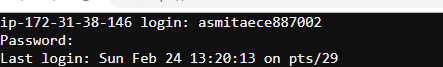
An RDD has partitions and each partitions has records. How to place the records in various partitions is controlled by partitioning. Partitioning is basically about choosing the distribution of keys in the RDD partitions. This is sometimes needed as per project objectives and sometimes it is needed for optimization.

1st login to console with user id and pwd



Run Spark-shell

On the scala spark shell, lets first create an array of numbers from 1 to 10:

///below commands are run in web console

var nums = 1 to 10;

////Now, let convert in an RDD using parallelize:

var numsRdd = sc.parallelize(nums)

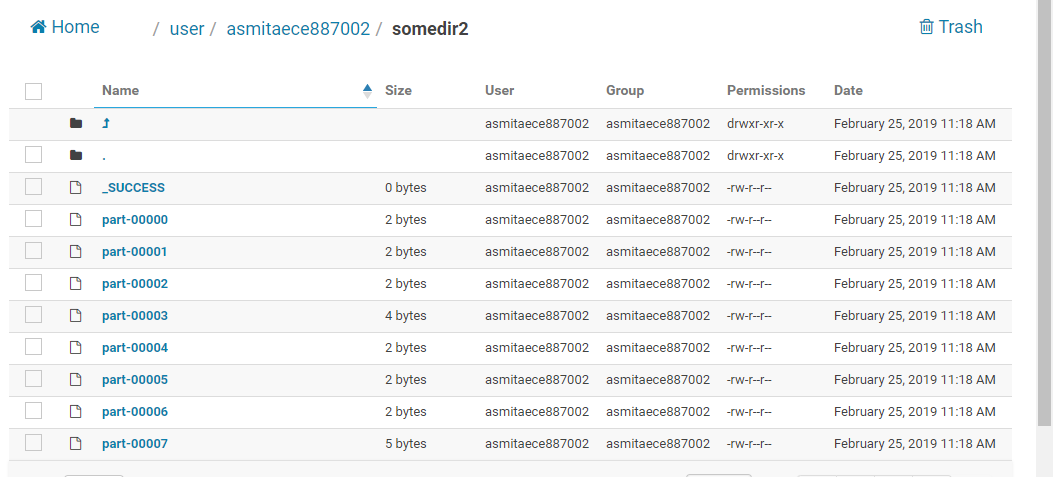
////To know about the number of partitions we can use numsRdd.partitions.length function:

**numsRdd.partitions.length**

///But to take a look at the data we can eithe save to HDFS

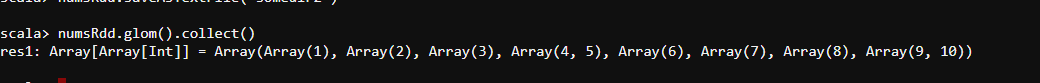
////using saveAsTextFile and then take a look at the actual data in the files created.

**numsRdd.saveAsTextFile("somedir2")**

****

**For each of the partitions , we can check from glom() using web console**

If your data is really small you can use glom to convert the partitions into arrary of arrays and then use collect. collect is generally impactical on huge data.

****

**Here we find , that glom creates array of arrays**

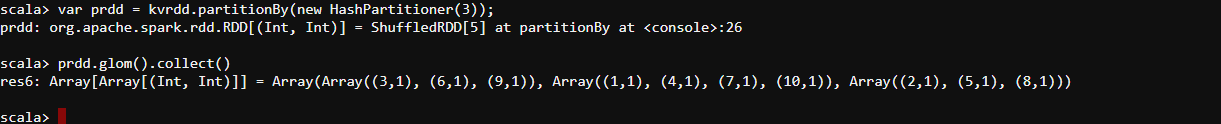
Since, only key-value RDD can be partitioned, let us create an RDD with key-value pairs with 1 as the value for every key.

Now, to partition, we need to specify partitioners. Partitioner are the classes or pieces of logic which specify how to distribute data between the partitions of RDD.

As part of core spark, we are provided with two partitioners:

HashPartitioner and

RangeParitioner

****

You can see it has partitioned the RDD into three parts.

Now, please notice that the records have being put into partitioned in round robin manner.

The hashPartitioner first computes the hashcode of the key and then computing the remainder of the hashcode after dividing by number of partitions. This remainder specifies the partition in which key needs to be put

Here key value is 1 , all those values which are divisible by 3 went into the 1st partition . hence hash value pf al these values are 0 so 3,6 ,9 went to 1st partition.

All those , whish resulted 1 as remainder , went into 2nd partition , that is 1,4,7 went to 2nd partition.

1/3 means 1 is remainder , 4/3 is 1 rema , 7/3 , 10/3

3rd partition , all those where remainder is 2

Here the ranges are completly overlapping

-----------------------------------------------------------------------------------------------------------------------

Sometimes, you may want to partition an RDD into non-overlapping partitions. This is where the range based partitioning comes into play.

 Lets repartition the wordsCount using rangePartitioner:

**import org.apache.spark.RangePartitioner**

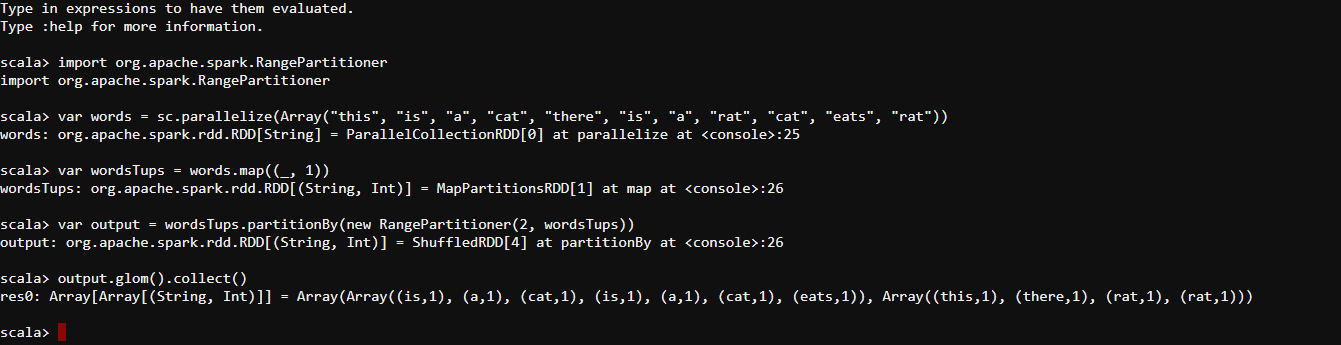
**var words = sc.parallelize(Array("this", "is", "a", "cat", "there", "is", "a", "rat", "cat", "eats", "rat"))**

**var wordsTups = words.map((\_, 1))**

**var output = wordsTups.partitionBy(new RangePartitioner(2, wordsTups))**

**output.glom().collect()**

**You can see that there are two partitions one having words less than or equal to is and second having keys bigger than "is".**

****