Introduction to Spark

October 9, 2017

1 Exercises

1. Start the Spark cluster

   ```
   $ /opt/spark-1.5.2-bin-hadoop2-hive2-r/sbin/start-all.sh
   ```

2. We are going to transform some data from HDFS.

3. Start Hadoop. In the VM, start a terminal and run the following command

   ```
   $ start-dfs.sh && start-yarn.sh
   ```

4. Check out the source codes.

   (a) If you find the `~/git/scala-bigdata-starter/` folder exists in your system,

   ```
   $ cd ~/git/scala-bigdata-starter
   $ git pull
   ```

   (b) otherwise, clone it from github.

   ```
   $ cd ~
   $ mkdir git
   $ cd git
   $ git clone http://github.com/luzhuomi/scala-bigdata-starter.git
   $ cd scala-bigdata-starter
   ```

5. Change to the following folder

   ```
   $ cd ~/git/scala-bigdata-starter/spark-examples/
   ```

1.1 Word count

1. Check out the wordcount example in `src/main/scala/spark/examples/WordCount.scala` which should be like as follows,

```
object SimpleApp {
  def main(args: Array[String]) = {
    val conf = new SparkConf().setAppName("Wordcount Application")
    val sc = new SparkContext(conf)
    val textFile = sc.textFile("hdfs://localhost:9000/input/")
    val counts = textFile.flatMap(line => line.split(" "))
    .map((word:String) => (word, 1))
    .reduceByKey(_ + _)
    counts.saveAsTextFile("hdfs://localhost:9000/output/")
  }
}
```

which loads the text files from the `input` folder in HDFS. For each line found in the text files, we split the line by spaces into words. That would give us RDD of lists of words. The `flatMap` collapse the inner lists. For each word in the RDD, we associate it with the number `1`. Then we shuffle and group them by words. Finally, the `1`s are aggregated by `+`.

2. Before running the code, we need to make sure the folders are created and the data are copied into HDFS.

```
$ hdfs dfs -mkdir /input
$ hdfs dfs -rm -r /input/TheCompleteSherlockHolmes.txt
$ hdfs dfs -rm -r /output
$ hdfs dfs -put  ~/git/scala-bigdata-starter/spark-examples/data/wordcount/\
TheCompleteSherlockHolmes.txt /input/
```

   Note that the two `hdfs dfs -rm -r` commands will fail if the files and folders do not exist, which are normal.

3. To compile the code

```
$ sbt package
```

4. To execute the code

```
$ /opt/spark-1.5.2-bin-hadoop2-hive2-r/bin/spark-submit  --class SimpleApp \
target/scala-2.10/spark-examples_2.10-0.1.0.jar
```

5. To observe the output

```
$ hdfs dfs -get /output .
$ cat output/part-*
```

1.2 Transformation

We consider an example of transforming data using Spark

1. Check out the example in `src/main/scala/spark/examples/Transform.scala`

```
object Transform {
  val hdfs_nn = "127.0.0.1"
  def main(args: Array[String]) = {
    val conf = new SparkConf().setAppName("ETL (Transform) Example")
    val sc   = new SparkContext(conf)
    // load the file
    val input:RDD[String] = sc.textFile(s"hdfs://${hdfs_nn}:9000/data/transform/")
    // split by spaces
    val tokenizeds:RDD[Array[String]] = input.map(line => line.split(" "))
    tokenizeds.cache()

    // process all the ones
    val ones = tokenizeds
    .filter(tokenized => tokenized(0) == "1")
    .map(tokenized => {
      val x = (tokenized(1).split(":"))(1)
      val y = (tokenized(2).split(":"))(1)
      List(x,y).mkString("\t")
    })
    ones.saveAsTextFile(s"hdfs://${hdfs_nn}:9000/output/ones")

    val zeros = tokenizeds
    .filter(tokenized => tokenized(0) == "0")
    .map(tokenized => {
      val x = (tokenized(1).split(":"))(1)
      val y = (tokenized(2).split(":"))(1)
      List(x,y).mkString("\t")
    })
    zeros.saveAsTextFile(s"hdfs://${hdfs_nn}:9000/output/zeros")
  }
}
```

The above program parses and transform a data file in the format of

```
<label> 0:<x-value> 1:<y-value>
...
<label> 0:<x-value> 1:<y-value>
```

into two output files where in the `ones` we find all the rows with label equals to `1` and in the `zeros` we find all the rows with label `0`. The output files are in the format of

```
<x-value>    <y-value>
...
<x-value>    <y-value>
```

For instance, given the input file as

```
1 0:102 1:230
0 0:123 1:56
0 0:22  1:2
1 0:74 1:102
```

The output files in `ones` will be as

```
102    230
74     102
```

and those in `zeros` will be as

```
123    56
22     2
```

2. Before running the code, we need to make sure the folders are created and the data are copied into HDFS.

```
$ hdfs dfs -mkdir /data/transform/
$ hdfs dfs -rm -r /output
$ hdfs dfs -rm -r /data/transform/input.txt
$ hdfs dfs -put  ~/git/scala-bigdata-starter/spark-examples/data/transform\
 /input.txt /data/transform/
```

3. To compile

```
$ sbt package
```

4. To execute the code

```
$ /opt/spark-1.5.2-bin-hadoop2-hive2-r/bin/spark-submit  --class Transform \
target/scala-2.10/spark-examples_2.10-0.1.0.jar
```

5. To observe the output

```
$ hdfs dfs -get /output .
$ cat output/ones/part-*
$ cat output/zeros/part-*
```

1.3 Extraction

In this section, we consider an example of extract the US addresses from an input file. The input file contains lines of text which could be addresses. If a US address is found in the line, the line and a "Y" is appended to the output, otherise the line and an "N" is appended to the output. To extract the US address we use a regular expression.

1. Consider the source code in `src/main/scala/spark/examples/Extract.scala`

```
object Extract {
  val opat = compile("^(.*) ([A-Za-z]{2}) ([0-9]{5})(-[0-9]{4})?$")
  val hdfs_nn = "10.1.0.1"
  // val hdfs_nn = "127.0.0.1"
  def main(args: Array[String]) = {

    opat match
    {
      case None    => println("Pattern compilation error." )
      case Some(p) =>
      {
        val conf = new SparkConf().setAppName("ETL (Extract) Example")
        val sc   = new SparkContext(conf)
        // load the file
        val input:RDD[String] = sc.textFile(s"hdfs://${hdfs_nn}:9000/data/extract/")

        val extracted = input.map(l => {
          exec(p,l.trim) match
          {
            case Some(env) => List(l,"Y").mkString("\t")
            case None => List(l,"N").mkString("\t")
          }
        })
        extracted.saveAsTextFile(s"hdfs://${hdfs_nn}:9000/output/extracted")
      }
    }
  }
}
```

2. Before running the code, we need to make sure the folders are created and the data are copied into HDFS.

```
$ hdfs dfs -mkdir /data/extract/
$ hdfs dfs -rm -r /output
$ hdfs dfs -rm -r /data/extract/input.txt
$ hdfs dfs -put ~/git/scala-bigdata-starter/spark-examples/\
data/extract/input.txt /data/extract/
```

3. To compile

   ```
   $ sbt assembly
   ```

   Note that we use `assembly` instead of `package` due to the need of external libraries such as `scala-pderiv` which is an efficient regular expression matching library based on partial derivative. To ensure that the library is distributed to all the workers, we have to bundle all the depended jars into the main jar.

4. To execute

   ```
   $ /opt/spark-1.5.2-bin-hadoop2-hive2-r/bin/spark-submit --class Extract\
   target/scala-2.10/spark-examples-assembly-0.1.0.jar
   ```

5. To observe the output

   ```
   $ hdfs dfs -get /output .
   $ cat output/extracted/part-*
   ```