

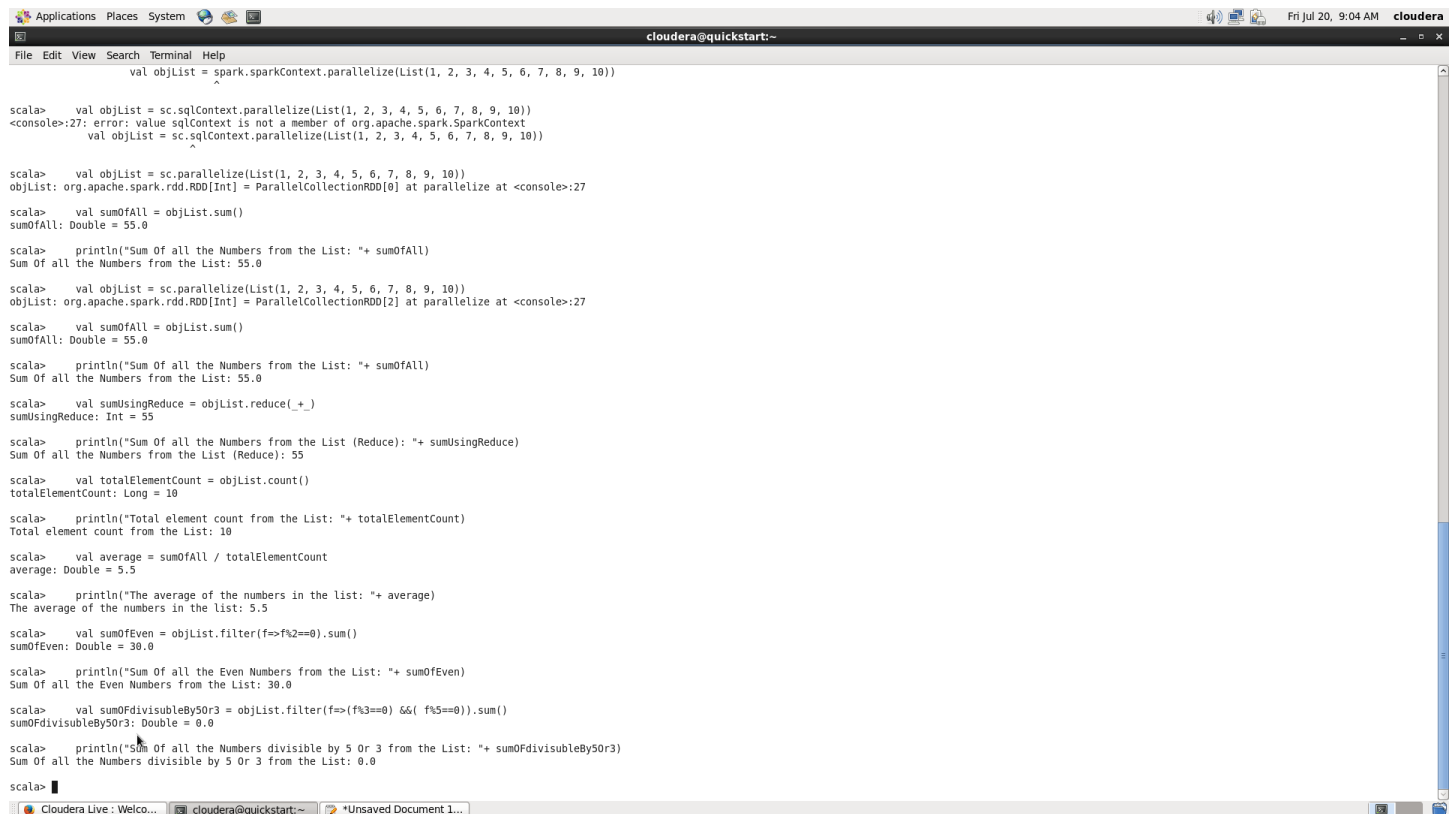
Assignment 18 – INTRODUCTION TO SPARK

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
- 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

Please find the code & answers in below screenshot:



```
Applications Places System cloudera@quickstart:~
File Edit View Search Terminal Help
val objList = spark.sparkContext.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))

scala> val objList = sc.sqlContext.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
<console>:27: error: value sqlContext is not a member of org.apache.spark.SparkContext
val objList = sc.sqlContext.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))

scala> val objList = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
objList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val sumOfAll = objList.sum()
sumOfAll: Double = 55.0

scala> println("Sum Of all the Numbers from the List: "+ sumOfAll)
Sum Of all the Numbers from the List: 55.0

scala> val objList = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
objList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:27

scala> val sumOfAll = objList.sum()
sumOfAll: Double = 55.0

scala> println("Sum Of all the Numbers from the List: "+ sumOfAll)
Sum Of all the Numbers from the List: 55.0

scala> val sumUsingReduce = objList.reduce(_+_ )
sumUsingReduce: Int = 55

scala> println("Sum Of all the Numbers from the List (Reduce): "+ sumUsingReduce)
Sum Of all the Numbers from the List (Reduce): 55

scala> val totalElementCount = objList.count()
totalElementCount: Long = 10

scala> println("Total element count from the List: "+ totalElementCount)
Total element count from the List: 10

scala> val average = sumOfAll / totalElementCount
average: Double = 5.5

scala> println("The average of the numbers in the list: "+ average)
The average of the numbers in the list: 5.5

scala> val sumOfEven = objList.filter(f=>f%2==0).sum()
sumOfEven: Double = 30.0

scala> println("Sum of all the Even Numbers from the List: "+ sumOfEven)
Sum Of all the Even Numbers from the List: 30.0

scala> val sumOfDivisibleBy5Or3 = objList.filter(f=>(f%3==0) && ( f%5==0)).sum()
sumOfDivisibleBy5Or3: Double = 0.0

scala> println("Sum Of all the Numbers divisible by 5 Or 3 from the List: "+ sumOfDivisibleBy5Or3)
Sum Of all the Numbers divisible by 5 Or 3 from the List: 0.0

scala>
```

Assignment 18 – INTRODUCTION TO SPARK

Task 2

1) Pen down the limitations of MapReduce.

MapReduce cannot handle:

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing

Below are some details regarding the issues / limitations of Map Reduce

1. Issues with Small Files

Hadoop is not suited for small data. **(HDFS) Hadoop distributed file system** lacks the ability to efficiently support the random reading of small files because of its high capacity design

2. Slow Processing Speed

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. Support for Batch Processing only

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.

4. No Real-time Data Processing

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

5. No Delta 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 is the input to the next stage).

6. High 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.

7. Not Easy to Use

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.

Assignment 18 – INTRODUCTION TO SPARK

8. No Caching

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.

9. Security

Hadoop can be challenging in managing the complex application. If the user doesn't know how to enable platform who is managing the platform, your data could be at huge risk. At storage and network levels, Hadoop is missing encryption, which is a major point of concern. Hadoop supports **Kerberos authentication**, which is hard to manage.

10. No 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.

11. Vulnerable by Nature

Hadoop is entirely written in **java**, a language most widely used, hence java been most heavily exploited by cyber criminals and as a result, implicated in numerous security breaches.

12. Lengthy Line of Code

Hadoop has 1,20,000 line of code, the number of lines produces the number of bugs and it will take more time to execute the program.

2 What is RDD? Explain few features of RDD?

RDD (Resilient Distributed Dataset): is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is **logically partitioned** across many servers so that they can be computed on different nodes of the cluster.

Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Features Of RDD

1. In-memory Computation

Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program.

3. Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.

Assignment 18 – INTRODUCTION TO SPARK

4. Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5. Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

6. Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

7. Coarse-grained Operations

We apply coarse-grained transformations to RDD, i.e. the operation applies to the whole dataset not on an individual element in the data set of RDD.

8. Parallel

RDD, process the data in parallel over the cluster.

9. Location-Stickiness

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

10. Typed

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

11. No limitation

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

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

RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

Transformations

Spark RDD Transformations are functions that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.

Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

1. Narrow Transformations

It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

2. Wide Transformations

Assignment 18 – INTRODUCTION TO SPARK

It is the result of `groupByKey()` and `reduceByKey()` like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.

Actions

An Action in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. `First()`, `take()`, `reduce()`, `collect()`, the `count()` is some of the Actions in spark.

1. 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. In 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 Eg: we need to square all the values in the list

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(", "))
```

2. 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.

`flatMap()` example:

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

3. filter(func)

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

`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())
```

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`.

4. 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.

```
val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
```

Assignment 18 – INTRODUCTION TO SPARK

```
val common = rdd1.intersection(rdd2)
common.foreach(println)
```

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

5. distinct()

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

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(", "))
```

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

6. groupByKey()

When we use groupByKey() on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.

Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory.

groupByKey() 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)
```

Note – The groupByKey() will group the integers on the basis of same key(alphabet). After that collect() action will return all the elements of the dataset as an Array.

7. 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:

```
val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b", 1), ("b", 1), ("b", 1), ("b", 1)))
val y = x.reduceByKey((key, value) => (key + value))
y.collect()
```

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

8. 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),
```

Assignment 18 – INTRODUCTION TO SPARK

```
("computer",65), ("maths",85)))
```

```
val sorted = data.sortByKey()
```

```
sorted.foreach(println)
```

Note – In above code, `sortByKey()` transformation sort the data RDD into Ascending order of the Key(String).

9. 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.

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(", "))
```

1.Count()

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

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())
```

Note – In above code `flatMap()` function maps line into words and count the word “Spark” using `count()` Action after filtering lines containing “Spark” from `mapFile`.

2.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(", "))
```

Note – `join()` transformation in above code will join two RDDs on the basis of same key(alphabet). After that `collect()` action will return all the elements to the dataset as an Array.

3.CountByValue()

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

countByValue() example:

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

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

```
result.foreach(println).
```