PYSPARK ASSIGNMENT

- HARSHINI V

TRANSFORMATION AND ACTION IN PYSPARK:

The Resilient Distributed Dataset (RDD) is the foundational data structure in PySpark, designed for efficient distributed computing. RDDs are low-level objects known for their high efficiency in performing distributed tasks. This explanation does not cover PySpark basics, such as creating RDDs and DataFrames. If you're unfamiliar with these concepts, note that RDDs support two types of operations: **Transformations** and **Actions**.

Actions:

Actions are operations that generate a single value or a non-RDD result by acting on the transformed RDD. These operations trigger the execution of the DAG, ending the laziness of the transformations and yielding the final output.

1. collect ():

The collect () action on an RDD returns a list of all the elements of the RDD. It's a great asset for displaying all the contents of our RDD.

```
from pyspark import SparkContext
from pyspark.sql import  SparkSession

sc =SparkContext.getOrCreate()
spark = SparkSession.builder.appName('pyspark first program').getOrCreate()

#create the rdd

rdd = sc.parallelize([('C',85,76,87,91), ('B',85,76,87,91), ("A", 85,78,96,92), ("A", 92,76,89,96)], 4)
mydata = ['Division','English','Mathematics','Physics','Chemistry']
marks_df = spark.createDataFrame(rdd, schema=mydata)

print(rdd)



rdd.collect()
```

```
▸ (2) Spark Jobs
▸ marks_df: pyspark.sql.dataframe.DataFrame = [Division: string, English: long ... 3 more fields]
ParallelCollectionRDD[125] at readRDDFromInputStream at PythonRDD.scala:435
Out[1]: [('C', 85, 76, 87, 91),
 ('B', 85, 76, 87, 91),
 ('A', 85, 78, 96, 92),
 ('A', 92, 76, 89, 96)]
```

2. count ():

The count () action on an RDD is an operation that returns the number of elements of our RDD. This helps in verifying if a correct number of elements are being added in an RDD.

```
rdd = sc.parallelize([('C',85,76,87,91), ('B',85,76,87,91), ("A", 85,78,96,92), ("A", 92,76,89,96)], 4)
mydata = ['Division','English','Mathematics','Physics','Chemistry']
rdd.count()
```

▸ (1) Spark Jobs

```
Out[7]: 4
```

3. first ():

The first() action on an RDD returns the first element from our RDD. This can be helpful when we want to verify if the exact kind of data has been loaded in our RDD as per the requirements.

```
rdd = sc.parallelize([('C',85,76,87,91), ('B',85,76,87,91), ("A", 85,78,96,92), ("A", 92,76,89,96)], 4)
rdd.first()
```

▸ (1) Spark Jobs

```
Out[8]: ('C', 85, 76, 87, 91)
```

4. take ():

The take(n) action on an RDD returns n number of elements from the RDD. The 'n' argument takes an integer which refers to the number of elements we want to extract from the RDD.

```
rdd = sc.parallelize([('C',85,76,87,91), ('B',85,76,87,91), ("A", 85,78,96,92), ("A", 92,76,89,96)], 4)
rdd.take(2)
```

▸ (2) Spark Jobs

```
Out[9]: [('C', 85, 76, 87, 91), ('B', 85, 76, 87, 91)]
```

5. saveAsTextFile():

The saveAsTextFile() Action is used to serve the resultant RDD as a text file. We can also specify the path to which file needed to be saved. This helps in saving our results especially when we are working with a large amount of data.

```
rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
rdd.saveAsTextFile('file.txt')
```

▸ (1) Spark Jobs

Transformations:

Transformations are operations that take an RDD as input and produce another RDD as output. They are immutable, meaning the original RDD remains unchanged, and the transformations generate a new RDD. These operations follow a lazy evaluation model, creating a Directed Acyclic Graph (DAG) for computations, which only executes when an action is applied.

1. map ():

Transformation As the name suggests, map() transformation maps a value to the elements of an RDD. The map() transformation takes in an anonymous function and applies this function to each of the elements in the RDD.

```
rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
r= rdd.map(lambda x: x+10)
r.collect()
```

▸ (1) Spark Jobs

```
Out[15]: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

2. filter ():

Transformation A .filter() transformation is an operation in PySpark for filtering elements from a PySpark RDD. The .filter() transformation takes in an anonymous function with a condition.

```
rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
rdd.filter(lambda x: x%2==0).collect()
```

▸ (1) Spark Jobs

```
Out[17]: [2, 4, 6, 8, 10]
```

3. union ():

Transformation The .union() transformation combines two RDDs and returns the union of the input two RDDs. This can be helpful to extract elements from similar characteristics from two RDDs into a single RDD.

```
rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
r=rdd.filter(lambda x: x%2==0)
p=rdd.filter(lambda x: x%5==0)
r.union(p).collect()
```

▸ (1) Spark Jobs

```
Out[21]: [2, 4, 6, 8, 10, 5, 10]
```

4. flatMap ():

Transformation The .flatMap() transformation peforms same as the .map() transformation except the fact that .flatMap() transformation return seperate values for each element from original RDD.

```
rdd = sc.parallelize(["Hey there", "This is pyspark examples"])
rdd.flatMap(lambda x:x.split(" ")).collect()
```

▸ (1) Spark Jobs

```
Out[1]: ['Hey', 'there', 'This', 'is', 'pyspark', 'examples']
```

PANDAS DATAFRAMES:

The following screenshots are codes based on Selecting, Renaming, Filtering Data in a Pandas DataFrame.

1. Create the spark dataframe:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('pyspark - example join').getOrCreate()
data = [(('Ram'), '1991-04-01', 'M', 3000),(('Mike'), '2000-05-19', 'M', 4000),(('Rohini'), '1978-09-05', 'M', 4000),(('Maria'), '1967-12-01', 'F',
4000),(('Jenis'), '1980-02-17', 'F', 1200)]
columns = ["Name", "DOB", "Gender", "salary"]
df = spark.createDataFrame(data=data,schema=columns)
df.withColumnRenamed("DOB","date of birth").show()
df.withColumnRenamed("DOB","date of birth").withColumnRenamed("Name","personname").show()
```

(6) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Name: string, DOB: string ... 2 more fields]

```
+------+-------------+------+------+
|  Name|date of birth|Gender|salary|
+------+-------------+------+------+
|   Ram|   1991-04-01|     M|  3000|
|  Mike|   2000-05-19|     M|  4000|
|Rohini|   1978-09-05|     M|  4000|
| Maria|   1967-12-01|     F|  4000|
| Jenis|   1980-02-17|     F|  1200|
+------+-------------+------+------+

+----------+-------------+------+------+
|personname|date of birth|Gender|salary|
+----------+-------------+------+------+
|       Ram|   1991-04-01|     M|  3000|
|      Mike|   2000-05-19|     M|  4000|
|    Rohini|   1978-09-05|     M|  4000|
|     Maria|   1967-12-01|     F|  4000|
|     Jenis|   1980-02-17|     F|  1200|
+----------+-------------+------+------+
```

```
data = df.selectExpr("Gender as category","DOB","Name as name","salary")

data.show()
```

(3) Spark Jobs

data: pyspark.sql.dataframe.DataFrame = [category: string, DOB: string ... 2 more fields]

```
+--------+----------+------+------+
|category|       DOB|  name|salary|
+--------+----------+------+------+
|       M|1991-04-01|   Ram|  3000|
|       M|2000-05-19|  Mike|  4000|
|       M|1978-09-05|Rohini|  4000|
|       F|1967-12-01| Maria|  4000|
|       F|1980-02-17| Jenis|  1200|
+--------+----------+------+------+
```

2. Select the 'salary' as 'Amount' using aliasing:

```
from pyspark.sql.functions import col
data = df.select(col("Name"),col("DOB"),
                 col("Gender"),
col("salary").alias('Amount'))
data.show()
```

(3) Spark Jobs

data: pyspark.sql.dataframe.DataFrame = [Name: string, DOB: string ... 2 more fields]

```
+------+----------+------+------+
|  Name|       DOB|Gender|Amount|
+------+----------+------+------+
|   Ram|1991-04-01|     M|  3000|
|  Mike|2000-05-19|     M|  4000|
|Rohini|1978-09-05|     M|  4000|
| Maria|1967-12-01|     F|  4000|
| Jenis|1980-02-17|     F|  1200|
+------+----------+------+------+
```

PYSPARK VIEWS:

In PySpark, **views** are temporary representations of DataFrames or tables that allow you to query data using SQL syntax without physically persisting the data to disk. Views are useful for running SQL queries within a PySpark application, enabling seamless integration between PySpark and SQL-like operations.

Types of Views in PySpark

1. **Global Temporary View**
    - Created using the createGlobalTempView() method.
    - It is associated with a global temporary database (global_temp) that is accessible across all sessions within the same application.
    - It is especially useful when you want the view to be shared across multiple PySpark sessions.
2. **Temporary View**
    - Created using the createOrReplaceTempView() method.
    - It is session-scoped, meaning the view is only accessible within the session in which it was created.
    - Once the session ends, the view is automatically dropped.

EXAMPLES OF PYSPARK VIEWS:

```
from pyspark.sql import SparkSession
spark = SparkSession \
.builder \
.appName("SparkByExamples.com") \
.enableHiveSupport() \
.getOrCreate()
data = [("James","Smith","USA","CA"),
("Michael","Rose","USA","NY"),
("Robert","Williams","USA","CA"),
("Maria","Jones","USA","FL")
]
columns = ["firstname","lastname","country","state"]
sampleDF = spark.sparkContext.parallelize(data).toDF(columns)
sampleDF.createOrReplaceTempView("Person")
sampleDF.createOrReplaceTempView("mydata")
sampleDF.show()
```

```
▸ (5) Spark Jobs
▸ sampleDF:  pyspark.sql.dataframe.DataFrame = [firstname: string, lastname: string ... 2 more fields]
+---------+--------+-------+-----+
|firstname|lastname|country|state|
+---------+--------+-------+-----+
|    James|   Smith|    USA|   CA|
|  Michael|    Rose|    USA|   NY|
|   Robert|Williams|    USA|   CA|
|    Maria|   Jones|    USA|   FL|
+---------+--------+-------+-----+
```