CSCI 5702/7702– Fall 2019 – Assignment 0: Spark Tutorial

Due: 9/9/2019 – 8:59 pm

The purpose of this tutorial is (1) to get you started with Spark and (2) to get you acquainted with the code and homework submission system. Completing the tutorial is optional, but by handing in the results in time students will earn extra credit (1% will be added to your total grade at the end of the semester). This tutorial is to be completed individually.

Setting up a stand-alone Spark instance

- Download and install Spark 2.4.3 on your machine
- Unpack the compressed TAR ball.
- Download and install JDK. Note that Spark is incompatible with some JDK versions. To be on the safe side, download JDK 8.
- If you plan to use Scala, you will also need to download and install Scala.
- If you plan to use python **(recommended)**, you will need python 3.4 or higher.

Running the Spark shell

Spark gives you two different ways to run your applications. The easiest is using the Spark shell, a REPL that lets you interactively compose your application. The Spark shell supports two languages: Scala/Java and python. In this tutorial, we will only discuss using python and Scala. We highly recommend **python as both the language itself and the python Spark API are straightforward**.

You can also use IDEs such as Jupyter Notebook. However, the installation process would be more complicated. If you use online tutorials or youtube videos, make sure that the videos/spark version used in the tutorials are not too old since they may not be valid for the newer versions of Spark.

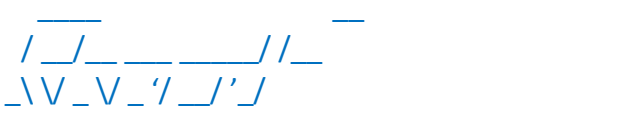
2.1 Spark Shell for Python

To start the Spark shell for python, do the following:

1. Open a terminal window on Mac or Linux or a command window on Windows.
2. Change into the directory where you unpacked the Spark binary.
3. Run: bin/pyspark on Mac or Linux or bin\pyspark on Windows.

As the Spark shell starts, you may see large amounts of logging information displayed on the screen, possibly including several warnings. You can ignore that output for now. Regardless, the startup is complete when you see something like:

Welcome to



Using Python version *** (default, ***) SparkSession

available as 'spark'. >>>

The Spark shell is a full python interpreter and can be used to write and execute regular python programs. For example:

>>> print "Hello!" Hello!

The Spark shell can also be used to write Spark applications in python. To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

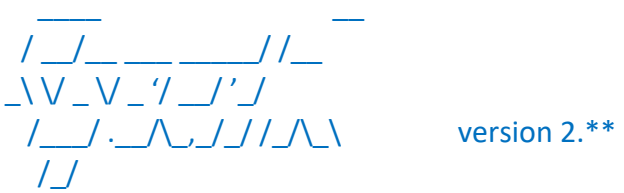
2.2 Spark Shell for Scala

To start the Spark shell for Scala, do the following:

1. Open a terminal window on Mac or Linux or a command window on Windows.
2. Change into the directory where you unpacked the Spark binary.
3. Run: bin/spark-shell on Mac or Linux or bin\spark-shell on Windows.

As the Spark shell starts, you may see large amounts of logging information displayed on the screen, possibly including several warnings. You can ignore that output for now. Regardless, the startup is complete when you see something like:

Welcome to



Using Scala version ***

(Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131)

Type in expressions to have them evaluated.

Type :help for more information.

scala>

The Spark shell is a full Scala interpreter and can be used to write and execute regular Scala programs. For example:

```
scala> print("Hello!") Hello!
```

The Spark shell can also be used to write Spark applications in Scala. (Surprise!) To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

3. Word Count

The typical "Hello, world!" app for Spark applications is known as word count. The map/reduce model is particularly well suited to applications like counting words in a document. In this section, you will see how to develop a word count application in python, Java, and Scala. Before reading this section, you should read through the Spark programming guide if you haven't already.

All operations in Spark operate on data structures called RDDs, Resilient Distributed Datasets. An RDD is nothing more than a collection of objects. If you read a file into an RDD, each line will become an object (a string, actually) in the collection that is the RDD. If you ask Spark to count the number of elements in the RDD, it will tell you how many lines are in the file. If an RDD contains only two-element tuples (i.e., key-value pairs), the RDD is known as a "pair RDD" and offers some additional functionality. The first element of each tuple is treated as a key and the second element as a value. Note that all RDDs are immutable, and any operations that would mutate an RDD will instead create a new RDD.

4.1 Word Count in python

For this example, you will create your application in an editor instead of using the Spark shell. The first step of every such Spark application is to create a Spark context:

```
import re import sys from pyspark

import SparkConf, SparkContext

conf = SparkConf()

sc =SparkContext(conf=conf)
```

Next, you'll need to read the target file into an RDD:

```
lines = sc.textFile(sys.argv[1])
```

Note that you can also hardcode the filepath in your script instead of using argv if you use an IDE. You now have an RDD filled with strings, one per line of the file.

Next, you'll want to split the lines into individual words:

```
words = lines.flatMap(lambda l: re.split(r'[^\w]+', l))
```

The flatMap() operation first converts each line into an array of words and then makes each of the words an element in the new RDD. If you asked Spark to count the number of elements in words RDD, it would tell you the number of words in the file.

Next, you'll want to replace each word with a tuple of that word and the number 1. The reason will become clear shortly.

```
pairs = words.map(lambda w: (w, 1))
```

The map() operation replaces each word with a tuple of that word and the number 1. The pairs RDD is a pair RDD where the word is the key, and all of the values are the number 1.

Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their values:

```
counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)
```

The reduceByKey() operation keeps adding elements' values together until there are no more to add for each key (word).

Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(sys.argv[2]) sc.stop()
```

The complete file should look like:

```
import re

import sys from pyspark import SparkConf, SparkContext

conf = SparkConf()

sc = SparkContext(conf=conf)

lines = sc.textFile(sys.argv[1])

words = lines.flatMap(lambda l: re.split(r'[^\w]+', l))

pairs = words.map(lambda w: (w, 1))

counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)

counts.saveAsTextFile(sys.argv[2]) sc.stop()
```

Save it in a file called wc.py.

4.2 Word Count in Java

Before you start, create a new Maven project for your application. The bare minimum requirements are a pom.xml file and a source code directory. The source code directory should be path/to/project/src/main/java, and the pom.xml file should contain something like:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.stanford.cs246</groupId>
  <artifactId>WordCount</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
  </dependencies>
</project>
```

Once you have your project created, you can begin writing the application. The first step of every Java Spark application is to create a Spark context:

```
import java.util.Arrays;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD; import scala.Tuple2;

public class WordCount {
```

```
        public static void main(String[] args) throws Exception {
                SparkConf conf = new SparkConf();
                JavaSparkContext sc = new JavaSparkContext(conf);
        ...
        }
}
```

Next, you'll need to read the target file into an RDD:

```
JavaRDD<String> lines = sc.textFile(args[0]);
```

You now have an RDD filled with strings, one per line of the file.

Next, you'll want to split the lines into individual words:

```
JavaRDD<String> words = lines.flatMap(l -> Arrays.asList(l.split("[^\\w]+")).iterator());
```

The flatMap() operation first converts each line into an array of words, and then makes each of the words an element in the new RDD. If you asked Spark to count the number of elements in the words RDD, it would tell you the number of words in the file. Note that the lambda argument to the method must return an iterator, not a list or array.

Next, you'll want to replace each word with a tuple of that word and the number 1. The reason will become clear shortly.

```
JavaPairRDD<String, Integer> pairs = words.mapToPair(w -> new Tuple2<>(w, 1));
```

The mapToPair() operation replaces each word with a tuple of that word and the number 1. The pairs RDD is a pair RDD where the word is the key, and all of the values are the number 1. Note that the type of the RDD is now JavaPairRDD. Also, note that the use of the Scala Tuple2 class is the normal and intended way to perform this operation.

Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their values:

```
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((n1, n2) -> n1 + n2);
```

The reduceByKey() operation keeps adding elements' values together until there are no more to add for each key (word).

Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(args[1]);
sc.stop();
```

The complete file should look like:

```
import java.util.Arrays; import
org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf; import
org.apache.spark.api.java.JavaPairRDD; import
org.apache.spark.api.java.JavaRDD; import
scala.Tuple2;

public class WordCount {
   public static void main(String[] args) throws Exception {
      SparkConf conf = new SparkConf();
      JavaSparkContext sc = new JavaSparkContext(conf);

      JavaRDD<String> lines = sc.textFile(args[0]);
      JavaRDD<String> words = lines.flatMap(l ->
            Arrays.asList(l.split("[^\\w]+")).iterator());
      JavaPairRDD<String, Integer> pairs =
            words.mapToPair(w -> new Tuple2<>(w, 1));
      JavaPairRDD<String, Integer> counts =
            pairs.reduceByKey((n1, n2) -> n1 + n2);

      counts.saveAsTextFile(args[1]);
      sc.stop();
   }
}
```

4.3 Word Count in Scala

Before you start, create a new Maven project for your application. The bare minimum requirements are a pom.xml file and a source code directory. The source code directory should be path/to/project/src/main/scala, and the pom.xml file should contain something like:

```
<?xml version="1.0" encoding="UTF-8"?>
   <project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>edu.stanford.cs246</groupId>
   <artifactId>WordCount</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
```

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
    <dependency>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <version>2.11</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Once you have your project created, you can begin writing the application. The first step of every Scala Spark application is to create a Spark context:

```
org.apache.spark.{SparkConf,SparkContext}

import org.apache.spark.SparkContext._
```

```
object WordCount {
        def main(args: Array[String]) {
        val conf = new SparkConf();
        val sc = new SparkContext(conf)
        ...
        }
}
```

Next, you'll need to read the target file into an RDD:

```
val lines = sc.textFile(args(0))
```

You now have an RDD filled with strings, one per line of the file. Next you'll want to split the lines into individual words:

```
val words = lines.flatMap(l => l.split("[^\\w]+"))
```

The flatMap() operation first converts each line into an array of words, and then makes each of the words an element in the new RDD. If you asked Spark to count the number of elements in the words RDD, it would tell you the number of words in the file.

Next, you'll want to replace each word with a tuple of that word and the number 1. The reason will become clear shortly.

```
val pairs = words.map(w => (w, 1))
```

The map() operation replaces each word with a tuple of that word and the number 1. The pairs RDD is a pair RDD where the word is the key, and all of the values are the number 1.

Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their

```
values: val counts = pairs.reduceByKey((n1, n2) => n1 + n2)
```

The reduceByKey() operation keeps adding elements' values together until there are no more to add for each key (word).

Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(args(1))
```

```
sc.stop
```

The complete file should look like:

```
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.SparkContext._

object WordCount {
   def main(args:
      Array[String]) { val conf
      = new SparkConf(); val
      sc = new
      SparkContext(conf)

      val lines = sc.textFile(args(0)) val words =
      lines.flatMap(l => l.split("[^\\w]+")) val pairs =
      words.map(w => (w, 1)) val counts =
      pairs.reduceByKey((n1, n2) => n1 + n2)

      counts.saveAsTextFile(args(1))
      sc.stop
   }
}
```

4. Task: Write your own Spark Job (Graded)

Now you will write your first Spark job to accomplish the following task:

Write a Spark application which outputs the number of words that start with each letter. This means that for every letter we want to count the total number of (nonunique) words that begin with that letter. In your implementation ignore the letter case, *i.e.*, consider all words as lower case. You can ignore all non-alphabetic characters.

Run your program over the same input data as above.

5. Submission Instruction

Submit a zip folder on Canvas before the deadline, including the followings:

1. A single file that contains your script. For example, it should be a .py file if you used Python. Add comments for the lines that you add/change from the provided word count sample.
2. The results generated by your script as a txt file (Spark may split the results into multupile files. Merge them all into a sainge .txt file).

- Please download your assignment after submission and make sure it is not corrupted. We won't be responsible for corrupted submissions and will not be able to take a resubmission after the deadline.

- You are highly encouraged to ask your question on the designated channel for Assignment 0 on MS Teams. Please DO NOT include your solutions in the comments you share on the board. Feel free to help other students with general questions.

- If you need help from the TAs, please email them.