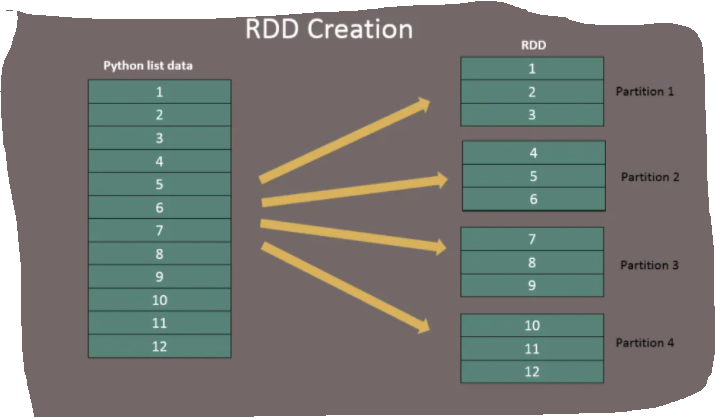
**Creating RDD**

RDD’s are created primarily in two different ways,

* [parallelizing an existing collection](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd/) and
* [referencing a dataset in an external storage system](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/) (HDFS)

#### Create RDD using sparkContext.parallelize()

This function loads the existing collection from your driver program into parallelizing RDD.



## #Create RDD from parallelize

## data = [1,2,3,4,5,6,7,8,9,10,11,12]

# rdd=spark.sparkContext.parallelize(data)

#### Create RDD using sparkContext.textFile()

#### #Create RDD from external Data source

#### rdd2 = spark.sparkContext.textFile("/path/textFile.txt")

#### rdd2 = spark.sparkContext.textFile("/a.txt")

**getNumPartitions()** – This a RDD function which returns a number of partitions our dataset split into.

print("initial partition count:"+str(rdd.getNumPartitions()))

sparkContext.parallelize([1,2,3,4,56,7,8,9,12,3], 10)// to set partitions manually

## PySpark RDD Operations

## **RDD transformations –** Transformations are lazy operations, instead of updating an RDD, these operations return another RDD. **RDD actions –** operations that trigger computation and return RDD values.

First, create an RDD by reading a text file

1. rdd = spark.sparkContext.textFile("/b.txt")

hadoop is fun

fun with hadoop

**flatMap**– flatMap() transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

rdd2 = rdd.flatMap(lambda x: x.split(" "))

This is the first line---🡪(this is the,first,line)

This is the second line;

This is the last line.

scala> val textFile = sc.textFile("b.txt")

scala> textFile.map(line => line.split(" ")).count()

res2: Long = 3

scala> textFile.flatMap(line => line.split(" ")).count()

res3: Long = 15

scala> textFile.map(line => line.split(" "))

Array(Array(This, is, the, first, line;), Array(This, is, the, second, line;), Array(This, is, the, last, line.))

textFile.flatMap(line => line.split(" "))

Array(This, is, the, first, line, This, is, the, second, line;, This, is, the, last, line.)

**map**– map() transformation is used the apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.

In our word count example, we are adding a new column with value 1 for each word, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

rdd3 = rdd2.map(lambda x: (x,1))

**reduceByKey** – reduceByKey() merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

rdd5 = rdd4.reduceByKey(lambda a,b: a+b)

**filter** – filter() transformation is used to filter the records in an RDD. In our example we are filtering all words starts with “a”.

rdd4 = rdd3.filter(lambda x : 'an' in x[1])

print(rdd4.collect())

### RDD Actions with example

[RDD Action operations](https://sparkbyexamples.com/pyspark/pyspark-rdd-actions/) return the values from an RDD to a driver program. In other words, any RDD function that returns non-RDD is considered as an action.

**count**() – Returns the number of records in an RDD

# Action - count

print("Count : "+str(rdd6.count()))

**first**() – Returns the first record.

# Action - first

firstRec = rdd6.first()

print("First Record : "+str(firstRec[0]) + ","+ firstRec[1])

rdd.take(3)

**take**() – Returns the record specified as an argument.

# Action - take

data3 = rdd6.take(3)

**collect**() – Returns all data from RDD as an array. Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.

# Action - collect

data = rdd6.collect()

**saveAsTextFile**() – Using saveAsTestFile action, we can write the RDD to a text file.

rdd6.saveAsTextFile("/tmp/wordCount")

## rdd.saveasTextFile(‘/storing/Wordcount’)

## Broadcast variables are useful for when you have a large dataset that you want to use across all the worker nodes. Instead of having to send out the entire dataset, only the variable is sent out.

## In the same shell from the last section, create a broadcast variable. Type in

## broadcastVar = sc.broadcast(Array(1,2,3))

## To get the value, type in:

## broadcastVar.value

## Accumulators are variables that can only be added through an associative operation. It is used to implement counters and sum efficiently in parallel. Spark natively supports numeric type accumulators and standard mutable collections. Programmers can extend these for new types. Only the driver can read the values of the accumulators. The workers can only invoke it to increment the value.

## Create the accumulator variable. Type in:

## accum = sc.accumulator(0)

## Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

## sc.parallelize(Array(1,2,3,4)).foreach(x => accum += x)

## To get the current value of the accumulator variable, type in:

## accum.value

## You should get a value of 10.

## This command can only be invoked on the driver side. The worker nodes can only increment the accumulator