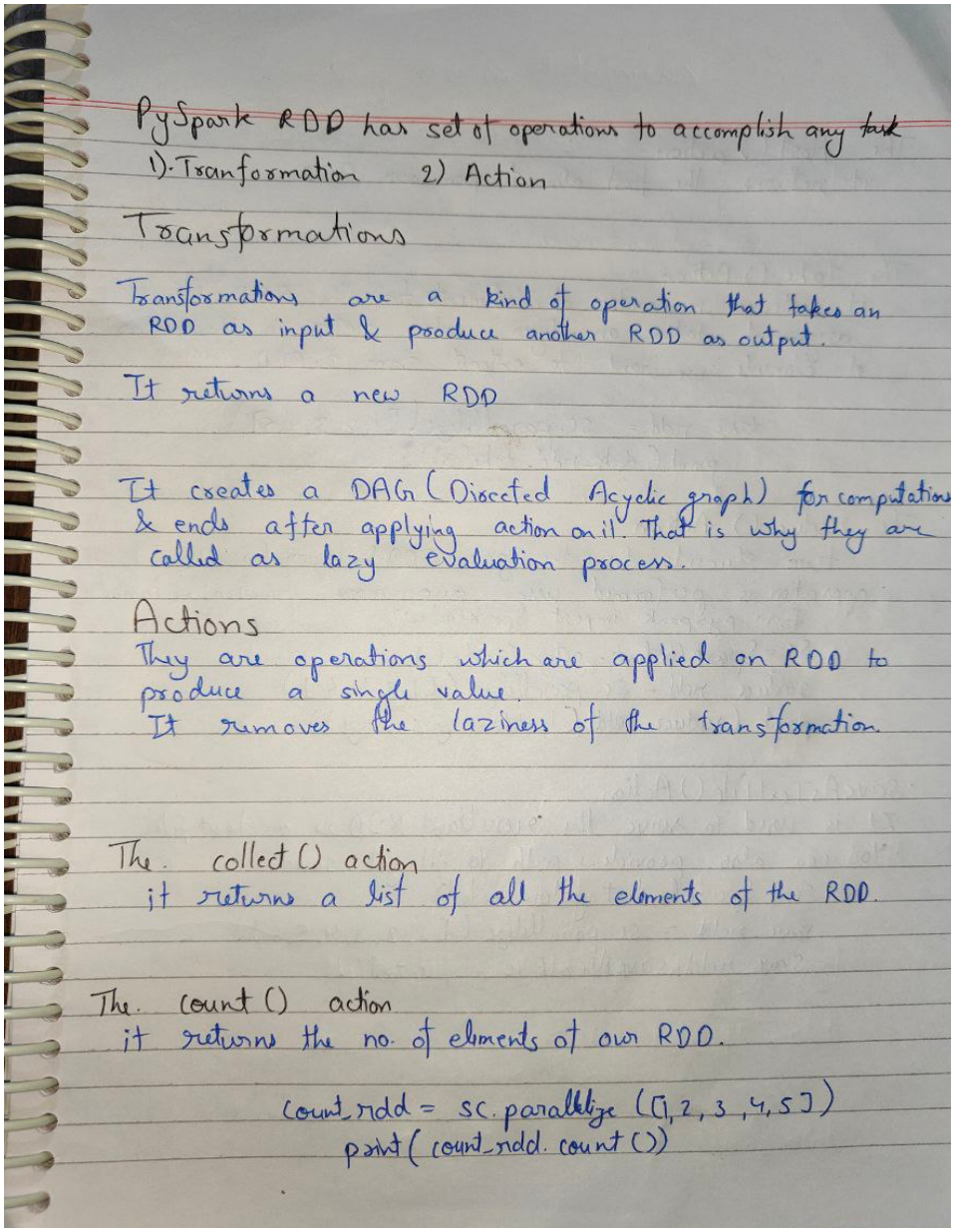
Assignment-13

Name-Ajay Chaudhary
Batch-Data Engineering(Batch-1)

Handwritten notes-



PySpark RDD has set of operations to accomplish any task
1). Tranformation 2) Action

Transformations

Transformations are a kind of operation that takes an RDD as input & produce another RDD as output.

It returns a new RDD

It creates a DAG (Directed Acyclic graph) for computations & ends after applying action on it. That is why they are called as lazy evaluation process.

Actions

They are operations which are applied on RDD to produce a single value.
It removes the laziness of the transformation.

The collect() action
it returns a list of all the elements of the RDD.

The count() action
it returns the no. of elements of our RDD.

```
count_rdd = sc.parallelize([1,2,3,4,5])
print(count_rdd.count())
```

The. first() action
it returns the first element from our RDD

The. take() Action
it returns n number of elements from the RDD. The 'n' argument takes an integer which refer to the number of elements we want to extract from the RDD.

```
take_rdd = sc.parallelize([1,2,3,4,5])
print(take_rdd.take(3))
```

The. reduce() Action
two elements from the given RDD & operates. This operation is performed using anonymous function or lambda.

```
from pyspark import SparkContext
sc = SparkContext.getorCreate()
reduce_rdd = sc.parallelize([1,3,4,6])
print(reduce_rdd(lambda x,y : x+y))
```

· SaveAsTextFile() Action.
It is used to serve the resultant RDD as a text file. You can also provide path to file get saved.

```
sc = SparkContext.getorCreate()
save_rdd = sc.parallelize([1,2,3,4,5,6])
save_rdd.saveAsTextFile('file.txt')
```

Transformations in RDDs

The .map() transformation

- .map() transformation maps a value to the elements of an RDD. The .map() function takes an anonymous fnction & applies to each element of the RDD.

```
my_rdd = sc.parallelize([1,2,3,4])
print(my_rdd.map(lambda x: x+10).collect())
```

The .filter() Transformation

It is an operation for filtering elements from a PySpark RDD

it returns an RDD having elements which passed the condition

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

The .union() transformation

It combines ~~two~~ RDDs & return the union of the input RDDs.

```
union_inp = sc.parallelize([2,4,5,6,7,8,9])
union_rdd_1 = ~~union_rdd~~ union_inp.filter(lambda x: x%2==0
union_rdd_2 = union_inp.filter(lambda x: x%3==0)
print(union_rdd_1.union(union_rdd_2).collect())
```

The .flatMap() transformation

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

Renaming Columns

Method 1:

Using withColumnRenamed() — existing

```
df.withColumnRenamed("DOB", "Date of Birth").show()
```

"Date of Birth" ↑ new column

Renaming multiple columns

```
df.withColumnRenamed("Gender", "Sex").
    withColumnRenamed("salary", "Amount").show()
```

Method 2:

Using selectExpr() — Renaming

```
data = df.selectExpr("Name as name", "DOB", "Gender", "Salary")
data.show()
```

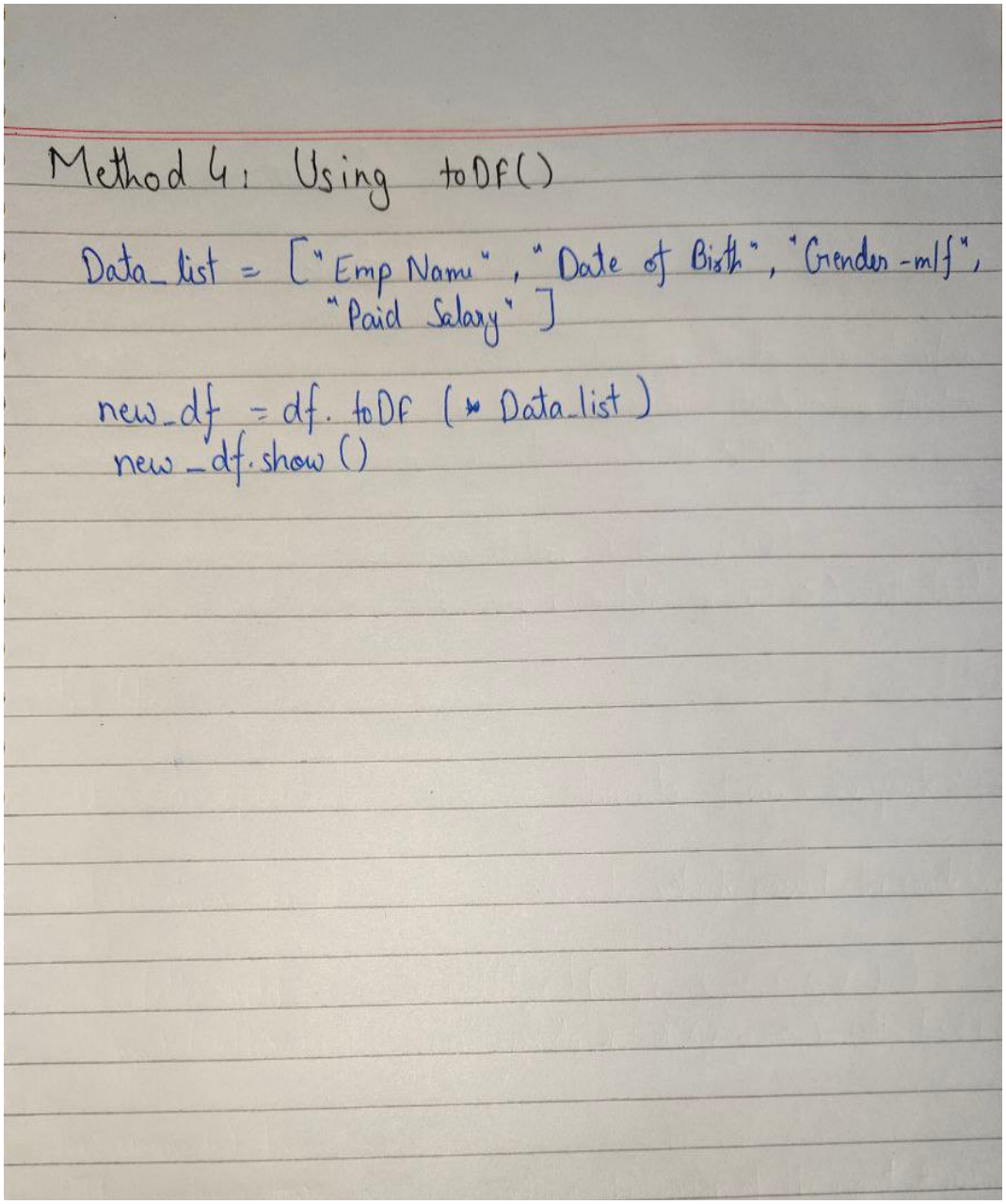
Method 3:

Using select() method

```
from pyspark.sql.functions import col

data = df.select(col("Name"), col("DOB"),
                 col("Gender"),
                 col("salary").alias('Amount'))
data.show()
```

↑ Renaming



PySpark RDD Operations

Resilient Distributed Dataset or RDD in a PySpark is a core data structure of PySpark. PySpark RDD's is a low-level object and are highly efficient in performing distributed tasks.

PySpark RDD has a set of operations to accomplish any task. These operations are of two types:
1. Transformations
2. Actions

Actions in PySpark RDDs

Actions are a kind of operation which are applied on an RDD to produce a single value.

1. The .collect() Action
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.

[36]:
```
#.collect action
collect_rdd = sc.parallelize([1,2,3,4,5])
print(collect_rdd.collect())
```

```
[1, 2, 3, 4, 5]
```

2. The .count() Action
The **.count()** action on an RDD is an operation that returns the number of elements of our RDD.

```
.count() action
```

[5]:
```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
count_rdd = sc.parallelize([1,2,3,4,5,5,6,7,8,9])
print(count_rdd.count())
```

```
10
```

3. The .first() Action
The **.first()** action on an RDD returns the first element from our RDD.

```
.first() action
```

[7]:
```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
count_rdd = sc.parallelize([1,2,3,4,5,5,6,7,8,9])
print(count_rdd.first())
```

```
1
```

4. The .take() Action
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

```
.take() action
```

```
[8]: from pyspark import SparkContext
     sc = SparkContext.getOrCreate()
     count_rdd = sc.parallelize([1,2,3,4,5,5,6,7,8,9])
     print(count_rdd.take(4))
```

```
[1, 2, 3, 4]
```

5. The .reduce() Action

The **.reduce()** Actiontakes two elements from the given RDD and operates. This operation is performed using an anonymous function or lambda.

```
.reduce() action
```

```
[6]: from pyspark import SparkContext
     sc = SparkContext.getOrCreate()
     reduce_rdd = sc.parallelize([1,3,4,6])
     print(reduce_rdd.reduce(lambda x, y : x + y))
```

```
14
```

6. The .saveAsTextFile() Action

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.

```
.saveAsTextFile() action
```

```
[3]: from pyspark import SparkContext
     sc = SparkContext.getOrCreate()
     save_rdd = sc.parallelize([1,2,3,4,5,6])
     save_rdd.saveAsTextFile('file.txt')
```

Transformations in PySpark RDDs

Transformations are the kind of operations that are performed on an RDD and return a new RDD.

1. The .map() Transformation

The **.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.

```
.map() Transformation
```

```
[10]: #.map() Transformation
my_rdd = sc.parallelize([1,2,3,4])
print(my_rdd.map(lambda x: x+ 100).collect())
```

```
[101, 102, 103, 104]
```

2. The .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

```
.filter() transformation
```

```
[11]: #.filter() transformation
filter_rdd = sc.parallelize([2, 3, 4, 5, 6, 7])
print(filter_rdd.filter(lambda x: x%2 == 0).collect())
```

```
[2, 4, 6]
```

3. The .union() Transformation

The **.union()** transformation combines two RDDs and returns the union of the input two RDDs.

```
[ ]: .union() transformation
```

```
[13]: #.union() transformation
union_inp = sc.parallelize([2,4,5,6,7,8,9])
union_rdd_1 = union_inp.filter(lambda x: x % 2 == 0)
union_rdd_2 = union_inp.filter(lambda x: x % 3 == 0)
print(union_rdd_1.union(union_rdd_2).collect())
```

```
[Stage 12:==============================>                    (4 + 4) / 8]
[2, 4, 6, 8, 6, 9]
```

4. The .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.

```
.flatMap()transformation
```

```
#.flatMap()transformation
flatmap_rdd = sc.parallelize(["Hey there", "This is PySpark RDD Transformations","Hello World!!"])
(flatmap_rdd.flatMap(lambda x: x.split(" ")).collect())
```

```
['Hey',
 'there',
 'This',
 'is',
 'PySpark',
 'RDD',
 'Transformations',
 'Hello',
 'World!!']
```

Transformations in Pair RDDs

Since Pair RDDs are created from multiple tuples, we need to use operations that make use of

1. The .reduceByKey() Transformation

The **.reduceByKey()** transformation performs multiple parallel processes for each key in the data and combines the values for the same keys. It uses an anonymous function or lambda to perform the task.

[31]:
```
#Transformation in pair RDDs
#.reduceByKey() Transformation
marks_rdd = sc.parallelize([('Rahul', 25), ('Swati', 26), ('Shreya', 22), ('Abhay', 29),
                            ('Rohan', 22), ('Rahul', 23), ('Swati', 19), ('Shreya', 28), ('Abhay', 26), ('Rohan', 22)])
print(marks_rdd.reduceByKey(lambda x, y: x + y).collect())
```

```
[Stage 46:>                                                         (0 + 4) / 4]
[('Rohan', 44), ('Rahul', 48), ('Swati', 45), ('Shreya', 50), ('Abhay', 55)]
```

2. The .sortByKey() Transformation

The **.sortByKey()** transformation sorts the input data by keys from key-value pairs either in ascending or descending order. It returns a unique RDD as a result.

```
#.sortByKey() Transformation

marks_rdd = sc.parallelize([('Rahul', 25), ('Swati', 26), ('Shreya', 22), ('Abhay', 29),
                            ('Rohan', 22), ('Rahul', 23), ('Swati', 19), ('Shreya', 28), ('Abhay', 26), ('Rohan', 22)])
print(marks_rdd.sortByKey('ascending').collect())
```

```
[('Abhay', 29), ('Abhay', 26), ('Rahul', 25), ('Rahul', 23), ('Rohan', 22), ('Rohan', 22), ('Shreya', 22), ('Shreya', 28), ('Swati', 26), ('S
wati', 19)]
```

3. The .groupByKey() Transformation

The **.groupByKey()** transformation groups all the values in the given data with the same key together.

[33]:
```
#.groupByKey() Transformation
marks_rdd = sc.parallelize([('Rahul', 25), ('Swati', 26), ('Shreya', 22), ('Abhay', 29),
                            ('Rohan', 22), ('Rahul', 23), ('Swati', 19), ('Shreya', 28), ('Abhay', 26), ('Rohan', 22)])
dict_rdd = marks_rdd.groupByKey().collect()
for key, value in dict_rdd:
    print(key, list(value))
```

```
Rohan [22, 22]
Rahul [25, 23]
Swati [26, 19]
Shreya [22, 28]
Abhay [29, 26]
```

Actions in Pair RDDs

1. The countByKey() Action

The **.countByKey()** option is used to count the number of values for each key in the given data. This action returns a dictionary and one can extract the keys and values by iterating over the extracted dictionary using loops. Since we are getting a dictionary as a result, we can also use the dictionary methods such as **.keys()**, **.values()** and **.items()**.

[34]:
```
# The countByKey() Action
marks_rdd = sc.parallelize([('Rahul', 25), ('Swati', 26), ('Rohan', 22), ('Rahul', 23), ('Swati', 19),
                            ('Shreya', 28), ('Abhay', 26), ('Rohan', 22)])
dict_rdd = marks_rdd.countByKey().items()
for key, value in dict_rdd:
    print(key, value)
```

```
Rahul 2
Swati 2
Rohan 2
Shreya 1
Abhay 1
```

[16]:
```
from pyspark.sql import SparkSession

# Create a spark session
spark = SparkSession.builder.appName('PySpark_example').getOrCreate()

# Create data in dataframe
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)]

# Column names in dataframe
columns = ["Name", "DOB", "Gender", "salary"]

# Create the spark dataframe
df = spark.createDataFrame(data=data,schema=columns)

# Print the dataframe
df.show()
```

Method 1: Using withColumnRenamed()

We will use of withColumnRenamed() method to change the column names of pyspark data frame.

```
renaming column using .withColumnRenamed()
```

```
[35]: df.withColumnRenamed("DOB","DateOfBirth").show()
```

```
+------+-----------+------+------+
|  Name|DateOfBirth|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|
+------+-----------+------+------+
```

Renaming multiple column names

```
Renaming multiple columns
```

```
[19]: df.withColumnRenamed("Gender","Sex").withColumnRenamed("salary","Amount").show()
```

```
+------+----------+---+------+
|  Name|       DOB|Sex|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|
+------+----------+---+------+
```

Method 2: Using selectExpr()

Renaming the column names using **selectExpr()** method

```
Using selectExpr()
```

```
[26]: data = df.selectExpr("Name as name","DOB","Gender","salary")
data.show()
```

```
+------+----------+------+------+
|  name|       DOB|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|
+------+----------+------+------+
```

Method 3: Using select() method

Selects the cols in the dataframe and returns a new DataFrame.

Here we Rename the column name ‘salary’ to ‘Amount’

```
Using select()method
```

[28]:
```
from pyspark.sql.functions import col

data = df.select(col("Name"),col("DOB"),
                 col("Gender").alias('Sex'),
                 col("salary").alias('Amount'))
data.show()
```

```
+------+----------+---+------+
|  Name|       DOB|Sex|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|
+------+----------+---+------+
```

Method 4: Using toDF()

This function returns a new DataFrame that with new specified column names.

```
using toDF() method
```

[29]:
```
Data_list = ["Emp Name","Date of Birth",
             " Gender-m/f","Paid salary"]

new_df = df.toDF(*Data_list)
new_df.show()
```

```
+--------+-------------+-----------+-----------+
|Emp Name|Date of Birth| Gender-m/f|Paid 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|
+--------+-------------+-----------+-----------+
```