**PYSPARK ASSIGNMENT**

**Name**: S. Nikhil Sai

**Date**: 20-11-2024

1. **Action And Transformations**

from pyspark import SparkContext

from pyspark.sql import SparkSession

sc=SparkContext.getOrCreate()

spark1 =SparkSession.builder.appName('Pyspark RDD program').getOrCreate()

rdd = sc.parallelize([('C', 87, 89 ,85, 94),('A',90,87,67,96),('B',90,98,95,79),('D',98,90,78,92)])

sub=['Division','English','Maths','Science','Social']

A computer screen shot of a black background

Description automatically generated with medium confidence

**Explanation:** This code sets up a PySpark environment using SparkContext and SparkSession. It creates an RDD containing tuples of division names and subject marks. The sc.parallelize() function converts the list into an RDD for distributed processing. A list sub defines the subject names corresponding to the marks. This setup prepares data for further analysis or processing in PySpark.

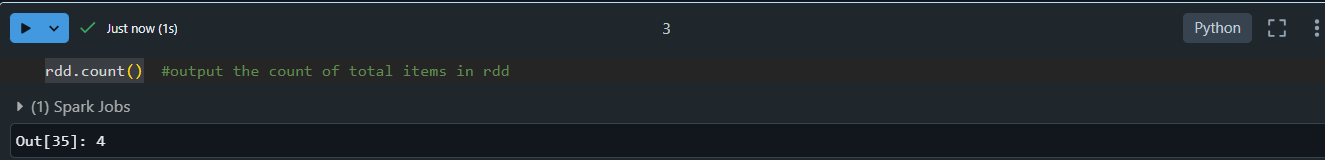
rdd.collect()

A black rectangular object with a blue background

Description automatically generated with medium confidence

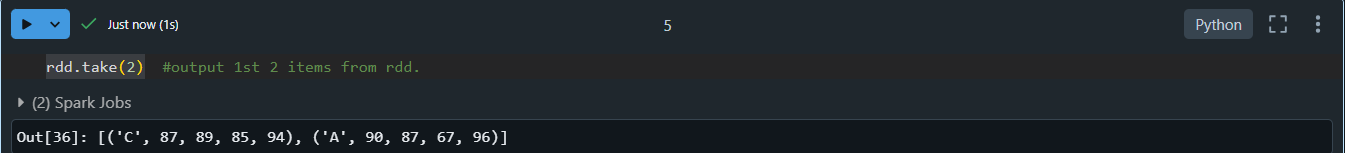
**Expalantion:** The `rdd.collect()` function gathers all elements of the RDD and returns them as a list to the driver program. For the given RDD, it retrieves the tuples containing division names and subject marks. It’s suitable for small datasets but not recommended for large ones due to memory constraints.

rdd.count()



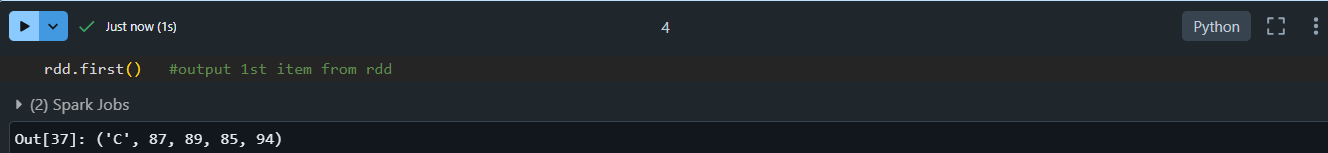
**Explanation:** The `rdd.count()` function returns the total number of elements (rows) in the RDD. For the given RDD, it will output `4`, as there are four tuples in the dataset.

rdd.take(2)



**Explanation:** The rdd.take(2) function retrieves the first 2 elements (rows) from the RDD. It’s useful for quickly viewing a subset of the data.

rdd.first()



**Explanation:** The rdd.first() function retrieves the first element (row) of the RDD. It’s useful to check the initial element of the dataset.

reduce\_rdd=sc.parallelize([1,2,3,4,5,6,7,8,9,10])

reduce\_rdd.reduce(lambda x,y: x+y)

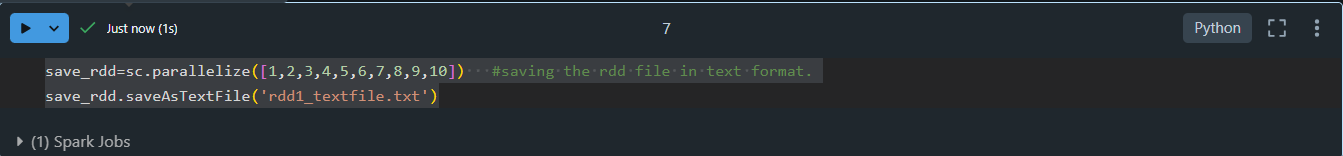
A screen shot of a computer

Description automatically generated

**Explanation:** The code creates an RDD containing numbers from 1 to 10. The `reduce()` function applies a binary operation (`x + y`) to combine all elements, summing them up. It works iteratively, adding two numbers at a time until only one result remains. For this RDD, the sum of numbers is 55. The `reduce()` function is useful for aggregation tasks.

save\_rdd=sc.parallelize([1,2,3,4,5,6,7,8,9,10])   #saving the rdd file in text format.

save\_rdd.saveAsTextFile('rdd1\_textfile.txt')

****

**Explanation:** The code creates an RDD with numbers from 1 to 10 using `sc.parallelize()`. The `saveAsTextFile()` function is then used to save the RDD's contents to a text file named `rdd1\_textfile.txt`. The data will be saved in the form of multiple text files (one for each partition) inside a folder named `rdd1\_textfile.txt`. This function is useful for exporting RDD data to the file system.

map\_rdd=sc.parallelize([1,2,3,4,5,6,7,8,9,10])       #map function

map\_rdd.map(lambda x: x+10).collect()

A screen shot of a computer

Description automatically generated

Explanation: The code creates an RDD with numbers from 1 to 10 using `sc.parallelize()`. The `map()` function applies a transformation to each element, adding 10 to each number. The `collect()` function then retrieves the transformed data as a list. The result will be a list of numbers with 10 added to each element: `[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]`.

filter\_rdd2= sc.parallelize(['Rahul', 'Riya','Swathi', 'Harish'])

print(filter\_rdd2.filter(lambda x:x.startswith('R')).collect())

A screenshot of a computer

Description automatically generated

**Explanation:** The code creates an RDD with a list of names. The `filter()` function is used to filter out the names that start with the letter 'R' by applying the lambda function `x.startswith('R')`. The `collect()` function then retrieves the filtered names as a list.

union\_rdd=sc.parallelize([1,2,3,4,5,6,7,8,9,10])

union\_rdd1=union\_rdd.filter(lambda x: x%2==0)

union\_rdd2=union\_rdd.filter(lambda x: x%2!=0)

union\_rdd1.union(union\_rdd2).collect()

A black rectangular object with a number

Description automatically generated

**Explanation:** The code creates an RDD union\_rdd with numbers from 1 to 10. It then filters the even numbers (union\_rdd1) and the odd numbers (union\_rdd2). The union() function combines the two filtered RDDs, returning all elements from both, without duplication. The collect() function retrieves the combined result as a list. This combines the even and odd numbers into a single RDD.

flatmap\_rdd=sc.parallelize(['Hey there', 'This is pyspark RDD transformation'])

print(flatmap\_rdd.flatMap(lambda x: x.split(' ')).collect())  #using flatmap

A screenshot of a computer program

Description automatically generated

**Explanation:** The code creates an RDD with two strings. The flatMap() function is used to transform each string into a list of words by splitting them at spaces. Unlike map(), which returns an RDD of lists, flatMap() "flattens" the result, creating a single list of words. The collect() function retrieves the result as a list. This breaks the strings into individual words and combines them into a flat structure.

marks\_rdd = sc.parallelize([('Rahul', 25), ('Swati', 26), ('Shreya', 22), ('Abhay', 29), ('Rohan', 22), ('Rahul', 23), ('Swati', 19), ('Shreya', 28), ('Abhay', 26), ('Rohan', 22)])

marks\_rdd.reduceByKey(lambda x, y: x + y).collect()

A screen shot of a computer

Description automatically generated

**Explanation:** The code creates an RDD with tuples containing names and ages. The reduceByKey() function is used to aggregate the values (ages) for each unique key (name) by applying the lambda function lambda x, y: x + y, which sums the ages. The collect() function retrieves the result as a list of tuples with names and their total ages.

print(marks\_rdd.sortByKey('ascending').collect())

A screenshot of a computer

Description automatically generated

**Explanation:** The code attempts to sort the RDD marks\_rdd by its keys (names) in ascending order using sortByKey(). However, there is an issue: the sortByKey() function requires a parameter specifying whether the sorting should be ascending or descending, typically as a boolean (True for ascending, False for descending).

marks\_list= marks\_rdd.groupByKey().collect()

for i,j in marks\_list:

  print(i,list(j))

A black and grey rectangular object

Description automatically generated

**Explanation:** The code groups the marks\_rdd by keys (names) using the groupByKey() function, which creates an RDD of key-value pairs where each key is associated with a list of values. The collect() function retrieves the grouped data as a list. Then, the for loop iterates over the grouped data and prints each key (name) along with the list of values (ages) associated with it.

marks\_count = marks\_rdd.countByKey().items()

for key, value in marks\_count:

  print(key, value)

A black and grey background

Description automatically generated with medium confidence

**Explanation**: The code uses countByKey() to count the number of occurrences of each key (name) in the marks\_rdd. It returns a dictionary-like structure where keys are the names, and values are the counts of how many times each name appears. The items() method retrieves the key-value pairs from the dictionary, and the for loop prints each key (name) and its corresponding count.

**2) Selecting, Renaming, Filtering Data in a Pandas DataFrame**

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('pyspark-select,filetr,rename pandas').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"]

A black screen with colorful lines

Description automatically generated with medium confidence

**Explanation:** The code creates a SparkSession named spark and initializes an RDD (list of tuples) with employee data, including names, dates of birth, gender, and salaries. The columns are labeled as "Name", "DOB", "Gender", and "salary". This is the initial step to create a DataFrame in PySpark.

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

df.withColumnRenamed("DOB","date of birth").show()         #column rename

df.withColumnRenamed("DOB","date of birth").withColumnRenamed("Name","personname").show()

A screenshot of a computer

Description automatically generated

**Explanation:** The code creates a PySpark DataFrame `df` from a list of tuples with columns `"Name"`, `"DOB"`, `"Gender"`, and `"salary"`. It then uses `withColumnRenamed()` to rename the `"DOB"` column to `"date of birth"`. The second `withColumnRenamed()` call chains another renaming, changing `"Name"` to `"personname"`. The `show()` method is used to display the DataFrame after each renaming operation. This demonstrates how to rename columns in PySpark DataFrames.

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

data.show()

A screenshot of a computer

Description automatically generated

**Explanation:** The code demonstrates how to use the selectExpr() function in PySpark to create aliases for column names in a DataFrame. Specifically, it renames the columns "Gender" to "category" and "Name" to "name". The selectExpr() method allows for column renaming and transformations using SQL expressions.

from pyspark.sql.functions import col

data = df.select(col("Name"),col("DOB"),

                 col("Gender"),

                 col("salary").alias('Amount'))

data.show()

A screenshot of a computer

Description automatically generated

Explanation: The code uses the col() function from pyspark.sql.functions to select specific columns from the DataFrame df, and it renames the "salary" column to "Amount" using the alias() method. This is a more explicit way to select and rename columns compared to selectExpr().

**3) Pyspark view and temp view.**

from pyspark.sql import SparkSession

spark = SparkSession \

.builder \

.appName("Spark\_view and temp.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()

A screenshot of a computer

Description automatically generated

**Explanation:** The code creates a Spark session with Hive support enabled, allowing for SQL operations on Spark DataFrames. It then defines a list of tuples, data, containing sample information about individuals. This list is converted into a DataFrame using parallelize() and toDF(), and the DataFrame is registered as two temporary views, "Person" and "mydata", using createOrReplaceTempView(). These views allow SQL queries to be executed on the DataFrame. The show() method is used to display the contents of the DataFrame. This demonstrates working with temporary views and DataFrames in PySpark.

spark.sql("select \* from person").show()

spark.sql("select \* from mydata").show()

A screenshot of a computer

Description automatically generated

**Explanation:** The code executes SQL queries on two temporary views, `"Person"` and `"mydata"`, created earlier from the same DataFrame. Both queries select all columns (`\*`) from the respective views. The `show()` method displays the results of these queries, showing the data in tabular form. Since both views contain the same data, the results of the queries are identical. This demonstrates how to run SQL queries on temporary views in PySpark.