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 all numbers

- find the total elements in the list

- calculate the average of the numbers in the list

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

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

Solution:

List of numbers:

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

Here sc is spark context object. sc.parallelize is used to create a RDD on integers within the list as input.

2)sum of all numbers:

```
scala> x.sum
res0: Double = 33.0

scala> x.reduce((x,y)=>x+y)
res1: Int = 33
```

Explanation: Spark reduce operation is almost similar as reduce method in Scala. It is an ***action operation*** of RDD which means it will trigger all the lined up transformation on the base RDD (or in the DAG) which are not executed and than execute the action operation on the last RDD. This operation is also a wide operation. In the sense the execution of this operation results in distributing the data across the multiple partitions.

It accepts a function with (which accepts two arguments and returns a single element) which should be Commutative and Associative in mathematical nature. That intuitively means, this function produces same result when repetitively

applied on same set of RDD data with multiple partitions irrespective of element's order.

find the total elements in the list

counting the elements

Explanation: Counting the number of elements within RDD using built-in function count.

```
scala> x.count
res2: Long = 6
```

- calculate the average of the numbers in the list

```
scala> x.mean
res3: Double = 5.5

scala> val y =x.sum/x.count
y: Double = 5.5
```

Explanation: Finding the average of elements within rdd dividing total by length of rdd **or** by using built-in function mean

find the sum of all the even numbers in the list

Explanation: Using built-in filter function to filter even number by taking modulus of 2 & then finding the sum of even elements within rdd using built-in function sum.

```
scala> val z= x.filter(a=> a%2==0)
z: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at filter at <console>:26

scala> val y = z.sum
y: Double = 30.0
```

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

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

scala> val z =y.sum
z: Double = 19.0

scala> val y =x.filter(a=> ((a%3==0)&&(a%5==0)))
y: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[8] at filter at <console>:26

scala> val z =y.sum
z: Double = 0.0

scala>
```


Task 2

1) Pen down the limitations of MapReduce.

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

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

Answers: Limitations of MapReduce:

1. Processing speed

In **Hadoop**, with a parallel and distributed algorithm, MapReduce process large data sets. MapReduce algorithm contains two important tasks: Map and Reduce and, MapReduce require lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce.

2. Data processing

Hadoop **MapReduce** is designed for *Batch processing*, that means it take huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing high volume of data, but depending on the size of the data being processed and computational power of the system, output can be delayed significantly. Hadoop is not suitable for *Real-time data processing*.

3. Latency

In Hadoop, 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.

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 add one such as hive and pig, make working with MapReduce a little easier for adopters.

5. Caching

In Hadoop, MapReduce cannot cache the intermediate data in-memory for a further requirement which diminishes the performance of hadoop

6. 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?

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.

Features of RDD?

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

Lazy Evaluation

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do.

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. Thus, we can easily recover the lost data.

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.

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.

Parallel

Rdd, process the data parallelly over the cluster.

Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD.
The **DAGScheduler**places the partitions in such a way that task is close to data as much as possible. Thus speed up computation.

Coarse-grained Operation

We apply coarse-grained transformations to RDD**.** Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the data set of RDD.

Typed

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

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.

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

Map:
Map will take each row as input and return an RDD for the row. The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

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).

Flat map:

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.

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

In above code, flatMap() function splits each line when space occurs.

Filter:

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:

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

- ***Note*** – In above code, flatMap function map line into words and then count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

ReduceByKey:

reduceByKey(func, [numTasks])

When we use **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.

reduceByKey() example:

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

- ***Note –*** The above code will parallelize the Array of String. It will then map each word with count 1, then reduceByKey will merge the count of values having the similar key.

Distict:

It returns a new dataset that contains the **distinct** elements 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:

```
1. val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
2. val result = rdd1.distinct()
3. println(result.collect().mkString(", "))
```

- ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'"nov",2014).

Intersection:

With the **intersection()** function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant ***rdd1.intersection(rdd2)***will have elements (spark).

Intersection() example:

```
1. val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014, (16,"feb",2014)))
2. val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
3. val comman = rdd1.intersection(rdd2)
4. comman.foreach(Println)
```

- ***Note –*** The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

Sort By Key:

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:

```
1. val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82), ("computer",65), ("maths",85)))
2. val sorted = data.sortByKey()
3. 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:

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

- ***Note*** – The join() transformation will join two different RDDs on the basis of Key.