Assignment 3

Name: **E.R Harish**

I. Actions in PySpark RDDs

1. The .collect() Action

```python
rdd = sc.parallelize([1, 2, 3])
print(rdd.collect())
```

(1) Spark Jobs

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

Summary:
Returns all elements of the RDD as a list. Useful for debugging and small datasets.

2. The .count() Action

```python
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.parallelize([1, 2, 3, 4])
print(rdd.count())
```

(1) Spark Jobs

```
4
```

Summary:
Counts the total number of elements in an RDD. Helps verify dataset size.

3. The .first() Action

```python
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.parallelize([10, 20, 30])
print(rdd.first())
```

(2) Spark Jobs

```
10
```

Summary:
Retrieves the first element of an RDD. Good for quick data validation.

4. The .take() Action

```python
rdd = sc.parallelize([5, 6, 7, 8])
print(rdd.take(2))
```

(2) Spark Jobs

```
[5, 6]
```

Summary:
Fetches the first n elements of the RDD. Useful for inspecting a subset of data.

5. The .reduce() Action

```python
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.parallelize([1, 2, 3])
print(rdd.reduce(lambda x, y: x + y))  # Sum of elements
```

(1) Spark Jobs

```
6
```

Summary:
Aggregates elements using a specified binary operation (e.g., summing all values).

6. The .saveAsTextFile() Action

```python
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.parallelize([10, 20, 30])
rdd.saveAsTextFile('files.txt')
```

(1) Spark Jobs
Job 7 View (Stages: 1/1)

Summary:
Saves the RDD’s content as a text file in the specified directory. Creates partitions as separate files.

II. Transformations in PySpark RDDs

1. The .map() Transformation

```python
rdd = sc.parallelize([1, 2, 3])
print(rdd.map(lambda x: x * 2).collect())
```

(1) Spark Jobs

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

Summary:
Applies a function to each element and returns a new RDD. Example: Add 10 to every number.

2. The .filter() Transformation

```python
rdd = sc.parallelize([1, 2, 3, 4])
print(rdd.filter(lambda x: x % 2 == 0).collect())  # Even numbers
```

(1) Spark Jobs

```
[2, 4]
```

Summary:
Filters elements based on a condition, returning a new RDD. Example: Retain only even numbers.

3. The .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())
```

(1) Spark Jobs

```
[2, 4, 6, 8, 6, 9]
```

Summary:
Combines two RDDs into one containing all elements from both RDDs.

4. The .flatMap() Transformation

```
rdd = sc.parallelize(["hello world", "PySpark RDD"])
print(rdd.flatMap(lambda x: x.split(" ")).collect())
```

(1) Spark Jobs

```
['hello', 'world', 'PySpark', 'RDD']
```

Summary:
Similar to .map(), but flattens the output. Useful for splitting strings into words.

III. Transformations in Pair RDDs

1. The .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())
```

(1) Spark Jobs

```
[('Shreya', 50), ('Swati', 45), ('Rahul', 48), ('Abhay', 55), ('Rohan', 44)]
```

Summary:
Combines values for each key using a binary operation (e.g., summing marks for students with the same name).

2. The .sortByKey() Transformation

Just now (1s) 12 Python

```
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())
```

(3) Spark Jobs

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

Summary:
Sorts key-value pairs by keys in ascending or descending order.

3. The .groupByKey() Transformation

Just now (1s) 13 Python

```
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))
```

(1) Spark Jobs

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

Summary:
Groups values by their keys, returning an RDD with each key and its associated list of values.

IV. Actions in Pair RDDs

1. The countByKey() Action

```python
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)
```

(1) Spark Jobs

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

Summary:

Counts the number of values for each key and returns a dictionary.

I. **Working with Pandas**

1. Selecting, Renaming, and Filtering Data in a DataFrame.

Just now (9s) 15 Python

```python
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("PySparkExample").getOrCreate()

data = [('Ravi', 25, 'Delhi'), ('Meena', 30, 'Mumbai'), ('Arun', 22, 'Chennai')]
columns = ['Name', 'Age', 'City']

df = spark.createDataFrame(data, columns)

# Select single column
df.select('Name').show()

# Select multiple columns
df.select('Name', 'Age').show()
```

(6) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Name: string, Age: long ... 1 more field]

```
+-----+
| Name|
+-----+
| Ravi|
|Meena|
| Arun|
+-----+

+-----+---+
| Name|Age|
+-----+---+
| Ravi| 25|
|Meena| 30|
| Arun| 22|
+-----+---+
```

Summary:

- **Selecting Columns:**
  Retrieve specific columns using df['column_name'] or df[['col1', 'col2']].

- **Renaming Columns:**
  Use .rename(columns={'old_name': 'new_name'}) to rename columns.

- **Filtering Rows:**
  Apply conditions like df[df['column'] > value] to filter rows based on criteria.

2. Manipulating, Dropping, Sorting, Aggregating, Joining, and Grouping DataFrames

```python
# Add a new column with calculated values
df_with_new_col = df.withColumn('AgeNextYear', df['Age'] + 1)
df_with_new_col.show()
# Drop a column
df_dropped = df.drop('City')
df_dropped.show()
# Sort by Age in descending order
df_sorted = df.orderBy(df['Age'].desc())
df_sorted.show()
# Group by City and calculate the average age
df.groupBy('City').avg('Age').show()
# Joining two DataFrames
data2 = [('Delhi', 'North'), ('Mumbai', 'West'), ('Chennai', 'South')]
columns2 = ['City', 'Region']

df2 = spark.createDataFrame(data2, columns2)

joined_df = df.join(df2, on='City', how='inner')
joined_df.show()
```

```
+-----+---+-------+-----------+
| Name|Age|   City|AgeNextYear|
+-----+---+-------+-----------+
| Ravi| 25|  Delhi|         26|
|Meena| 30| Mumbai|         31|
| Arun| 22|Chennai|         23|
+-----+---+-------+-----------+


+-----+---+
| Name|Age|
+-----+---+
| Ravi| 25|
|Meena| 30|
| Arun| 22|
+-----+---+
```

Summary:

- **Manipulating Data:**
  Modify values or create new columns using operations like df['new_col'] = df['col'] * 2.
- **Dropping Data:**
  Use .drop(columns=['col']) to remove columns or .drop(index) for rows.

- **Sorting Data:**
  Sort values using .sort_values(by='col', ascending=True/False).
- **Aggregations:**
  Apply functions like .sum(), .mean(), or .agg({'col1': 'sum', 'col2': 'max'}).
- **Joining DataFrames:**
  Combine DataFrames with .merge() for relational joins or .concat() for stacking.
- **Grouping Data:**
  Use .groupby('col').agg() for operations like grouping and applying aggregations.

3. Applying Functions in a DataFrame

Just now (2s) 19 Python

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Define a UDF to add "Mr./Ms." prefix
def add_prefix(name):
    return f"Mr./Ms. {name}"

add_prefix_udf = udf(add_prefix, StringType())

# Apply UDF
df_with_prefix = df.withColumn('NameWithPrefix', add_prefix_udf(df['Name']))
df_with_prefix.show()
```

(3) Spark Jobs

df_with_prefix: pyspark.sql.dataframe.DataFrame = [Name: string, Age: long ... 2 more fields]

```
+-----+---+-------+--------------+
| Name|Age|   City|NameWithPrefix|
+-----+---+-------+--------------+
| Ravi| 25|  Delhi|  Mr./Ms. Ravi|
|Meena| 30| Mumbai| Mr./Ms. Meena|
| Arun| 22|Chennai|  Mr./Ms. Arun|
+-----+---+-------+--------------+
```

Summary:

- **Element-wise Operations:**
  Use .apply() to apply a function to rows or columns, e.g., df['col'].apply(lambda x: x*2).
- **Row/Column-wise Operations:**
  Apply functions row-wise (axis=1) or column-wise (axis=0).
- **Vectorized Operations:**
  Leverage NumPy or Pandas for efficient operations directly on columns, e.g., df['col'] * 10.

I. PySpark: Creating Local and Temporary Views

1. Creating Temporary Views

```python
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("PySparkExample").getOrCreate()

data = [('Ravi', 25, 'Delhi'), ('Meena', 30, 'Mumbai'), ('Arun', 22, 'Chennai')]
columns = ['Name', 'Age', 'City']

df = spark.createDataFrame(data, columns)

# Create a temporary view
df.createOrReplaceTempView("people")

# Query the view using SQL
spark.sql("SELECT * FROM people WHERE Age > 25").show()
```

```
(3) Spark Jobs
df: pyspark.sql.dataframe.DataFrame = [Name: string, Age: long ... 1 more field]
+-----+---+------+
| Name|Age|  City|
+-----+---+------+
|Meena| 30|Mumbai|
+-----+---+------+
```

Summary:

- **Temporary Views:** Session-scoped views created using createOrReplaceTempView. Useful for querying DataFrames with SQL.

2. Creating Global Temporary Views

```python
# Create a global temporary view
df.createGlobalTempView("global_people")

# Query the global view using SQL (prefix with `global_temp.`)
spark.sql("SELECT * FROM global_temp.global_people").show()
```

```
(3) Spark Jobs
+-----+---+-------+
| Name|Age|   City|
+-----+---+-------+
| Ravi| 25|  Delhi|
|Meena| 30| Mumbai|
| Arun| 22|Chennai|
+-----+---+-------+
```

Summary:

- Global Temporary Views: Accessible across multiple sessions within the same Spark application using createGlobalTempView. Use the global_temp prefix to query them.