PySpark RDD Activity

Learning outcomes:

Understand Transformations, Actions in Spark.
Understand and execute the Spark commands written in Python.

This document explains the basic operations on RDDs. There are two kind of operations that can be performed on RDD namely **Transformations** and **Actions**.

Spark introduces the concept of an **RDD (Resilient Distributed Dataset)**, an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel. An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program.

RDDs support two types of operations:

Transformations are operations (such as map, filter, join, union, and so on) that are performed on an RDD and which yield a new RDD containing the result.

Actions are operations (such as reduce, count, first, and so on) that return a value after running a computation on an RDD.

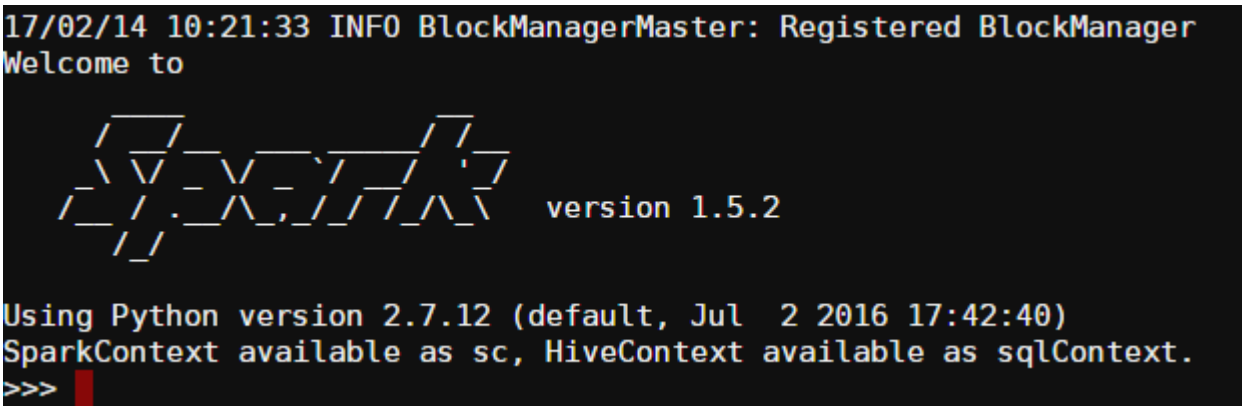
Transformations in Spark are "lazy", meaning that they do not compute their results right away. Instead, they just "remember" the operation to be performed and the dataset (e.g., file) to which the operation is to be performed. The transformations are only actually computed when an action is called and the result is returned to the driver program. This design enables Spark to run more efficiently. For example, if a big file was transformed in various ways and passed to first action, Spark would only process and return the result for the first line, rather than do the work for the entire file.

To open the PySpark shell type 'pyspark' at the command prompt using the terminal in the VM/Cluster:

Command:

$pyspark

The output should be as the following



Spark Context :

Type 'sc' in the pyspark shell

>>>sc

<pyspark.context.SparkContext object at 0x39e3150>

Parallelize (Constructs Rdd):

Using the spark context, Call the parallelize method on the Spark Context. Passing a List object into parallelize method. An RDD is created and stored in the variable intRdd.

Command(s) to Execute from the pyspark shell

>>>intRdd = sc.parallelize(*[10, 20, 30, 40, 50])*

Map (Transformation):

Using the intRdd created in the above step, perform a map operation using the lambda function on the Rdd. Map is a **Transformation.**

Command(s) to Execute from the pyspark shell

>>>mapRdd = intRdd.map(lambda x: x*2)

>>>***mapRdd.collect()***

[20, 40, 60, 80, 100]

Command(s) to Execute from the pyspark shell

>>>***sqrRdd =intRdd.map(lambda x: x*x)***

>>>sqrRdd.collect()

filter(Transformation):

The filter operation evaluates a boolean function for each data item of the RDD and puts the items for which the function returned true into the resulting RDD. Filter is a **Transformation. Collect** is an Action.

Command(s) to Execute from the pyspark shell

>>>***numRdd = sc.parallelize([11,12,13,14,15,16,17,18])***

>>>***filterRdd1 = numRdd.filter(lambda x : x%2 == 1)***

>>>filterRdd1.collect()

>>>filterRdd2 = numRdd.filter(lambda x : x%2 == 0)

>>>filterRdd2.collect()

reduceByKey (Transformation):

Spark RDD reduceByKey function merges the values for each key using an associative reduce function. Basically reduceByKey function works only for RDDs which contains key and value pairs kind of elements (i.e. RDDs having tuple or Map as a data element).

Command(s) to Execute from the pyspark shell

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

>>>y = x.reduceByKey(lambda a, b: a + b)

>>>y.collect()

[***('a', 3), ('b', 5)]***

flatMap (Transformation) :

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

First perform the Map Operation in pySpark on the Sentences.

Command(s) to Execute from the pyspark shell

>>>***sentRdd = sc.parallelize(["Welcome to INSOFE", "This is Lab Session", "We are doing pySpark Activity"])***

>>>listRdd = sentRdd.map(lambda x: x.split(' '))

>>>listRdd.collect()

Now, perform flatMap operation on sentRdd created above that will return List of Words in the following case.

Command(s) to Execute from the pyspark shell

>>>worListRdd = sentRdd.flatMap(lambda x: x.split(' '))

>>>worListRdd.collect()

Map &flatMap difference:

map: It returns a new RDD by applying a function to each element of the RDD. Function in map can return only one item.

flatMap: Similar to map, it returns a new RDD by applying a function to each element of the RDD, but output is flattened.

Also, function in flatMap can return a list of elements (0 or more)

Example1:-

>>>sc.parallelize([3,4,5]).map(lambda x: range(1,x)).collect()

Output:

[[1, 2], [1, 2, 3], [1, 2, 3, 4]]

>>>sc.parallelize([3,4,5]).flatMap(lambda x: range(1,x)).collect()

Output: notice o/p is flattened out in a single list

[1, 2, 1, 2, 3, 1, 2, 3, 4]

Example 2:

>>>sc.parallelize([3,4,5]).map(lambda x: [x, x*x]).collect()

Output:

[[3, 9], [4, 16], [5, 25]]

>>>sc.parallelize([3,4,5]).flatMap(lambda x: [x, x*x]).collect()

Output: notice flattened list

[3, 9, 4, 16, 5, 25]

groupByKey(Transformation):

Spark groupByKey function returns a new RDD. The returned RDD gives back an object which allows to iterate over the results. The results of groupByKey returns a list by calling list() on values.

Command(s) to Execute from the pyspark shell

```
>>> example = sc.parallelize([('x',1), ('x',1), ('y', 1), ('z', 1)])

>>>example.groupByKey()

PythonRDD[28] at RDD at PythonRDD.scala:43

>>>itrRdd = example.groupByKey()

>>>itrRdd.collect()

>>>itrRdd.map(lambda x : (x[0], list(x[1]))).collect()
```

groupBy (Transformation) :

groupBy function returns an RDD of grouped items. This operation will return the new RDD which basically is made up with a KEY (which is a group) and list of items of that group (in a form of Iterator). Order of element within the group may not same when you apply the same operation on the same RDD over and over.

Command(s) to Execute from the pyspark shell

```
>>>namesRdd = sc.parallelize(["Joseph", "Jimmy", "Tina",
"Thomas","James","Cory","Christine", "Jackeline", "Juan"])
>>> result =namesRdd.groupBy(lambda word: word[0]).collect()

>>>sorted([(x, sorted(y)) for (x, y) in result])
```

Note: As noticed above groupBy is applied to single set of values. However, groupByKey is applied to pair of values in Rdd or pairedRdds.

mapValues (Transformation) :

Apply a function to each value of a pair RDD without changing the key.

Command(s) to Execute from the pyspark shell

```
>>>namesRdd = sc.parallelize(["dog", "tiger", "lion", "cat", "panther", "eagle"])

>>>pairRdd = namesRdd.map(lambda x :(len(x), x))

>>>result =  pairRdd.mapValues(lambda y: "Animal name is " + y)

>>>result.collect()
```

Output : [(3, 'Animal name is dog'), (5, 'Animal name is tiger'), (4, 'Animal name is lion'), (3, 'Animal name is cat'), (7, 'Animal name is panther'), (5, 'Animal name is eagle')]

join (pair Rdd Transformation):

Command(s) to Execute from the pyspark shell

```
>>>rdd = sc.parallelize([("red",20),("red",30),("blue", 100)])

>>>rdd2 = sc.parallelize([("red",40),("red",50),("yellow", 10000)])

>>>rdd.join(rdd2).collect()
```

Output :[('red', (20, 40)), ('red', (20, 50)), ('red', (30, 40)), ('red', (30, 50))]

With PairRDDs:

Inner join & Output Join:

Command(s) to Execute from the pyspark shell

>>>rdd1 = sc.parallelize([("Mercedes", "E-Class"), ("Toyota", "Corolla"), ("Renault", "Duster")])

>>>rdd2 = sc.parallelize([("Mercedes", "C-Class"), ("Toyota", "Prius"), ("Toyota", "Etios")])

>>>innerJoinRdd = rdd1.join(rdd2)

Output: [('Mercedes', ('E-Class', 'C-Class')), ('Toyota', ('Corolla', 'Prius')), ('Toyota', ('Corolla', 'Etios'))]

>>>outerJoinRdd = rdd1.leftOuterJoin(rdd2)

Output:[('Mercedes', ('E-Class', 'C-Class')), ('Toyota', ('Corolla', 'Prius')), ('Toyota', ('Corolla', 'Etios')), ('Renault', ('Duster', None))]

Union:

Combines the values in various Rdds to form a cohesive unit

Command(s) to Execute from the pyspark shell

>>> d1= [('k1', 1), ('k2', 2), ('k3', 5)]

>>> d1

[('k1', 1), ('k2', 2), ('k3', 5)]

>>> d2= [('k1', 3), ('k2',4), ('k4', 8)]

>>> d2

[('k1', 3), ('k2', 4), ('k4', 8)]

```
>>> rdd1 = sc.parallelize(d1)

>>> rdd1.collect()

[('k1', 1), ('k2', 2), ('k3', 5)]

>>> rdd2 = sc.parallelize(d2)

>>> rdd2.collect()

[('k1', 3), ('k2', 4), ('k4', 8)]

>>> rdd3 = rdd1.union(rdd2)

>>> rdd3.collect()

[('k1', 1), ('k2', 2), ('k3', 5), ('k1', 3), ('k2', 4), ('k4', 8)]

>>> rdd4 = rdd3.reduceByKey(lambda x,y: x+y)

>>> rdd4.collect()

[('k3', 5), ('k2', 6), ('k1', 4), ('k4', 8)]
```

Reading a file in PySpark:

To read a file in spark using the spark context, use "sc.textFile(<path to File>)"

Command(s) to Execute from the pyspark shell

Note: The file should be placed in the local file system.

Reading the file from the local file system

```
>>rdd1= sc.textFile('file:///home/yugandhar1458/sample.txt')
```

Note: The file should be placed in the hdfs file system.

Reading the file from the hdfs location

```
>>>rdd2 = sc.textFile('hdfs://ip-172-31-53-48/user/yugandhar1458/dir1/hdfsdata.txt')
```

collect (Action):

Collect action returns the results or the value. When an action is called transformations are executed.

Command(s) to Execute from the pyspark shell

>>>rdd1 = sc.textFile('hdfs://ip-172-31-53-48/user/yugandhar1458/dir1/hdfsdata.txt')

>>>rdd1.collect()

Output:

[u'hello world', u'hello world1', u'hello world2']

first (Action):

Returns the first line of the file.

Command(s) to Execute from the pyspark shell

>>>rdd1=sc.textFile('hdfs://ip-172-31-53-48/user/yugandhar1458/dir1/hdfsdata.txt')

>>>rdd1.first()

Output:

u'hello world'

take(Action):

Extracts the first *n* items of the RDD and returns them as an list.

Command(s) to Execute from the pyspark shell

>>>rdd1=sc.textFile('hdfs://ip-172-31-53-48/user/yugandhar1458/dir1/hdfsdata.txt')

>>>rdd1.take(3)

takeOrdered(Action):

Orders the data items of the RDD using their inherent implicit ordering function and returns the first *n* items as an array.

Command(s) to Execute from the pyspark shell

>>>rdd1 = sc.parallelize(["dog", "cat", "ape", "salmon", "gnu"***])***

>>>rdd1.takeOrdered(3)

reduce (Action):

This function provides the well-known *reduce* functionality in Spark. Please note that any function *f* you provide, should be commutative in order to generate reproducible results.

Command(s) to Execute from the pyspark shell

>>>intVals = range(1,15)

>>>intVals

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>>numRdd = sc.parallelize(intVals)

>>>cSum = numRdd.reduce(lambda a, b: a + b)

>>>cSum

Word Count Program in pySpark:

Command(s) to Execute from the pyspark shell

>>> file =sc.textFile('hdfs://ip-172-31-53-48/user/yugandhar1458/dir1/hdfsdata.txt')

>>>file.collect()

>>>word_tokens = file.flatMap(lambda line: line.split(' ')).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

>>>word_tokens.collect()

>>>word_tokens.count()

>>>sorted = word_tokens.sortByKey()

>>>sorted.collect()

>>>sortedDescending = word_tokens.sortByKey(False)

>>>sortedDescending.collect()

RDD Transformations

map() - Return a new RDD by applying a function to each element of this RDD.

pipe() - Return an RDD created by piping elements to a forked external process.

reduceByKey() - Merge the values for each key using an associative 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 hash-partitioned with **numPartitions** partitions, or the default parallelism level if **numPartitions** is not specified.

intersection() - Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.

filter() - Return a new RDD containing only the elements that satisfy a predicate.

repartition() - Return a new RDD that has exactly numPartitions partitions.

cartesian() - Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements *(a, b)* where **a** is in **self** and **b** is in **other**.

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

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

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

mapPartitions() - Return a new RDD by applying a function to each partition of this RDD.

join() - Return an RDD containing all pairs of elements with matching keys in **self** and **other**.

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in **self** and (k, v2) is in **other**.

distinct() - Return a new RDD containing the distinct elements in this RDD.

sample() - Return a sampled subset of this RDD.

union() - Return the union of this RDD and another one.

coalesce() - Return a new RDD that is reduced into *numPartitions* partitions.

sortByKey() - Sorts this RDD, which is assumed to consist of (key, value) pairs.

Most transformations are element-wise.

RDD Actions

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

first() - Return the first element in this RDD.

takeSample() - Return a fixed-size sampled subset of this RDD.

takeOrdered() - Get the N elements from a RDD ordered in ascending order or as specified by the optional key function.

foreach() - Applies a function to all elements of this RDD.

collect() - Return a list that contains all of the elements in this RDD.

take() - Take the first num elements of the RDD.

It works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

saveAsTextFile() - Save this RDD as a text file, using string representations of elements.

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

count() - Return the number of elements in this RDD.

saveAsSequenceFile() - Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system

Actions trigger execution and transformations are applied.

http://spark.apache.org/docs/1.5.2/api/python/pyspark.html