Harshil D Patel (0501)

```
! pip install wget
```

```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9655 sha256=09e46706a5fcc73c861f139e8f9e181f18e0bb882f6e0b4f77a77669e7
  Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505ca1c81cd1d9208ae2064675d97582078e6c769
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

```
!pip install pyspark
```

```
Collecting pyspark
  Downloading pyspark-3.5.0.tar.gz (316.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 316.9/316.9 MB 3.9 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspark) (0.10.9.7)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.5.0-py2.py3-none-any.whl size=317425345 sha256=dbf06a0d9b424cfca463acb941df67899bc68b49d
  Stored in directory: /root/.cache/pip/wheels/41/4e/10/c2cf2467f71c678cfc8a6b9ac9241e5e44a01940da8fbb17fc
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.0
```

Resilient Distributed Dataset (RDD) is the fundamental building block of spark. It is an immutable distributed collection of objects. All the Data frame or Dataset are stored as RDD under the hood.

A dataset in Spark is called as RDD because:

Resilient: In case of any node in the cluster goes down, Spark can recover the parts of the RDDs from the input and pick up from where it left off because data is available in memory of other nodes. RDD can be reconstructed from the data available in memory of other node, that is why it is called as resilient.

Distributed: Each RDD is broken into multiple pieces called partitions, and these partitions are divided across the clusters. This partitioning process is done automatically by Spark, so you don’t need to worry about all the details about how your data is partitioned across the cluster.

Immutable: They cannot be changed after they are created. Immutability rules out a significant set of potential problems due to updates from multiple threads at once. RDD are immutable, similar to data frames that are built on top of RDD. You cannot change a RDD, but you can apply transformation and create a new RDD.

Creating a RDD

Load RDDs from external storage: This approach is recommended for creating RDD from large datasets. This is done by executing textFile() function on SparkContext. The external storage is usually a distributed file system from different HDFS and data sources. The user can upload any csv file and read the csv file by specifying the path in the function textFile() for spark context object. In the following example, the big data is fetched directly from online sources using wget library.

```
#Downloading the dataset using wget
import wget
wget.download("https://github.com/futurexskill/bigdata/raw/master/retailstore_large.zip","retail.zip")

#Extracting the contents from the zip file in retail directory
from zipfile import ZipFile
with ZipFile('retail.zip','r') as file:
     file.extractall('retaildir')

#Creating a spark object
from pyspark import SparkConf, SparkContext
if __name__ == "__main__":
    conf=SparkConf().setAppName("first").setMaster("local") #working on a local envt, not on a cluster
    sc=SparkContext(conf=conf) #session name sc

#Creating a RDD from csv file
RetailRDD=sc.textFile("retaildir/retailstore_large.csv")

#Displaying the type
print("Type:",type(RetailRDD))
```

```
Type: <class 'pyspark.rdd.RDD'>
```

Note: If set master has the value as local[2], two processes will be executed for doing the computation. Both the processes first split the data and then reduce by join addition. The final result is the total of both the processes. However, it should be noted that if it is executed on the cluster, two machines will be deployed.

Actions for RDD

Two operations can be executed on RDD: Actions and Transformations. Actions are operations that return a result to the driver program or write it to storage, and kick off a computation. Transformations return RDDs, whereas actions return some other data type. Some of the most common actions include reduce(), collect(), first(), take(), count(), countbyvalue(), top(), saveAsTextFile() etc. It should be noted that actions will also force the evaluation of the transformations required for the RDD they were called on. Even though new RDDS can be defined any time, they are only computed by Spark in a lazy fashion.Transformations also return the output in the form of RDD and are lazily evaluated, meaning that Spark will not begin to execute until it sees an action

```
#Count action helps to count number of rows in an RDD
print("Number of rows:",RetailRDD.count())

#First action displays the first row
print("Header of RDD:\n",RetailRDD.first())

#Take action displays the specified number of elements from an RDD
print("First 2 rows of RDD:\n",RetailRDD.take(2))

#Top Action for fetching rows on sorted basis of length of row
print("Sorted 4 rows of RDD:\n",RetailRDD.top(4))
```

```
Number of rows: 1048576
Header of RDD:
 CustomerID,Age,Salary,Gender,Country
First 2 rows of RDD:
 ['CustomerID,Age,Salary,Gender,Country', '1,18,20000,Male,Germany']
Sorted 4 rows of RDD:
 ['CustomerID,Age,Salary,Gender,Country', '999999,51.72727273,12600,Male,France', '999998,50.34545455,34000,Male,France', '999997,48.963
```

It is important to transform the dataset before applying advanced analysis. This can be done using traditional technologies also but, Spark is used for doing transformation of high volume of data and when it is required to do high level of processing. Transformations and actions are different because of the way how spark computes RDDs. Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.Transformations on RDD:Some of the most common transformations include map(), mapValues(), mapPartitions(), filter(), flatMap(), sample(), randomSplit(), distinct(), coalesce() and repartition() etc.

```
#Map Transformation accepts input parameter as a function. The function can be any user defined function or one line lambda function.
#Using lambda function for transformation
CountryRDD=RetailRDD.map(lambda x: x.split(",")[4])
print(CountryRDD.take(2))

#Using Count by Value Action on a RDD
print("Using Count by value :",)
result=CountryRDD.countByValue() #output is a dictionary
print(result.items())
for word,count in result.items():
    print(word, " occurred ", count, " times")

#saveAsTextFile is used to write on distributed storage system (HDFS or local system)
CountryRDD.saveAsTextFile("newRetail4.csv")
print("File saved")
```

```
['Country', 'Germany']
Using Count by value :
dict_items([('Country', 1), ('Germany', 278528), ('France', 294911), ('England', 475136)])
Country  occurred  1  times
Germany  occurred  278528  times
France  occurred  294911  times
England  occurred  475136  times
File saved
```

```
#Practical Exercise:  Determine the number of male and female customers
GenderRDD=RetailRDD.map(lambda x: x.split(",")[3])

#Using Count by Value Action on a RDD
print("Using Count by value :",)
result=GenderRDD.countByValue() #output is a dictionary
print(result.items())
for word,count in result.items():
    print(word, " occurred ", count, " times")

CountryRDD.saveAsTextFile("newRetail5_Gender.csv")
print("File saved")
```

```
Using Count by value :
dict_items([('Gender', 1), ('Male', 544195), ('Female', 504380)])
Gender  occurred  1  times
Male  occurred  544195  times
Female  occurred  504380  times
File saved
```

```
#Map transformation with user defined function on RDD created from csv
#Creating a user defined function
def retail_func(myrecord):
    id=myrecord.split(",")[0]
    salary=myrecord.split(",")[2]
    country=myrecord.split(",")[4]
    return "Salary of "+ id +" is "+ salary+ " and belongs to "+country

#Map transformation with user defined function
mapRDD=RetailRDD.map(retail_func)
print(mapRDD.take(3))
```

```
['Salary of CustomerID is Salary and belongs to Country', 'Salary of 1 is 20000 and belongs to Germany', 'Salary of 2 is 22000 and belon
```

```
#Filter transformation for filtering rows
#Using filter for filtering the header
header=RetailRDD.first()
FinalRDD=RetailRDD.filter(lambda x: x!=header)
print("First few rows:\n",FinalRDD.take(2))

#Using 'in' for filtering categorical variable
#Observe the difference between two results.
#Approach 1: When the string is directly searched
FranceRDD=FinalRDD.filter(lambda x:'France' in x)
print("Number of rows containing France:",FranceRDD.count())
print("First few rows:\n",FranceRDD.take(2))


#Approach 2: When the string is directly searched from the filtered RDD
CountryRDD=FinalRDD.map(lambda x: x.split(",")[4])
GermanyRDD=CountryRDD.filter(lambda x:'Germany' in x)
print("Number of rows containing Germany:",GermanyRDD.count())
print("First few rows:\n",GermanyRDD.take(2))


#Using filter for numeric variable
SalaryRDD=FinalRDD.map(lambda x: x.split(",")[2])
ResultRDD=SalaryRDD.filter(lambda x:int(x)>60000)
print("Number of rows where salary > 60000:",ResultRDD.count())
print("First few rows:\n",ResultRDD.take(2))

#Using 'and' Logical operators for filtering multiple conditions
LogicalRDD1=SalaryRDD.filter(lambda x:(int(x)>40000) & (int(x)< 50000))
print("Number of rows between 40K and 50K:",LogicalRDD1.count())
print("First few rows:\n",LogicalRDD1.take(2))

#Using 'or' Logical operators for filtering multiple conditions
LogicalRDD2=SalaryRDD.filter(lambda x:(int(x)>50000)|(int(x) < 20000))
print("Number of rows with high salary:",LogicalRDD2.count())
print("First few rows:\n",LogicalRDD2.take(2))
```

```
First few rows:
 ['1,18,20000,Male,Germany', '2,19,22000,Female,France']
Number of rows containing France: 294911
First few rows:
 ['2,19,22000,Female,France', '5,22,50000,Male,France']
Number of rows containing Germany: 278528
First few rows:
 ['Germany', 'Germany']
Number of rows where salary > 60000: 41942
First few rows:
 ['65000', '65000']
Number of rows between 40K and 50K: 209711
First few rows:
 ['42000', '45000']
Number of rows with high salary: 377485
First few rows:
 ['2600', '4300']
```

```
#Using multiple filters for multiple categorical, continuous variables
MaleRDD=FinalRDD.filter(lambda x: 'Male' in x)
print("Number of male customers:", MaleRDD.count())
print("First few rows:\n",MaleRDD.take(2))

MaleEngRDD=MaleRDD.filter(lambda x: 'England' in x)
print("Number of male customers from England:",MaleEngRDD.count())
print("First few rows:\n",MaleEngRDD.take(2))

MaleEngSalRDD=MaleEngRDD.map(lambda x: x.split(",")[2])
MaleEngSalHighRDD=MaleEngSalRDD.filter(lambda x:int(x)>50000)
print("Number of male customers from England having salary>50000:", MaleEngSalHighRDD.count())
print("First few rows:\n",MaleEngSalHighRDD.take(2))
```

```
Number of male customers: 544195
First few rows:
 ['1,18,20000,Male,Germany', '4,21,2600,Male,England']
Number of male customers from England: 246591
First few rows:
 ['4,21,2600,Male,England', '14,35.14545455,7600,Male,England']
Number of male customers from England having salary>50000: 29594
```

```
First few rows:
 ['65000', '65000']
```

```
#Using and operation for searching two strings together
MaleEngRDD2=FinalRDD.filter(lambda x: ('Male' in x) & ('England' in x))
print("Number of male customers:", MaleEngRDD2.count())
print("First few rows:\n",MaleEngRDD2.take(2))
```

```
Number of male customers: 246591
First few rows:
 ['4,21,2600,Male,England', '14,35.14545455,7600,Male,England']
```

```
#Exercise: Determine the number of female customers from France where age is greater than 40
FemaleRDD=FinalRDD.filter(lambda x: 'Female' in x)

FemaleFranceRDD=FemaleRDD.filter(lambda x: 'France' in x)

FemaleFranceAgeRDD=FemaleFranceRDD.map(lambda x: x.split(",")[1])
FemaleFranceAge40RDD=FemaleFranceAgeRDD.filter(lambda x:float(x)>40)
print("Number of female customers from France having age>40:", FemaleFranceAge40RDD.count())
```

```
Number of female customers from France having age>40: 141854
```

```
#Exercise: Search for multiple conditions: Display the information of
#country, age and salary of records where age > 30 and salary > 40000
#Create a user defined function

def retail_func(myrecord):

    age=myrecord.split(",")[1]
    salary=myrecord.split(",")[2]
    country=myrecord.split(",")[4]

    return float(age), int(salary), country

#Map transformation with user defined function

mapRDD=FinalRDD.map(retail_func)
print(mapRDD.take(3))
result=mapRDD.filter(lambda x:(x[0]>30) & (x[1] > 40000))
print(result.take(3))
print(result.count())
```

```
[(18.0, 20000, 'Germany'), (19.0, 22000, 'France'), (20.0, 24000, 'England')]
[(36.52727273, 55000, 'France'), (43.43636364, 42000, 'France'), (50.34545455, 60000, 'Germany')]
377479
```

```
#Determining Total salary using map transformation and reduce action
SalaryRDD=FinalRDD.map(lambda x: x.split(",")[2])
Total_Salary=SalaryRDD.reduce(lambda x,y: int(x)+int(y))
print("Total salary:",Total_Salary)
```

```
Total salary: 37106829800
```

```
#Exercise: Determine the maximum age of male customers from Germany
MaleRDD=FinalRDD.filter(lambda x: 'Male' in x)
print("Number of Male customers:", MaleRDD.count())

MaleGerRDD=MaleRDD.filter(lambda x: 'Germany' in x)
print("Number of Male customers from Germany:",MaleGerRDD.count())

AgeRDD=FinalRDD.map(lambda x: x.split(",")[1])
Max_Age=AgeRDD.reduce(lambda x,y: max(x,y))
print("Max Age:",Max_Age)
```

```
Number of Male customers: 544195
Number of Male customers from Germany: 144551
Max Age: 66.92727273
```

```
#Sample Transformation
SampleRDD=FinalRDD.sample(withReplacement=False,fraction=0.00001,seed=1)
print("Number of rows in sample:",SampleRDD.count())
```

```
Number of rows in sample: 9
```

```
#Flatmap transformation
FlatRDD=SampleRDD.flatMap(lambda x:x.split(","))
print("Using FlatMap transformation for counting:\n",FlatRDD.count())
print("Using FlatMap transformation:\n",FlatRDD.take(8))
```

```
Using FlatMap transformation for counting:
 45
Using FlatMap transformation:
 ['20964', '55.87272727', '7600', 'Male', 'England', '45816', '47.58181818', '40000']
```

```
#Random Split Transformation
RandomRDD=FinalRDD.randomSplit(weights=[0.2,0.4,0.3,0.1], seed=1000)
```