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

| data = [1,2,3,4,5,6,7,8,9,10,11,12]  rdd=spark.sparkContext.parallelize(data)  rdd.count() |
| --- |

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

| rdd2 = spark.sparkContext.textFile("file:///home/imran/customer.txt")  rdd2.count() |
| --- |

#### **Creating empty RDD with partition**

| rdd10=spark.sparkContext.parallelize(data, 10)  print("initial partition count:"+str(rdd10.getNumPartitions())) |
| --- |

## **Repartition and Coalesce**

Sometimes we may need to repartition the RDD, PySpark provides two ways to repartition; first using repartition() method which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffle data from minimum nodes, for examples if you have data in 4 partitions and doing coalesce(2) moves data from just 2 nodes.

Both of the functions take the number of partitions to repartition rdd as shown below. Note that the repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.

| reparRdd = rdd.repartition(4)  print("re-partition count:"+str(reparRdd.getNumPartitions())) |
| --- |

## **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.

### 

### **RDD Transformations with example**

| rdd = spark.sparkContext.textFile("file:///home/zidane/test.txt") |
| --- |

**flatMap** – flatMap() transformation flattens the RDD after applying the function and returns a new RDD. In 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(" ")) |
| --- |

**map** – map() transformation is used to 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.

| 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 = rdd3.reduceByKey(lambda a,b: a+b)  print(rdd5.collect()) |
| --- |

**sortByKey** – sortByKey() transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int, String]) using map transformation and apply sortByKey which ideally does sort on an integer value. And finally, foreach with println statements returns all words in RDD and their count as key-value pair

| rdd6 = rdd5.map(lambda x: (x[1],x[0])).sortByKey()  print(rdd6.collect()) |
| --- |

**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[0])  print(rdd4.collect()) |
| --- |

### **RDD Actions with example**

RDD Action operations 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

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

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

| # Action - first  firstRec = rdd6.first()  print("First Record : "+str(firstRec[0]) + ","+ firstRec[1]) |
| --- |

**max()** – Returns max record.

| # Action - max  datMax = rdd6.max()  print("Max Record : "+str(datMax[0]) + ","+ datMax[1]) |
| --- |

**reduce**() – Reduces the records to single, we can use this to count or sum.

| # Action - reduce  totalWordCount = rdd6.reduce(lambda a,b: (a[0]+b[0],a[1]))  print("dataReduce Record : "+str(totalWordCount[0])) |
| --- |

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

| # Action - take  data3 = rdd6.take(3)  for f in data3:  print("data3 Key:"+ str(f[0]) +", Value:"+f[1]) |
| --- |

**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()  for f in data:  print("Key:"+ str(f[0]) +", Value:"+f[1]) |
| --- |

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

| rdd6.saveAsTextFile("file:///home/zidane/wordCount") |
| --- |

### **RDD Cache**

PySpark RDD cache() method by default saves RDD computation to storage level `MEMORY\_ONLY` meaning it will store the data in the JVM heap as unserialized objects.

PySpark cache() method in RDD class internally calls persist() method which in turn uses sparkSession.sharedState.cacheManager.cacheQuery to cache the result set of RDD. Let’s look at an example.

| cachedRdd = rdd.cache() |
| --- |

### **RDD Persist**

PySpark persist() method is used to store the RDD to one of the storage levels MEMORY\_ONLY,MEMORY\_AND\_DISK, MEMORY\_ONLY\_SER, MEMORY\_AND\_DISK\_SER, DISK\_ONLY, MEMORY\_ONLY\_2,MEMORY\_AND\_DISK\_2 and more.

PySpark persist has two signatures: the first signature doesn’t take any argument which by default saves it to <strong>MEMORY\_ONLY</strong> storage level and the second signature which takes StorageLevel as an argument to store it to different storage levels.

| import pyspark  dfPersist = rdd.persist(pyspark.StorageLevel.MEMORY\_ONLY)  dfPersist.show(false) |
| --- |

### **RDD Unpersist**

PySpark automatically monitors every persist() and cache() calls you make and it checks usage on each node and drops persisted data if not used or by using least-recently-used (LRU) algorithm. You can also manually remove using the unpersist() method. unpersist() marks the RDD as non-persistent, and removes all blocks for it from memory and disk.

| rddPersist2 = rddPersist.unpersist() |
| --- |