Session 18: INTRODUCTION TO SPARK

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 numbers

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_151)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala>
```

- Find the total elements in the list

```
scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala> rdd1.count()
res1: Long = 10

scala>
```

acadgild@localhost:~

- Calculate the average of the numbers in the list

```
scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala> rdd1.count()
res1: Long = 10

scala> sum
res2: Int = 55

scala> val avg =sum.toFloat/res1
avg: Float = 5.5

scala>
```

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

```
scala> val evenNumberRdd= rdd1.filter(i => (i%2==0))
evenNumberRdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at filter at <console>:26

scala> evenNumberRdd.collect()
res7: Array[Int] = Array(2, 4, 6, 8, 10)

scala> evenNumberRdd.sum()
res8: Double = 30.0

scala>
```

acadgild@localhost:~

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

```
scala> val oddRdd = rdd1.filter(x => x % 3 == 0 || x % 5 == 0)
oddRdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at filter at <console>:26

scala> oddRdd.count()
res13: Long = 5

scala> oddRdd.collect()
res14: Array[Int] = Array(3, 5, 6, 9, 10)

scala>
```

Task 2

- Pen down the limitations of MapReduce

MapReduce is meant to handle batch processing
MapReduce cannot handle

- Interactive processing
- Real-time (stream) processing
- Iterative (delta) processing
- In-memory processing.
- Graph processing

I. *Issue with small files*
Hadoop is not suited for small data. HDFS lacks the ability to efficiently support the random reading of small files because of high capacity design

II. *Slow Processing speed*
Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

III. *Support for Batch processing only*
Hadoop supports only batch processing only, it does not process streamed data, and hence overall performance is slower.

IV. *No real time data processing*
Hadoop is not suitable for real time processing.

V. *No Data 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 in input to the next stage).

VI. *Latency*
Map takes 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 require lot of time to perform these tasks thereby increasing latency.

VII. *No Latency*
Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for further requirement which diminishes the performance of Hadoop.

- What is RDD? Explain the features of RDD?

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

i. Parallelizing the existing collection in driver program.
ii. Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).

iii. Creating an RDD from an existing RDDs

Feature of RDD

a. *In-memory computation*- The data inside the RDD are stored in memory for as long as you want to store. Keeping the data in memory improves the performance.
b. *Lazy evaluation*- the changes or the computation is performed only after the action is triggered.
c. *Fault Tolerance*- Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one.
d. *Immutability*- RDDs are immutable in nature i.e. once we create and RDD we cannot manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.
e. *Partitioning*- RDD partitions are 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.

- List down few spark RDD operations and explain each of them.

Apache spark supports two types of operations.

- Transformations and
- Actions

RDD Transformation is a function that produces new RDD from the existing RDDs. It takes the RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation.

Transformations are lazy in nature i.e. they gets executed when we call an action.

There are two type of transformation:

Narrow transformation – II the elements that are required to compute the records in single partition live in single partition of parent RDD. Narrow transformations are the result of *map()*, *flatmap*(), *union()*

Wide transformation- In wide transformation, all elements that are required to compute the records in single partition may live in many partitions of parent RDD. Wide transformations are result of *groupbyKey()* and *reducebyKey(),join(),intersection()*

RDD Action

An action is one of the ways of sending data from *Executor* to the *driver.* Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of a task. Some of the actions of spark are:

- count() : returns the number of elements in RDD
- collect() : is the common and simplest operation that returns our entire RDDs content to driver program.
- take(n) : returns n number of elements from RDD.
- reduce() : takes the two elements as input from the RDD and then produces the output of same type as that of input elements.