Use of Apache Spark RDD in Python

B17_IN 4410 - Big Data Analytics

Continuous Assignment 02

Group 05

Index Number	**Name**
174031R	De Silva K S M
174051D	Jayarathna P.A.T.L.
174146A	Samson H.
174005R	Ahamed M.B.T
174097X	Madurangi K.P.G.I

Introduction

'Data is the new oil' is one of the famous statements made in the 21st century however raw data is not valuable unless we process them and generate insights that help decision making. Apache Spark is an open source big data framework that helps us tackle the problem of processing big data by harnessing the power of distributed computing. In this tutorial, we look into use of Resilient Distributed Datasets (RDD) in Apache Spark.

RDD

RDD which stands for Resilient Distributed Dataset is the fundamental building block of PySpark. Each record in RDD is divided into logical partitions and can be computed on different nodes of the cluster. RDD is somewhat similar to lists in Python but unlike Python lists which is computed as a one process, RDD is computed in several processes dispersed across numerous physical servers.

RDDs are fault-tolerant. RDD can not be changed once created which is called being immutable. Data abstraction is another feature of RDD. Also while performing transformations in RDD, PySpark handles the parallelism by default making it convenient to the user.

Benefits of PySpark RDD

1. In-memory processing

In RDD, data is loaded from the disk and stored in memory. When performing transformations, it is possible to use previous computations. In-memory processing increases the execution speed and is considered as the main advantage of RDD against MapReduce.

2. Fault-tolerant

PySpark works on HDFS therefore a certain RDD operation fails, it reloads the data from auxiliary partitions. This allows applications to run seamlessly.

3. Lazy Evaluation

Data available in RDD is not executed until any action is carried out on them. In other words Sparks does not evaluate each transformation as they arrive, instead it queues them and only performs evaluation once an action is called upon them. It optimizes the performance.

4. Immutable and partitions

All the records are partitioned and each partition is logically divided and immutable. It ensures the consistency of data.

Limitations of PySpark RDD

1. Run-time safety

RDD does not allow the check the error at runtime. Also there is no static-typing

2. Memory issues

Storage issues can arise when there is not enough memory to store RDD in memory or even in disk. Therefore RDD is not suitable for applications such as storage systems in web applications.

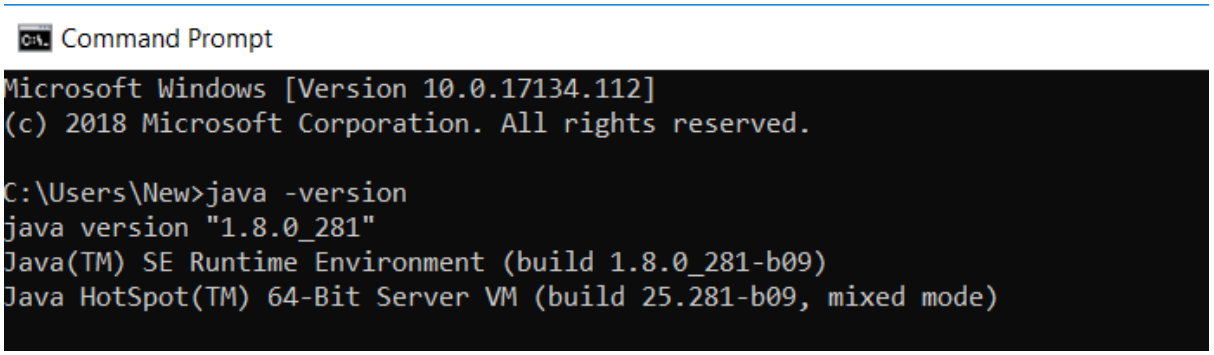
Apache Spark installation

Prerequisite

- Install Java 8

You can download Java 8 using the following link.
https://java.com/en/download/

If you have already installed Java, then check whether your Java version is 8. You can check the Java version by typing "**java -version**" in the command prompt.



- Install python

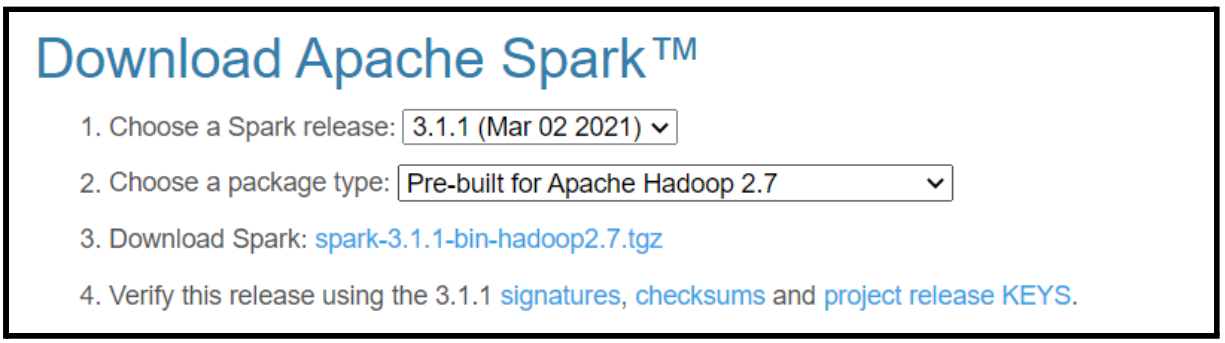
You can download the latest python version using the following link.
https://www.python.org/downloads/

- Install Apache Hadoop

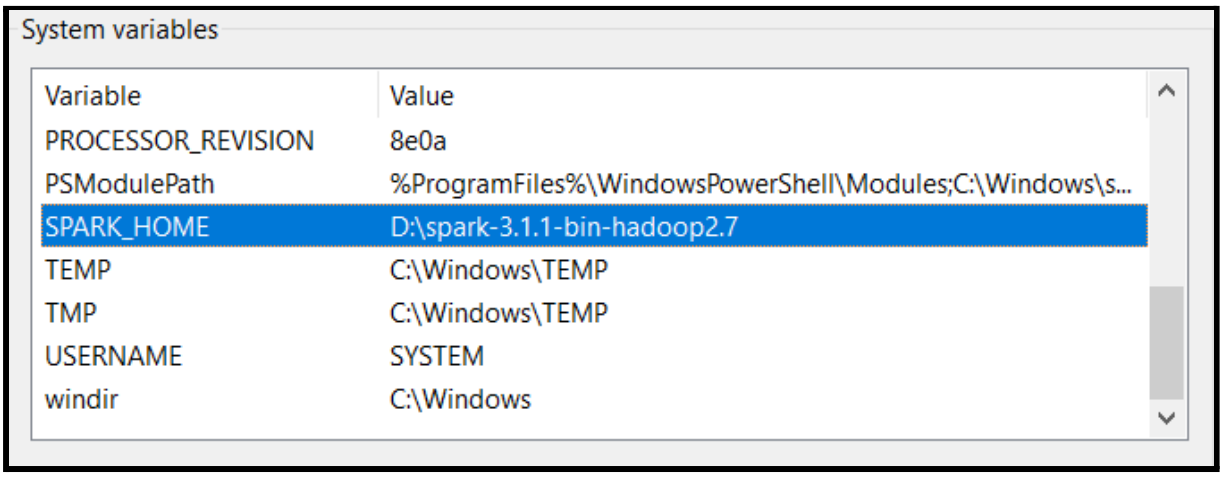
Use the following link to download Apache Hadoop.
https://hadoop.apache.org/releases.html

Steps to install Apache Spark

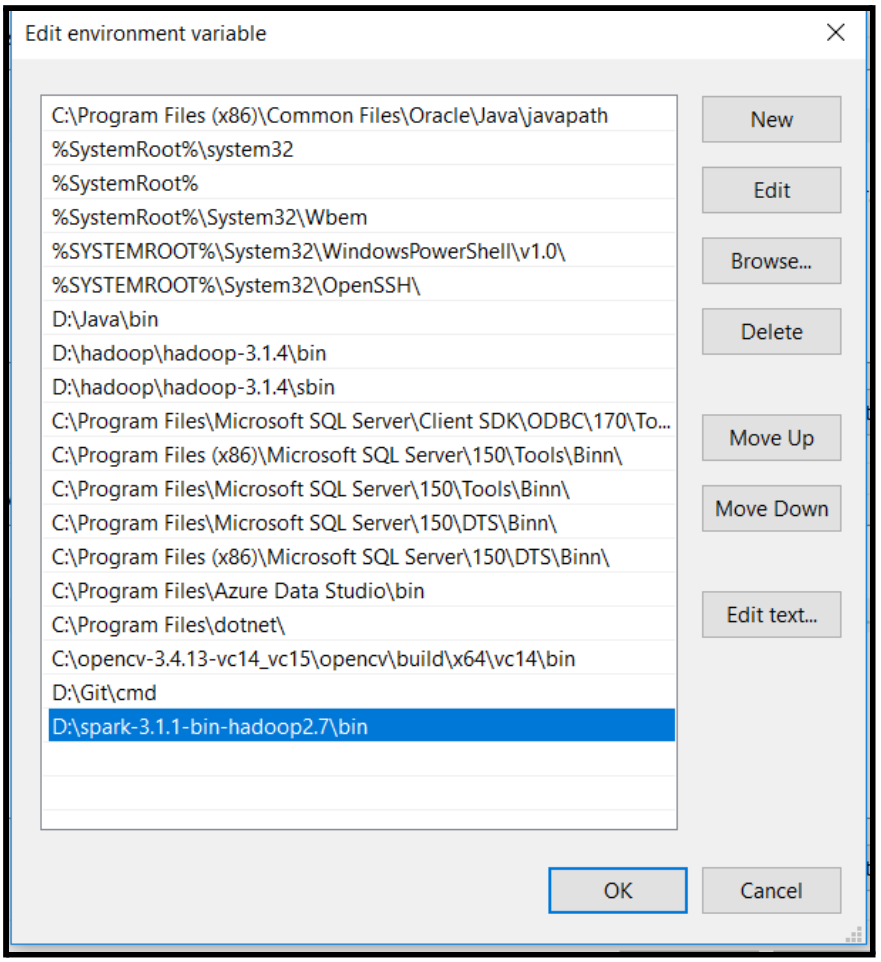
1. Download Apache spark from the following link. When downloading, you should select the most suitable package type according to your Hadoop version.
   https://spark.apache.org/downloads.html



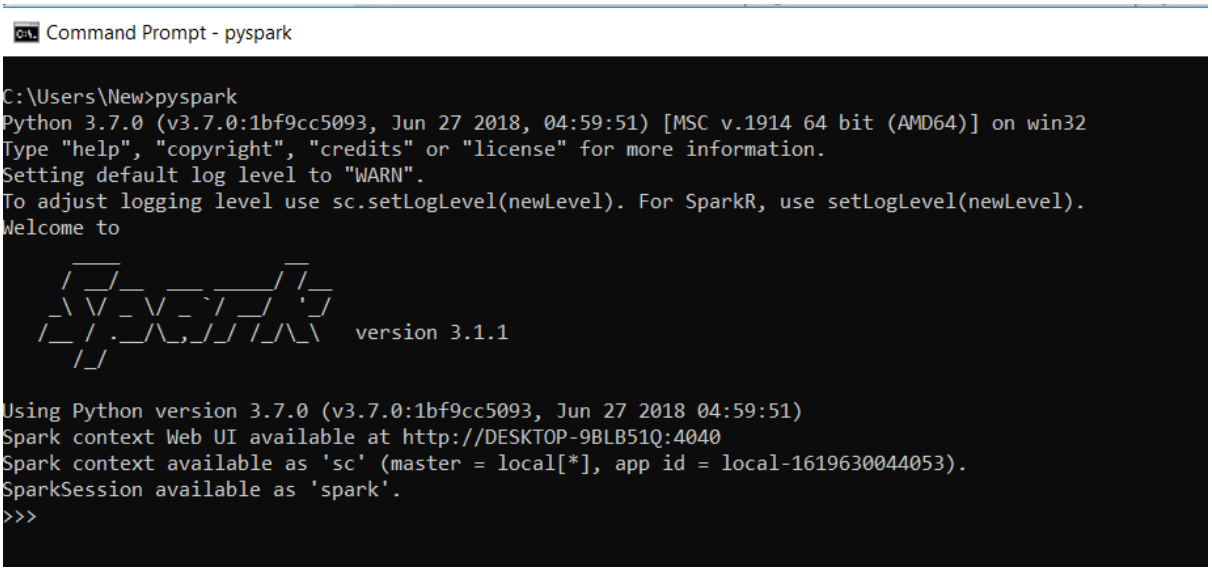
2. Extract the downloaded .tgz file to the C: drive..
3. Configure environment variables as follows.



Enter the path to the Spark bin folder as follows or you can use **%SPARK_HOME%bin** also.



4. Open a command prompt as administrator and run **pyspark** command to check whether the Apache Spark is installed successfully. If Spark is installed successfully you will get the below window.



Initialize spark

Spark context acts as the main entry point for Spark functionality and its' connection to the Spark cluster. Therefore, at the beginning of a Spark program, you have to create a 'sparkContext' object.

```
conf = SparkConf().setAppName("RDD operations").setMaster("local[*]")
sc = SparkContext(conf = conf)
```

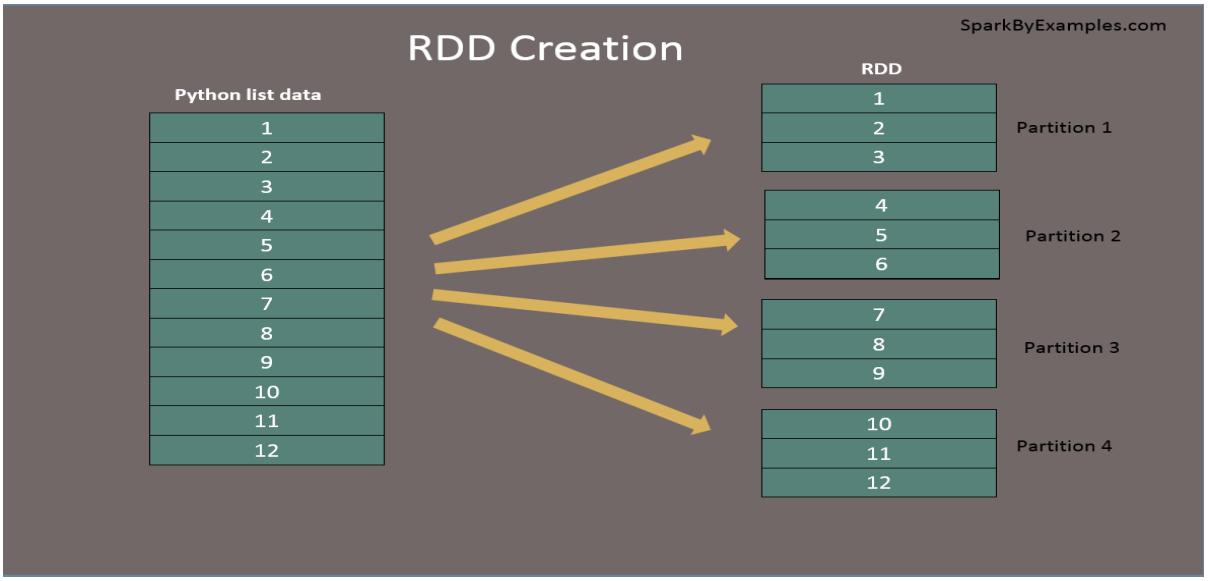
You can set app name using 'setAppName()' function and number of cores which is allocated to run the program through 'setMaster()' function. **local[*]** means we allocate all cores to run the Spark program.

Create RDDs

Spark provides two ways to create a RDD.

1. **Parallelizing an existing collection of Data**

In this method we create RDD from a list of collections of Data. **sparkContext** provides a function **parallelize()** to create RDD. This approach loads the existing data from the driver into parallelizing RDD. When you already have data in-memory, loaded from a file or from a database, this approach is used.



Each dataset in RDD is divided into logical partitions as in the above diagram. Divided data might be computed on different nodes. Another signature feather of parallelize() function is that it additionally takes integer argument to designate the number of partitions. Partitions can be defined as the basic units of parallelism in PySpark.

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
rdd = sc.parallelize(data)
rdd_collect = rdd.collect()
print("Number of Partitions: "+str(rdd.getNumPartitions()))
print("Action: First element: "+str(rdd.first()))
print(rdd_collect)
```

Output:-
N0. of Partitions: 4

Action: 1st element: 1

[1,2,3,4,5,6,7,8,9,10,11,12]

2. Referencing a dataset in an external storage system

Using Spark, distributed datasets can be created from any storage source which is supported by Hadoop, including your HDFS, HBase, Cassandra, Amazon S3 and even your local file system. etc. Spark supports SequenceFiles, text files and any other Hadoop InputFormat.

- **textFile()** method allows us to read a text (.txt) file into RDD.

```
distFile = sc.textFile("dataset.txt")
```

- **wholeTextFiles()** function returns a PairRDD. In this pair Key is the file path and Value is the file content

```
distFile = sc.wholeTextFiles("dataset.txt")
```

Creating empty RDD

- emptyRDD creates an empty RDD with no partition.

```
distFile = sc.emptyRDD
```

- parellelize also creates an empty RDD but with partition

```
distFile = sc.parallelize([],10)
```

Note: Data partitioning will be based on the resource availability. It would create number of partitions similar to the number of cores available on the system when it is run in a PC.

getNumPartitions() returns the number of partitions the dataset split into.

RDD operations

There are two types of operations that can be done with RDD.

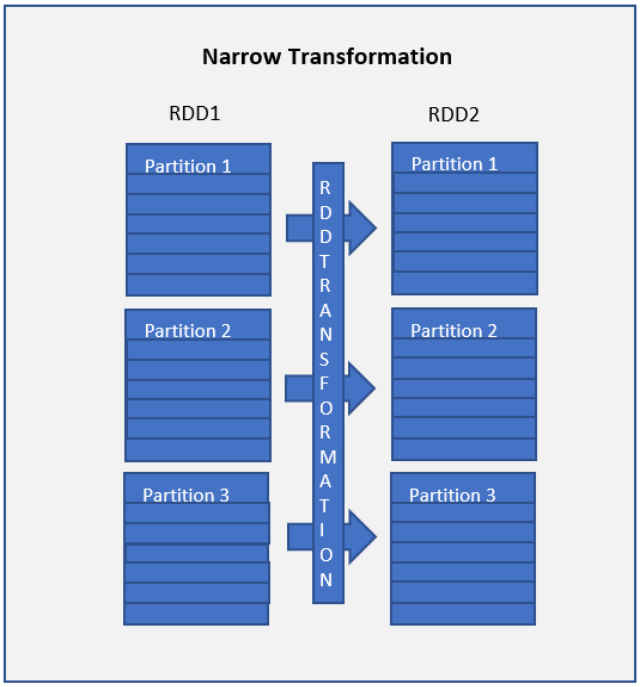
1. Transformation

PySpark RDD Transformations are lazy operations therefore used to transform or update from one RDD into another. When carried out, PySpark RDD transformations result in a single or multiple new RDD.

There are two types are transformations as,

1. Narrow Transformation

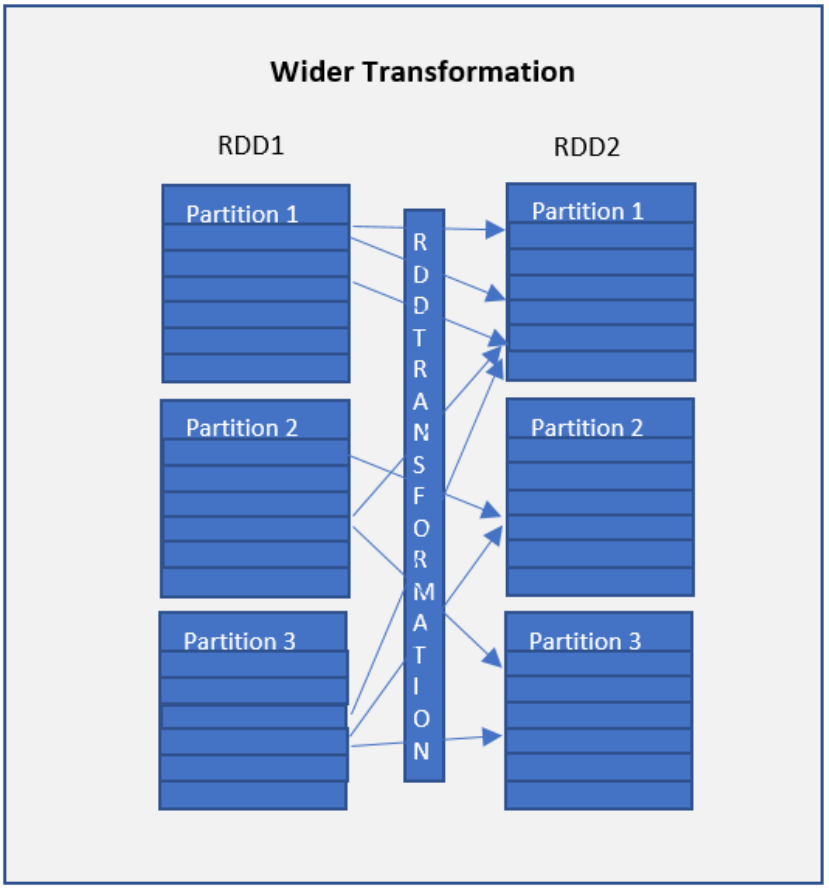
In this type of transformation there will not be any data movement between partitions.



map(), **flatMap**(), **mapPartition**(), **filter**(), **union**() are some narrow transformation functions.

2. Wider Transformation

In this type of transformations there will be data movements between partitions when a transformation occurs Wider transformation is also known shuffle transformations.



groupByKey(), **aggregate**(), **aggregateByKey**() , **join**(), **repartition**() are some wider transformation functions

Note: Wider transformations are expensive operations due to shuffling.

Let us assume there is a dataset given as follows (dataset.txt).

10	5	8	9	8
1	17	3	6	7
23	19	5	2	15
4	20	9	2	12
18	8	22	5	7

You have to return a new dataset which includes only numbers greater than 10. For that you have to use RDD transformation functions which are flatMap(func), map(func) and filter(func).

- ❖ flatMap(func)

When we want to output multiple elements from each input element we should use flatMap(func).

According to the given dataset you have to output all numbers of each line by splitting from tab spaces and you can use the below code line to do it.

```
numbers = distFile.flatMap(lambda line: line.split("\t"))
```

It will return a new dataset of all separated string typed numbers as a sequence.

Output:-
['10', '5', '8', '9', '8', '1', '17', '3', '6', '7', '23', '19', '5', '2', '15', '4', '20', '9', '2', '12', '18', '8', '22', '5', '7']

- ❖ map(func)

map(func) is used to return a new dataset formed by passing each element of the given dataset through a function.

Because the returned dataset is including string typed numbers you have to convert each element to integer. Therefore, using map(func) you should pass each element to int() function.

```
validNumbers = numbers.map(lambda number: int(number))
```

It will return a new dataset of all separated integers from a given dataset.

Output:-
[10, 5, 8, 9, 8, 1, 17, 3, 6, 7, 23, 19, 5, 2, 15, 4, 20, 9, 2, 12, 18, 8, 22, 5, 7]

- ❖ filter(func)

Filter(func) is used to return a new dataset which is a subset of the given dataset. As mentioned above, you have to filter all numbers which are greater than 10. Therefore, you can use the below code line to do it.

```
filterNumber = validNumbers.filter(lambda number: number > 10)
```

It will return a new dataset of all integers which are greater than 10, from the given dataset.
Output:-
[17, 23, 19, 15, 20, 12, 18, 22]

Let us take another dataset as follows (Dataset2.txt).

. 10 20 30

5 10 15

15 30 45

Now we are going to do few operations between two datasets (dataset.txt and Dataset2.txt)

```
distFile = sc.textFile("dataset.txt")
numbers = distFile.flatMap(lambda line: line.split("\t"))

distFile2 = sc.textFile("dataset2.txt")
numbers2 = distFile2.flatMap(lambda line: line.split("\t"))
```

- ❖ intersection(dataset2)

intersection(dataset2) function returns a new dataset which includes common elements of two datasets.

You can take the intersection of two datasets using the following code line.

```
intersections = numbers.intersection(numbers2)
```

Output:-
['10', '20', '15', '5']

- union(dataset2)

To return a new dataset which includes all elements of given two elements, union(dataset2) function can be used.

You can use the following code line to take the union of two datasets.

```
unions = numbers.union(numbers2)
```

Output:-
['10', '5', '8', '9', '8', '1', '17', '3', '6', '7', '23', '19', '5', '2', '15', '4', '20', '9', '2', '12', '18', '8', '22', '5', '7', '10', '20', '30', '5', '10', '15', '15', '30', '45']

- distinct()

distinct() function is used to return a new dataset which has only distinct elements from a given dataset.

Let us take a new distinct dataset from the previously created 'unions' dataset.

```
distincts = unions.distinct()
```

Output:-
['10', '4', '20', '12', '3', '6', '7', '23', '15', '18', '30', '8', '9', '1', '17', '19', '22', '45', '5', '2']

Even though Spark operations are capable of working on RDDs consisting of any type of objects, there are few transformation operations in Spark RDD which consider key-value pairs.

To do those types of operations let us create a new dataset which contains (K,V) pairs from dataset.txt. We take each element dataset.txt as keys and assign 1 as value for each key.

```
distFile = sc.textFile("dataset.txt")
numbers = distFile.flatMap(lambda line: line.split("\t"))
validNumbers = numbers.map(lambda number: int(number))
pairs = validNumbers.map(lambda s: (s, 1))
```

[(10, 1), (5, 1), (8, 1), (9, 1), (8, 1), (1, 1), (17, 1), (3, 1), (6, 1), (7, 1), (23, 1), (19, 1), (5, 1), (2, 1), (15, 1), (4, 1), (20, 1), (9, 1), (2, 1), (12, 1), (18, 1), (8, 1), (22, 1), (5, 1), (7, 1)]

- groupByKey()

groupByKey() is used to return a dataset of (K,iterable<V>) pairs from a dataset of (K,V) pairs.

Using the below code line you can return a new dataset of (K,iterable<V>) pairs grouping by keys. mapValues(list) is used to convert iterable<V> as a list to visible elements of iterable<V>.

```
groups = pairs.groupByKey().mapValues(list)
```

Output:-
[(10, [1]), (8, [1, 1, 1]), (6, [1]), (2, [1, 1]), (4, [1]), (20, [1]), (12, [1]), (18, [1]), (22, [1]), (5, [1, 1, 1]), (9, [1, 1]), (1, [1]), (17, [1]), (3, [1]),(7, [1, 1]), (23, [1]), (19, [1]), (15, [1])]

- ❖ reduceByKey(func)

ReduceByKey(func) returns a dataset of (K,U) pairs from a dataset of (K,V) pairs. In this U is the aggregation of Vs which are related to the same keys.

```
counts = pairs.reduceByKey(lambda a, b: a + b)
```

Output:-
[(10, 1), (8, 3), (6, 1), (2, 2), (4, 1), (20, 1), (12, 1), (18, 1), (22, 1), (5, 3), (9, 2), (1, 1), (17, 1), (3, 1), (7, 2), (23, 1), (19, 1), (15, 1)]

- ❖ sortByKey([ascending])

sortByKey() returns a sorted dataset of (K,V) pairs from a dataset of (K,V) pairs. The returned dataset is sorted by keys in descending or ascending order.

```
sorts = counts.sortByKey()
```

Output:-
[(1, 1), (2, 2), (3, 1), (4, 1), (5, 3), (6, 1), (7, 2), (8, 3), (9, 2), (10, 1), (12, 1), (15, 1), (17, 1), (18, 1), (19, 1), (20, 1), (22, 1), (23, 1)]

Note : All examples explain in colab.

2. Actions

Actions compute a result based on a RDD such as the first element of a dataset, count of elements in a dataset etc. Any RDD Operations that return values other than RDD are actions. let's use the same dataset used in Transformation here.

- ❖ collect()

Returns the complete dataset as an array. However, it is important to keep in mind that when you are dealing with large datasets with millions of data points it is not advisable to use this action because you might run out of memory when executing this action.

```
print(distFile.collect())
```

Output:-
['10\t5\t8\t9\t8', '1\t17\t3\t6\t7', '23\t19\t5\t2\t15', '4\t20\t9\t2\t12', '18\t8\t22\t5\t7']

```
for element in distFile.collect():
  print(element)
```

Output:-

10	5	8	9	8
1	17	3	6	7
23	19	5	2	15
4	20	9	2	12
18	8	22	5	7

To do the following actions let's use **validNumbers** RDD and **pairs** RDD created in the Transaction Section.

```
print(validNumbers.collect())
```

Output:-
[10, 5, 8, 96, 8, 1, 17, 3, 6, 7, 23, 19, 5, 2, 15, 4, 20, 9, 2, 12, 18, 8, 22, 5, 7]

```
print(pairs.collect())
```

Output:-
[(10, 1), (5, 1), (8, 1), (9, 1), (8, 1), (1, 1), (17, 1), (3, 1), (6, 1), (7, 1), (23, 1), (19, 1), (5, 1), (2, 1), (15, 1), (4, 1), (20, 1), (9, 1), (2, 1), (12, 1), (18, 1), (8, 1), (22, 1), (5, 1), (7, 1)]

- ❖ count()

Returns the number of records in the RDD as the name suggests.

```
print("Count is "+str(validNumbers.count()))
print("Count is "+str(pairs.count()))
```

Output:-
Count is 25
Count is 25

- ❖ countApprox()

This method returns an approximate count of elements in the datasets and returns incomplete when the execution time meets timeout.

```
print("countApprox is "+str(validNumbers.countApprox(1200)))
print("countApprox is "+str(pairs.countApprox(1200)))
```

Output:-
countApprox is 25
countApprox is 25

- ❖ countApproxDistint()

This method also returns an approximate count of elements but takes only the distinct elements into the consideration. If the same element exists twice or more in the dataset it will only amount to one when counting.

```
print("countApproxDistinct is"+str(validNumbers.countApproxDistinct()))
```

```
print("countApproxDistinct is "+str(pairs.countApproxDistinct()))
```

Output:-
countApproxDistinct is 17
countApproxDistinct is 18

- ❖ countByValue()

Returns the count of each distinct value in the RDD as a dictionary of pairs. (value, count)

```
print("countByValue :  "+str(validNumbers.countByValue()))
print("countByValue :  "+str(pairs.countByValue()))
```

Output:-
countByValue : defaultdict(<class 'int'>, {10: 1, 5: 3, 8: 3, 9: 2, 1: 1, 17: 1, 3: 1, 6: 1, 7: 2, 23: 1, 19: 1, 2: 2, 15: 1, 4: 1, 20: 1, 12: 1, 18: 1, 22: 1})
countByValue : defaultdict(<class 'int'>, {(10, 1): 1, (5, 1): 3, (8, 1): 3, (9, 1): 2, (1, 1): 1, (17, 1): 1, (3, 1): 1, (6, 1): 1, (7, 1): 2, (23, 1): 1, (19, 1): 1, (2, 1): 2, (15, 1): 1, (4, 1): 1, (20, 1): 1, (12, 1): 1, (18, 1): 1, (22, 1): 1})

- ❖ first()

first() returns the very first element of the dataset.

```
print("first element :  "+str(validNumbers.first()))
print("first element :  "+str(pairs.first()))
```

Output:-
first element : 10
first element : (10, 1)

- ❖ min()

As the action name suggests, this returns the minimum element in the dataset.

```
print("min is  "+str(validNumbers.min()))
print("min is  "+str(pairs.min()))
```

Output:-
min is 1
min is (1, 1)

- ❖ max()

Returns the maximum element in the dataset.

```
print("max is  "+str(validNumbers.max()))
print("max is  "+str(pairs.max()))
```

Output:-
max is 23
max is (23, 1)

- top()

top(n) returns the top n elements by taking elements in their descending order.

```
print("top : "+str(validNumbers.top(3)))
print("top : "+str(pairs.top(3)))
```

Output:-
top : [23, 22, 20]
top : [(23, 1), (22, 1), (20, 1)]

- takeOrdered()

Similar to top() action, takeOrdered(n) returns top n elements but taking the elements in their ascending order.

It is not advisable to use top() and takeOrdered() action of large datasets because all the records are loaded into the memory of the driver when performing these two actions.

```
print("takeOrdered : "+ str(validNumbers.takeOrdered(2)))
print("takeOrdered : "+ str(pairs.takeOrdered(2)))
```

Output:-
takeOrdered : [1, 2]
takeOrdered : [(1, 1), (2, 1)]

- reduce()

Reduces the elements of the dataset through a binary operator. This can be used to calculate sum and count.

```
from operator import add
redRes=validNumbers.reduce(add)
print(redRes)
```

Output:- 245

- treeReduce()

Reduces the elements of the dataset through a multilevel tree pattern.

```
add = lambda x, y: x + y
redRes=validNumbers.treeReduce(add)
print(redRes)
```

Output:- 245

- ❖ aggregate ()

aggregate() aggregates the elements of each partition and then the results for all the partitions as the name suggests. This is carried out using a given combine function out of "combOp" and a neutral "zero value."

```
seqOp = (lambda x, y: x + y)
combOp = (lambda x, y: x + y)
agg=validNumbers.aggregate(0, seqOp, combOp)
print(agg)
```

Output:- 245

Note : All examples explain in colab.

RDD Persistence

When we call actions on the same RDD multiple times, RDDs and all of its dependencies are recomputed each time an action is called on the RDD. The results can be delayed as actions are called on the same dataset many times.

Spark provides a solution to overcome the above issue called as RDD persistence. If you want to reuse an RDD in multiple actions, you can persist an RDD by keeping it in memory using persist() method.

Let us check the below code to get an idea about RDD persistence.

```
from pyspark import SparkContext, SparkConf, StorageLevel

if __name__ == "__main__":
    conf = SparkConf().setAppName("persist").setMaster("local[*]")
    sc = SparkContext(conf = conf)

    inputIntegers = [1, 2, 3, 4, 5]
    integerRdd = sc.parallelize(inputIntegers)

    integerRdd.persist(StorageLevel.MEMORY_ONLY)

    integerRdd.reduce(lambda x, y: x*y)
    integerRdd.count()
```

According to the above example, the RDD named as 'integerRdd' persists in the memory when the first action(reduce) is called on the RDD. Then, at the movement that the next action(count) is called on the RDD, it does not need to parallelize again because the previously created RDD persists in the memory.

You can use **integerRdd.cache()** instead of **integerRdd.persist(StorageLevel.MEMORY_ONLY)**.

There are different storage levels in Spark RDD.

Storage Level	Meaning
MEMORY_ONLY (default level)	Store RDD as deserialized Java objects in the JVM. If there are some partitions that do not fit in the memory then they will not be cached. If those missing partitions are needed, each time they will be recomputed.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. In this,All the partitions which are not fitted in the memory will be stored on the disk. When they are needed, read them from the disk.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). When compared with deserialized objects, this is more space-efficient but needs more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but store partitions on the disk which are not fitted in the memory.
DISK_ONLY	Only disk is used to store RDD partitions
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	The difference of these is they replicate each partition on two cluster nodes.

You should go through the below points to select the most suitable storage level.

- Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.
- If the RDDs can fit comfortably with the default storage level, MEMORY_ONLY is the ideal option. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using MEMORY_ONLY_SER to make the objects much more space-efficient, but still reasonably fast to access.
- Do not save to disk unless the functions that computed your datasets are expensive, or they filter a significant amount of the data.

Exercise

Question:-

You have been given a dataset of air quality levels in India as follows.

With the knowledge of Spark RDD, you should write a code to,

1. Find the city which has the maximum average NH3 pollutant value.
2. Find the number of results(rows) for each pollutant type.
3. Find the top ten resulting rows which have maximum pollutant level in CO and store the result in a text file.

	A	B	C	D	E	F	G	H
1	Country	State	city	lastupdate	Avg	Max	Min	Pollutants
2	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	70	108	42	PM2.5
3	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	76	102	43	PM10
4	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	73	118	46	NO2
5	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	5	6	4	NH3
6	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	41	109	2	SO2
7	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	44	102	18	CO
8	India	Andhra_Pradesh	Amaravati	21-12-2018 03:00:00	29	35	12	OZONE
9	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	NA	NA	NA	PM2.5
10	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	NA	NA	NA	PM10
11	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	NA	NA	NA	NO2
12	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	NA	NA	NA	NH3
13	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	NA	NA	NA	SO2
14	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	30	103	2	CO
15	India	Andhra_Pradesh	Rajamahendravaram	21-12-2018 03:00:00	108	130	51	OZONE
16	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	46	72	28	PM2.5
17	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	64	83	40	PM10
18	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	61	89	36	NO2
19	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	2	3	2	NH3
20	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	10	12	6	SO2
21	India	Andhra_Pradesh	Tirupati	21-12-2018 03:00:00	16	25	6	CO

Answers:-

```
from pyspark import SparkContext, SparkConf

if __name__ == "__main__":
    conf = SparkConf().setAppName("Transformation").setMaster("local[*]")
    sc = SparkContext(conf = conf)

    distFile = sc.textFile("Air_Quality_dataset.csv")

    lines = distFile.flatMap(lambda line: line.split("\t"))
    row_data = lines.map(lambda row: row.split(","))

    #Find the city which has maximum average NH3 pollutant value
    row_NH3 = row_data.filter(lambda row: row[7] == "NH3")
    real_row_NH3 = row_NH3.filter(lambda row: row[4] != "NA")
    pair = real_row_NH3.map(lambda row: (int(row[4]), row))
    sorted_pairs = pair.sortByKey(False)
```

```
print("The city which has maximum average NH3 pollutant value is " + sorted_pairs.first()[1][2])

#Find the number of results for each pollutant type
pair = row_data.map(lambda row: (row[7], row))
row_count = pair.countByKey()

for pollutant_type,count in row_count.items():
    print("Number of results for " + pollutant_type + " : " +str(count))

#Find the top 10 resulting rows which have maximum pollutant level in CO and store the result in a text file.
row_CO = row_data.filter(lambda row: row[7] == "CO")
real_row_CO = row_CO.filter(lambda row: row[5] != "NA")
pair = real_row_CO.map(lambda row: (int(row[5]), row))
sorted_pairs = pair.sortByKey(False)
top_ten = sorted_pairs.take(10)

top_ten_row = []

for key,value in top_ten:
    top_ten_row.append(value)

sc.parallelize(top_ten_row).saveAsTextFile("top_ten_max_CO")
```

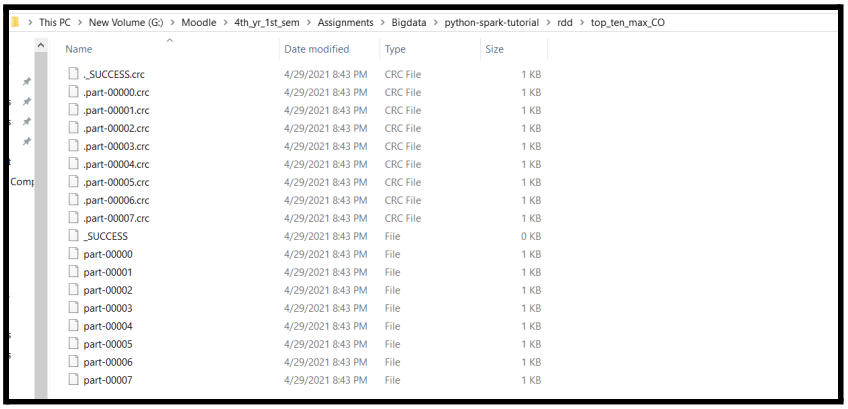
Output 1:-

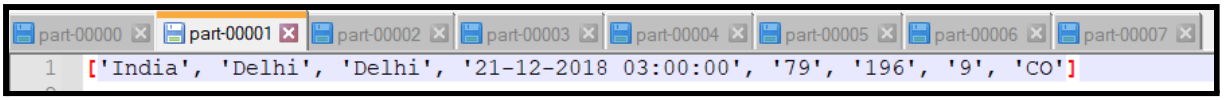
The city which has maximum average NH3 pollutant value is Nagpur

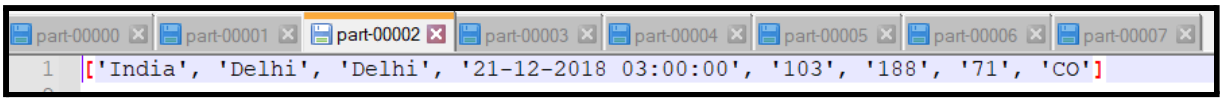
Output 2:-

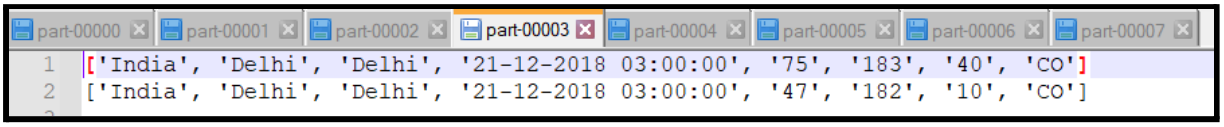
Number of results for PM2.5 : 127
Number of results for PM10 : 108
Number of results for NO2 : 129
Number of results for NH3 : 92
Number of results for SO2 : 121
Number of results for CO : 128
Number of results for OZONE : 119

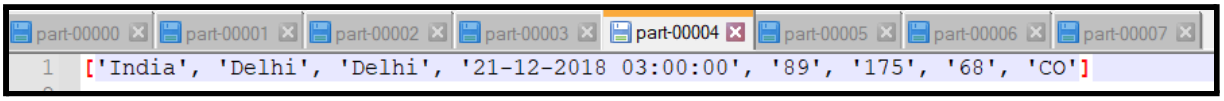
Output 3:-

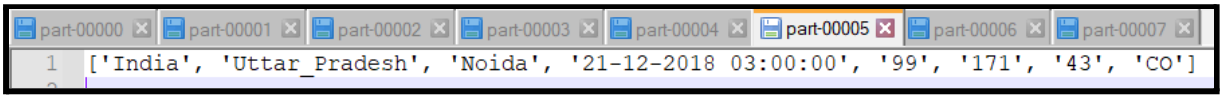


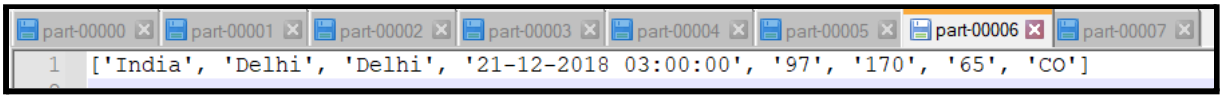


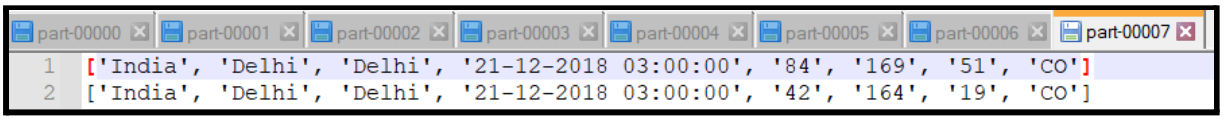












Note : Exercise answers in colab.

References

1. https://spark.apache.org/docs/latest/rdd-programming-guide.html
2. https://sparkbyexamples.com/pyspark-rdd/#rdd-actions
3. https://jaafarbenabderrazak-info.medium.com/introduction-to-apache-spark-rdds-using-python-1ce25452d59b
4. http://spark.apache.org/examples.html
5. https://cloudxlab.com