Practical 1 Title :- Practicals on RDD(REsilient Distributed Dataset) with scala operations and transformation Date

Aim : Learning RDD data format of scala and different transformation & action functions

Create RDD object using parallelize

```
scala> val rdd = sc.parallelize(Array(1,2,3,4,5,6,7,8,9,10))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at parallelize at <console>:23
scala>rdd.foreach(println)

scala> val rdd=spark.sparkContext.parallelize(Seq(("Java", 20000), ("Python", 100000),
   | ("Scala", 3000))) rdd: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[7] at parallelize at
<console>:22

scala> rdd.foreach(println)
(Python,100000) (0 + 2) / 2]
(Java,20000)
(Scala,3000)
```

ReadText File in rdd object and display its content

```
scala> val rd=sc.textFile("c:/test/test1.txt")
rd: org.apache.spark.rdd.RDD[String] = c:/test/test1.txt MapPartitionsRDD[8] at textFile at
<console>:23

scala> rd.foreach(println)
used for big data analysis
This is a spark tutorial
From sparkbyexample
spark is a open source tool

scala> rd.collect
res7: Array[String] = Array(This is a spark tutorial, From sparkbyexample, spark is a open
source tool, used for big data analysis)
```

Split the file on spaces to get the words

```
scala> val rdd2 = rd.flatMap(f=>f.split(" ")) rdd2:
```

```
org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[10] at flatMap at
<console>:23
scala> rdd2.collect
res11: Array[String] = Array(This, is, a, spark, tutorial, From, sparkbyexample, spark, is, a, open,
source, tool, used, for, big, data, analysis)
```

Read Number of partitions

```
scala> rdd2.getNumPartitions
res12: Int = 2
```

Create new partitions scala>

```
val rd2=rdd2.repartition(4)
rd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[18] at repartition at <console>:23

scala> rd2.getNumPartitions
res15: Int = 4
```

Create RDD Object with key & value

```
scala> val rdd3= rdd2.map(m=>(m,1))
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[19] at map at <console>:23

scala> rdd3.collect
res16: Array[(String, Int)] = Array((This,1), (is,1), (a,1), (spark,1), (tutorial,1), (From,1),
(sparkbyexample,1), (spark,1), (is,1), (a,1), (open,1), (source,1), (tool,1), (used,1), (for,1),
(big,1), (data,1), (analysis,1))

scala> rdd3.getNumPartitions
res17: Int = 2
```

List the words starts with letter ‘s’

```
scala> val rdd4=rdd3.filter(l=>l._1.startsWith("s"))
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[20] at filter at <console>:23

scala> rdd4.foreach(println)
(spark,1)
(sparkbyexample,1)
```

```
(spark,1)
(source,1)
```

Sort RDD object by Key

```
scala> val rdd6=rdd5.map(l=>(l._2,l._1)).sortByKey()
rdd6: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[27] at sortByKey at <console>:23
```

#Actions

```
scala> rdd6.foreach(println)
(1,sparkbyexample)
(2,is)
(1,big)
(2,spark)
(1,From)
(2,a)
(1,data)
(1,This)
(1,tutorial)
(1,tool)
(1,open)
(1,analysis)
(1,source)
(1,for)
(1,used)
```

Find the length of RDD object

```
scala> rdd6.count()
res23: Long = 15
```

List the first element

```
scala> rdd6.first()
res24: (Int, String) = (1,sparkbyexample)
```

List the element with max key

```
scala> rdd6.max()
res26: (Int, String) = (2,spark)
```

Retrieve 5 elements from RDD

```
scala> rdd6.take(5)
res27: Array[(Int, String)] = Array((1,sparkbyexample), (1,big), (1,From), (1,data), (1,This))
```

Read CSV file in RDD

```
scala> val rddcsv=sc.textFile("D:/TYDS/sample.csv") rddcsv:
org.apache.spark.rdd.RDD[String] = D:/TYDS/sample.csv
MapPartitionsRDD[1] at textFile at <console>:23

scala> rddcsv.collect() res0: Array[String] =
Array(rno,name,class, 1,A,TYDS, 2,B,TYDS,
3,C,SYDS, 4,D,SYDS, 5,E,TYDS, 6,F,TYDS)

scala> val rdd=rddcsv.map(f=>{f.split(",")}) rdd:
org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at
map at <console>:23

scala> rdd.foreach(f=>println(f))
[Ljava.lang.String;@6bd94b6a
[Ljava.lang.String;@380e8b12
[Ljava.lang.String;@3b424f3
[Ljava.lang.String;@521ebb37
[Ljava.lang.String;@5af886ec
[Ljava.lang.String;@15c4d7ab
[Ljava.lang.String;@2ad5280e


scala> rdd.foreach(f=>println(f(0),f(1),f(2)))
(rno,name,class)
(4,D,SYDS)
```

```
(1,A,TYDS)
(5,E,TYDS)
(2,B,TYDS)
(3,C,SYDS)
(6,F,TYDS)
scala> val rdd1 = rdd.mapPartitionsWithIndex { (idx, iter) => if (idx ==
0) iter.drop(1) else iter } rdd1:
org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[4] at
mapPartitionsWithIndex at <console>:23

scala> rdd1.foreach(f=>println(f(0),f(1),f(2))
     | )
(4,D,SYDS)
(1,A,TYDS)
(5,E,TYDS)
(2,B,TYDS)
(6,F,TYDS)
(3,C,SYDS)
```

Reading Multiple CSV File scala> val
rddmcv=sc.textFile("D:/TYDS/sample.csv,D:/TYDS/sample1.csv"
) rddmcv: org.apache.spark.rdd.RDD[String] =
D:/TYDS/sample.csv,D:/TYDS/sample1.csv MapPartitionsRDD[10] at
textFile at <console>:23

scala> rddmcv.collect() res11: Array[String] =
Array(rno,name,class, 1,A,TYDS, 2,B,TYDS,

```
3,C,SYDS, 4,D,SYDS, 5,E,TYDS, 6,F,TYDS, class,Subject,
TYDS,DE, TYDS,CV, SYDS,DW, SYDS,LA, TYDS,RPA, TYDS,BF)
scala> rddmcv.foreach(f=>println(f))
rno,name,class
class,Subject
1,A,TYDS
TYDS,DE
2,B,TYDS
TYDS,CV
3,C,SYDS
SYDS,DW
4,D,SYDS
SYDS,LA
5,E,TYDS
TYDS,RPA
6,F,TYDS
TYDS,BF
```

b) RDD Action Function

```
scala> val listRdd = sc.parallelize(List(1,2,3,4,5,3,2)) listRdd:
org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at
parallelize at <console>:23

scala> listRdd.collect() res19: Array[Int] =
Array(1, 2, 3, 4, 5, 3, 2)

scala> def param0= (accu:Int, v:Int) => accu + v
param0: (Int, Int) => Int
```

```
scala> def param1= (accu1:Int,accu2:Int) => accu1 + accu2
param1: (Int, Int) => Int
scala> println("aggregate : "+listRdd.aggregate(0)(param0,param1))
aggregate : 20



scala> def param3= (accu:Int, v:(String,Int)) => accu + v._2
param3: (Int, (String, Int)) => Int

scala> def param4= (accu1:Int,accu2:Int) => accu1 + accu2
param4: (Int, Int) => Int

scala> println("aggregate :
"+inputRDD.aggregate(0)(param3,param4))
aggregate : 181

scala> listRdd.fold(0){(acc,v)=>
     | val sum =acc+v
     | sum
     | } res28:
Int = 20

scala> listRdd.first()
res29: Int = 1

scala> listRdd.min()
res31: Int = 1
```

```
scala> listRdd.max()
res32: Int = 5 scala>
listRdd.count() res33:
Long = 7



scala> listRdd.top(3) res35:
Array[Int] = Array(5, 4, 3)

scala> listRdd.collect() res36: Array[Int] =
Array(1, 2, 3, 4, 5, 3, 2)

scala> listRdd.take(3) res37:
Array[Int] = Array(1, 2, 3)
```

Practical 2

Title :- Practical on the DataFrame operations

Aim: Learning how to create dataframe and performing various operations on it.

a) Creating dataframe object

1. Create an empty DataFrame:

```
val df = spark.emptyDataFrame

df: org.apache.spark.sql.DataFrame = []
```

2. Adding or Updating a column in empty dataframe:

```
val data = Seq(("Java", "20000"), ("Python", "100000"), ("Scala", "3000"))

val rdd = spark.sparkContext.parallelize(data)

val schema = StructType( Array(

StructField("language", StringType,true),

StructField("users", StringType,true)

))

val rowRDD = rdd.map(attributes => Row(attributes._1, attributes._2))

val dfFromRDD3 = spark.createDataFrame(rowRDD,schema)

dfFromRDD3.printSchema()

dfFromRDD3.show()
```

3. Create a DataFrame using StructType & StructField schema

```
val data = Seq(

Row(8, "bat"),

Row(64, "mouse"),

Row(-27, "horse")

)

val schema = StructType(

List(
StructField("number", IntegerType, true),
StructField("word", StringType, true)
)
)
val df = spark.createDataFrame(
spark.sparkContext.parallelize(data), schema)
df.printSchema()
df.show()
```

4. Convert Spark RDD to DataFrame

```
import spark.implicits._
val columns = Seq("language","users_count")
val data = Seq(("Java", "20000"), ("Python", "100000"), ("Scala", "3000"))
val rdd = spark.sparkContext.parallelize(data)

val dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
dfFromRDD1.show()

val dfFromRDD1 = rdd.toDF("language","users_count")
dfFromRDD1.printSchema()
dfFromRDD1.show()

val columns = Seq("language","users_count")
val dfFromRDD2 = spark.createDataFrame(rdd).toDF(columns:_*)
dfFromRDD2.show()
```

5) Read CSV file into DataFrame

```
scala>val df = spark.read.csv("D:/archana/sample1.csv")

scala>df.printSchema()

root

 |-- _c0: string (nullable = true)

 |-- _c1: string (nullable = true)
```

Read CSV with Header

```
scala>val df = spark.read.option("header",true).csv("D:/archana/sample1.csv")

df: org.apache.spark.sql.DataFrame = [class: string, Subject: string]

 scala>df.printSchema()

root

 |-- class: string (nullable = true)

 |-- Subject: string (nullable = true)
```

```
val options = Map("inferSchema"->"true","delimiter"->",","header"->"true")

val df5 = spark.read.options(options).csv("D:/archana/sample1.csv ")

df5.printSchema()
```

5) Read CSV file into DataFrame

```
scala>val df = spark.read.csv("D:/archana/sample1.csv")

scala>df.printSchema()

root

 |-- _c0: string (nullable = true)

 |-- _c1: string (nullable = true)
```

Read CSV with Header

```
scala>val df = spark.read.option("header",true).csv("D:/archana/sample1.csv")
```

```
df: org.apache.spark.sql.DataFrame = [class: string, Subject: string]
```

```
 scala>df.printSchema()
root
 |-- class: string (nullable = true)
 |-- Subject: string (nullable = true)
```

You can use the options() method to specify multiple options at a time.

```
val options = Map("inferSchema"->"true","delimiter"->",","header"->"true")
val df5 = spark.read.options(options).csv("D:/archana/sample1.csv ")
df5.printSchema()
```

b) Different Operations on data frame

1. Drop a column in DataFrame:

```
dfFromRDD3.printSchema()
val df = dfFromRDD3.drop("users")
df.printSchema()
df.show()
```

2. Combine two or more DataFrames using union:

```
val data = Seq(
Row(1, "dog"),
Row(2, "cat"),
)
val df = spark.createDataFrame( spark.sparkContext.parallelize(data), schema)
df.show()
```

```
val data1 = Seq(
Row(18, "rose"),
Row(6, "lotus"),
)
val df1 = spark.createDataFrame(
spark.sparkContext.parallelize(data1), schema)
df1.show()
val df3 = df.union(df1)
df3.show()
```

3. Pivot and Unpivot a DataFrame:

```
val data = Seq(("Banana",1000,"USA"), ("Carrots",1500,"USA"),
("Beans",1600,"USA"),
("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"),
("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"),
("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico"))
import spark.sqlContext.implicits._
val df = data.toDF("Product","Amount","Country")
df.show()
```

Pivot Spark DataFrame:

```
val pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.show()
```

Unpivot Spark DataFrame:

```
val unPivotDF = pivotDF.select($"Product",expr("stack(3, 'Canada', Canada, 'China',
China, 'Mexico', Mexico) as (Country, Total)")).where("Total is not null")
unPivotDF.show()
```

Practical 3

Title : Practical on the Spark Array and Map operations

Aim : <u>ArrayType in Spark DataFrames</u>

<u>Creating DataFrames with ArrayType Columns</u>

```
import org.apache.spark.sql.types.{ArrayType, IntegerType, StringType, StructField, StructType}
import org.apache.spark.sql.{Row, SparkSession}

val spark = SparkSession.builder.appName("ArrayTypeExample").getOrCreate()
val data = Seq(
  Row(Array("apple", "banana", "cherry")),
  Row(Array("cucumber", "tomato", "lettuce"))
)
val schema = StructType(Array(
  StructField("fruits_vegetables", ArrayType(StringType), nullable = true)
))
val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)
df.show(false)
```

Output

```
// +-------------------+
// |  fruits_vegetables |
// +-------------------+
// | [apple, banana, cherry] |
// | [cucumber, tomato, lettuce] |
// +-------------------+
```

Accessing Elements in an ArrayType Column

```
val selectedElementDf = df.withColumn("first_fruit", element_at(col("fruits_vegetables"), 1))
selectedElementDf.show()
```

Output

```
// +-------------------+-----------+
// |  fruits_vegetables| first_fruit |
// +-------------------+-----------+
// | [apple, banana, cherry] |    apple |
// | [cucumber, tomato, lettuce] | cucumber |
// +-------------------+-----------+
```

Adding and Removing Elements

You can modify arrays by adding or removing elements using functions like `array_union`, `array_except`, and `array_intersect`.

```
val newArrayDf = df.withColumn("more_fruits", array_union(col("fruits_vegetables"), array(lit("kiwi"))))
newArrayDf.show(false)
```

Outtput

```
// +------------------------+
// |    fruits_vegetables    |      more_fruits      |
// +------------------------+
// | [apple, banana, cherry] | [apple, banana, cherry, kiwi] |
// | [cucumber, tomato, lettuce] | [cucumber, tomato, lettuce, kiwi] |
// +------------------------+
```

Exploding Arrays

Often, you'll want to transform each element of an array into a separate row. This is done using the `explode` function. It converts an ArrayType column into a set of rows, one for each element in the array.

```
val explodedDf = df.withColumn("exploded", explode(col("fruits_vegetables")))
explodedDf.show(false)
```

Output

```
// +------------------------+---------+
// |    fruits_vegetables    | exploded |
// +------------------------+---------+
// | [apple, banana, cherry] |    apple |
// | [apple, banana, cherry] |   banana |
// | [apple, banana, cherry] |   cherry |
// | [cucumber, tomato, lettuce] | cucumber |
// | [cucumber, tomato, lettuce] |   tomato |
// | [cucumber, tomato, lettuce] |  lettuce |
// +------------------------+---------+
```

<u>**Creating DataFrames with MapType Columns**</u>

Defining MapType Columns

To create a DataFrame with a `MapType` column, we first need to define the schema. A `MapType` requires three components: a data type for the key, a data type for the value, and a boolean to specify whether the value can contain nulls. Below is how you define a `MapType` in Scala:

```
import org.apache.spark.sql.types.{ArrayType, IntegerType, StringType, StructField, StructType}

import org.apache.spark.sql.types._

import org.apache.spark.sql.Row


// Create a schema with a MapType column

val schema = StructType( StructField("id", IntegerType, nullable = false) :: StructField("properties",
MapType(StringType, StringType), nullable = true))

// Create some sample data

 val data = Seq( Row(1, Map("color" -> "red", "size" -> "M")),

 Row(2, Map("color" -> "blue", "size" -> "L")),

Row(3, Map("color" -> "green", "size" -> "S")) )

// Create a DataFrame

val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)

 // Show the DataFrame

df.show(false)
```

Creating an Array of Struct Column

Let's assume you have a DataFrame with individual columns that you want to group into an array of structs.

```
import org.apache.spark.sql.types._

import org.apache.spark.sql.Row

import org.apache.spark.sql.functions._


val schema = StructType(

  StructField("id", IntegerType, nullable = false) ::

  StructField("attributes", ArrayType(

    StructType(

      StructField("name", StringType, nullable = false) ::

      StructField("value", StringType, nullable = true) :: Nil

    )

  ), nullable = true) :: Nil

)


val data = Seq(

  Row(1, Array(Row("color", "red"), Row("size", "M"))),

  Row(2, Array(Row("color", "blue"), Row("size", "L"), Row("brand", "Nike"))),

  Row(3, Array(Row("color", "green"), Row("size", "S"), Row("brand", "Adidas")))

)


val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)


df.show(false)
```

Expected Output Before Exploding

```
+---+-------------------------------------+
|id |attributes                           |
+---+-------------------------------------+
|1  |[{color, red}, {size, M}]            |
|2  |[{color, blue}, {size, L}, {brand, Nike}]|
|3  |[{color, green}, {size, S}, {brand, Adidas}]|
+---+-------------------------------------+
// Explode the 'attributes' array of structs
val explodedDF = df.withColumn("attribute", explode($"attributes"))

// Show the exploded DataFrame
explodedDF.show(false)
```

Output After Exploding

The explode function will create a new row for each struct in the attributes array:

```
+---+-------------------------------------------+---------------+
|id |attributes                                 |attribute      |
+---+-------------------------------------------+---------------+
|1  |[{color, red}, {size, M}]                  |{color, red}   |
|1  |[{color, red}, {size, M}]                  |{size, M}      |
|2  |[{color, blue}, {size, L}, {brand, Nike}]   |{color, blue}  |
|2  |[{color, blue}, {size, L}, {brand, Nike}]   |{size, L}      |
|2  |[{color, blue}, {size, L}, {brand, Nike}]   |{brand, Nike}  |
|3  |[{color, green}, {size, S}, {brand, Adidas}]|{color, green} |
|3  |[{color, green}, {size, S}, {brand, Adidas}]|{size, S}      |
|3  |[{color, green}, {size, S}, {brand, Adidas}]|{brand, Adidas}|
+---+-------------------------------------------+---------------+
```

Creating a DataFrame with a nested array (an array within an array) involves defining a schema that includes a nested ArrayType. Below is a step-by-step guide to creating such a DataFrame in Spark.

Define the Schema

Let's define a schema where one of the columns is a nested array. The schema will include an ArrayType that contains another ArrayType as its element type.

```
import org.apache.spark.sql.types._

import org.apache.spark.sql.Row


// Define the schema with a nested array

val schema = StructType(

  StructField("id", IntegerType, nullable = false) ::

  StructField("nested_array", ArrayType(

    ArrayType(StringType)

  ), nullable = true) :: Nil

)
```

Create Sample Data

Next, create some sample data that matches the schema. Each row will contain a nested array.

scala

Copy code

```
// Create sample data with nested arrays

val data = Seq(

  Row(1, Array(Array("a", "b", "c"), Array("d", "e"))),

  Row(2, Array(Array("x", "y"), Array("z"))),

  Row(3, Array(Array("1", "2", "3", "4"), Array("5", "6")))

)


// Create the DataFrame

val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)


// Show the DataFrame

df.show(false)
```

Output

The show(false) command will output the following DataFrame:

```
+---+------------------------+
|id |nested_array            |
+---+------------------------+
|1  |[[a, b, c], [d, e]]     |
|2  |[[x, y], [z]]           |
|3  |[[1, 2, 3, 4], [5, 6]]  |
+---+------------------------+
```

Step 1: Explode the Outer Array

```
val explodedOuterDF = df.withColumn("outer_array", explode($"nested_array"))

// Show the DataFrame after exploding the outer array
explodedOuterDF.show(false)
```

Expected Output After Exploding the Outer Array

```
+---+-------------+
|id |outer_array  |
+---+-------------+
|1  |[a, b, c]    |
|1  |[d, e]       |
|2  |[x, y]       |
|2  |[z]          |
|3  |[1, 2, 3, 4] |
|3  |[5, 6]       |
+---+-------------+
```

Explode Nested array to Single array

```
val explodedInnerDF = explodedOuterDF.withColumn("inner_element", explode($"outer_array"))

// Show the final exploded DataFrame
explodedInnerDF.show(false)
```

Output

```
+---+----------------------+------------+-------------+
|id |nested_array          |outer_array |inner_element|
+---+----------------------+------------+-------------+
|1  |[[a, b, c], [d, e]]   |[a, b, c]   |a            |
|1  |[[a, b, c], [d, e]]   |[a, b, c]   |b            |
|1  |[[a, b, c], [d, e]]   |[a, b, c]   |c            |
|1  |[[a, b, c], [d, e]]   |[d, e]      |d            |
|1  |[[a, b, c], [d, e]]   |[d, e]      |e            |
|2  |[[x, y], [z]]         |[x, y]      |x            |
|2  |[[x, y], [z]]         |[x, y]      |y            |
|2  |[[x, y], [z]]         |[z]         |z            |
|3  |[[1, 2, 3, 4], [5, 6]]|[1, 2, 3, 4]|1            |
|3  |[[1, 2, 3, 4], [5, 6]]|[1, 2, 3, 4]|2            |
|3  |[[1, 2, 3, 4], [5, 6]]|[1, 2, 3, 4]|3            |
|3  |[[1, 2, 3, 4], [5, 6]]|[1, 2, 3, 4]|4            |
|3  |[[1, 2, 3, 4], [5, 6]]|[5, 6]      |5            |
|3  |[[1, 2, 3, 4], [5, 6]]|[5, 6]      |6            |
+---+----------------------+------------+-------------+
```

Convert array of String to a String column

Step 1: Create the DataFrame with an Array of Strings

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
import org.apache.spark.sql.functions._

// Define the schema with an array of strings
val schema = StructType(
  StructField("id", IntegerType, nullable = false) ::
  StructField("string_array", ArrayType(StringType), nullable = true) :: Nil
)

// Create sample data
val data = Seq(
  Row(1, Array("apple", "banana", "cherry")),
  Row(2, Array("dog", "elephant")),
  Row(3, Array("fish", "giraffe", "hippo"))
)

// Create the DataFrame
val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)

// Show the original DataFrame
df.show(false)
```

Output Before Conversion

plaintext

Copy code

```
+---+----------------------+
|id |string_array          |
+---+----------------------+
|1  |[apple, banana, cherry]|
|2  |[dog, elephant]       |
|3  |[fish, giraffe, hippo]|
+---+----------------------+
```

Step 2: Convert Array of Strings to a Single String Column

Now, use the concat_ws function to concatenate the elements of the string_array into a single string with a specified separator, such as a comma.

```
// Convert the array of strings into a single string column
```

```
val dfWithString = df.withColumn("string_column", concat_ws(", ", $"string_array"))


// Show the final DataFrame

dfWithString.show(false)
```

<u>Output</u>

```
+---+----------------------+---------------------+
|id |string_array          |string_column        |
+---+----------------------+---------------------+
|1  |[apple, banana, cherry]|apple, banana, cherry|
|2  |[dog, elephant]       |dog, elephant        |
|3  |[fish, giraffe, hippo]|fish, giraffe, hippo |
+---+----------------------+---------------------+
```

Practical 4

Title :- Spark SQL Aggregate Functions

Aim :- Learning Spark SQL aggregate functions like count,distinct,sum, avg, min, max.

Introduction to Spark SQL and Aggregate Functions

Spark SQL is a module in Apache Spark that integrates relational processing with Spark's functional programming API.

It supports querying data via SQL as well as the DataFrame API, which can be used in Scala, Java, Python, and R.

Aggregate functions in Spark SQL are built-in methods that allow you to compute a single result from a set of input values.

These functions are commonly used to summarize data by performing operations like counting, summing, averaging, and finding the minimum or maximum value within a data set.

Essential Aggregate Functions in Spark SQL

Spark SQL provides a wide range of aggregate functions to help perform various aggregation operations on DataFrames/Datasets. Some of the essential functions include:

– `count()`: Returns the count of rows for a column or group of columns.
– `countDistinct()`: Returns the count of distinct rows for a column or group of columns.
– `sum()`: Computes the sum of a column.
– `avg()`: Computes the average value of a column. – `min()`: Returns the minimum value of a column.
– `max()`: Returns the maximum value of a column.

```
#Creating Dataframe
```

```
val columns = Seq("name","age","city")

val data = Seq(("Alice", 29, "New York"),("Bob", 35, "Los Angeles"),
("Charlie", 23, "San Francisco"),("Charlie", 39, "New
York"),("Kim", 25, "Los Angeles"))

val rdd =sc.parallelize(data) val

personsDF= rdd.toDF(columns:_*)

personsDF.show()
```

Output :-

```
+-------+---+-------------+ |    name|age|     city|
+-------+---+-------------+ | Alice| 29|    New York| |    Bob| 35|
Los Angeles| |Charlie| 23|San Francisco| |Charlie| 39|
New York| |    Kim| 25| Los Angeles| +-------+---+-------------+
```

Using distinct()

```
personsDF.select("name").distinct().show()
```

Output ➖

```
| name|
+-------+
| Alice|
|    Bob|
|Charlie|
|    Kim|
+-------+
```

Agg() method :-

`agg` is a method used to apply one or more aggregate functions to a DataFrame.

The code block demonstrates how to use `count()` to get the total number of rows in `personsDF` and `countDistinct()` to get the number of distinct names in the same DataFrame.

Using count() and countDistinct()

```
import org.apache.spark.sql.functions._

// Counting all rows scala>
personsDF.agg(count("*")).show()
+--------+
|count(1)|
+--------+
|       5|
+--------+




// Counting distinct names scala>
personsDF.agg(countDistinct("name")).show()
+--------------------+
|count(DISTINCT name)|
+--------------------+
|                   4|
+--------------------+
```

Using aggregate functions in select

```
personsDF.select(count("*"),sum("age"), avg("age"),
min("age"),max("age")).show()
```

This would output the sum, average, minimum, and maximum age of all persons in our DataFrame:

```
+--------+--------+--------+--------+
|sum(age)|avg(age)|min(age)|max(age)|
+--------+--------+--------+--------+
|      87|    29.0|      23|      35|
+--------+--------+--------+--------+
```

Grouping Data with Aggregate Functions

Similar to SQL "GROUP BY" clause, Spark groupBy() function is used to collect the identical data into groups on DataFrame/Dataset and perform aggregate functions on the grouped data.
Aggregate functions are often used together with the `groupBy` operation, which groups the DataFrame using the specified columns, then, aggregate functions can be applied to each group. This is powerful for performing computations on categorical data:

#calculate the number of person in each city using `count()`

```
personsDF.groupBy("city").count().show()
```

#Calculate the minimum `age` of each city using `min()`

```
personsDF.groupBy("city").min("age").show()
```

#Calculate the maximum `age` of each city using `max()`

```
personsDF.groupBy("city").max("age").show()
```

#Calculate the average `age` of each city using `avg()`

```
personsDF.groupBy("city").avg( "age").show()
```

#Calculate the total `age` of each city using `mean()`

```
personsDF.groupBy("city").sum( "age").show()
```

Using Multiple aggregate functions together :-

```
personsDF.groupBy("city").agg(
count("*").as("num_people"),
avg("age").as("average_age"),
min("age").as("min_age"),
max("age").as("max_age")
).show()
```

Here, we are grouping the DataFrame by city and calculating the number of people and the average age in each city. We use `alias` to name the resultant columns. The output could be something like this:

```
+-------------+----------+-----------+-------+-------+
|    city|num_people|average_age|min_age|max_age|
+-------------+----------+-----------+-------+-------+
|    New York| 2|   34.0|      29|  39| | Los Angeles|
2|   30.0|     25|  35| |San Francisco| 1|   23.0|
23|  23|
+-------------+----------+-----------+------+-------+
```

GroupBy and aggregate on multiple DataFrame columns

```
val simpleData = Seq(("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)
  )
 val df =
simpleData.toDF("employee_name","department","state","salary","a
ge","bonus")
  df.show()
```

//GroupBy on multiple columns

```
df.groupBy("department","state")
    .sum("salary","bonus")
    .show(false)
```

This yields the below output.

```
+----------+-----+-----------+----------+
|department|state|sum(salary)|sum(bonus)|
+----------+-----+-----------+----------+
|Finance   |NY   |162000     |34000     |
|Marketing |NY   |91000      |21000     |
|Sales     |CA   |81000      |23000     |
|Marketing |CA   |80000      |18000     |
|Finance   |CA   |189000     |47000     |
|Sales     |NY   |176000     |30000     |
+----------+-----+-----------+----------+
```

Using filter on aggregate data

Similar to SQL “HAVING” clause, On Spark DataFrame we can use either the where() or filter() function to filter the rows of aggregated data.

```
df.groupBy("department")
    .agg(
      sum("salary").as("sum_salary"),
      avg("salary").as("avg_salary"),
      sum("bonus").as("sum_bonus"),
      max("bonus").as("max_bonus"))
    .where(col("sum_bonus") >= 50000)
    .show(false)
```

This removes the sum of a bonus that has less than 50000 and yields below output.

```
+----------+----------+-----------------+---------+---------+
|department|sum_salary|avg_salary       |sum_bonus|max_bonus|
+----------+----------+-----------------+---------+---------+
|Sales     |257000    |85666.66666666667|53000    |23000    |
|Finance   |351000    |87750.0          |81000    |24000    |
+----------+----------+-----------------+---------+---------+
```

Window Functions and Aggregate Functions

In addition to the standard aggregate functions, Spark SQL also supports window functions which allow you to perform calculations across sets of rows that are related to the current row. This is similar to aggregate functions, but rather than performing a single aggregate operation across the whole dataset, window functions perform calculations for each partition of data.

```
import org.apache.spark.sql.expressions.Window

val windowSpec = Window.partitionBy("city").orderBy("age")
val windowedDF = personsDF.withColumn("rank",
rank().over(windowSpec)) windowedDF.show()
```

This code snippet creates a window specification that partitions the data by the `city` column and orders it by the `age` column within each partition. Then, it adds a new column `rank` that displays the rank of each row within its partition. The hypothetical output could look like:

```
+-------+---+-------------+----+
|   name|age|         city|rank|
+-------+---+-------------+----+
|    Bob| 35| Los Angeles|    1|
| Alice| 29|   New York| 1|
|Charlie| 23|San Francisco|   1|
+-------+---+-------------+----+
```

Practical No.5

Aim : Spark SQL Joins, Spark SQL Schema, StructType & SQL Functions

Joins are used to combine rows from two or more DataFrames based on a related column.

```
import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.functions._


// Create DataFrames

val df1 = Seq(

  (1, "Alice"),

  (2, "Bob"),

  (3, "Carol")

).toDF("id", "name")


val df2 = Seq(

  (1, "HR"),

  (2, "Finance"),

  (4, "Engineering")

).toDF("id", "department")


df1.show()

df2.show()
```

Output:

```
+---+-----+
| id| name|
+---+-----+
|  1|Alice|
|  2|  Bob|
|  3|Carol|
+---+-----+
+---+-----------+
| id|department |
+---+-----------+
|  1|       HR |
|  2|   Finance |
|  4|Engineering|
+---+-----------+
```

Inner Join two Data Frames

Returns only the rows with matching keys in both DataFrames.

```
val innerJoinDF = df1.join(df2, df1("id") === df2("id"))
innerJoinDF.show()
```

Self Join

To perform a self-join on a DataFrame in Spark, you'll essentially join the DataFrame with itself based on some condition. Self-joins are often used to compare rows within the same DataFrame.

Let's demonstrate this with the DataFrames we created earlier (df1 and df2).

Scenario

Let's assume we want to perform a self-join on df1 to compare rows within df1 itself. We'll join df1 with itself based on a condition. For simplicity, let's perform a self-join to compare rows where the id values are equal (which will essentially return the same rows twice, but it's useful for understanding how to apply self-joins).

```
// Alias the DataFrame for self-joining
val df1Alias1 = df1.alias("df1Alias1")
val df1Alias2 = df1.alias("df1Alias2")

// Perform the self-join
val selfJoinDF = df1Alias1
  .join(df1Alias2, df1Alias1("id") === df1Alias2("id"))
  .select(
   df1Alias1("id").as("id1"),
   df1Alias1("name").as("name1"),
   df1Alias2("id").as("id2"),
   df1Alias2("name").as("name2")
  )

selfJoinDF.show()
```

Output

Since df1 has only unique id values and we are joining on the same column, the output will show rows where each row is joined with itself:

```
+---+-----+---+-----+
|id1|name1|id2|name2|
+---+-----+---+-----+
|  1|Alice|  1|Alice|
|  2|  Bob|  2|  Bob|
|  3|Carol|  3|Carol|
+---+-----+---+-----+
```

Date and Time Functions

In Apache Spark, date and time functions are very useful for processing and manipulating date and time data. When using the Spark Shell, you can work with these functions directly in Scala. Here's a guide on how to use some common date and time functions with examples.

```
import org.apache.spark.sql.functions._
#Creating a DataFrame with a Date Column:
val df = Seq(("2024-08-30", 1)).toDF("date", "value")
#Converting String to Date:
val dfWithDate = df.withColumn("date", to_date(col("date"), "yyyy-MM-dd"))

#Extracting Date Components:
val dfWithDateComponents = dfWithDate
```

```
  .withColumn("year", year(col("date")))
  .withColumn("month", month(col("date")))
  .withColumn("day", dayofmonth(col("date")))
```

#Adding Days to a Date:

```
val dfWithFutureDate = dfWithDate.withColumn("future_date", date_add(col("date"), 10))
```

#Calculating Date Difference:

```
val dfWithDateDiff = dfWithFutureDate.withColumn("days_diff", datediff(col("future_date"),
col("date")))
```

#Formatting Dates:

```
val dfFormattedDate = dfWithDate.withColumn("formatted_date", date_format(col("date"),
"yyyy/MM/dd"))
```

#Truncating Dates:

```
val dfTruncatedDate = dfWithDate.withColumn("truncated_date", trunc(col("date"), "MM"))
```

#Show Results

```
dfWithDateComponents.show()
dfWithFutureDate.show()
dfWithDateDiff.show()
dfFormattedDate.show()
dfTruncatedDate.show()
```

dfWithDateComponents.show()

```
+----------+-----+
|      date|value|
+----------+-----+
|2024-08-30|    1|
+----------+-----+
scala> dfWithFutureDate.show()
+----------+-----+-----------+
|      date|value|future_date|
+----------+-----+-----------+
|2024-08-30|    1| 2024-09-09|
+----------+-----+-----------+
```

```
scala> dfWithDateDiff.show()
+----------+-----+-----------+---------+
|      date|value|future_date|days_diff|
+----------+-----+-----------+---------+
|2024-08-30|    1| 2024-09-09|       10|
+----------+-----+-----------+---------+
```

```
scala> dfFormattedDate.show()
+----------+-----+--------------+
|      date|value|formatted_date|
+----------+-----+--------------+
|2024-08-30|    1|    2024/08/30|
+----------+-----+--------------+
```

```
scala> dfTruncatedDate.show()
+----------+-----+--------------+
|      date|value|truncated_date|
+----------+-----+--------------+
|2024-08-30|    1|    2024-08-01|
+----------+-----+--------------+
```

String Functions

```
import org.apache.spark.sql.functions._

// Create a DataFrame with a string column
val df = Seq(("hello world", 1)).toDF("text", "value")
df.show()
+-----------+-----+
|       text|value|
+-----------+-----+
|hello world|    1|
+-----------+-----+
```

// Convert string to uppercase

```
val dfUppercase = df.withColumn("uppercase", upper(col("text")))
dfUppercase.show()
+-----------+-----+-----------+
|       text|value|  uppercase|
+-----------+-----+-----------+
|hello world|    1|HELLO WORLD|
+-----------+-----+-----------+
```

// Convert string to lowercase

```
val dfLowercase = df.withColumn("lowercase", lower(col("text")))
dfLowercase.show()
+-----------+-----+-----------+
|       text|value|  lowercase|
+-----------+-----+-----------+
|hello world|    1|hello world|
+-----------+-----+-----------+
```

// Split a string

```
val dfSplit = df.withColumn("split", split(col("text"), " "))

dfSplit.show()

+-----------+-----+--------------+
|       text|value|         split|
+-----------+-----+--------------+
|hello world|    1|[hello, world]|
+-----------+-----+--------------+
```

Map Functions

In Spark, the Map passes each element of the source through a function and forms a new distributed dataset.

In this example, we add a constant value 10 to each element.

Create an RDD using parallelized collection.

```
val data = sc.parallelize(List(10,20,30))

data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26
```

Apply the map function and pass the expression required to perform.

```
val mapfunc = data.map(x => x+10)

mapfunc: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map at <console>:26

mapfunc.collect

res12: Array[Int] = Array(20, 30, 40)
```

Aggregate Functions

// Create an RDD

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5))

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:26
```

a. Counting Elements:

```
val count = rdd.count()

println(s"Count: $count")  // Outpvalut: Count: 5
```

b. Sum of Elements:

```
val sum = rdd.sum()
```

```
println(s"Sum: $sum")  // Output: Sum: 15.0
```

c. Average of Elements:

```
val mean = rdd.mean()

println(s"Mean: $mean")  // Output: Mean: 3.0
```

d. Minimum and Maximum:

```
val min = rdd.min()

val max = rdd.max()

println(s"Min: $min, Max: $max")  // Output: Min: 1, Max: 5
```

<u>Sort Functions</u>

#Create an RDD

```
val rdd = sc.parallelize(List((1, "Alice"), (3, "Bob"), (2, "Charlie"), (4, "David")))
```

a. Sorting by Key:Ascending Order

```
val sortedByKey = rdd.sortByKey()

sortedByKey.collect()

 O/P : res14: Array[(Int, String)] = Array((1,Alice), (2,Charlie), (3,Bob), (4,David))
```

b. Sorting in Descending Order:

```
val sortedByKeyDesc = rdd.sortByKey(false)

sortedByKeyDesc.collect()

res15: Array[(Int, String)] = Array((4,David), (3,Bob), (2,Charlie), (1,Alice))
```

Practical No. 6

Aim : SPARK SQL

. 1. createDataFrame()

- Purpose: Creates a DataFrame from various sources like a list, a Pandas DataFrame, or an existing RDD.

```
// Create a DataFrame from a Seq of tuples

val df = Seq((1, "Alice"), (2, "Bob")).toDF("id", "name")

df.show()

+---+-----+

| id| name|

+---+-----+

|  1|Alice|

|  2|  Bob|

+---+-----+
```

2. where() & filter()

- Purpose: Filter rows based on a condition.

```
val filteredDF = df.where(col("id") > 1)

filteredDF.show()

+---+----+

| id|name|

+---+----+

|  2| Bob|

+---+----+

// or

val filteredDF = df.filter(col("id")<2)

filteredDF.show()

+---+-----+

| id| name|

+---+-----+

|  1|Alice|
```

3. withColumn()

- Purpose: Adds or replaces a column in the DataFrame with a new column.

```
val updatedDF = df.withColumn("name_upper", upper(col("name")))

updatedDF.show()

+---+-----+----------+

| id| name|name_upper|

+---+-----+----------+

|  1|Alice|     ALICE|

|  2|  Bob|       BOB|

+---+-----+----------+
```

4. withColumnRenamed()

- Purpose: Renames an existing column in the DataFrame.

```
val renamedDF = df.withColumnRenamed("name", "full_name")

renamedDF.show()

+---+---------+

| id|full_name|

+---+---------+

|  1|    Alice|

|  2|      Bob|

+---+---------+
```

5. drop()

- Purpose: Drops one or more columns from the DataFrame.

```
val dfDropped = df.drop("name")

dfDropped.show()

+---+

| id|

+---+

|  1|

|  2|

+---+
```

6. distinct()

- Purpose: Returns a new DataFrame with duplicate rows removed.

```
val distinctDF = df.distinct()

distinctDF.show()

scala> distinctDF.show()

+---+-----+
| id| name|
+---+-----+
|  1|Alice|
|  2|  Bob|
+---+-----+
```

7. groupBy()

- Purpose: Groups the DataFrame by one or more columns and allows aggregation.

```
val groupedDF = df.groupBy("id").count()

groupedDF.show()

scala> groupedDF.show()

+---+-----+
| id|count|
+---+-----+
|  1|    1|
|  2|    1|
+---+-----+
```

8. join()

- Purpose: Joins two DataFrames on a specified column or condition.

```
val joinedDF = df1.join(df2, Seq("id"), "inner")

joinedDF.show()
```

9. map() vs mapPartitions()

- Purpose: Transformations on RDDs.

- map(): Applies a function to each element in the RDD.

```
val rdd = spark.sparkContext.parallelize(Seq(1, 2, 3))

val mappedRDD = rdd.map(x => x * 2)

mappedRDD.collect

res26: Array[Int] = Array(2, 4, 6)
```

mapPartitions(): Applies a function to each partition of the RDD. Useful for operations that need to process an entire partition at once.

```
val partitionedRDD = mappedRDD.mapPartitions(iter => Iterator(iter.sum))

partitionedRDD.collect

res27: Array[Int] = Array(0, 0, 0, 2, 0, 0, 0, 4, 0, 0, 0, 6)
```

10. foreach()

- foreach(): Applies a function to each element in the RDD.

```
rdd.foreach(println)
```

11. pivot()

- Purpose: Performs pivoting of data, which is useful for creating a summary table where values are aggregated across different columns.

```
val df = spark.createDataFrame(Seq(

  ("A", "2024-01", 1),

  ("A", "2024-02", 2),

  ("B", "2024-01", 3),

  ("B", "2024-02", 4)

)).toDF("category", "month", "value")


val pivotedDF = df.groupBy("category").pivot("month").sum("value")

pivotedDF.show()
```

Output:

```
+--------+-------+-------+

|category|2024-01|2024-02|

+--------+-------+-------+

|       B|      3|      4|

|       A|      1|      2|
```

12. union()

- Purpose: Merges two DataFrames or RDDs with the same schema.

```
val df1 = Seq(
  (1, "Alice"),
  (2, "Bob"),
  (3, "Carol")
).toDF("id", "name")

val df2 = Seq(
  (1, "HR"),
  (2, "Finance"),
  (4, "Engineering")
).toDF("id", "department")

val unionDF = df1.union(df2)
unionDF.show()
```

Output:

```
+---+-----------+
| id|       name|
+---+-----------+
|  1|      Alice|
|  2|        Bob|
|  3|      Carol|
|  1|         HR|
|  2|    Finance|
|  4|Engineering|
+---+-----------+
```

13. collect()

- Purpose: Returns all the elements of the DataFrame or RDD as a list to the driver. Use with caution as it can cause out-of-memory errors for large datasets.

```
scala> val collectedData = df.collect()
```

Output:

```
collectedData: Array[org.apache.spark.sql.Row] = Array([A,2024-01,1], [A,2024-02,2], [B,2024-01,3], [B,2024-02,4])
```

14. cache()

- Purpose: Caches or persists DataFrame or RDD in memory or on disk for faster access in subsequent actions.

```
scala> df.cache()
```

Output:

```
res21: df.type = [category: string, month: string ... 1 more field]
```

Creating and Using UDFs

Define a UDF (User Defined Function): To define a UDF, you first need to create a function and then register it as a UDF with Spark.

```
// Define a simple function

def addOne(x: Int): Int = x + 1

// Register the function as a UDF

val addOneUDF = udf(addOne _)
```

Use the UDF: Apply the UDF to a DataFrame or Dataset.

```
// Create a DataFrame

val df = Seq(1, 2, 3, 4).toDF("number")

// Apply the UDF

val dfWithAddedOne = df.withColumn("number_plus_one", addOneUDF(col("number")))

dfWithAddedOne.show()
+------+----------------+

|number|number_plus_one|

+------+----------------+

|     1|              2|

|     2|              3|

|     3|              4|

|     4|              5|
```

Practical No :07 : Spark Data Source API

Aim : Read data from JSON Files

1.Open notepad ++

2. Language-> select J(means JSON Java Script Object Notation)

3. Open cmd type spark-shell

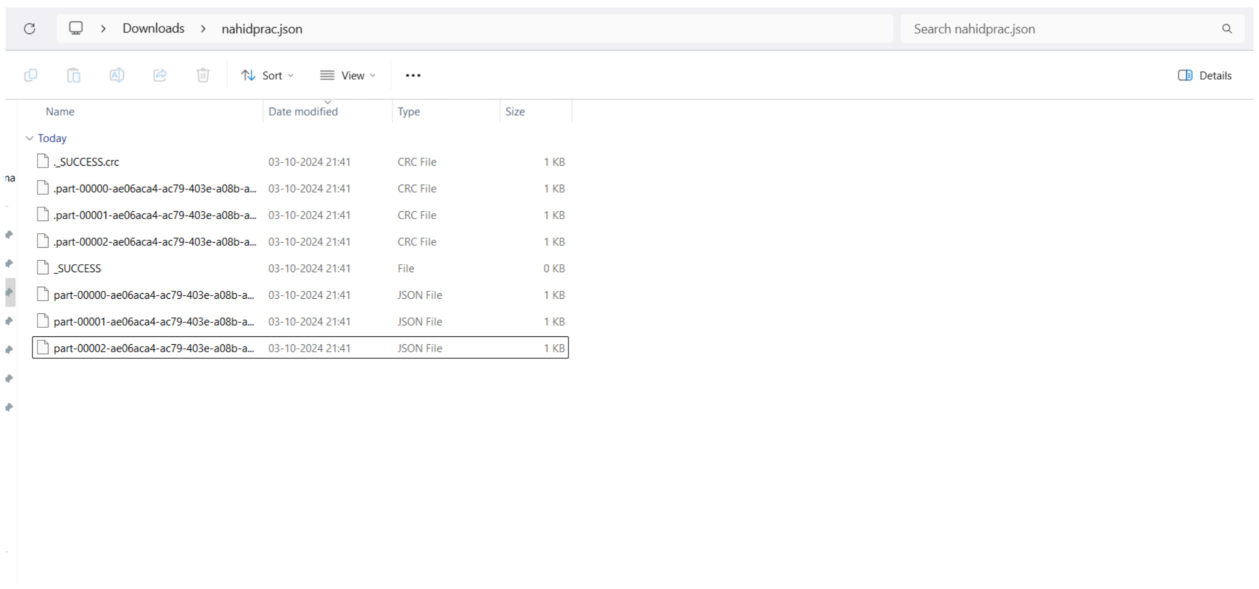
4. Type

```
val a =spark.read.json("/C:/Users/Lenovo/Desktop/tydsprac.json")

a: org.apache.spark.sql.DataFrame = [a: struct<b: bigint>]


scala> a.show()

+---+

|  a|

+---+

|{1}|

|{2}|

|{3}|

+---+
```

Write data to a JSON File

```
 scala>val data = Seq(("John", 30), ("Jane", 25), ("Sam", 22))

O/p: data: Seq[(String, Int)] = List((John,30), (Jane,25), (Sam,22))


scala> val df = spark.createDataFrame(data).toDF("name", "age")
```

```
O/p: df: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> df.show()

+----+---+

|name|age|

+----+---+

|John| 30|

|Jane| 25|

| Sam| 22|

+----+---+

scala>
df.write.mode("overwrite").json("C:/Users/Lenovo/Desktop/TYBSCDS.json")
```



Name	Date modified	Type	Size
Today			
._SUCCESS.crc	03-10-2024 21:41	CRC File	1 KB
.part-00000-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	CRC File	1 KB
.part-00001-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	CRC File	1 KB
.part-00002-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	CRC File	1 KB
_SUCCESS	03-10-2024 21:41	File	0 KB
part-00000-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	JSON File	1 KB
part-00001-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	JSON File	1 KB
part-00002-ae06aca4-ac79-403e-a08b-a...	03-10-2024 21:41	JSON File	1 KB

<u>Reading CSV FILE</u>

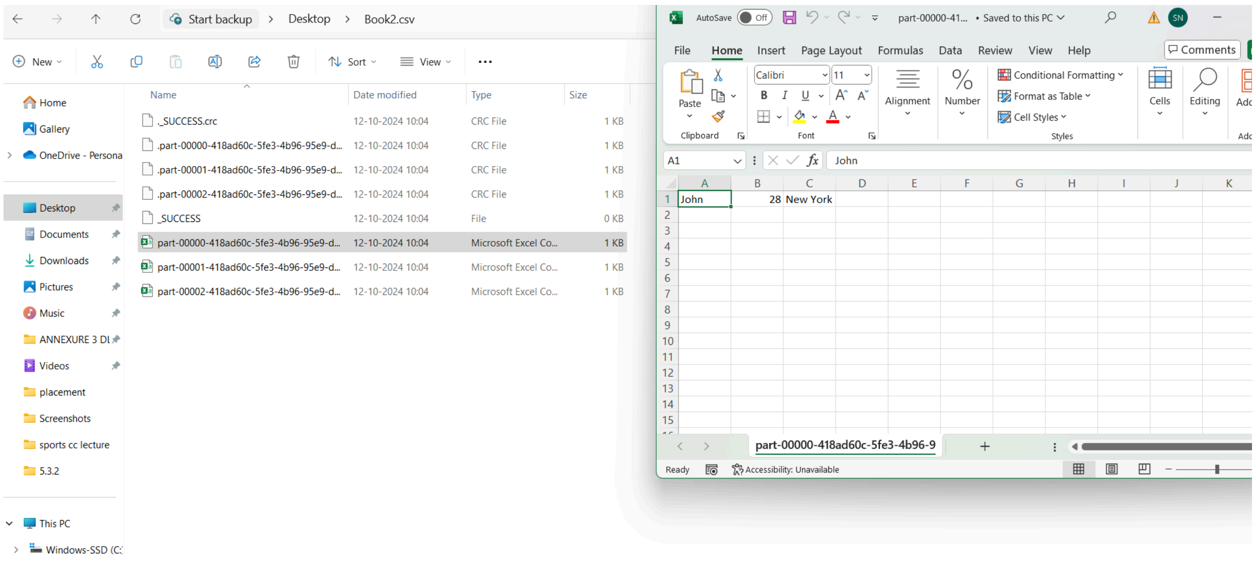
```
val df = sc.textFile("C:/Users/Nahid Shaikh/OneDrive/Desktop/Book2.csv")
```

df.collect()

res1: Array[String] = Array(Roll No,Name, 1,A, 2,B, 3,C)

Writing CSV File

```
// Load or create your DataFrame. Here's an example of creating a DataFrame.

val data = Seq(

  ("John", 28, "New York"),

  ("Anna", 23, "London"),

  ("Mike", 35, "San Francisco")

)



val df = spark.createDataFrame(data).toDF("Name", "Age", "City")



// Write DataFrame to a CSV file

df.write.mode("overwrite").csv("C:/Users/Nahid Shaikh/OneDrive/Desktop/Book2.csv")
```



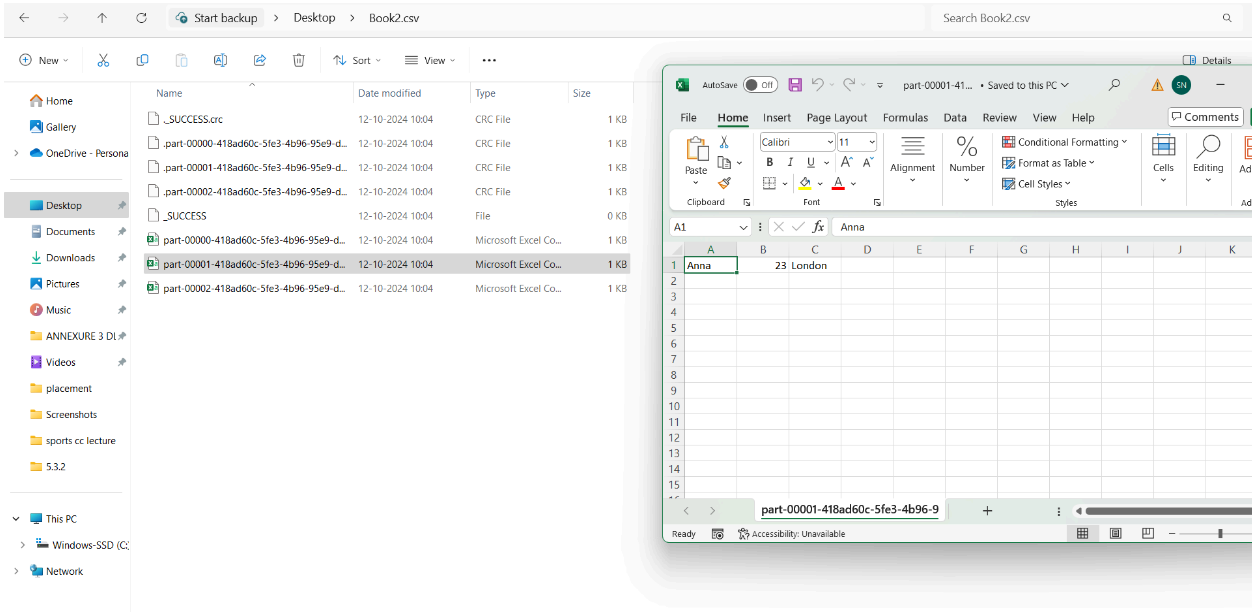
Start backup > Desktop > Book2.csv

Name	Date modified	Type	Size
._SUCCESS.crc	12-10-2024 10:04	CRC File	1 KB
.part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
_SUCCESS	12-10-2024 10:04	File	0 KB
part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB

part-00000-41... • Saved to this PC

A1 | John

	A	B	C
1	John	28	New York



Start backup > Desktop > Book2.csv

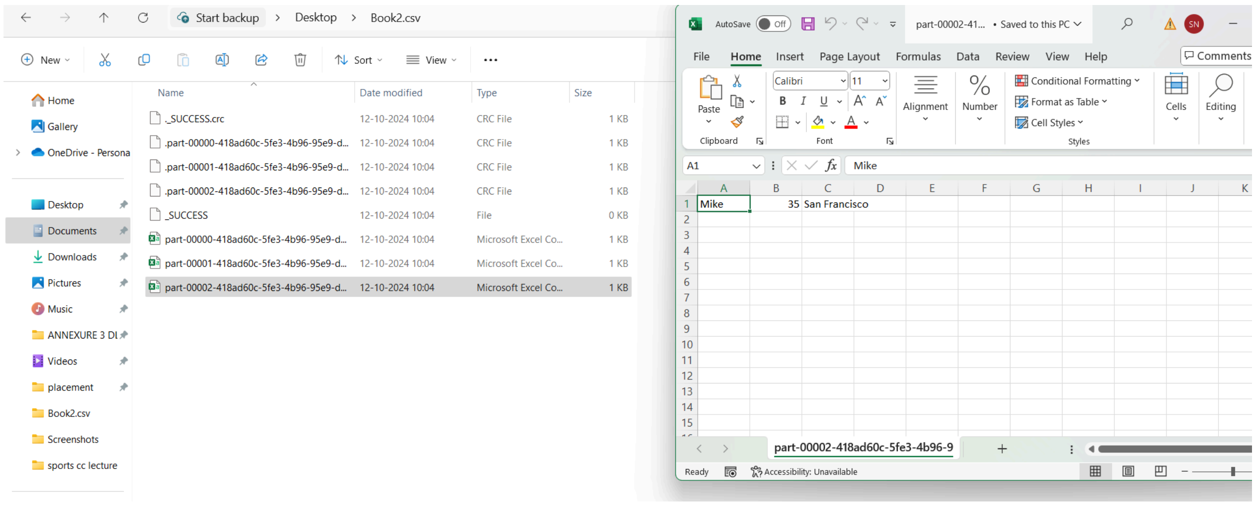
Name	Date modified	Type	Size
._SUCCESS.crc	12-10-2024 10:04	CRC File	1 KB
.part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
_SUCCESS	12-10-2024 10:04	File	0 KB
part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB

part-00001-41... • Saved to this PC

A1: Anna

	A	B	C
1	Anna	23	London

Sheet: part-00001-418ad60c-5fe3-4b96-9



Start backup > Desktop > Book2.csv

Name	Date modified	Type	Size
._SUCCESS.crc	12-10-2024 10:04	CRC File	1 KB
.part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
.part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	CRC File	1 KB
_SUCCESS	12-10-2024 10:04	File	0 KB
part-00000-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00001-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB
part-00002-418ad60c-5fe3-4b96-95e9-d...	12-10-2024 10:04	Microsoft Excel Co...	1 KB

part-00002-41... • Saved to this PC

A1: Mike

	A	B	C
1	Mike	35	San Francisco

Sheet: part-00002-418ad60c-5fe3-4b96-9

Practical NO :8

Spark MLlib : Demonstrate use of Estimator, Transformer, and Param

<u>What is Spark MLib</u>

Spark MLlib is Spark's scalable machine learning library, providing tools for building and deploying machine learning models. It leverages the distributed computing power of Apache Spark to handle large-scale data processing and machine learning tasks.

Estimator: An algorithm or method that fits a model to data. In Spark, Estimators are used to train models or create Transformers.

Transformer: An abstraction that transforms data into a new form. Transformers are often used for feature extraction or data manipulation.

Param: A parameter that can be configured on an Estimator or Transformer. Params control the behavior and performance of these models or transformers.

<u>Program</u>

```
import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.Row

import org.apache.spark.ml.feature.{Tokenizer, HashingTF}

import org.apache.spark.ml.classification.LogisticRegression


// Create Spark session

val spark = SparkSession.builder

  .appName("Spark MLlib Example")

  .getOrCreate()


import spark.implicits._


// Sample data

val data = Seq(

  ("I love Spark", 1.0),

  ("I hate Spark", 0.0),

  ("Spark is awesome", 1.0),

  ("I don't like Spark", 0.0)

)

val df = data.toDF("text", "label")
```

```
df.show()
```

Output

```
+------------------+-----+
|              text|label|
+------------------+-----+
|      I love Spark|  1.0|
|      I hate Spark|  0.0|
|  Spark is awesome|  1.0|
|I don't like Spark|  0.0|
```

```
// Tokenizer:separate the words
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
//Convert tokenizer into dataframe
val dfWords = tokenizer.transform(df)
dfWords.show(truncate = false)
```

Output

```
+------------------+-----+----------------------+
|text              |label|words                 |
+------------------+-----+----------------------+
|I love Spark      |1.0  |[i, love, spark]      |
|I hate Spark      |0.0  |[i, hate, spark]      |
|Spark is awesome  |1.0  |[spark, is, awesome]  |
|I don't like Spark|0.0  |[i, don't, like, spark]|
+------------------+-----+----------------------+
```

```
// HashingTF
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("features")
  .setNumFeatures(20)

val dfFeatures = hashingTF.transform(dfWords)
dfFeatures.show(truncate = false)
```

<u>Output :</u>

```
+------------------+-----+----------------------+----------------------------------+
|text              |label|words                 |features                          |
+------------------+-----+----------------------+----------------------------------+
|I love Spark      |1.0  |[i, love, spark]      |(20,[0,6,16],[1.0,1.0,1.0])       |
|I hate Spark      |0.0  |[i, hate, spark]      |(20,[6,9,16],[1.0,1.0,1.0])       |
|Spark is awesome  |1.0  |[spark, is, awesome]  |(20,[5,6,9],[1.0,1.0,1.0])        |
|I don't like Spark|0.0  |[i, don't, like, spark]|(20,[1,6,10,16],[1.0,1.0,1.0,1.0])|
+------------------+-----+----------------------+----------------------------------+
```

```
// Predictions
val predictions = model.transform(dfFeatures)
predictions.select("text", "label","prediction").show(truncate = false)
+------------------+-----+----------+
|text              |label|prediction|
+------------------+-----+----------+
|I love Spark      |1.0  |1.0       |
|I hate Spark      |0.0  |0.0       |
|Spark is awesome  |1.0  |1.0       |
|I don't like Spark|0.0  |0.0       |
+------------------+-----+----------+
```