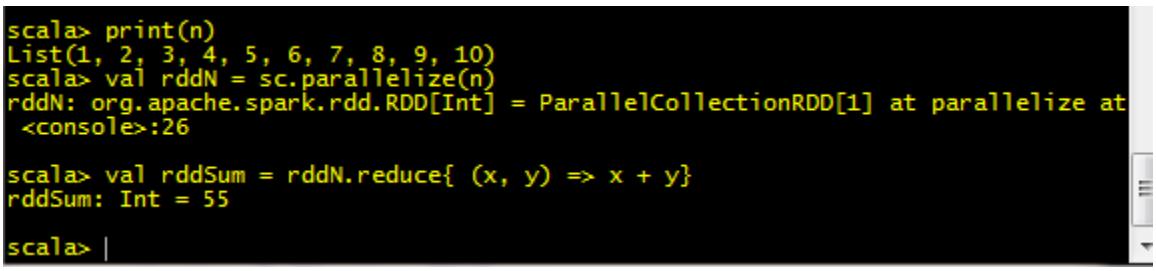
Big Data Hadoop and Spark Development: Session 18 Assignment 1

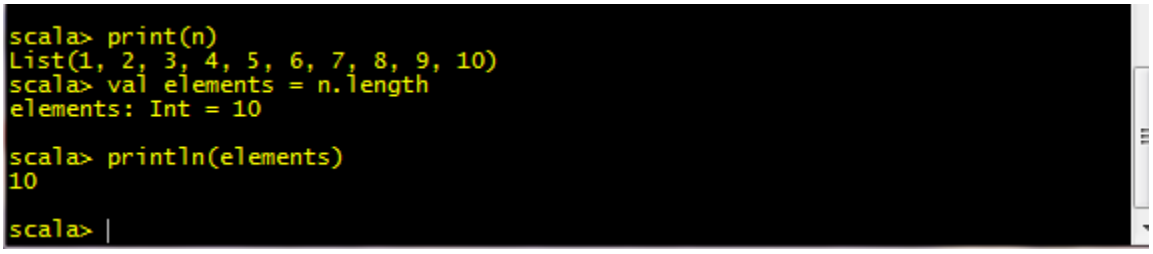
Task 1

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

- Find the sum of all numbers



- Find the total elements in the list



```
scala> rddN.count()
res13: Long = 10

scala>

scala>
```

- Calculate the average of the numbers in the list

```
scala> println(elements)
10

scala> print(n)
List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> println(sum)
55

scala> println(elements)
10

scala> val avg = n/elements
<console>:27: error: value / is not a member of List[Int]
       val avg = n/elements
                   ^

scala> val avg = sum/elements
avg: Int = 5
```

```
scala> val rddAvg = sc.parallelize(n)
rddAvg: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize
at <console>:26

scala> rddAvg.reduce(_ + _)
res0: Int = 55

scala> rddAvg.count.toDouble()
<console>:29: error: Double does not take parameters
       rddAvg.count.toDouble()
                            ^

scala> rddAvg.count.toDouble
res2: Double = 10.0

scala> rddAvg.reduce(_ + _)/rddAvg.count.toDouble
res3: Double = 5.5

scala>
```

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

```
scala> print(n)
List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> rddN = sc.parallelize(n)
<console>:28: error: not found: value rddN
val $ires7 = rddN
             ^
<console>:26: error: not found: value rddN
       rddN = sc.parallelize(n)
       ^

scala> val rddN = sc.parallelize(n)
rddN: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at
 <console>:26

scala> val evenN = rddN.filter(i => (i%2==0))
evenN: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at filter at <console
>:28

scala> val sumEvenN = evenN.reduce{ (x, y) => x + y}
sumEvenN: Int = 30

scala> print(sumEvenN)
30
scala> |
```

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

```
scala> print(n)
List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> rddDivisible = sc.parallelize(n)
<console>:28: error: not found: value rddDivisible
val $ires8 = rddDivisible
             ^
<console>:26: error: not found: value rddDivisible
       rddDivisible = sc.parallelize(n)
       ^

scala> val rddDivisible = sc.parallelize(n)
rddDivisible: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:26

scala> val rddDiv35 = rddDivisible.filter( i => ((i%3==0) || (i%5==0)))
rddDiv35: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at filter at <console>:28

scala> val numEleDiv35 = rddDiv35.count()
numEleDiv35: Long = 5

scala> println(numEleDiv35)
5

scala> |
```

Task 2

1. Pen down the limitations of MapReduce

The value of MapReduce is the ability to perform parallel operations across multiple nodes. The value of performing parallel operations across multiple notes is the foundation for both decentralized and distributed processing environments. The ability to process data on multiple nodes in parallel creates the capability to process large volumes of data in shorter periods of time using this distributed processing framework. Although MapReduce is a strong tool for processing large volumes of data in parallel, there are limitations to MapReduce.

Limitations of MapReduce include (a) slow processing, (b) batch only processing, and (c) learning curve. MapReduce leverages the Hadoop file system (HDFS) to process the data within a distributed environment. In addition, MapReduce has two main components; the mapping component and the reduce component. As MapReduce processes the data on each node it constantly reads and writes data to the disk of each node as it iterates through the MapReduce stages; input, map, shuffle, sort, reduce, output. This high disk I/O function causes MapReduce to have inherently slow processing. Another limitation of MapReduce is batch only processing. MapReduce is designed to read and process a batch of data at one time. MapReduce requires the data to remain static throughout processing in order to function properly. If data on one or more nodes changed during the mapping, shuffling, or sorting, then MapReduce has no way of re-processing the deltas without re-running the MapReduce. Therefore, data must remain static for the duration of batch processing, which is why MapReduce can only process batches data. Another limitation of MapReduce is the learning curve. MapReduce has many primitive features. However, using these features require individuals to learn the MapReduce processing structure to be effective in using MapReduce properly. When dealing with large dataset, many people are accustomed to using RDBMS environment to write queries using SQL to obtain needed information. Although tools such as Pig and Hive use an SQL-like language to ease the burden of learning MapReduce, natively MapReduce does not use SQL. For this reason, MapReduce require a learning curve to use properly.

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

RDD stands for Resilient Distributed Dataset. RDD is an immutable fault-tolerant data structure within Spark that reads a partition of data. RDD offers an advantage over MapReduce because it supports in-memory processing. This in-memory processing creates two distinct advantages; faster processing and streamed processing across multiple nodes. Some features of RDD includes (a) in-memory processing, fault tolerance, and (c) partitioning. In-memory processing provides an ability to process data in RAM instead of performing physical disk operations. Processing data in RAM significantly reduces the time to process large volumes of data compared to physical disk processing. Another feature of RDD is fault tolerance. RDD can dynamically reproduce lost data when there is a node failure. This reproduction of lost data is because each

dataset is immutable and RDD remembers the lineage of the operation. In the event of a lost input partition, RDD can re-compute the original dataset using lineage operations. Another feature of RDD is partition. Partitions are logical chunks of mutable data that is used for input to the RDD operations. RDD performs transformations on the partition to execute business logic.

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

RDD operations include Map, FlatMap, Filter, Distinct, GroupByKey, and Join. The RDD mapping function iterates over each line or element within the RDD. Using map, users can define functions that will operate on each element within the RDD. Another operation in RDD is FlatMap. Although Map and FlatMap are similar, FlatMap operates on the entire list of elements in the RDD, whereas Map operates on each element at a time. The Filter operation provides the ability to define a new RDD based on a subset of data derived from the parent RDD. Filters operate similar to the WHERE clause in an RDBMS system. The RDD distinct operation is designed to remove duplicate elements from the RDD. Like the filter operation, distinct creates a new dataset based on the parent RDD with duplicate elements removed. The groupByKey operation acts on RDD dataset defined using key/value pairs. The groupByKey operations groups the key/value pairs within the RDD by key; the groupByKey is a transformation operation.