NAME: Meith Navlakha
SAP: 60004190068
BDI Practicals

EXPERIMENT 7

AIM: To create RDD, perform various operations and find occurrence of each word.

THEORY:

Apache Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

Features of Apache Spark

Apache Spark has following features.

- Speed − Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.
- Supports multiple languages − Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
- Advanced Analytics − Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

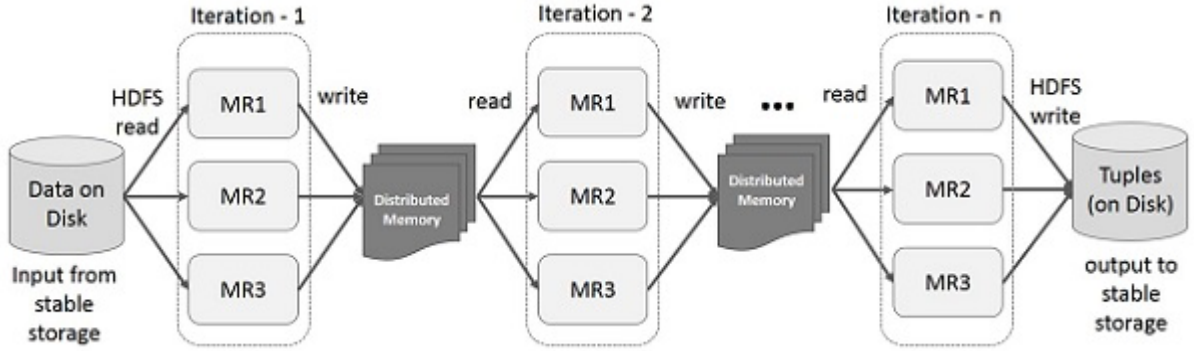
Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

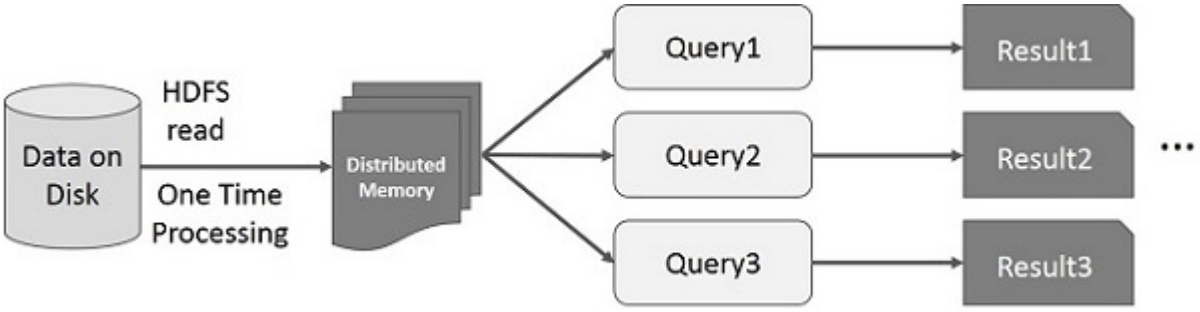
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

IMPLEMENTATION:

1) Initialize Spark

```
[ ] !apt-get install openjdk-8-jdk-headless -qq > /dev/null
    !wget -q https://archive.apache.org/dist/spark/spark-3.0.0/spark-3.0.0-bin-hadoop3.2.tgz
    !tar xf spark-3.0.0-bin-hadoop3.2.tgz
    !pip install -q findspark
```

```
[ ] import os

    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["SPARK_HOME"] = "/content/spark-3.0.0-bin-hadoop3.2"
```

```
[ ] import findspark
    findspark.init()
```

```
[ ] from pyspark import SparkContext , SparkConf
    from pyspark.sql import SparkSession

    conf = SparkConf().set('spark.ui.port', '4050')
```

```
    sc = SparkContext(conf=conf)
    spark = SparkSession.builder.master("local[*]").getOrCreate()
```

2) Dataset

```
[ ] import pandas as pd
    import numpy as np

    df = pd.read_csv('wineQuality.csv')
```

```
[ ] df.describe()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	
	6463.000000	6463.000000	6463.000000	6463.000000	6463.000000	6463.000000	6463.000000	6463.000000	6
	7.217755	0.339589	0.318758	5.443958	0.056056	30.516865	115.694492	0.994698	
	1.297913	0.164639	0.145252	4.756852	0.035076	17.758815	56.526736	0.003001	
	3.800000	0.080000	0.000000	0.600000	0.009000	1.000000	6.000000	0.987110	
	6.400000	0.230000	0.250000	1.800000	0.038000	17.000000	77.000000	0.992330	
	7.000000	0.290000	0.310000	3.000000	0.047000	29.000000	118.000000	0.994890	
	7.700000	0.400000	0.390000	8.100000	0.065000	41.000000	156.000000	0.997000	
	15.900000	1.580000	1.660000	65.800000	0.611000	289.000000	440.000000	1.038980	

NAME: Meith Navlakha
SAP: 60004190068
BDI Practicals

3) Create RDD

1) Using parallelize

```
[ ] RDD1 = sc.parallelize(df)
    rddCollect = RDD1.collect()
    print("Number of Partitions: "+str(RDD1.getNumPartitions()))
    print("Action: First element: "+str(RDD1.first()))
    print(rddCollect)
```

```
Number of Partitions: 2
Action: First element: type
['type', 'fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides',
```

```
[ ] print("is Empty RDD? : "+str(RDD1.isEmpty()))
```

```
is Empty RDD? : False
```

2) read.csv

```
[ ] RDD2 = spark.read.csv('wineQuality.csv',inferSchema=True, header =True).rdd
    RDD2
```

```
MapPartitionsRDD[14] at javaToPython at NativeMethodAccessorImpl.java:0
```

3) textfile

```
[ ] RDD3 = sc.textFile('wineQuality.csv')
```

```
[ ] RDD3
```

```
wineQuality.csv MapPartitionsRDD[107] at textFile at NativeMethodAccessorImpl.java:0
```

PART A: TRANSFORMATION

4) Map

```
[ ] RDD2.map(lambda x:x).collect()
```

```
Row(_c0=336, type='white', fixed acidity=6.3, volatile acidity=0.23, citric acid=0.33, res
Row(_c0=337, type='white', fixed acidity=5.8, volatile acidity=0.27, citric acid=0.27, res
Row(_c0=338, type='white', fixed acidity=5.9, volatile acidity=0.26, citric acid=0.4, resi
Row(_c0=339, type='white', fixed acidity=6.6, volatile acidity=0.18, citric acid=0.35, res
Row(_c0=340, type='white', fixed acidity=7.4, volatile acidity=0.2, citric acid=0.43, resi
Row(_c0=341, type='white', fixed acidity=8.0, volatile acidity=0.24, citric acid=0.36, res
Row(_c0=342, type='white', fixed acidity=6.4, volatile acidity=0.26, citric acid=0.42, res
Row(_c0=343, type='white', fixed acidity=5.4, volatile acidity=0.31, citric acid=0.47, res
Row(_c0=344, type='white', fixed acidity=5.4, volatile acidity=0.29, citric acid=0.47, res
Row(_c0=345, type='white', fixed acidity=7.1, volatile acidity=0.145, citric acid=0.33, re
Row(_c0=346, type='white', fixed acidity=5.6, volatile acidity=0.34, citric acid=0.1, resi
Row(_c0=347, type='white', fixed acidity=6.7, volatile acidity=0.19, citric acid=0.41, res
Row(_c0=348, type='white', fixed acidity=7.8, volatile acidity=0.18, citric acid=0.46, res
Row(_c0=349, type='white', fixed acidity=7.6, volatile acidity=0.17, citric acid=0.45, res
Row(_c0=350, type='white', fixed acidity=6.3, volatile acidity=0.12, citric acid=0.36, res
Row(_c0=351, type='white', fixed acidity=7.3, volatile acidity=0.33, citric acid=0.4, resi
Row(_c0=352, type='white', fixed acidity=5.5, volatile acidity=0.335, citric acid=0.3, res
Row(_c0=353, type='white', fixed acidity=7.3, volatile acidity=0.33, citric acid=0.4, resi
Row(_c0=354, type='white', fixed acidity=5.8, volatile acidity=0.4, citric acid=0.42, resi
Row(_c0=355, type='white', fixed acidity=7.3, volatile acidity=0.22, citric acid=0.37, res
Row(_c0=356, type='white', fixed acidity=7.3, volatile acidity=0.22, citric acid=0.37, res
Row(_c0=357, type='white', fixed acidity=6.1, volatile acidity=0.36, citric acid=0.33, res
Row(_c0=358, type='white', fixed acidity=10.0, volatile acidity=0.2, citric acid=0.39, res
Row(_c0=359, type='white', fixed acidity=6.9, volatile acidity=0.24, citric acid=0.34, res
Row(_c0=360, type='white', fixed acidity=6.4, volatile acidity=0.24, citric acid=0.32, res
Row(_c0=361, type='white', fixed acidity=7.1, volatile acidity=0.365, citric acid=0.14, re
```

```
RDD2.filter(lambda x : x['volatile acidity'] > 1.2 ).collect()
```

```
[Row(_c0=5024, Unnamed: 0=5024, type='red', fixed acidity=8.2, volatile acidity=1.33, 
 Row(_c0=5025, Unnamed: 0=5025, type='red', fixed acidity=8.1, volatile acidity=1.33, 
 Row(_c0=5570, Unnamed: 0=5570, type='red', fixed acidity=9.8, volatile acidity=1.24, 
 Row(_c0=6197, Unnamed: 0=6197, type='red', fixed acidity=7.6, volatile acidity=1.58, 
```

5) flatmap

```
[ ] # flatmap
    rdd_filtered_map = RDD2.flatMap(lambda x : [x['citric acid']/2 , x['pH']/14] )
    rdd_filtered_map.take(10)
```

```
[0.18,
 0.21428571428571427,
 0.17,
 0.2357142857142857,
 0.2,
 0.23285714285714285,
 0.16,
 0.22785714285714284,
 0.16,
 0.22785714285714284]
```

6) sortby

```
# sortby
rdd_sorted = RDD2.sortBy(lambda x: x['fixed acidity'])
rdd_sorted.take(20)
```

```
[Row(_c0=4259, Unnamed: 0=4259, type='white', fixed acidity=3.8, volatile acidity=0
 Row(_c0=4787, Unnamed: 0=4787, type='white', fixed acidity=3.9, volatile acidity=0
 Row(_c0=2872, Unnamed: 0=2872, type='white', fixed acidity=4.2, volatile acidity=0
 Row(_c0=3265, Unnamed: 0=3265, type='white', fixed acidity=4.2, volatile acidity=0
 Row(_c0=4446, Unnamed: 0=4446, type='white', fixed acidity=4.4, volatile acidity=0
 Row(_c0=4786, Unnamed: 0=4786, type='white', fixed acidity=4.4, volatile acidity=0
 Row(_c0=4847, Unnamed: 0=4847, type='white', fixed acidity=4.4, volatile acidity=0
 Row(_c0=2625, Unnamed: 0=2625, type='white', fixed acidity=4.5, volatile acidity=0
 Row(_c0=2321, Unnamed: 0=2321, type='white', fixed acidity=4.6, volatile acidity=0
 Row(_c0=4943, Unnamed: 0=4943, type='red', fixed acidity=4.6, volatile acidity=0.5
 Row(_c0=3710, Unnamed: 0=3710, type='white', fixed acidity=4.7, volatile acidity=0
 Row(_c0=3915, Unnamed: 0=3915, type='white', fixed acidity=4.7, volatile acidity=0
 Row(_c0=4470, Unnamed: 0=4470, type='white', fixed acidity=4.7, volatile acidity=0
 Row(_c0=4679, Unnamed: 0=4679, type='white', fixed acidity=4.7, volatile acidity=0
 Row(_c0=4792, Unnamed: 0=4792, type='white', fixed acidity=4.7, volatile acidity=0
 Row(_c0=4993, Unnamed: 0=4993, type='red', fixed acidity=4.7, volatile acidity=0.6
 Row(_c0=862, Unnamed: 0=862, type='white', fixed acidity=4.8, volatile acidity=0.3
 Row(_c0=864, Unnamed: 0=864, type='white', fixed acidity=4.8, volatile acidity=0.3
 Row(_c0=3556, Unnamed: 0=3556, type='white', fixed acidity=4.8, volatile acidity=0
 Row(_c0=3726, Unnamed: 0=3726, type='white', fixed acidity=4.8, volatile acidity=0
```

NAME: Meith Navlakha
SAP: 60004190068
BDI Practicals

PART B: ACTIONS

7) first

```
[ ]  # first
     rdd_sorted.first()
```

```
Row(_c0=4259, Unnamed: 0=4259, type='white', fixed acidity=3.8, volatile acidity=0.31,
```

8) take

```
# take
rdd_sorted.take(10)
```

```
[Row(_c0=4259, Unnamed: 0=4259, type='white', fixed acidity=3.8, volatile acidity=0.31, citric acid
 Row(_c0=4787, Unnamed: 0=4787, type='white', fixed acidity=3.9, volatile acidity=0.225, citric aci
 Row(_c0=2872, Unnamed: 0=2872, type='white', fixed acidity=4.2, volatile acidity=0.17, citric acid
 Row(_c0=3265, Unnamed: 0=3265, type='white', fixed acidity=4.2, volatile acidity=0.215, citric aci
 Row(_c0=4446, Unnamed: 0=4446, type='white', fixed acidity=4.4, volatile acidity=0.46, citric acid
 Row(_c0=4786, Unnamed: 0=4786, type='white', fixed acidity=4.4, volatile acidity=0.32, citric acid
 Row(_c0=4847, Unnamed: 0=4847, type='white', fixed acidity=4.4, volatile acidity=0.54, citric acid
 Row(_c0=2625, Unnamed: 0=2625, type='white', fixed acidity=4.5, volatile acidity=0.19, citric acid
 Row(_c0=2321, Unnamed: 0=2321, type='white', fixed acidity=4.6, volatile acidity=0.445, citric aci
 Row(_c0=4943, Unnamed: 0=4943, type='red', fixed acidity=4.6, volatile acidity=0.52, citric acid=0
```

9) reduce

```
# reduce
data = np.arange(10)
print(data)
sc.parallelize(data).reduce(add)
```

```
[0 1 2 3 4 5 6 7 8 9]
45
```

10) join

```
# join
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
x.join(y).collect()
```

```
[('b', (2, 5)), ('a', (1, 3)), ('a', (1, 4))]
```

11) groupByKey

```
# groupByKey
rdd = sc.parallelize([('B',5),('B',4),('A',3),('C',1),('A',2),('A',1)])
rdd = rdd.groupByKey()
[(j[0], list(j[1])) for j in rdd.collect()]
```

```
[('C', [1]), ('B', [5, 4]), ('A', [3, 2, 1])]
```

12) reduceByKey

```
RDD4 = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("c",2), ("a",2) , ("d",1)])
```

```
# reduceByKey
from operator import add
sorted(RDD4.reduceByKey(add).collect())
```

```
[('a', 4), ('b', 1), ('c', 2), ('d', 1)]
```

```
# countByKey
sorted(RDD4.countByKey().items())
```

```
[('a', 3), ('b', 1), ('c', 1), ('d', 1)]
```

13) countByKey

```
# countByKey
sorted(RDD4.countByKey().items())
```

```
[('a', 3), ('b', 1), ('c', 1), ('d', 1)]
```

Part C: Word Count using RDD in spark

Data

```
rdd = sc.textFile("content.txt")
rdd.collect()
```

```
['Love, hate, or feel meh about Harry Potter, it's hard to argue that J.K. Rowling filled the books
 'intentional writing choices. From made up words to the meanings of names to the well-scripted fir
 'Rowling wanted to the writing to match the intricate fantasy world she created for the now-iconic
 'these choices, I'll be taking a closer look at the first line of Harry Potter, as well as the las
 'novels']
```

Tokenize

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
rdd2.take(10)
```

```
['Love,',
 'hate,',
 'or',
 'feel',
 'meh',
 'about',
 'Harry',
 'Potter,',
 'it’s',
 'hard']
```

Map with 1

```
rdd3 = rdd2.map(lambda x: (x,1))
rdd3.take(10)
```

```
[('Love,', 1),
 ('hate,', 1),
 ('or', 1),
 ('feel', 1),
 ('meh', 1),
 ('about', 1),
 ('Harry', 1),
 ('Potter,', 1),
 ('it’s', 1),
 ('hard', 1)]
```

ReduceByKey

```
rdd4 = rdd3.reduceByKey(lambda a,b: a+b)
rdd4.take(10)
```

```
[('hate,', 1),
 ('feel', 1),
 ('meh', 1),
 ('Potter,', 2),
 ('it’s', 1),
 ('filled', 1),
 ('books', 1),
 ('', 4),
 ('intentional', 1),
 ('writing', 2)]
```

sortByKey

```
rdd5 = rdd4.map(lambda x: (x[1],x[0])).sortByKey()
rdd5.collect()
```

```
[(1, 'hate,'),
 (1, 'feel'),
 (1, 'meh'),
 (1, 'it's'),
 (1, 'filled'),
 (1, 'books'),
 (1, 'intentional'),
 (1, 'choices.'),
 (1, 'names'),
 (1, 'match'),
 (1, 'world'),
 (1, 'created'),
 (1, 'boy'),
 (1, 'wizard.'),
 (1, 'these'),
 (1, 'choices,'),
 (1, 'I'll'),
 (1, 'look'),
 (1, 'at'),
 (1, 'line'),
 (1, 'Love,'),
 (1, 'or'),
 (1, 'about'),
 (1, 'hard'),
 (1, 'argue'),
 (1, 'that'),
 (1, 'J.K.'),
 (1, 'with'),
 (1, 'From'),
 (1, 'made'),
 (1, 'up'),
 (1, 'words'),
 (1, 'meanings'),
 (1, 'well-scripted'),
 (1, 'and'),
 (1, 'lines'),
```

CONCLUSION:

Thus, in this experiment we explored and learnt about pySpark. We learnt what exactly is Spark, the RDD(Resilient Distributed Dataset), it's actions and transformation operations. In this experiment, we finally coded Word Count for a given data. The code and output has been observed and attached. Thus, we have successfully, completed our experiment, creating RDD and performing word occurrence for a given data.