Assignment 18 Introduction to Spark

TASK 1

Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- find the sum of all numbers

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

scala> list.sum
res0: Int = 55
```

- find the total elements in the list

```
scala> list.length
res1: Int = 10
```

- calculate the average of the numbers in the list

```
scala> val res0=55.0
res0: Double = 55.0

scala> val res1=10.0
res1: Double = 10.0

scala> val avg = res0/res1
avg: Double = 5.5
```

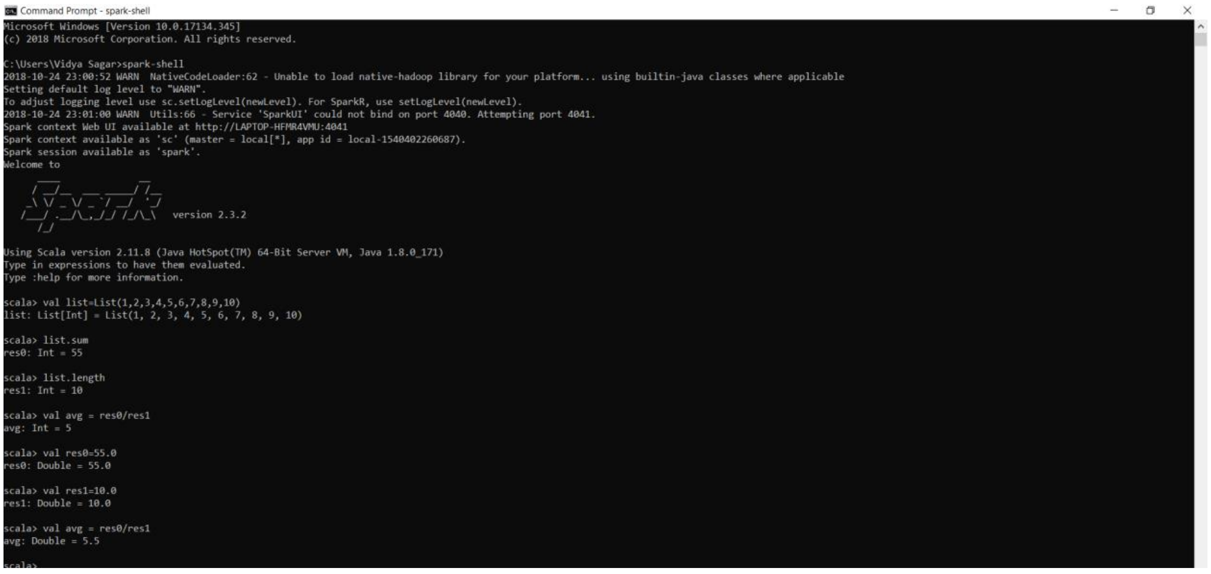
- find the sum of all the even numbers in the list

```
scala> val e = list.filter(x => x % 2 == 0)
e: List[Int] = List(2, 4, 6, 8, 10)

scala> e.sum
res2: Int = 30
```

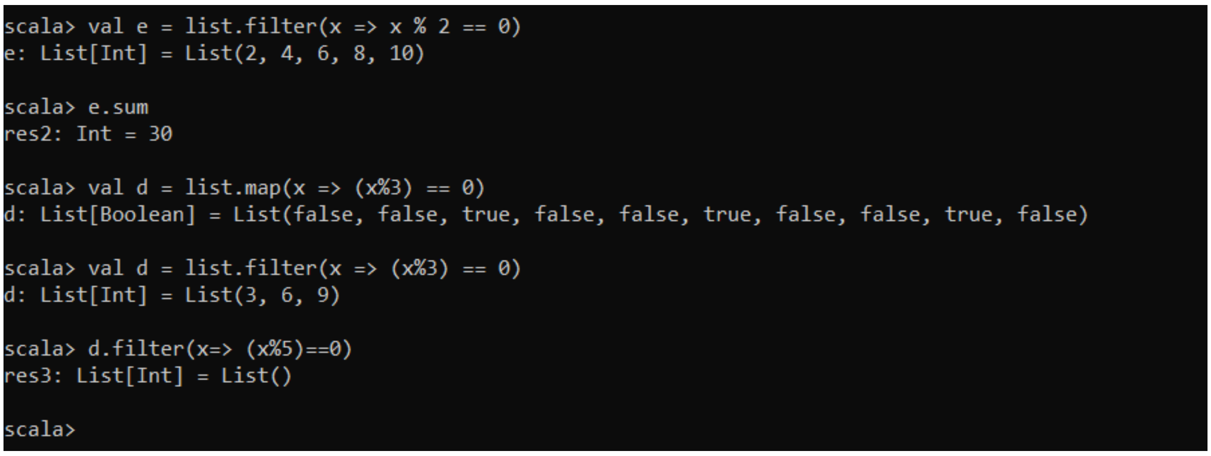
- find the total number of elements in the list divisible by both 5 and 3

```
scala> val d = list.filter(x => (x%3) == 0)
d: List[Int] = List(3, 6, 9)

scala> d.filter(x=> (x%5)==0)
res3: List[Int] = List()
```

Output screen



```
Microsoft Windows [Version 10.0.17134.345]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Vidya Sagar>spark-shell
2018-10-24 23:00:52 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
2018-10-24 23:01:00 WARN  Utils:66 - Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://LAPTOP-HFMR4VMU:4041
Spark context available as 'sc' (master = local[*], app id = local-1540402260687).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.2
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_171)
Type in expressions to have them evaluated.
Type :help for more information.

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

scala> list.sum
res0: Int = 55

scala> list.length
res1: Int = 10

scala> val avg = res0/res1
avg: Int = 5

scala> val res0=55.0
res0: Double = 55.0

scala> val res1=10.0
res1: Double = 10.0

scala> val avg = res0/res1
avg: Double = 5.5

scala>
```



```
scala> val e = list.filter(x => x % 2 == 0)
e: List[Int] = List(2, 4, 6, 8, 10)

scala> e.sum
res2: Int = 30

scala> val d = list.map(x => (x%3) == 0)
d: List[Boolean] = List(false, false, true, false, false, true, false, false, true, false)

scala> val d = list.filter(x => (x%3) == 0)
d: List[Int] = List(3, 6, 9)

scala> d.filter(x=> (x%5)==0)
res3: List[Int] = List()

scala>
```

TASK 2

1) ***Pen down the limitations of MapReduce.***

Map Reduce is the first frame work for processing and used for batch processing and it would not care about the time it took to complete the job. Spark is introduced to cover the time constraint.

There are many phases like mappers reducers and sort & shuffle which writes on disk and writing on the disk takes more time so spark was introduced which runs in memory.

Below are few points Map reduce cannot handle
1.Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

2.No Real-time Data Processing

3.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)

4.In Hadoop, 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.

5.Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.
----

2) *What is RDD? Explain few features of RDD?*

Apache Spark RDD

RDD stands Resilient Distributed Dataset. RDDs are the fundamental abstraction of Apache Spark. It is an immutable distributed collection of the dataset. Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute These partitions. RDDs are a read-only partitioned collection of record. we can create RDD in three ways:

· Parallelizing already existing collection in driver program.
· Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
· Creating RDD from already existing RDDs.

There are two operations in RDD namely transformation and Action.
Sparkling Features of Spark RDD

There are several advantages of using RDD. Some of them are-

1.In-memory computation

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes. refer this comprehensive guide to Learn Spark in-memory computation in detail.

2.Immutability

RDDS are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

3.Persistence

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function. Follow this guide for the detailed study of RDD persistence in Spark.

4.Partitioning

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

5.Typed

We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

6.No limitation

we can have any number of RDD. there is no limit to its number. the limit depends on the size of disk and memory.

Hence, using RDD we can recover the shortcoming of Hadoop MapReduce and can handle the large volume of data, as a result, it decreases the time complexity of the system. Thus the above-mentioned features of Spark RDD make them useful for fast computations and increase the performance of the system.

3.List down few Spark RDD operations and explain each of them.

Spark RDD Operations

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.

Transformation

map(func)

The map function iterates over every line in RDD and split into new RDD.
Using map() transformation we take in any function, and that function is applied to every element of RDD.

n the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For example, in RDD {1, 2, 3, 4, 5} if we apply “rdd.map(x=>x+2)” we will get the result as (3, 4, 5, 6, 7).

flatMap()

With the help of flatMap() function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.

Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

```
val data = spark.read.textFile("spark_test.txt").rdd

val flatmapFile = data.flatMap(lines => lines.split(" "))

flatmapFile.foreach(println)
```

filter(func)

Spark RDD filter() function returns a new RDD, containing only the elements that meet a predicate. It is a narrow operation because it does not shuffle data from one partition to many partitions.

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

Filter() example:

```
val data = spark.read.textFile("spark_test.txt").rdd

val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")

println(mapFile.count())
```

mapPartitions(func)

The MapPartition converts each partition of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

union(dataset)

With the union() function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

For example, the elements of RDD1 are (Spark, Spark,Hadoop, Flink) and that of RDD2 are (Big data, Spark, Flink) so the resultant rdd1.union(rdd2) will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

Union() example:

val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))

val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))

val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))

val rddUnion = rdd1.union(rdd2).union(rdd3)

rddUnion.foreach(Println)

distinct()

It returns a new dataset that contains the distinctelements of the source dataset. It is helpful to remove duplicate data.

For example, if RDD has elements (Spark, Spark, Hadoop, Flink), then rdd.distinct() will give elements (Spark, Hadoop, Flink).

Distinct() example:

val rdd1 =
park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov"
,2014)))

val result = rdd1.distinct()

println(result.collect().mkString(", "))

sortByKey()

When we apply the sortByKey() function on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

sortByKey() example:

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

val sorted = data.sortByKey()

sorted.foreach(println)

join()

The Join is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.

The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

Join() example:

```
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(","))
```

Action

count()

Action count() returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

Count() example:

```
val data = spark.read.textFile("spark_test.txt").rdd

val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")

println(mapFile.count())
```

collect()

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

Collect() example:

```
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(","))

take(n)

The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

Take() example:

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)

top()

If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top()use default ordering of data.

Top() example:

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)

countByValue()

The countByValue() returns, many times each element occur in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.countByValue()” will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

countByValue() example:

val data = spark.read.textFile("spark_test.txt").rdd

val result= data.map(line => (line,line.length)).countByValue()

result.foreach(println)

reduce()

The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

Reduce() example:

val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))

val sum = rdd1.reduce(_+_)

println(sum)

fold()

The signature of the fold() is like reduce(). Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation. The key difference
between fold() andreduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.

For example, zero is an identity for addition; one is identity element for multiplication. The return type of fold() is same as that of the element of RDD we are operating on.

For example, rdd.fold(0)((x, y) => x + y).

Fold() example:

val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))

val additionalMarks = ("extra", 4)

val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2

("total", add)

}

println(sum)

foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the driver. In this case, foreach()function is useful. For example, inserting a record into the database.

Foreach() example:

```
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)
```