ASSIGNMENT

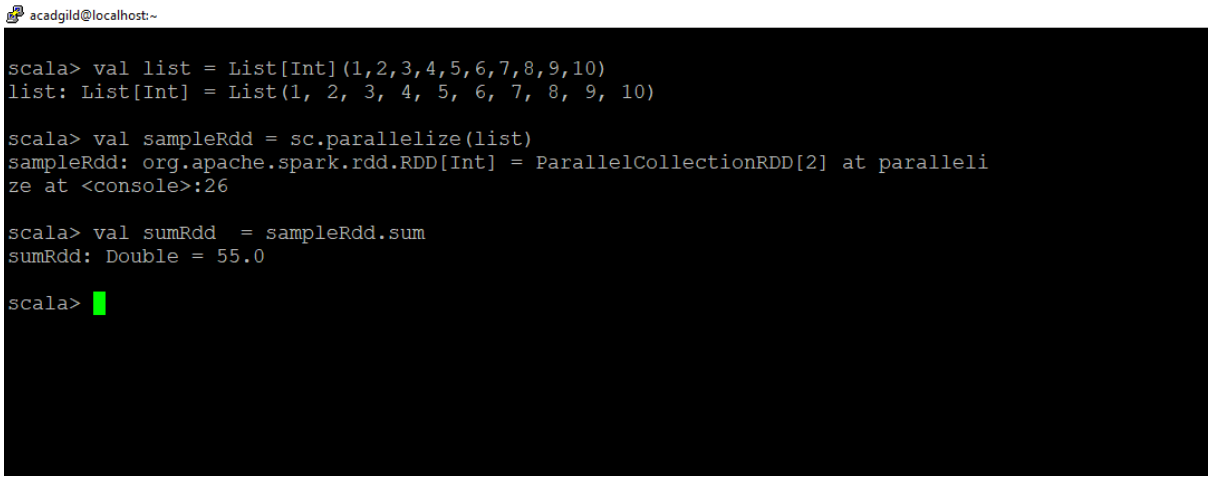
Task1:

1. a. Command used:

val list = List[Int](1,2,3,4,5,6,7,8,9,10)

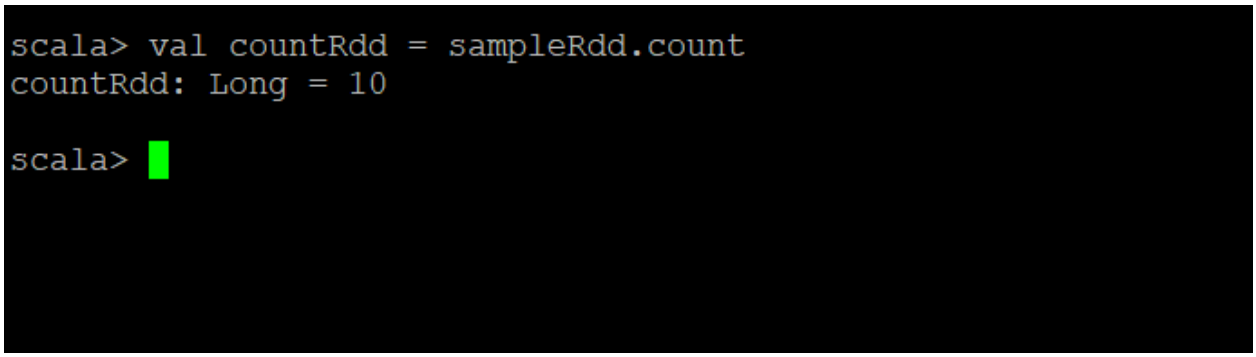
val sampleRdd = sc.parallelize(list)

val sumRdd = sampleRdd.sum



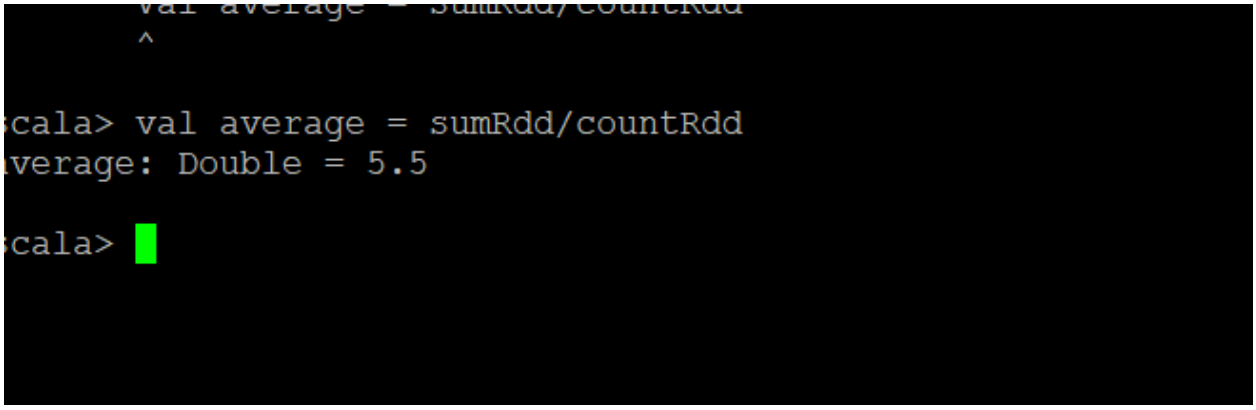
b. Command used:

val countRdd = sampleRdd.count



c. Command used:

val average = sumRdd/countRdd



d. val sumEven = sampleRdd.filter(_%2==0).sum

```
scala> val sumEven = sampleRdd.filter(_%2==0).sum
sumEven: Double = 30.0

scala>
```

e. val count = sampleRdd.filter(x=> x%3==0 || x%5==0).count

➔ val count = sampleRdd.filter(x=> x%3==0 && x%5==0).count

```
scala> val count = sampleRdd.filter(x=> x%3==0 || x%5==0).count
count: Long = 5

scala> val count = sampleRdd.filter(x=> x%3==0 && x%5==0).count
count: Long = 0

scala>
```

Task2

1. **Limitations of mapreduce**

➔ **Sol:** **a.** MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing **speed**.

b. Intermediate results in a processing operation are stored on a disk which slows down map reduce whereas spark stores the intermediate results in memory.

c. Small files are the major problem in HDFS. A small file is significantly smaller than the HDFS block size (default 128MB). HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the **NameNode** will be overloaded since it stores the namespace of HDFS.

d. **No delta iteration**: Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

e. **No real time processing** : Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result.

f. **latency**: In mapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data.

In **MapReduce**, Map takes a set of data and converts it into another set of data, where individual element are broken down into **key value pair** and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

g. **not easy to use**: In MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

h. **Security:** Hadoop can be challenging in managing the complex application. If the user doesn't know how to enable platform who is managing the platform, your data could be at huge risk. At storage and network levels, Hadoop is missing encryption, which is a major point of concern. Hadoop supports **Kerberos authentication**, which is hard to manage.
HDFS **supports access control lists** (ACLs) and a traditional file permissions model. However, third party vendors have enabled an organization to leverage **Active Directory Kerberos** and **LDAP** for authentication.

i. No abstraction: Hadoop does not have any type of abstraction so MapReduce developers need to hand code for each and every operation which makes it very difficult to work

2. What is RDD? Explain few features of RDD?

Sol: 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.

i. **In-Memory:** It is possible to store data in spark RDD. Storing of data in spark RDD is size as well as quantity independent. We can store as much data we want in any size. In-memory computation means operate information in the main RAM. It requires operating across jobs, not in complicated databases. Since operating jobs in databases slow the drive.

ii. **Lazy evaluations:** By its name, it says that on calling some operation, execution process doesn't start instantly. To trigger the execution, an action is a must. Since that action takes place, data inside RDD cannot get transform or available. Through DAG, Spark maintains the record of every operation performed whereas DAG refers to **Directed Acyclic Graph**.

iii. **Immutable and read only:** Since, RDDs are immutable. That property helps to maintain consistency when we perform further computations. As we cannot

make any change in RDD once created, it can only get transformed into new RDDs. This is possible through its transformations processes.

iv. **Cacheable or persistence**: We can store all the data in persistent storage, memory, and disk. Memory (most preferred) and disk (less Preferred because of its slow access speed). We can also extract it directly from memory. Hence this property of RDDs makes them useful for fast computations. Therefore, we can perform multiple operations on the same data. Also, leads reusability which also helps to compute faster.

v. **Partitioned**: Each dataset is logically partitioned and distributed across nodes over the cluster. They are just partitioned to enhance the processing, not divided internally. This arrangement of partitions provides parallelism.

vi. **Fault tolerance**: While working on any node, if we lost any RDD it recovers itself. When we apply different transformations on RDDs, it creates a logical execution plan. The logical execution plan is generally known as lineage graph. As a consequence, we may lose RDD as if any fault arises in the machine. So by applying the same computation on that node of the lineage graph, we can recover our same dataset again. As a matter of fact, this process enhances its property of Fault Tolerance.

vii. **Location stickiness**: RDDs supports placement preferences. That refers information of the location of RDD. That DAG(Directed Acyclic Graph) scheduler use to place computing partitions on. DAG helps to manage the tasks as much close to the data to operate efficiently. This placing of data also enhances the speed of computations.

viii. **Coarse-grained operations:** RDDs support coarse-grained operations. That means we can perform an operation on entire cluster once at a time.

ix. **No limitations:** There is no specific number that limits the usage of RDD. We can use as much RDDs we require. It totally depends on the size of its memory or disk

3. List down few spark RDD operations and explain them?

Sol: Two types of **Apache Spark** RDD operations are- Transformations and Actions.
A **Transformation** is a function that produces new **RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation.

a) **Transformations**:

i. Map()-> val spark = SparkSession.builder.appName("mapExample").master("local").getOrCreate()

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
mapFile.foreach(println)
```

ii. flatMap() ->

```
val data = spark.read.textFile("spark_test.txt").rdd
val flatmapFile = data.flatMap(lines => lines.split(" "))
flatmapFile.foreach(println)
```

iii. filter() ->

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())
```

iv. groupByKey->

```
val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)
```

v. reduceByKey() ->

```
val words = Array("one","two","two","four","five","six","six","eight","nine","ten")
val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(_+_)
data.foreach(println)
```

vi. sortByKey() ->

```
val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)
```

b) Actions:

i. **count():**

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count()
```

ii. **collect():**

```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))
```

iii. **take()->** val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)

```
val group = data.groupByKey().collect()

val twoRec = result.take(2)

twoRec.foreach(println)
```

iv. **top() ->** val data = spark.read.textFile("spark_test.txt").rdd

```
val mapFile = data.map(line => (line,line.length))

val res = mapFile.top(3)

res.foreach(println)
```