**RDD:**

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

Use parallelize to convert an array to RDD.

rdd = sc.parallelize(["b", "a", "c"])

**Map:**

Return a new RDD by applying a function to each element of this RDD.

mrdd = sc.parallelize(["b", "a", "c"])

sorted(mrdd.map(lambda x: (x, 1)).collect())

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

**Syntax:**

lambda arguments : expression

**FlatMap:**

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

fmrdd = sc.parallelize([2, 3, 4])

sorted(fmrdd.flatMap(lambda x: range(1, x)).collect())

**Range: range(start, end=None, step=1, numSlices=None)**

Create a new RDD of int containing elements from start to end (exclusive), increased by **step** every element.

sorted(fmrdd.flatMap(lambda x: [(x, x), (x, x)]).collect())

**Filter:**

Return a new RDD containing only the elements that satisfy a predicate.

frdd = sc.parallelize([1, 2, 3, 4, 5])

frdd.filter(lambda x: x % 2 == 0).collect()

**GroupBy:**

Return an RDD of grouped items.

gbrdd = sc.parallelize([1, 1, 2, 3, 5, 8])

gbresult = gbrdd.groupBy(lambda x: x % 2).collect()

sorted([(x, sorted(y)) for (x, y) in gbresult])

In python, the sorted() function returns a sorted list of the specified iterable object.

**Syntax:**

sorted(iterable, key=key, reverse=reverse)

**GroupByKey:**

Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.

gbkrdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(gbkrdd .groupByKey().mapValues(len).collect())

sorted(gbkrdd .groupByKey().mapValues(list).collect())

**mapValues:**

Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD’s partitioning.

x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])

def f(x):

return len(x)

x.mapValues(f).collect()

**mapPartition:**

Return a new RDD by applying a function to each partition of this RDD.

mprdd = sc.parallelize([1, 2, 3, 4], 2)

def f(iterator):

yield sum(iterator)

mprdd.mapPartitions(f).collect()

**mapPartitionwithIndex:**

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

mprdd = sc.parallelize([1, 2, 3, 4], 4)

def f(splitIndex, iterator):

yield splitIndex

mprdd.mapPartitionsWithIndex(f).sum()

**Glom:**

Return an RDD created by coalescing all elements within each partition into a list.

rdd = sc.parallelize([1, 2, 3, 4], 2)

sorted(rdd.glom().collect())

**ReducebyKey:**

Merge the values for each key using an associative and commutative reduce function.

from operator import add

rbkrdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(rbkrdd.reduceByKey(add).collect())

**Sample:**

Return a sampled subset of this RDD.

slrdd = sc.parallelize(range(100), 4)

slrdd.sample(False, 0.1, 81).collect()

**Union:**

Return the union of this RDD and another one.

uxrdd = sc.parallelize([1, 2, 3])

uyrdd = sc.parallelize([4, 5, 6])

uxrdd.union(uyrdd).collect()

**Join:**

Return an RDD containing all pairs of elements with matching keys in self and other.

x = sc.parallelize([("a", 1), ("b", 4)])

y = sc.parallelize([("a", 2), ("a", 3)])

sorted(x.join(y).collect())

**Distinct:**

Return a new RDD containing the distinct elements in this RDD.

sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())

**Coalesce:**

Return a new RDD that is reduced into numPartitions partitions.

sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()

sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()

**Keyby:**

Creates tuples of the elements in this RDD by applying f.

x = sc.parallelize(range(0,3)).keyBy(lambda x: x\*x)

y = sc.parallelize(zip(range(0,5), range(0,5)))

[(x, list(map(list, y))) for x, y in sorted(x.cogroup(y).collect())]

**PartitionBy:**

Return a copy of the RDD partitioned using the specified partitioner.

pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1]).map(lambda x: (x, x))

sets = pairs.partitionBy(2).glom().collect()

len(set(sets[0]).intersection(set(sets[1])))

**Zip:**

Zips this RDD with another one, returning key-value pairs with the first element in each RDD second element in each RDD, etc. Assumes that the two RDDs have the same number of partitions and the same number of elements in each partition (e.g. one was made through a map on the other).

zx = sc.parallelize(range(0,5))

zy = sc.parallelize(range(1000, 1005))

zx.zip(zy).collect()

**Actions:**

Actions are RDD methods that return a value to a driver program.

**Reduce:**

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

from operator import add

sc.parallelize([1, 2, 3, 4, 5]).reduce(add)

sc.parallelize((2 for \_ in range(10))).map(lambda x: 1).cache().reduce(add)

sc.parallelize([]).reduce(add)

**Reduce By Key:**

Merge the values for each key using an associative and commutative reduce function.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

Output will be partitioned with *numPartitions* partitions, or the default parallelism level if *numPartitions* is not specified. Default partitioner is hash-partition.

from operator import add

rbkrdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(rbkrdd.reduceByKey(add).collect())

**Aggregate:**

Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral “zero value.”

The functions op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object allocation; however, it should not modify t2.

The first function (seqOp) can return a different result type, U, than the type of this RDD. Thus, we need one operation for merging a T into an U and one operation for merging two U

[**RDD.aggregate**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.aggregate.html#pyspark.RDD.aggregate)(zeroValue, seqOp, combOp)

seqOp = (lambda x, y: (x[0] + y, x[1] + 1))

combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))

sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)

sc.parallelize([]).aggregate((0, 0), seqOp, combOp)

**Take:**

Take the first “num” elements of the RDD. This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver’s memory.

sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)

**Collect:**

Return a list that contains all of the elements in this RDD. This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver’s memory.

Rdd.collect()

**Max:**

Find the maximum item in this RDD.

maxrdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])

maxrdd.max()

maxrdd.max(key=str)

**Sum:**

Add up the elements in this RDD.

sc.parallelize([1.0, 2.0, 3.0]).sum()

**Count:**

Return the number of elements in this RDD.

sc.parallelize([2, 3, 4]).count()

**CountByKey:**

Count the number of elements for each key, and return the result to the master as a dictionary.

cntrdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(cntrdd.countByKey().items())

**LookUp:**

Return the list of values in the RDD for key “key”. This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

pairRdd = sc.parallelize([("a", 1), ("b", 2), ("c", 3), ("a", 11), ("b", 22), ("a",1)])

pairRdd.lookup(“a”)

**Broadcast Variable:**

broadcastVar = sc.broadcast([1, 2, 3])

broadcastVar.value

broadcastVar.unpersist()

broadcastVar.destroy()

**Accumulator:**

accum = sc.accumulator(0)

accum

sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))

accum.value

accum1 = sc.accumulator(0)

def g(x):

r=accum1.add(x)

return r

data= sc.parallelize([2, 3, 3, 3])

res=data.map(g)

accum1.value

res.collect()

accum1.value