flatten, explode, col, concat\_ws

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("ArrayOperationsExample")**.**getOrCreate()

*# Sample data with a nested array column*

data **=** [

(1, [["James", "John"], ["Alice", "Bob"]]),

(2, [["Michael", "Tom"], ["Rose", "Jen"]])

]

*# Create DataFrame*

df **=** spark**.**createDataFrame(data, ["id", "nested\_array"])

*# Show the original DataFrame*

print("Original DataFrame:")

df**.**show(truncate**=False**)

*# a) Flatten the nested array*

df\_flattened **=** df**.**withColumn("flattened\_array", flatten(col("nested\_array")))

print("Flattened Array DataFrame:")

df\_flattened**.**show(truncate**=False**)

*# b) Explode the nested array*

df\_exploded **=** df**.**withColumn("exploded\_array", explode(col("nested\_array")))

print("Exploded Array DataFrame:")

df\_exploded**.**show(truncate**=False**)

*# c) Convert array of strings to a single string column*

df\_string\_col **=** df\_flattened**.**withColumn("array\_to\_string", concat\_ws(", ", col("flattened\_array")))

print("Array to String DataFrame:")

df\_string\_col**.**show(truncate**=False**)

Slip 3

1.

a) Create a data frame with today’s date and timestamp b) Display the hours, minutes and seconds from the timestamp

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** current\_timestamp, hour, minute, second

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("DateTimeExample")**.**getOrCreate()

*# a) Create DataFrame with today's date and timestamp using select*

df **=** spark**.**sql("SELECT current\_timestamp() as timestamp")

*# Show the original DataFrame*

print("Original DataFrame:")

df**.**show(truncate**=False**)

*# b) Display the hours, minutes, and seconds from the timestamp*

df\_time\_extracted **=** df**.**select(

col("timestamp"),

hour(col("timestamp"))**.**alias("hours"),

minute(col("timestamp"))**.**alias("minutes"),

second(col("timestamp"))**.**alias("seconds")

)

print("Extracted Time DataFrame:")

df\_time\_extracted**.**show(truncate**=False**)

2.

For the following employee data showing name, dept and salary, perform the given operations: Data: ("James", "Sales", 3000), ("Michael", "Sales", 4600), ("Robert", "Sales", 4100), ("Maria", "Finance", 3000), ("James", "Sales", 3000), ("Scott", "Finance", 3300), ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000), ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100), (Jason", "Sales", 9000), ("Alice", "Finance", 3700), ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000) a) Create a data frame for the above data b) Display average salary c) Display number of unique departments d) Display number of employees with unique salary

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** avg, countDistinct

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("EmployeeDataExample")**.**getOrCreate()

*# a) Create a DataFrame for the employee data*

data **=** [

("James", "Sales", 3000),

("Michael", "Sales", 4600),

("Robert", "Sales", 4100),

("Maria", "Finance", 3000),

("James", "Sales", 3000),

("Scott", "Finance", 3300),

("Jen", "Finance", 3900),

("Jeff", "Marketing", 3000),

("Kumar", "Marketing", 2000),

("Saif", "Sales", 4100),

("Jason", "Sales", 9000),

("Alice", "Finance", 3700),

("Jenniffer", "Finance", 8900),

("Jenson", "Marketing", 9000)

]

columns **=** ["name", "dept", "salary"]

employee\_df **=** spark**.**createDataFrame(data, columns)

*# Show the original DataFrame*

print("Employee DataFrame:")

employee\_df**.**show(truncate**=False**)

*# b) Display average salary*

average\_salary **=** employee\_df**.**select(avg("salary")**.**alias("average\_salary"))

print("Average Salary:")

average\_salary**.**show()

*# c) Display number of unique departments*

unique\_departments **=** employee\_df**.**select("dept")**.**distinct()**.**count()

print("Number of Unique Departments:", unique\_departments)

*# d) Display number of employees with unique salary*

unique\_salaries **=** employee\_df**.**select("salary")**.**distinct()**.**count()

print("Number of Employees with Unique Salary:", unique\_salaries)

Slip 4

1

1. Create a data frame containing today’s date, date 2022-01-31, date 2021-03-22, date 2024-01-31, date 2023-11-11. b) Store the date in the format MM-DD-YYYY. c) Display the dates in the format DD/MM/YYYY d) Find the number of months between each of the dates and today’s date

from pyspark.sql import SparkSession

from pyspark.sql.functions import current\_date, to\_date, date\_format, months\_between

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("DateExample").getOrCreate()

# Define the list of dates, including today's date

data = [("Today's Date", current\_date()),

("2022-01-31", to\_date(lit("2022-01-31"))),

("2021-03-22", to\_date(lit("2021-03-22"))),

("2024-01-31", to\_date(lit("2024-01-31"))),

("2023-11-11", to\_date(lit("2023-11-11")))]

# Create a DataFrame with the dates

df = spark.createDataFrame(data, ["Event", "Date"])

# b) Store the date in the format MM-DD-YYYY

df\_mm\_dd\_yyyy = df.withColumn("Date\_MM\_DD\_YYYY", date\_format("Date", "MM-dd-yyyy"))

# c) Display the dates in the format DD/MM/YYYY

df\_dd\_mm\_yyyy = df\_mm\_dd\_yyyy.withColumn("Date\_DD\_MM\_YYYY", date\_format("Date", "dd/MM/yyyy"))

# d) Find the number of months between each of the dates and today’s date

df\_with\_months\_between = df\_dd\_mm\_yyyy.withColumn("Months\_Between", months\_between(current\_date(), "Date"))

# Show the final DataFrame

df\_with\_months\_between.select("Event", "Date\_MM\_DD\_YYYY", "Date\_DD\_MM\_YYYY", "Months\_Between").show(truncate=False)

2.

a) Create data frame with a column that contains JSON string. b) Convert the JSON string into Struct type or Map type. c) Extract the Data from JSON and create them as new columns. d) Convert MapType or Struct type to JSON string

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** col, from\_json, to\_json, struct

**from** pyspark.sql.types **import** StructType, StructField, StringType, IntegerType

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("JsonStringExample")**.**getOrCreate()

*# a) Create a DataFrame with a column that contains JSON strings*

data **=** [

(1, '{"name": "James", "age": 30, "city": "New York"}'),

(2, '{"name": "Maria", "age": 25, "city": "Los Angeles"}'),

(3, '{"name": "Robert", "age": 28, "city": "Chicago"}')

]

*# Define columns for the DataFrame*

columns **=** ["id", "json\_string"]

*# Create DataFrame*

json\_df **=** spark**.**createDataFrame(data, columns)

*# Show the original DataFrame*

print("Original DataFrame with JSON strings:")

json\_df**.**show(truncate**=False**)

*# b) Convert the JSON string into Struct type*

*# Define the schema for the JSON*

json\_schema **=** StructType([

StructField("name", StringType(), **True**),

StructField("age", IntegerType(), **True**),

StructField("city", StringType(), **True**)

])

*# Convert JSON string to StructType*

json\_struct\_df **=** json\_df**.**withColumn("json\_data", from\_json(col("json\_string"), json\_schema))

*# Show DataFrame after conversion*

print("DataFrame with JSON converted to StructType:")

json\_struct\_df**.**show(truncate**=False**)

*# c) Extract the data from JSON and create them as new columns*

extracted\_df **=** json\_struct\_df**.**select(

col("id"),

col("json\_data.name")**.**alias("name"),

col("json\_data.age")**.**alias("age"),

col("json\_data.city")**.**alias("city")

)

*# Show DataFrame with extracted columns*

print("DataFrame with extracted columns:")

extracted\_df**.**show(truncate**=False**)

*# d) Convert Struct type to JSON string*

json\_string\_df **=** extracted\_df**.**withColumn("json\_string", to\_json(

struct("name", "age", "city")))

*# Show final DataFrame with JSON string*

print("DataFrame with StructType converted back to JSON string:")

json\_string\_df**.**show(truncate**=False**)

Slip 5

1.

a) Create a data frame containing today’s date, date 2022-01-31, date 2021-03-22, date 2024-01-31 b) Add 5 days to each date and display the result. c) Display the new dates after subtracting 10 days from each date. d) For each date, display year, month, dayofweek, dayofmonth, dayofyear, next\_day,weekofyear

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** col, date\_add, date\_sub, year, month, dayofweek, dayofmonth, dayofyear, next\_day, weekofyear, to\_date

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("DateExample")**.**getOrCreate()

*# Collect current date as a string*

current\_date\_str **=** spark**.**sql("SELECT current\_date()")**.**collect()[0][0]**.**strftime('%Y-%m-%d')

*# a) Create a DataFrame with today's date and specific dates*

data **=** [

(current\_date\_str,), *# Current date as string*

("2022-01-31",),

("2021-03-22",),

("2024-01-31",)

]

columns **=** ["date"]

*# Create DataFrame with string dates and convert to DateType*

date\_df **=** spark**.**createDataFrame(data, columns)**.**withColumn("date", to\_date(col("date"), "yyyy-MM-dd"))

*# Show the original DataFrame*

print("Original DataFrame:")

date\_df**.**show(truncate**=False**)

*# b) Add 5 days to each date*

date\_df\_with\_added\_days **=** date\_df**.**withColumn("date\_plus\_5\_days", date\_add(col("date"), 5))

print("Dates After Adding 5 Days:")

date\_df\_with\_added\_days**.**show(truncate**=False**)

*# c) Subtract 10 days from each date*

date\_df\_with\_subtracted\_days **=** date\_df\_with\_added\_days**.**withColumn("date\_minus\_10\_days", date\_sub(col("date"), 10))

print("Dates After Subtracting 10 Days:")

date\_df\_with\_subtracted\_days**.**show(truncate**=False**)

*# d) Display year, month, dayofweek, dayofmonth, dayofyear, next\_day, weekofyear*

date\_analysis **=** date\_df\_with\_subtracted\_days**.**select(

col("date"),

year(col("date"))**.**alias("year"),

month(col("date"))**.**alias("month"),

dayofweek(col("date"))**.**alias("dayofweek"),

dayofmonth(col("date"))**.**alias("dayofmonth"),

dayofyear(col("date"))**.**alias("dayofyear"),

next\_day(col("date"), 'Sunday')**.**alias("next\_day"),

weekofyear(col("date"))**.**alias("weekofyear")

)

print("Date Analysis:")

date\_analysis**.**show(truncate**=False**)

2.

Refer to the employee.json file. Perform the following operations: a) Print the names of employees above 25 years of age. b) Print the number of employees of different ages.

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** col

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("EmployeeJsonExample")**.**getOrCreate()

*# Load the employee JSON file into a DataFrame*

employee\_df **=** spark**.**read**.**json(r"C:\Users\HomePC\Desktop\pyspark test\js1.json")

*# Show the structure of the DataFrame*

print("Employee DataFrame Schema:")

employee\_df**.**printSchema()

*# a) Print the names of employees above 25 years of age*

employees\_above\_25 **=** employee\_df**.**filter(col("age") **>** 25)**.**select("name")

print("Employees Above 25 Years of Age:")

employees\_above\_25**.**show(truncate**=False**)

*# b) Print the number of employees of different ages*

age\_count **=** employee\_df**.**groupBy("age")**.**count()

print("Number of Employees of Different Ages:")

age\_count**.**show(truncate**=False**)

Slip 6

1.

Create two dataframes one for employee and other for dept. Perform a) Left anti join b) Self join c) Left semi join

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("JoinExample").getOrCreate()

# Sample Employee Data

employee\_data = [(1, "John", "HR"),

(2, "Alice", "Finance"),

(3, "Bob", "IT"),

(4, "David", "Finance"),

(5, "Eve", "HR")]

# Sample Department Data

dept\_data = [("HR", "Human Resources"),

("Finance", "Financial Department"),

("IT", "Information Technology"),

("Sales", "Sales Department")]

# Creating Employee DataFrame

employee\_df = spark.createDataFrame(employee\_data, ["emp\_id", "emp\_name", "dept"])

# Creating Department DataFrame

dept\_df = spark.createDataFrame(dept\_data, ["dept", "dept\_name"])

# Show the DataFrames

print("Employee DataFrame:")

employee\_df.show()

print("Department DataFrame:")

dept\_df.show()

# Left Anti Join: Employees whose departments are not in dept\_df

left\_anti\_join\_df = employee\_df.join(dept\_df, employee\_df.dept == dept\_df.dept, "left\_anti")

print("Left Anti Join:")

left\_anti\_join\_df.show()

# Self Join: Find pairs of employees from the same department

self\_join\_df = employee\_df.alias("emp1").join(

employee\_df.alias("emp2"),

on=[employee\_df["dept"] == employee\_df["dept"]],

how="inner"

).select("emp1.emp\_name", "emp2.emp\_name", "emp1.dept")

print("Self Join (Employees from the same department):")

self\_join\_df.show()

# Left Semi Join: Employees whose departments exist in dept\_df

left\_semi\_join\_df = employee\_df.join(dept\_df, employee\_df.dept == dept\_df.dept, "left\_semi")

print("Left Semi Join:")

left\_semi\_join\_df.show()

2.

a) Create two case classes – Student and Address b) Create schema from these case classes

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("CaseClassSchema").getOrCreate()

# a) Define 'Student' and 'Address' classes in Python

class Student:

def \_\_init\_\_(self, student\_id, name, age, address):

self.student\_id = student\_id

self.name = name

self.age = age

self.address = address

class Address:

def \_\_init\_\_(self, street, city, zip\_code):

self.street = street

self.city = city

self.zip\_code = zip\_code

# b) Define schema for 'Address' and 'Student' classes using StructType

# Schema for Address class

address\_schema = StructType([

StructField("street", StringType(), True),

StructField("city", StringType(), True),

StructField("zip\_code", StringType(), True)

])

# Schema for Student class

student\_schema = StructType([

StructField("student\_id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("age", IntegerType(), True),

StructField("address", address\_schema, True)

])

# Example: Creating a DataFrame using the schema

# Sample data

data = [(1, "John Doe", 20, ("123 Main St", "New York", "10001")),

(2, "Alice Smith", 22, ("456 Oak Ave", "Los Angeles", "90001")),

(3, "Bob Johnson", 21, ("789 Maple Rd", "Chicago", "60601"))]

# Create DataFrame

df = spark.createDataFrame(data, schema=student\_schema)

# Show DataFrame

df.show(truncate=False)

Slip 7

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, LongType

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("MapKeysExample").getOrCreate()

# Define the schema for the DataFrame

schema = StructType([

StructField("emp\_id", IntegerType(), True),

StructField("dept", StringType(), True),

StructField("properties", StructType([

StructField("salary", LongType(), True),

StructField("location", StringType(), True)

]), True)

])

# Create sample data

data = [

(1, "HR", (50000, "New York")),

(2, "Finance", (70000, "San Francisco")),

(3, "IT", (60000, "Chicago"))

]

# Create the DataFrame

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

# Show the DataFrame

print("DataFrame:")

df.show(truncate=False)

# Register the DataFrame as a SQL temporary view

df.createOrReplaceTempView("employee\_table")

# Write a SQL query to return the map keys (field names of the 'properties' struct)

# In this case, since 'properties' is a struct, we extract the column names from the struct.

query = """

SELECT

emp\_id,

dept,

properties.salary,

properties.location

FROM employee\_table

"""

# Execute the SQL query

result = spark.sql(query)

# Show the result

print("Query Result:")

result.show(truncate=False)

2.

For the following employee data showing name, dept and salary, perform the given operations: Data: ("James", "Sales", 3000), ("Michael", "Sales", 4600), ("Robert", "Sales", 4100), ("Maria", "Finance", 3000), ("James", "Sales", 3000), ("Scott", "Finance", 3300), ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000), ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100), (Jason", "Sales", 9000), ("Alice", "Finance", 3700), ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000) a) Create a data frame for the above data b) Find the highest salary value c) Find the lowest salary value d) Find the standard deviation for the salary

**from** pyspark.sql **import** SparkSession

**from** pyspark.sql.functions **import** col, max, min, stddev

*# Initialize Spark session*

spark **=** SparkSession**.**builder**.**master("local")**.**appName("EmployeeSalaryExample")**.**getOrCreate()

*# Sample employee data*

data **=** [

("James", "Sales", 3000),

("Michael", "Sales", 4600),

("Robert", "Sales", 4100),

("Maria", "Finance", 3000),

("James", "Sales", 3000),

("Scott", "Finance", 3300),

("Jen", "Finance", 3900),

("Jeff", "Marketing", 3000),

("Kumar", "Marketing", 2000),

("Saif", "Sales", 4100),

("Jason", "Sales", 9000),

("Alice", "Finance", 3700),

("Jenniffer", "Finance", 8900),

("Jenson", "Marketing", 9000)

]

*# Create DataFrame with schema*

columns **=** ["name", "dept", "salary"]

employee\_df **=** spark**.**createDataFrame(data, columns)

*# Show the DataFrame*

print("Employee DataFrame:")

employee\_df**.**show(truncate**=False**)

*# b) Find the highest salary value*

highest\_salary **=** employee\_df**.**select(max("salary"))**.**collect()[0][0]

print(f"Highest Salary: {highest\_salary}")

*# c) Find the lowest salary value*

lowest\_salary **=** employee\_df**.**select(min("salary"))**.**collect()[0][0]

print(f"Lowest Salary: {lowest\_salary}")

*# d) Find the standard deviation for the salary*

stddev\_salary **=** employee\_df**.**select(stddev("salary"))**.**collect()[0][0]

print(f"Standard Deviation of Salary: {stddev\_salary}")

Slip 9

1.

Create a Spark RDD using 5 different Functions

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("RDDExample").getOrCreate()

# Create an RDD from a list

data = [("John", 28), ("Alice", 23), ("Bob", 34), ("David", 29), ("Eva", 22)]

rdd = spark.sparkContext.parallelize(data)

# 1. Filter: Keep only records where age > 25

rdd\_filtered = rdd.filter(lambda x: x[1] > 25)

# 2. Map: Add a new field that categorizes employees as "Senior" or "Junior" based on age

rdd\_mapped = rdd\_filtered.map(lambda x: (x[0], x[1], "Senior" if x[1] > 30 else "Junior"))

# 3. FlatMap: Split names into individual characters

rdd\_flatmapped = rdd\_mapped.flatMap(lambda x: list(x[0]))

# 4. ReduceByKey (not directly applicable here since there are no key-value pairs)

# To demonstrate, let’s create an RDD with counts for each letter in names

rdd\_letter\_counts = rdd\_flatmapped.map(lambda x: (x, 1)).reduceByKey(lambda a, b: a + b)

# 5. Collect: Bring the RDD data back to the driver and print the results

result = rdd\_letter\_counts.collect()

# Show the result

print("Letter counts in employee names:")

print(result)

2.

Write example for following Spark RDD Actions: a. aggregate b. treeAggregate c. fold d. reduce e. collect

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("RDDActionExamples").getOrCreate()

# Create an RDD

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

# 1. aggregate: Perform a sum and count aggregation

zero\_value = (0, 0) # Initial value (sum, count)

seq\_op = lambda acc, x: (acc[0] + x, acc[1] + 1) # Operation within partitions

comb\_op = lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]) # Operation between partitions

result\_aggregate = rdd.aggregate(zero\_value, seq\_op, comb\_op)

print("Aggregate result (sum and count):", result\_aggregate)

# 2. treeAggregate: Similar to aggregate but with a tree structure to minimize shuffle

result\_tree\_agg = rdd.treeAggregate(zero\_value, seq\_op, comb\_op, depth=2)

print("Tree Aggregate result (sum and count):", result\_tree\_agg)

# 3. fold: Aggregates elements of each partition and combines results with a neutral "zero" value

result\_fold = rdd.fold(0, lambda acc, x: acc + x)

print("Fold result (sum):", result\_fold)

# 4. reduce: Combines elements of the RDD using a specified associative binary function (sum in this case)

result\_reduce = rdd.reduce(lambda x, y: x + y)

print("Reduce result (sum):", result\_reduce)

# 5. collect: Collects the entire RDD and brings it to the driver as a Python list

result\_collect = rdd.collect()

print("Collect result:", result\_collect)

# Stop the Spark session

spark.stop()

Slip 10

1.

Write example for following Spark RDD Actions: a. count b. countApproxDistinct c. first d. top e. Min

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("RDDActionExamples").getOrCreate()

# Create an RDD

rdd = spark.sparkContext.parallelize([10, 20, 30, 40, 50, 50, 60, 10, 70, 80, 90])

# 1. count: Returns the number of elements in the RDD

result\_count = rdd.count()

print("Count result:", result\_count)

# 2. countApproxDistinct: Returns an approximate count of distinct elements in the RDD

result\_count\_approx\_distinct = rdd.countApproxDistinct()

print("Approximate Distinct Count result:", result\_count\_approx\_distinct)

# 3. first: Returns the first element of the RDD

result\_first = rdd.first()

print("First element:", result\_first)

# 4. top: Returns the top 'n' elements in the RDD, sorted in descending order

result\_top = rdd.top(3) # Top 3 elements

print("Top 3 elements:", result\_top)

# 5. min: Returns the minimum element of the RDD

result\_min = rdd.min()

print("Min element:", result\_min)

# Stop the Spark session

spark.stop()

2.

Write Spark Pair RDD Functions.

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("PairRDDExamples").getOrCreate()

# Create a Pair RDD

pair\_rdd = spark.sparkContext.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4), ('c', 5)])

# 1. mapValues: Multiply each value by 2

result\_map\_values = pair\_rdd.mapValues(lambda x: x \* 2).collect()

print("mapValues result:", result\_map\_values)

# 2. reduceByKey: Sum the values for each key

result\_reduce\_by\_key = pair\_rdd.reduceByKey(lambda x, y: x + y).collect()

print("reduceByKey result:", result\_reduce\_by\_key)

# 3. groupByKey: Group all values by key

result\_group\_by\_key = pair\_rdd.groupByKey().mapValues(list).collect()

print("groupByKey result:", result\_group\_by\_key)

# 4. flatMapValues: Split each value into a range of values

result\_flat\_map\_values = pair\_rdd.flatMapValues(lambda x: range(1, x + 1)).collect()

print("flatMapValues result:", result\_flat\_map\_values)

# 5. keys: Retrieve only the keys from the pair RDD

result\_keys = pair\_rdd.keys().collect()

print("keys result:", result\_keys)

# 6. values: Retrieve only the values from the pair RDD

result\_values = pair\_rdd.values().collect()

print("values result:", result\_values)

# 7. sortByKey: Sort the RDD by key

result\_sort\_by\_key = pair\_rdd.sortByKey().collect()

print("sortByKey result:", result\_sort\_by\_key)

# Stop the Spark session

spark.stop()

Slip 11

1.

Get new dates by adding 4 days, and subtracting 7 days in below dates "2020-01-02","2023-01-15","2025-01-30"

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, date\_add, date\_sub

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("DateManipulation").getOrCreate()

# Create a DataFrame with the given dates

data = [("2020-01-02",), ("2023-01-15",), ("2025-01-30",)]

columns = ["date"]

df = spark.createDataFrame(data, columns)

# Add 4 days to each date

df\_with\_added\_days = df.withColumn("date\_plus\_4\_days", date\_add(col("date"), 4))

# Subtract 7 days from each date

df\_with\_subtracted\_days = df\_with\_added\_days.withColumn("date\_minus\_7\_days", date\_sub(col("date"), 7))

# Show the result

df\_with\_subtracted\_days.show(truncate=False)

# Stop the Spark session

spark.stop()

2.

Use the Operation Read CSV file on RDD with Scala operation

import org.apache.spark.sql.SparkSession

object ReadCsvToRDD {

def main(args: Array[String]): Unit = {

// Initialize Spark session

val spark = SparkSession.builder

.appName("Read CSV to RDD Example")

.master("local[\*]") // Use all available cores

.getOrCreate()

// Path to the CSV file

val csvFilePath = "C:\\Users\\HomePC\\Desktop\\pyspark test\\data.csv"

// Read the CSV file into an RDD

val csvRDD = spark.sparkContext.textFile(csvFilePath)

// Show the contents of the RDD

println("CSV RDD Contents:")

csvRDD.collect().foreach(println)

// Optionally, you can split the lines into columns based on a delimiter

val header = csvRDD.first() // Get the header

val dataRDD = csvRDD

.filter(row => row != header) // Filter out the header

.map(line => line.split(",")) // Split by comma

// Show the data RDD contents

println("\nData RDD Contents (as arrays):")

dataRDD.collect().foreach(arr => println(arr.mkString(", ")))

// Stop the Spark session

spark.stop()

}

}

Slip 12

1.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, size

# Initialize Spark session

spark = SparkSession.builder \

.appName("Create Table with Array and Map") \

.master("local[\*]") \

.getOrCreate()

# Create a DataFrame with the required structure

data = [

("James", ["Java", "Scala"], {"hair": "black", "eye": "brown"}),

("Michael", ["Scala", "Java", None], {"hair": "brown", "eye": None})

]

columns = ["Name", "knownlanguages", "properties"]

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

# Show the original DataFrame

print("Original DataFrame:")

df.show(truncate=False)

# Perform operations on the array column

df\_with\_array\_ops = df \

.withColumn("first\_language", col("knownlanguages")[0]) \

.withColumn("languages\_count", size(col("knownlanguages")))

# Perform operations on the map column

df\_with\_map\_ops = df\_with\_array\_ops \

.withColumn("hair\_color", col("properties")["hair"]) \

.withColumn("eye\_color", col("properties")["eye"])

# Show the result DataFrame with operations

print("\nDataFrame after array and map operations:")

df\_with\_map\_ops.show(truncate=False)

# Stop the Spark session

spark.stop()

2.

Find current timestamp and hour, Minute, second separately for today's date

from pyspark.sql import SparkSession

from pyspark.sql.functions import current\_timestamp, hour, minute, second

# Initialize Spark session

spark = SparkSession.builder \

.appName("Current Timestamp Example") \

.master("local[\*]") \

.getOrCreate()

# Create a DataFrame with the current timestamp

current\_time\_df = spark.createDataFrame([(1,)], ["dummy"]) \

.select(current\_timestamp().alias("current\_timestamp"))

# Show the current timestamp

print("Current Timestamp:")

current\_time\_df.show(truncate=False)

# Extract hour, minute, and second

current\_time\_df = current\_time\_df \

.select(

col("current\_timestamp"),

hour(col("current\_timestamp")).alias("hour"),

minute(col("current\_timestamp")).alias("minute"),

second(col("current\_timestamp")).alias("second")

)

# Show the results

print("\nExtracted Hour, Minute, and Second:")

current\_time\_df.show(truncate=False)

# Stop the Spark session

spark.stop()

Slip 13

1.

Write a Maven dependencies for writing and Reading Avro Data File

**from** pyspark.sql **import** SparkSession

*# Initialize Spark session with Avro support*

spark **=** SparkSession**.**builder \

**.**appName("Avro Example") \

**.**config("spark.jars.packages", "org.apache.spark:spark-avro\_2.12:3.5.3") \

**.**getOrCreate()

*# Read Avro data into DataFrame*

df **=** spark**.**read**.**format("avro")**.**load(r"C:\Users\HomePC\Downloads\userdata1.avro")

df**.**show()

*# Write the DataFrame back to Avro format*

df**.**write**.**format("avro")**.**save(r"C:\Users\HomePC\Downloads\userdata3.avro")

2.

Create the following two data frames and apply Inner and Right Outer join

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder \

.appName("Join Example") \

.master("local[\*]") \

.getOrCreate()

# Create the first DataFrame (Employee DataFrame)

employee\_data = [

(1, "James", None, 2018, 101, "M"),

(2, "Michael", 1, 2019, 102, "M"),

(3, "Robert", 1, 2020, 101, "M"),

(4, "Maria", 2, 2021, 103, "F"),

(5, "Jen", None, 2017, 104, "F")

]

employee\_columns = ["emp\_id", "name", "superior\_emp\_id", "year\_joined", "emp\_dept\_id", "gender"]

employee\_df = spark.createDataFrame(employee\_data, schema=employee\_columns)

# Create the second DataFrame (Department DataFrame)

department\_data = [

(101, "HR"),

(102, "Finance"),

(103, "IT"),

(104, "Sales"),

(105, "Marketing") # This department does not have any employee

]

department\_columns = ["dept\_id", "dept\_name"]

department\_df = spark.createDataFrame(department\_data, schema=department\_columns)

# Show the Employee DataFrame

print("Employee DataFrame:")

employee\_df.show(truncate=False)

# Show the Department DataFrame

print("Department DataFrame:")

department\_df.show(truncate=False)

# Inner Join

inner\_join\_df = employee\_df.join(department\_df, employee\_df.emp\_dept\_id == department\_df.dept\_id, "inner")

print("\nInner Join Result:")

inner\_join\_df.select("emp\_id", "name", "dept\_name").show(truncate=False)

# Right Outer Join

right\_outer\_join\_df = employee\_df.join(department\_df, employee\_df.emp\_dept\_id == department\_df.dept\_id, "right")

print("\nRight Outer Join Result:")

right\_outer\_join\_df.select("emp\_id", "name", "dept\_name").show(truncate=False)

# Stop the Spark session

spark.stop()