

Basic API of Spark

Lab 3

October 26th, 2017

Jonghyun Bae(jonghbae@snu.ac.kr)

Computer Science and Engineering

Seoul National University

Index

- Spark execution model
- RDD creation
- Transformation function
- Action function
- Exercise

Before we start...

■ Please connect your VM using SSH

```
1 # Please your public IP address in xxx.xxx.xxx.xxx
2 student@computer:~$ ssh -X -i bde3.pem ubuntu@xxx.xxx.xxx.xxx
3 Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-125-generic x86_64)
4 [...snipp...]
5 ubuntu@ip-x-x-x:~$
```

Before we start...

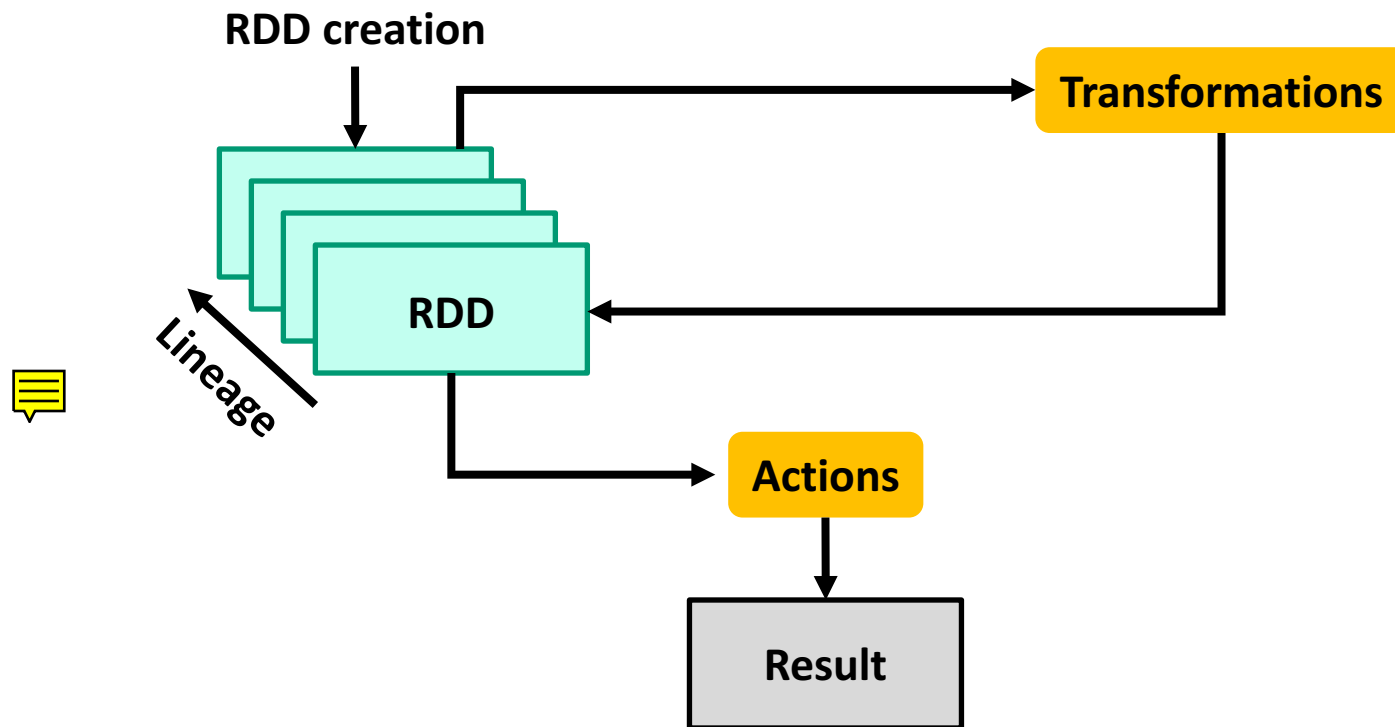
- Please start your Hadoop & Spark
- Open your pyspark shell

```
1  ubuntu@ip-x-x-x:~$ cd $HADOOP_HOME
2  ubuntu@ip-x-x-x:~/hadoop-2.7.4$ sbin/start-dfs.sh
   ubuntu@ip-x-x-x:~/hadoop-2.7.4$ cd $SPARK_HOME
2  ubuntu@ip-x-x-x:~/spark-2.1.0$ sbin/start-all.sh
3  ubuntu@ip-x-x-x:~/spark-2.1.0$ bin/pyspark
4  [... snipp ...]
5  >>>
```

Spark execution model (1)

■ Lazy evaluation

- There are no computation during "Transformation" function is entered
- Real computation is occurred when "Action" function is entered




* image from <http://data-flair.training/blogs/apache-spark-lazy-evaluation/>

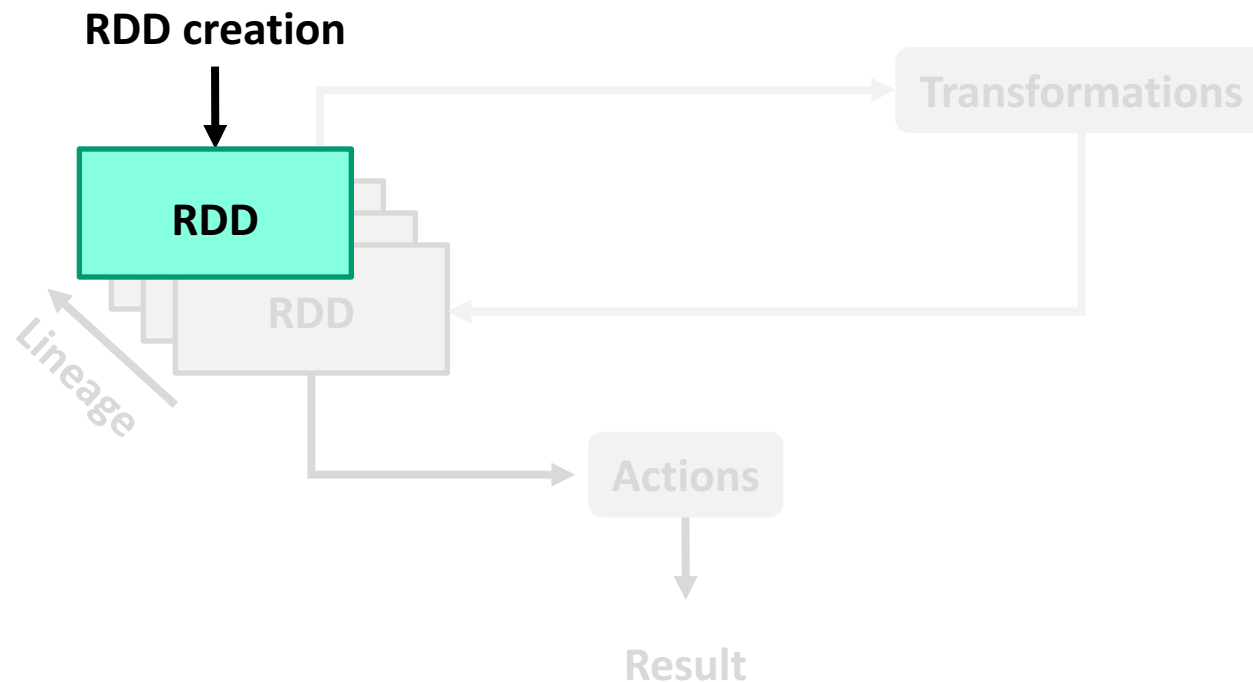
Spark execution model (2)

■ Why lazy evaluation?

- Increase manageability
- Save computation and increase speed
- Reduce complexities
- Optimize

RDD creation (1)

- **Resilient Distributed Dataset (RDD)*** 
 - Set of function lineage and data: lazy execution
 - Alpha and omega of Apache Spark



Matei Zaharia et al., Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12).

RDD creation (2)

- **SparkSession (a.k.a SparkContext in scala)**

- (...)

- SparkSession available as 'spark'.



- **Main entry point for Spark functionality**

- **Generally, used for creating RDD from filesystem**

RDD creation (3 - 1)

■ `parallelize(c)`

- Distribute a local Python collection to form an RDD

```
1 >>> test = sc.parallelize([0, 2, 3, 4, 6])
```



```
2 >>> test.collect()
```

```
3 [0, 2, 3, 4, 6]
```

```
4 >>>
```

RDD creation (3 - 2)

■ `parallelize(c, numSlices)`

- Distribute a local Python collection to form an RDD

```
1 >>> test2 = sc.parallelize([0, 2, 3, 4, 6], 5)
2 >>> test2.collect()
3 [0, 2, 3, 4, 6]
4 >>>
```



RDD creation (4 - 1)

■ range(start)

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

```
1 >>> sc.range(8).collect()
2 [0, 1, 2, 3, 4, 5, 6, 7]
3 >>>
```

RDD creation (4 - 2)

■ range(start, end)

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

```
1 >>> sc.range(2, 8).collect()
```

```
2 [2, 3, 4, 5, 6, 7]
```

```
3 >>>
```

RDD creation (4 - 3)

■ range(start, end, step)

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

```
1 >>> sc.range(2, 8, 2).collect()
```

```
2 [2, 4, 6]
```

```
3 >>>
```

RDD creation (5 - 1)

■ `textFile(name, minPartitions)`

- Read a text file from a **local file system**, HDFS

```
1 >>> sc.textFile("/home/ubuntu/spark-2.1.0/README.md").collect()
2 [u'# Apache Spark', u'', u'Spark is a (...)]
3 >>>
```

RDD creation (5 - 2)

■ `textFile(name, minPartitions)`

- Read a text file from a local file system, *HDFS*

```
1 >>> sc.textFile("hdfs://localhost:9000/input/sample1.txt").collect()
2 [(...)]
3 >>>
```

RDD creation (6)

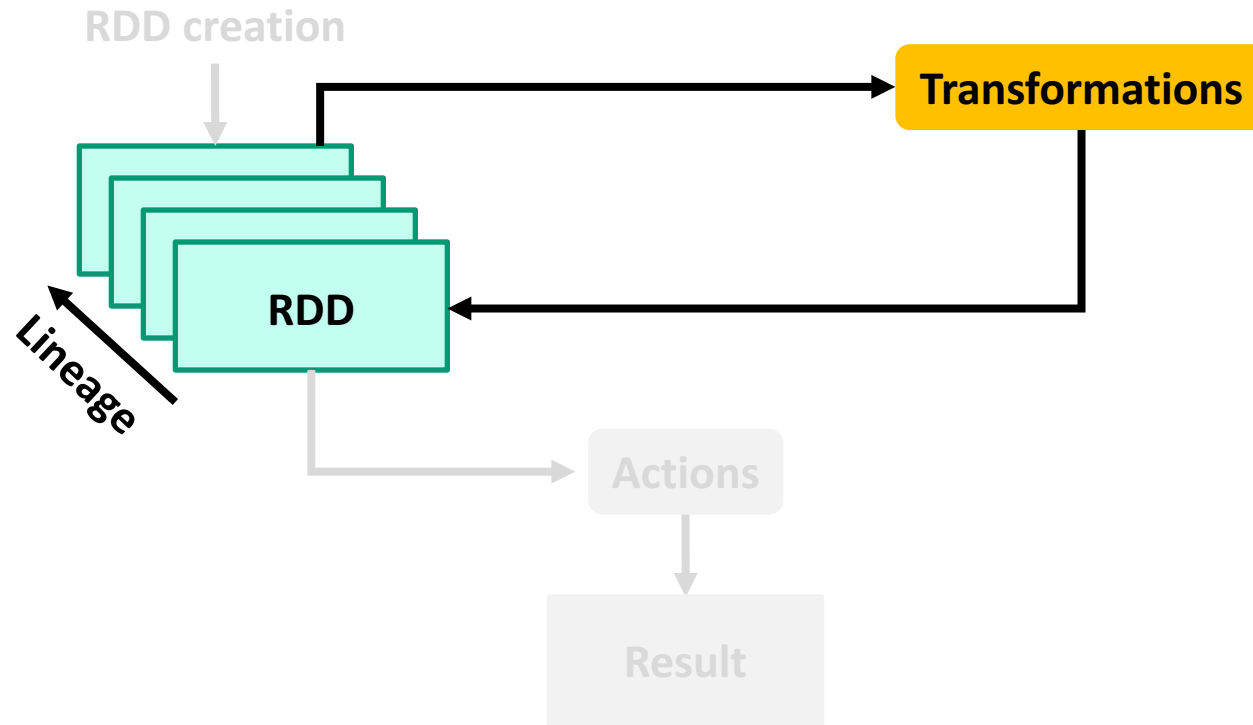
- **wholeTextFiles(path, minPartitions)**
 - Read a directory of text files from a local file system, HDFS

```
1 >>> sc.wholeTextFiles("hdfs://localhost:9000/input").collect()
2 [(...)]
3 >>>
```


Transformation function (1)



- Functions that take a RDD as the input and produce one or many RDDs as the output



Transformation function (2 - 1)

■ map(f)

- Return a new RDD by applying a function to each element of this RDD.
- Most frequently used in Spark

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

```
2 >>> rdd.map(lambda x: (x, 1)).collect()
```

```
[( 'b', 1), ( 'c', 1), ( 'a', 1)]
```



```
3 >>>
```



Transformation function (2 - 2)

■ map(f)

- Return a new RDD by applying a function to each element of this RDD.
- Most frequently used in Spark

```
1 >>> rdd = sc.parallelize(["I am a boy you are a girl", "how are you"])
2 >>> rdd.map(lambda line: line.split(" ")).collect()
[['I', 'am', 'a', 'boy', 'you', 'are', 'a', 'girl'], ['how', 'are', 'you']]
3 >>>
```



Transformation function (2 - 3)

■ map(f)

- Return a new RDD by applying a function to each element of this RDD.
- Most frequently used in Spark

```
1 >>> rdd = sc.parallelize([('a', 1), ('b', 2), ('c', 3)])  
2 >>> rdd.map(lambda (k, v): (k, v * 2)).collect()  
[('a', 2), ('b', 4), ('c', 6)]  
3 >>>
```



Transformation function (3 - 1)

■ flatMap(f)

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

```
1 >>> rdd = sc.parallelize([2, 3, 4])  
2 >>> rdd.flatMap(lambda x: range(1, x)).collect()  
  
[1, 1, 2, 1, 2, 3]  
  
>>>
```

```
2 >>> rdd.map(lambda x: range(1, x)).collect()  
  
[[1], [1, 2], [1, 2, 3]]
```

Transformation function (3 - 2)

■ flatMap(f)

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

```
1 >>> rdd = sc.parallelize(["I am a boy you are a girl", "how are you"])
2 >>> rdd.flatMap(lambda line: line.split(" ")).collect()
['I', 'am', 'a', 'boy', 'you', 'are', 'a', 'girl', 'how', 'are', 'you']
3 >>>
```

Transformation function (3 - 3)

■ flatMap(f)

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

```
1 >>> rdd = sc.parallelize(["I am a boy you are a girl", "how are you"])
2 >>> rdd.map(lambda line: line.split(" ")) \
3 ...     .flatMap(lambda x: [(x[i], x[i+1]), 1) for i in range (0, len(x)-1)])
4 ...     .collect()
['I', 'am', 'a', 'boy', 'you', 'are', 'a', 'girl', 'how', 'are', 'you']
5 >>>
```

Transformation function (4)

■ reduceByKey(f)

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

```
1 >>> from operator import add
2 >>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
3 >>> rdd.reduceByKey(add).collect()
[('a', 2), ('b', 1)]
4 >>> rdd.reduceByKey(lambda x, y: x + y).collect()
[('a', 2), ('b', 1)]
```


Transformation function (5 - 1)

■ sortBy(f, ascending = True)

- Sorts this RDD by the given keyfunc

```
1 >>> tmp = [('egg', 1), ('apple', 2), ('car', 3), ('dad', 4), ('bear', 5)]
2 >>> rdd = sc.parallelize(tmp)
3 >>> rdd.sortBy(lambda x: x[0], True).collect()
    [('apple', 2), ('bear', 5), ('car', 3), ('dad', 4), ('egg', 1)]
4 >>> rdd.sortBy(lambda x: x[0], False).collect()
    [('egg', 1), ('dad', 4), ('car', 3), ('bear', 5), ('apple', 2)]
5 >>> rdd.sortBy(lambda x: x[1], True).collect()
    [('egg', 1), ('apple', 2), ('car', 3), ('dad', 4), ('bear', 5)]
6
```

Transformation function (5 - 2)

■ sortByKey(ascending = True)

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

```
1 >>> tmp = [('egg', 1), ('apple', 2), ('car', 3), ('dad', 4), ('bear', 5)]
2 >>> rdd = sc.parallelize(tmp)
3 >>> rdd.sortByKey(True).collect()
[('apple', 2), ('bear', 5), ('car', 3), ('dad', 4), ('egg', 1)]
4 >>> rdd.sortByKey(False).collect()
[('egg', 1), ('dad', 4), ('car', 3), ('bear', 5), ('apple', 2)]
```

Transformation function (6)

■ filter(f)

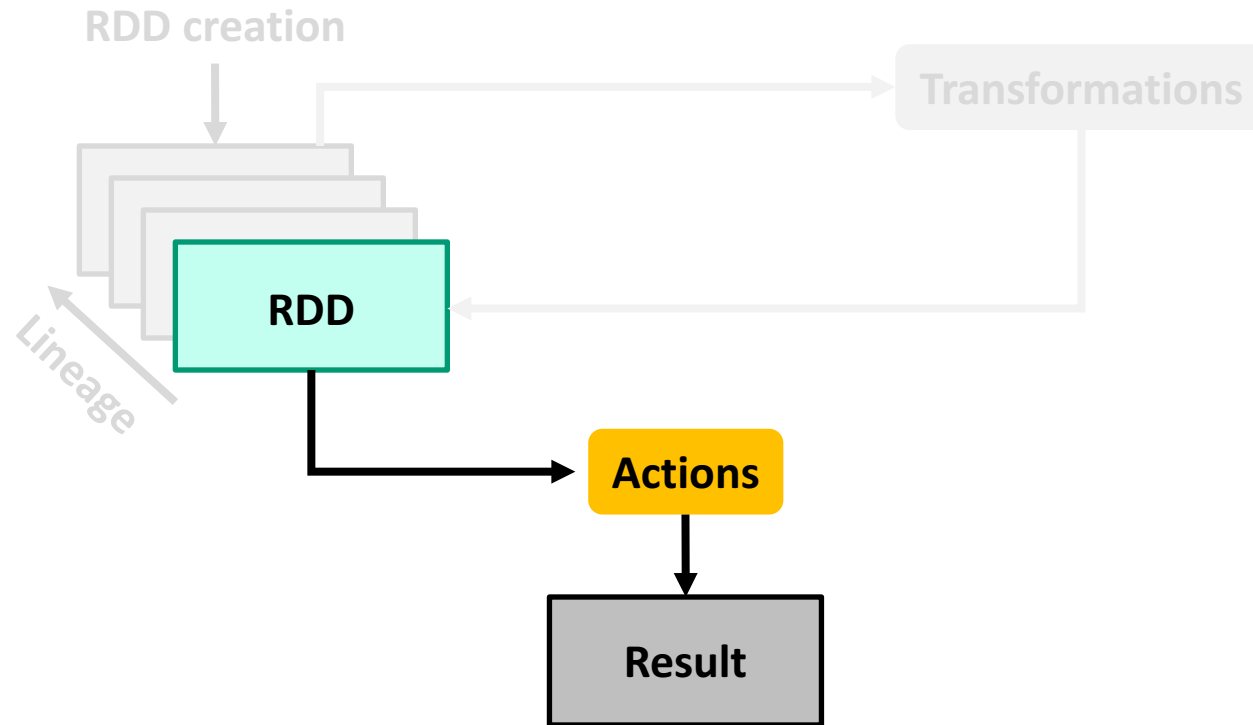
- Return a new RDD containing only the elements that satisfy a predicate.

```
1 >>> rdd = sc.parallelize([1, 2, 3, 4, 5])  
2 >>> rdd.filter(lambda x: x % 2 == 0).collect()  
  
[2, 4]  
3 >>>
```



Action function (1)

- Produce result values
- Real computation is occurred



Action function (2)

■ collect()

- Return a list that contains all of the elements in this RDD
- **WARNING: should only be used if the resulting array is expected to be small**

Action function (3)

■ count()

- Return the number of elements in this RDD

```
1 >>> rdd = sc.parallelize([1, 2, 3, 4, 5])
```

```
2 >>> rdd.count()
```

```
5
```

```
3 >>>
```

Action function (4)

■ `saveAsTextFile(path)`

- Save this RDD as a text file, using string representations of elements

```
1 >>> rdd = sc.parallelize(range(10))  
2 >>> rdd.saveAsTextFile("range")  
3 >>> rdd.saveAsTextFile("hdfs://localhost:9000/range")
```

Action function (5)

■ take(num)

- Take the first num elements of the RDD

```
1 >>> rdd = sc.parallelize(range(10))
```

```
2 >>> rdd.take(2)
```



```
[0, 1]
```

```
3 >>>
```


Exercise (1)

- Find out how many times the “**cracking**” and “**bucket**” were used
- Input file
 - `hdfs://localhost:9000/input/(sample1 ~ sample3.txt)`
- Please save your result in “**exercise3_1.txt**”

```
2 ubuntu@ip-x-x-x:~/hadoop-2.7.4$ vim exercise3_1.txt
```

Exercise (2)

- Find the unique words and sort alphabetical order
- Example
 - I am a boy and I am a man
 - (I, 1), (am, 1), (a, 1), (boy, 1), (and, 1), (man, 1)
- Input file
 - hdfs://localhost:9000/input/(sample1 ~ sample3.txt)
- Please save your result in “**exercise3_2.txt**”

```
2 ubuntu@ip-x-x-x:~/hadoop-2.7.4$ vim exercise3_2.txt
```