For Scala: <https://www.analyticsvidhya.com/blog/2017/01/scala/>

If you are working with Apache Spark then you would know that it has 4 different APIs support for different languages: **Scala, Java, Python and R**.

* **Lazy operation:** Operations which do not execute until we require results.
* **Spark Context:**  holds a connection with Spark cluster manager.
* **Driver and Worker:** A driver is in charge of the process of running the main() function of an application and creating the SparkContext.
* **In-memory computation:** Keeping the data in RAM instead of Hard Disk for fast processing.

Spark has three data representations viz RDD, Dataframe, Dataset. To use Apache Spark functionality, we must use one of them for data manipulation. Let’s discuss each of them briefly:

* **RDD:**RDD (Resilient Distributed Database) is a collection of elements, that can be divided across multiple nodes in a cluster for parallel processing. It is also fault tolerant collection of elements, which means it can automatically recover from failures. RDD is immutable, we can create RDD once but can’t change it.
* **Dataset:**It is also a distributed collection of data. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). As I have already discussed in my previous articles, dataset API is only available in Scala and Java. It is not available in Python and R.
* **DataFrame:** In Spark, a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame. It is mostly used for structured data processing. In Scala, a DataFrame is represented by a Dataset of Rows. A DataFrame can be constructed by wide range of arrays for example, existing RDDs, Hive tables, database tables.
* **Transformation:** Transformation refers to the operation applied on a RDD to create new RDD.
* **Action:** Actions refer to an operation which also apply on RDD that perform computation and send the result back to driver.
* **Broadcast:** We can use the Broadcast variable to save the copy of data across all node.
* **Accumulator:** In Accumulator, variables are used for aggregating the information.

## Create RDDs in Apache Spark

* Parallelizing already existing collection in driver program.
* Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
* Creating RDD from already existing RDDs.

### Creating a RDD from existing source

val data = Array(1, 2, 3, 4, 5,6,7,8,9,10)

val distData = sc.parallelize(data)

### Creating a RDD from External sources

You can create a RDD through external sources such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format. So let’s create a RDD from the text file:

val lines = sc.textFile("text.txt");

Spark RDD can contain Objects of any type.

There are two types of RDD Operations.

1. Transformations : Create a new RDD from an existing RDD
2. Actions : Run a computation or aggregation on the RDD and return a value to the driver program.

### Transformation are lazy

In Spark, Transformations are lazy. Lazy by meaning, they are not actually acted upon until an action is encountered. For an RDD, all transformations are kept in a queue and when an action is encountered, all the transformations and action are executed.

### RDDs are fault tolerant

In Spark, data is stored in RDDs. Transformations could be applied on RDDs or new RDDs could be created using actions. Unlike other big data frameworks, the intermediate data is not stored onto disk storage. Only the information about the transformations an RDD undergoes is stored. So, if in case a node goes down, Spark has information of what has to be done with the input data, hence fault tolerant. Because of this, it avoids time and resource costly reads and writes to persistent data storage.

Spark Shell is an interactive shell through which we can access Spark’s API. Spark provides the shell in two programming languages : Scala and Python.

~$ spark-shell

we shall start with option --master local[4] meaning the spark context of this spark shell acts as a **master** on **local** node with **4** threads.

|  |
| --- |
| $ spark-shell --master local[4] |
| **Spark context** available as sc, meaning you may access the spark context in the shell as variable named ‘sc’.  **Spark session** available as spark, meaning you may access the spark session in the shell as variable named ‘spark’. Map Transformations A map transformation is useful when we need to transform a RDD by applying a function to each element. So how can we use map transformation on ‘rdd’ in our case? Let’s calculate the length (number of characters) of each line in “text.txt”  val lines = sc.textFile("text.txt");  val Lenght = lines.map(s => s.length)  Length.collect() Read two text files to single RDD String files = "data/rdd/input/file1.txt, data/rdd/input/file2.txt, data/rdd/input/file3.txt"; lines = sc.textFile(files); filter Transformation Let’s filter out the lines in “text.txt” whose length is more than 5.  val lg5 = lines.filter(\_.length > 5)  val file = sc.textFile("catalina.out")  val errors = file.filter(line => line.contains("ERROR")) flatMap() Flat-Mapping is transforming each RDD element using a function that could return multiple elements to new RDD. Simple example would be applying a flatMap to Strings and using split function to return words to new RDD  lines.flatMap(line => line.split(" "))  sc.parallelize(List(1,2,3)).flatMap(x=>List(x,x,x))  org.apache.spark.rdd.RDD[Int] = FlatMappedRDD[373]  sc.parallelize(List(1,2,3)).map(x=>List(x,x,x))  res203: org.apache.spark.rdd.RDD[List[Int]] = MappedRDD[375] mapPartitions(func) Similar to map() transformation but in this case function runs separately on each partition (block) of RDD unlike map() where it was running on each element of partition. Hence mapPartitions are also useful when you are looking for performance gain (calls your function once/partition not once/element).   Suppose you have elements from 1 to 100 distributed among 10 partitions i.e. 10 elements/partition. map() transformation will call func 100 times to process these 100 elements but in case of mapPartitions(), func will be called once/partition i.e. 10 times.   Secondly, mapPartitions() holds the data in-memory i.e. it will store the result in memory until all the elements of the partition has been processed.   mapPartitions() will return the result only after it finishes processing of whole partition.   mapPartitions() requires an iterator input unlike map() transformation  sc.parallelize(1 to 9, 3).***map***(x=>(x, "Hello")).collect  res3: Array[(Int, String)] = Array((1,Hello), (2,Hello), (3,Hello), (4,Hello), (5,Hello), (6,Hello), (7,Hello), (8,Hello), (9,Hello))  sc.parallelize(1 to 9, 3).partitions.size  res95: Int = 3  sc.parallelize(1 to 9, 3).***mapPartitions***(x=>(Array("Hello").iterator)).collect  res7: Array[String] = Array(Hello, Hello, Hello)  sc.parallelize(1 to 9, 3).***mapPartitions***(x=>(List(x.next).iterator)).collect  res11: Array[Int] = Array(1, 4, 7)  In first example, I have applied map() transformation on dataset distributed between 3 partitions so that you can see function is called 9 times. In second example, when we applied mapPartitions(), you will notice it ran 3 times i.e. for each partition once. We had to convert string "Hello" into iterator because mapPartitions() takes iterator as input. In thirds step, I tried to get the iterator next value to show you the element. Note that next is always increasing value, so you can't step back.  sc.parallelize(1 to 9, 3).mapPartitions(x=>(List(x.next,x.next, "|").iterator)).collect  res18: Array[Any] = Array(1, 2, |, 4, 5, |, 7, 8, |) mapPartitionsWithIndex(func) Similar to mapPartitions, but good part is that you have index to see the partition position. For example,  val mp = sc.parallelize(List("One","Two","Three","Four","Five","Six","Seven","Eight","Nine"), 3)  mp: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[38] at parallelize at <console>:24  scala> mp.collect  res23: Array[String] = Array(One, Two, Three, Four, Five, Six, Seven, Eight, Nine)  scala> mp.mapPartitionsWithIndex((index, iterator) => {iterator.toList.map(x => x + "=>" + index ).iterator} ).collect  res26: Array[String] = Array(One=>0, Two=>0, Three=>0, Four=>1, Five=>1, Six=>1, Seven=>2, Eight=>2, Nine=>2)  Index 0 (first partition) has three values as expected, similarly other 2 partitions. union(otherDataset) Similar to SQL union, but it keeps duplicate data.  scala> val rdd1 = sc.parallelize(List("apple","***orange***","grapes","mango","***orange***"))  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[159] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(List("red","green","yellow"))  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[160] at parallelize at <console>:24  scala> rdd1.***union***(rdd2).collect  res116: Array[String] = Array(apple, ***orange***, grapes, mango, ***orange***, red, green, yellow)  scala> rdd2.***union***(rdd1).collect  res117: Array[String] = Array(red, green, yellow, apple, ***orange***, grapes, mango, ***orange***) intersection(otherDataset) Returns intersection of two datasets. For example,  scala> val rdd1 = sc.parallelize(-5 to 5)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[171] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(1 to 10)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[172] at parallelize at <console>:24  scala> rdd1.intersection(rdd2).collect  res119: Array[Int] = Array(4, 1, 5, 2, 3) distinct() Returns new dataset with distinct elements. For example, we don't have duplicate orange now.  scala> val rdd = sc.parallelize(List("apple","orange","grapes","mango","orange"))  rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[186] at parallelize at <console>:24  scala> rdd.distinct.collect  res121: Array[String] = Array(grapes, orange, apple, mango) groupByKey() As name says it groups the dataset (K, V) key-value pair based on Key and stores the value as Iterable, (K, V) => (K, Iterable(V)). It's very expensive operation and consumes lot of memory if dataset is huge. For example,  scala> val rdd = sc.parallelize(List("Hello Hello Spark Apache Hello Dataneb Dataneb Dataneb Spark"))  rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:24  scala> rdd.collect  res3: Array[String] = Array(Hello Hello Spark Apache Hello Dataneb Dataneb Dataneb Spark)  // Splitting the array and creating (K, V) pair  scala> val keyValue = rdd.flatMap(words => words.split(" ")).map(x=>(x,1))  keyValue: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[16] at map at <console>:25  // Iterable[Int] Value "1" tells number of occurrences of Key  scala> keyValue.***groupByKey***.collect  res12: Array[(String, Iterable[Int])] = Array((Spark,CompactBuffer(1, 1)), (Dataneb,CompactBuffer(1, 1, 1)), (Hello,CompactBuffer(1, 1, 1)), (Apache,CompactBuffer(1))) reduceByKey() Operates on (K, V) pair dataset, but reduce func must be of type (V, V) => V. For example, if you want to reduce all the values to get the total number of occurrences.  scala> rdd  .flatMap(words => words.split(" "))  .map(x=>(x,1))  .***reduceByKey***((x, y)=>x+y)  .collect  res14: Array[(String, Int)] = Array((Spark,2), (Dataneb,3), (Hello,3), (Apache,1)) aggregateByKey() It's similar to reduceByKey(), I hardly use this transformation because you can achieve the same with previous transformation. For example,  scala> rdd  .flatMap(words => words.split(" "))  .map(x=>(x,1))  .***aggregateByKey***(0)((x, y)=> x+y, (k, v)=> k+v )  .collect  res24: Array[(String, Int)] = Array((Spark,2), (Dataneb,3), (Hello,3), (Apache,1)) sortByKey() Called upon key-value pair, returns sorted by keys. For example,  scala> rdd  .flatMap(words => words.split(" "))  .distinct  .map(x => (x,1))  .***sortByKey***() -- by default Ascending  .collect  res36: Array[(String, Int)] = Array((Apache,1), (Dataneb,1), (Hello,1), (Spark,1))  scala> rdd  .flatMap(words => words.split(" "))  .distinct  .map(x => (x,1))  .***sortByKey***(false) -- Ascending order (false)  .collect  res37: Array[(String, Int)] = Array((Spark,1), (Hello,1), (Dataneb,1), (Apache,1)) join() It takes datasets of type key-value pair and works same like sql joins. For no match value will be None. For example,  scala> val rdd1 = sc.parallelize(List("Apple","Orange", "Banana", "Grapes", "Strawberry", "Papaya")).map(words => (words,1))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[96] at map at <console>:24  scala> val rdd2 = sc.parallelize(List("Apple", "Grapes", "Peach", "Fruits")).map(words => (words,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[98] at map at <console>:24  scala> rdd1.***join***(rdd2).collect  res40: Array[(String, (Int, Int))] = Array((Grapes,(1,1)), (Apple,(1,1)))  scala> rdd1.***rightOuterJoin***(rdd2).collect  res41: Array[(String, (Option[Int], Int))] = Array((Grapes,(Some(1),1)), (Peach,(None,1)), (Apple,(Some(1),1)), (Fruits,(None,1)))  scala> rdd1.***leftOuterJoin***(rdd2).collect  res43: Array[(String, (Int, Option[Int]))] = Array((Grapes,(1,Some(1))), (Banana,(1,None)), (Papaya,(1,None)), (Orange,(1,None)), (Apple,(1,Some(1))), (Strawberry,(1,None)))  scala> rdd1.***fullOuterJoin***(rdd2).collect  res44: Array[(String, (Option[Int], Option[Int]))] = Array((Grapes,(Some(1),Some(1))), (Peach,(None,Some(1))), (Banana,(Some(1),None)), (Papaya,(Some(1),None)), (Orange,(Some(1),None)), (Apple,(Some(1),Some(1))), (Fruits,(None,Some(1))), (Strawberry,(Some(1),None))) cartesian() Same like cartesian product, return all possible pairs of elements of dataset.  scala> val rdd1 = sc.parallelize(List("Apple","Orange", "Banana", "Grapes", "Strawberry", "Papaya"))  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[111] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(List("Apple", "Grapes", "Peach", "Fruits"))  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[112] at parallelize at <console>:24  scala> rdd1.***cartesian***(rdd2).collect  res46: Array[(String, String)] = Array((Apple,Apple), (Apple,Grapes), (Apple,Peach), (Apple,Fruits), (Orange,Apple), (Banana,Apple), (Orange,Grapes), (Banana,Grapes), (Orange,Peach), (Banana,Peach), (Orange,Fruits), (Banana,Fruits), (Grapes,Apple), (Grapes,Grapes), (Grapes,Peach), (Grapes,Fruits), (Strawberry,Apple), (Papaya,Apple), (Strawberry,Grapes), (Papaya,Grapes), (Strawberry,Peach), (Papaya,Peach), (Strawberry,Fruits), (Papaya,Fruits)) coalesce() coalesce and repartition both shuffles the data to increase or decrease the partition, but repartition is more costlier operation as it re-shuffles all data and creates new partition. For example,  scala> val distData = sc.parallelize(1 to 16, 4)  distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[128] at parallelize at <console>:24  // current partition size  scala> distData.partitions.size  res63: Int = 4  // checking data across each partition  scala> distData.mapPartitionsWithIndex((index, iter) => if (index == 0) iter else Iterator()).collect  res64: Array[Int] = Array(***1, 2, 3, 4***)  scala> distData.mapPartitionsWithIndex((index, iter) => if (index == 1) iter else Iterator()).collect  res65: Array[Int] = Array(5, 6, 7, 8)  scala> distData.mapPartitionsWithIndex((index, iter) => if (index == 2) iter else Iterator()).collect  res66: Array[Int] = Array(***9, 10, 11, 12***)  scala> distData.mapPartitionsWithIndex((index, iter) => if (index == 3) iter else Iterator()).collect  res67: Array[Int] = Array(13, 14, 15, 16)  // decreasing partitions to 2  scala> val coalData = distData.***coalesce***(2)  coalData: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[133] at coalesce at <console>:25  // see how shuffling occurred. Instead of moving all data it just moved 2 partitions.  scala> coalData.mapPartitionsWithIndex((index, iter) => if (index == 0) iter else Iterator()).collect  res68: Array[Int] = Array(***1, 2, 3, 4,*** 5, 6, 7, 8)  scala> coalData.mapPartitionsWithIndex((index, iter) => if (index == 1) iter else Iterator()).collect  res69: Array[Int] = Array(***9, 10, 11, 12,*** 13, 14, 15, 16) repartition() Notice how it re-shuffled everything to create new partitions as compared to previous RDDs - distData and coalData. Hence repartition is more costlier operation as compared to coalesce.  scala> val repartData = distData.***repartition***(2)  repartData: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[139] at repartition at <console>:25  // checking data across each partition  scala> repartData.mapPartitionsWithIndex((index, iter) => if (index == 0) iter else Iterator()).collect  res70: Array[Int] = Array(***1***, ***3***, 6, 8, ***9***, ***11***, 13, 15)  scala> repartData.mapPartitionsWithIndex((index, iter) => if (index == 1) iter else Iterator()).collect  res71: Array[Int] = Array(***2***, ***4***, 5, 7, ***10***, ***12***, 14, 16)  Now, as said earlier, RDDs are immutable so you can't change original RDD but you can always create a new RDD with spark [transformations](http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations) like map, flatmap, filter, groupByKey, reduceByKey, mapValues, sample, union, intersection, distinct, sortByKey etc.  RDDs transformations are broadly classified into two categories - Narrow & Wide transformation.   * In narrow transformation like map & filter, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. * In wide transformation like groupByKey and reduceByKey, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD.  ****Spark Actions**** When you want to work on actual dataset, you need to perform spark [actions](http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions) on RDDs like count, reduce, collect, first, takeSample, saveAsTextFile etc.   * Transformations are lazy in nature i.e. nothing happens when the code is evaluated. Meaning actual execution happens only when code is executed. RDDs are computed only when an action is applied on them. Also called as lazy evaluation. Spark evaluates the expression only when its value is needed by action. * When you call an action, it actually triggers transformations to act upon RDD, dataset or dataframe. After that RDD, dataset or dataframe is calculated in memory. In short, transformations will actually occur only when you apply an action. Before that it’s just line of evaluated code :)  reduce() It aggregates the elements of the dataset. For example,  *scala> val rdd = sc.parallelize(1 to 15).collect*  *rdd: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)*  *scala> val rdd = sc.parallelize(1 to 15).reduce(\_ + \_)*  *rdd: Int = 120*  *scala> val rdd = sc.parallelize(Array("Hello", "Dataneb", "Spark")).reduce(\_ + \_)*  *rdd: String = SparkHelloDataneb*  *scala> val rdd = sc.parallelize(Array("Hello", "Dataneb", "Spark")).map(x =>(x, x.length)).flatMap(l=> List(l.\_2)).collect*  *rdd: Array[Int] = Array(5, 7, 5)*  *scala> rdd.reduce(\_ + \_)*  *res96: Int = 17*  *scala> rdd.reduce((x, y)=>x+y)*  *res99: Int = 17* collect(), count(), first(), take() Collect returns all the elements of the dataset as an array. For example  *scala> sc.parallelize(1 to 20, 4).collect*  *res100: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)*  Counts the number of elements  *scala> sc.parallelize(1 to 20, 4).count*  *res101: Long = 20*  First returns the first element  *scala> sc.parallelize(1 to 20, 4).first*  *res102: Int = 1*  Take returns the number of elements you pass as argument  *scala> sc.parallelize(1 to 20, 4).take(5)*  *res104: Array[Int] = Array(1, 2, 3, 4, 5)* takeSample() It returns the random sample of size n. Boolean input is for with or without replacement. For example,  *scala> sc.parallelize(1 to 20, 4).takeSample(false,4)*  *res107: Array[Int] = Array(15, 2, 5, 17)*  *scala> sc.parallelize(1 to 20, 4).takeSample(false,4)*  *res108: Array[Int] = Array(12, 5, 4, 11)*  *scala> sc.parallelize(1 to 20, 4).takeSample(true,4)*  *res109: Array[Int] = Array(18, 4, 1, 18)* takeOrdered() It returns the elements in ordered fashion. For example,  *scala> sc.parallelize(1 to 20, 4).takeOrdered(7)*  *res117: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7)*  Just opposite to top() action  *scala> sc.parallelize(1 to 20, 4).top(7)*  *res118: Array[Int] = Array(20, 19, 18, 17, 16, 15, 14)* countByKey() It takes (key, value) pair and returns (key, count of key). For example,  *scala> sc.parallelize(Array("Apple","Banana","Grapes","Oranges","Grapes","Banana")).map(k=>(k,1)).countByKey()*  *res121: scala.collection.Map[String,Long] = Map(Grapes -> 2, Oranges -> 1, Banana -> 2, Apple -> 1)* saveAsTextFile() It saves the dataset as text files in local directory or HDFS etc. You can reduce the number of files by coalesce transformation.  *scala>sc.parallelize(Array("Apple","Banana","Grapes","Oranges","Grapes","Banana")).saveAsTextFile("sampleFruits.txt")*  *// Just one partition file with coalesce*  *scala>sc.parallelize(Array("Apple","Banana","Grapes","Oranges","Grapes","Banana")).coalesce(1).saveAsTextFile("newsampleFruits.txt")* saveAsObjectFile() It writes the data into simple format using Java serialization and you can load it again using sc.objectFile()  *scala> sc.parallelize(List(1,2)).saveAsObjectFile("/Users/Rajput/sample")* foreach() It is generally used when you want to carry out some operation on output for each element. Like loading each element into database.  *scala> sc.parallelize("Hello").collect*  *res139: Array[Char] = Array(H, e, l, l, o)*  *scala> sc.parallelize("Hello").foreach(x=>println(x))*  *// Output order of elements is not same every time*  *scala> sc.parallelize("Hello").foreach(x=>println(x))*  **Word-Count Example with Spark (Scala) Shell**  /\*\* map \*/  var map = sc.textFile("/path/to/text/file").flatMap(line => line.split(" ")).map(word => (word,1));    /\*\* reduce \*/  var counts = map.reduceByKey(\_ + \_);    /\*\* save the output to file \*/  counts.saveAsTextFile("/path/to/output/") |
|  |