Assignment Day 18

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

Ans:

list of numbers

val r_d = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10**))**

Explanation: sc is SparkContext object. sc.parallelize is used to create an RDD on integers within the list input.

the sum of all numbers

r_d.sum

r_d.reduce((x,y**)** => x+y**)**

Explanation: reduce aggregates the elements of the dataset using a function func (which takes **two arguments** and **returns one**). The function should be commutative and associative so that it can be computed correctly in parallel.

the total elements in the list

r_d.count

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

the average of the numbers in the list

r_d.sum/r_d.count

r_d.mean

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

the sum of all the even numbers in the list

r_d.filter(x => x%2==0**).sum**

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.

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

println("Total number of elements divisible by **both 3 and 5 is "+r_d.filter(**x => ((x%3==0) && (x%5==0**))).count)**

Explanation: Counting all such elements within the rdd whose elements are divisible by both 3 as well as 5 using filter function that returns count only for conditions that returns a boolean 1 or True.

Observation: Finding the elements in RDD which are divisible by either 3 or 5

println("Total number of elements divisible by **either 3 or 5 is "+r_d.filter(**x => ((x%3==0) || (x%5==0**))).count)**

ScreenShot:

```
scala> val r_d = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
r_d: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:24

scala>

scala> r_d.sum
res44: Double = 55.0

scala> r_d.reduce((x,y) => x+y)
res45: Int = 55

scala>

scala> r_d.count
res46: Long = 10

scala>

scala> r_d.mean
res47: Double = 5.5

scala>

scala> r_d.filter(x => x%2==0).sum
res48: Double = 30.0

scala>

scala> println("Total number of elements divisible by both 3 and 5 is "+r_d.filter(x => ((x%3==0) && (x%5==0))).count)
Total number of elements divisible by both 3 and 5 is 0

scala>

scala> println("Total number of elements divisible by either 3 or 5 is "+r_d.filter(x => ((x%3==0) || (x%5==0))).count)
Total number of elements divisible by either 3 or 5 is 5

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.

Ans:

1) Pen down the limitations of MapReduce.

Ans:

i. It is based on disk based computation which makes computation jobs slower

ii. It is only meant for single pass computation, not iterative computations. It requires a sequence of Map Reduce jobs to run iterative task

iii. Needs integration with several tools to solve big data usecases. Integration with Apache Storm is required for Stream data processing. Integration with Mahout required for Machine Learning

iv. Problems that cannot be trivially partitionable or recombinable becomes a candid limitation of MapReduce problem solving. For instance, Travelling Salesman problem.

v. Due to the fixed cost incurred by each MapReduce job submitted, application that requires low latency time or random access to a large set of data is infeasible.

vi. Tasks that has a dependency on each other cannot be parallelized, which is not possible through MapReduce.

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

Ans:

RDD is a logical reference of a dataset which is partitioned across many server machines in the cluster.It is the primary abstraction in Spark and is the core of Apache Spark. Immutable and partitioned collection of records, which can only be created by coarse grained operations such as map, filter, group-by, etc. Can only be created by reading data from a stable storage like HDFS or by transformations on existing RDD's.

Features of RDD:

i. Resilient, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures

ii. Distributed with data residing on multiple nodes in a cluster.

iii. Dataset is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects

iv. In-Memory, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.

v. Immutable or Read-Only, i.e. it does not change once created and can only be transformed using transformations to new RDDs.

vi. Lazy evaluated, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.

vii. Cacheable, i.e. we can hold all the data in a persistent "storage" like

memory (default and the most preferred) or disk (the least preferred due

to access speed).

viii. IParallel, i.e. process data in parallel.

ix. Typed — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].

x. Partitioned — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.

xi. Location-Stickiness — RDD can define placement preferences to

compute partitions (as close to the records as possible).

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

Ans:

i. map: 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.

Example:

tupleRDD has 4 fields (name, subject, grade, marks). Using map operations two fields are taken (subject, marks)

```
val studentMarksSubjectRDD = tupleRDD.map(t=> (t._2, t._4)
```

ii. 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. flatMap returns a collection of elements

Example:

Tis example takes a input file and split them into words

```
val rdd = sc.textFile("/home/acadgild/assignment_17.1/wordcount_input_file")
val rdd_words = rdd.flatMap(line=> line.split(" "))
```

iii. 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.

Example:

tupleRDD has 4 fields (name, subject, grade, marks). Using filter operation only students who are in grade-2 taken

```
val grade2StudentRDD = tupleRDD.filter(t=> t._3 == "grade-2")
```

iv. **mapPartition**: 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.

v. Union: 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.

Example:

In the example rdd1 has three dates, rdd2 has two dates and rdd3 has two dates, union operation is done on rdd1, rdd2, rdd3 to get new RDD rddUnion

```
val rdd1 = 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)
```

vi. 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.

Example:

In the example rdd1 has three dates, rdd2 has two dates and rdd2 has two dates, intersection operation is done on rdd1, rdd2 to get new RDD rddCommon

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

vii. **Distinct**: It returns a new dataset that contains the distinct elements of the source dataset. It is helpful to remove duplicate data.

Example:

In this example tuple RDD is created from a file which has student records. Distict is used to get distinct tuples

```
val baseRDD = sc.textFile("/home/acadgild/assignment_18/18_Dataset.txt")

val tupleRDD = baseRDD.map(x => (x.split(",")(0), x.split(",")(1), x.split(",")(2), x.split(",")(3).toInt))

val distinctTupleRDD = tupleRDD.distinct
```

viii. **ReduceByKey**: 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.

Example:

In this example, distinctGradeStudentMapCountRDD has tuples with first elementa as key grade and second element as value marks. Using reduceByKey operations Marks for each grade are summed and put to gradeStudentCountRDD

```
val gradeStudentCountRDD = distinctGradeStudentMapCountRDD.reduceByKey((x, y) => x+y)
```

ix. **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.

Example:

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

val sorted = data.sortByKey()

sorted.foreach(println)
```

x. Join:

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 advantage 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.

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(","))
```

********** **End** **********