Language Abstractions for Concurrent and Parallel Programming 2017 Assignment 3: MapReduce and Spark

Prepared by Dave Clarke

November 29, 2017

Due Date: Friday, January 12, 2018 at 18:00.

The goal of this assignment is to solve a small number of problems using both the MapReduce paradigm and the more expressive Spark model.

The assignment should be done in pairs.

Preliminaries: Select a Language and Install Spark

This assignment will use Apache Spark which has support for programming languages Java, Scala, R, and Python. You are free to do your assignment in whichever language you prefer.[1]

Unfortunately, Spark has not yet been installed on the departmental computers. Installing it is quite easy, in particular, because Spark can run without a Hadoop backend. For intructions on how to install Spark (for non-Mac users), go to `https://spark.apache.org/downloads.html` and follow the instructions there.

If you have a Mac, you can install the homebrew software (`https://brew.sh/`) and then run

```
brew install apache-spark
```

This will install everything you need including a shell for running Scala code.

If you want to use Python, you could check whether the command `pyspark` is available. If not, try:

```
pip install pyspark
```

or

```
pip3 install pyspark+
```

depending on your version of Python.

You can now test whether your installation works by running the shell. For Scala, try:

[1] I have programmed Spark using Scala and Python. I prefer using Python because Scala's complicated type system got in the way, and Python automatically handled serialisation/deserialisation, making it easier to focus on the core algorithms.

```
spark-shell
```

For Python, try:

```
pyspark
```

The easiest way to use Spark is via the Scala or Python shell.

Unfortunately, there is no Java shell, and you'll have to work out for yourself how to compile Java-Spark programs — see here `http://www.robertomarchetto.com/spark_java_maven_example`. Also, if you want to use R, you'll have to find out this for yourselves.

MapReduce Framework: Python and Scala

To keep the overhead of doing this assignment as low as possible, instead of using both Hadoop and Spark, you will instead use a simple MapReduce framework written in Spark for part of the assignment. The framework plus some additional utility functions and an example are provided in Python (Figure 1) and in Scala (Figure 2). (These files are provided with in the student portal.)

The MapReduce framework is packaged up into a single function called `transform`. It takes an RDD of key-value pairs and two **functions**, `mapper` and `reducer`, as parameters. These functions have the following generic behaviour:

- **mapper** takes a single pair consisting of a key and a value and returns a list of key-value pairs, where the types of the output keys and values are potentially different from the input types.

  *The framework will apply this **mapper** function to all key-value pairs in the RDD.*

- **reducer** takes a single pair of a key and a list of values (of the type produced by map). *The list of values for each key is a list of all the values returned for that key by all mappers in the map phase.* Based on this input, a single key-value pair is returned.

  *The framework will apply this **reducer** function to the lists of values associated with each key produced in the mapper.*

The Problems

For each of the following problems:

- develop a MapReduce solution using the framework given above, calling only the `transform` function, possibly multiple times, preceeded with some pre- and post-processing on the RDD; and
- develop a Spark solution that uses any of the functionality available in Spark.

If the specifications below do not provide enough information about the format of the input or output, make an intelligent design decision and document your choice in the report.

```
## Import SparkContext
from pyspark import SparkContext

## Set up SparkContext
sc = SparkContext("local", "Simple App")

## MapReduce Framework
def initialise(sc, inputFile, prepare):
    """Open a file and apply the prepare function to each line"""
    input = sc.textFile(inputFile)
    return input.map(prepare)

def finalise(data, outputFile):
    """store data in given file"""
    data.saveAsTextFile(outputFile)

## Core MapReduce function
# input: is an RDD of inkey, value pairs
# mapper: (key, value) => List of (midkey, midvalue),
# reducer : (midkey, List of midvalue) => (outkey, outvalue)
# (See Scala version for type signature)
# output: RDD of outkey, outvalue pairs.
#
def transform(input, mapper, reducer):
    """map reduce framework"""
    return input.flatMap(mapper).groupByKey().map(reducer)

## End of MapReduce Framework

# Complete wordcount example
def wordcount(sc, inputFile, outputFile):
    rdd = initialise(sc, inputFile, lambda line: ("NoKey", line))
    result = transform(rdd,
                       lambda (key, data): [(x,1) for x in data.split(" ")],
                       lambda (key, values): (key, sum(values)))
    finalise(result, outputFile)

## Run the workcount example -- requires text in file "input.txt"
## Output placed in directory "wordcount.out"
wordcount(sc, "input.txt", "wordcount.out")
```

Figure 1: Basic MapReduce framework in Python and WordCount example

```
object HadoopMock {
    def initialise[t:ClassTag](sc : SparkContext, inputFile : String,
                               prepare : String => t) : RDD[t] = {
       val input = sc.textFile(inputFile)
       input.map(prepare)
    }

    def finalise[t](data : RDD[t], outputFile : String) {
      data.saveAsTextFile(outputFile)
    }

    // transform: core map reduce function
    // (inkey, s)  -- types of input key and input data
    // (midkey, t) -- types of key output from map phase and each data item
    //    -- reduce phase received (midkey, Iterable[t])
    // (outkey, u) -- types of final output key and final output type

    def transform[inkey, s, midkey : ClassTag, t : ClassTag, outkey, u]
            (input : RDD[(inkey, s)],
             mapper : ((inkey, s)) => Iterable[(midkey, t)],
             reducer : ((midkey, Iterable[t])) => (outkey, u)) : RDD[(outkey, u)] = {
        input.flatMap(mapper).groupByKey().map(reducer)
    }
    // end of MapReduce

    // Word count example
    def wordcount(sc : SparkContext, inputFile : String, outputFile : String) {
        val rdd = initialise(sc, inputFile, line => ("NoKey", line))
        val result = transform[String, String, String, Int, String, Int](rdd,
                               {case (key, line) => line.split("\\s+").map( x => (x,1))},
                               {case (key, values) => (key, values.sum)} )
        finalise(result, outputFile)
    }
}

## set up wordcount example -- won't run yet
lazy val wc = HadoopMock.wordcount(sc, "input.txt", "wordcount.out")

## actually run the example
wc
```

Figure 2: Basic MapReduce framework in Scala and WordCount example

Exercise 1: Word Count

The input will be drawn from the files present in the directory specified at the command line. Each file will consist of lines of text. Each line needs to be separated into words, and each word needs to be converted to lower case. These are the words that will be counted. **You may base your solution on the code provided above.**

The output is a list of words along with their count (in one or more files):

```
a 10
annie 1
apple 12
bark 12
...
```

This list does *not* need to be sorted.

Large volumes of text can be found at `https://www.gutenberg.org/`.

Exercise 2: Graph Conversion

Graphs produced by the utility program GraphGen (see below) are in the form of edge lists:

```
(1, 3)
(2, 1)
(6, 2)
...
```

Many graph problems require a different input format, such as an adjacency list.

The objective of this problem is to convert a graph in edge list format (distributed across a number of separate files) into adjacency list format. In this format, each output line has the form:

```
node1 # node2 node3 node4 ...
```

For example, the line

```
1 # 4 9 5
```

indicates that node 1 is connected to nodes 4, 9 and 5. (As our graphs are undirected, node 4, for instance, should then also include node 1 in its adjacency list.)

Exercise 3: Common Friends

A *social network* is an undirected graph where the nodes represent people, and an edge between two nodes represents that the corresponding two people are friends. (We do not allow loops or multi-edges in social networks.) Two people a and b in a social network have a common friend c whenever edges between (a, c) and (b, c) exist. (Note that this does not require that a and b are friends.)

The goal of this question is to compute the common friends for all pairs of people in a social network.

Output format (in one or more files):

```
friend1 friend2 # commonfriend1 commonfriend2 ...
...
```

A good implementation will avoid duplicates—that is, avoid computing the common friends for both (a, b) and (b, a).

Exercise 4: Counting Number of Triangles in a Social Network

A *triangle* in an undirected graph is a collection of three nodes that are each connected to the other two. For instance, a, b, c form a triangle if (a, b), (a, c) and (b, c) are edges in the graph. (A triangle is called a 3-*clique* in graph-theoretic terms.)

The goal of this exercise is to calculate the number of triangles that each person in a social network appears in.

Output format (in one or more files):

```
person number_of_triangles_person_is_in
...
```

This is the hardest question.

Exercise 5: Inverted Index

An inverted index gives a mapping from words to the locations where those words occur in a document. For this exercise, we will consider that a document is split into many files, each containing a page of the document, and that the inverted index maps words to the pages they occur on.

Develop a function that produces an inverted index. **You may use the code presented in lectures.**

Input: the name of a directory containing several text files numbered `1.txt`, `2.txt`, etc, such that each text file corresponds to a page of some text.

Output (in one or more files):

```
word # page1 page2 ...
...
```

For example, the line

```
fancy # 4 13 22
```

indicates that the word 'fancy' occurs on pages 4, 13, and 22 (equivalently, in files `4.txt`, `13.txt`, and `22.txt`.)

Each line should be sorted in increasing order and contain no duplicates.

Useful Tools

You have been provided the following utilities (Java programs) to help you generate and prepare input data for your programs:

GraphGen generates graphs that have a structure similar to social networks. This data can be used in Exercises 3–5. The program takes two numbers n and k, where n is the number of nodes and k is the minimum number of friends each node has. (Typically n is large and k is small.)

The resulting graph is printed to standard output one edge per line, where each edge has the format

```
(node1, node2)
```

For example,

```
(1, 3)
(2, 1)
(6, 2)
...
```

The assignment assumes that **all graphs are undirected**. This means that if nodes 1 and 2 are connected, only one of the two edges $(1, 2)$ and $(2, 1)$ will be present in the output. Take this into account when designing your functions!

Compile GraphGen using

```
javac GraphGen.java
```

Run GraphGen using, for example,

```
java GraphGen 10000 4 > graph.txt
```

Note that the "`> graph.txt`" part is required as GraphGen outputs the graph to the standard output stream.

FileSplitter takes two arguments, a name of a file `file.ext` and a number n. It splits the file into n approximately equal sized files called `file_1.ext`, ..., `file_n.ext`

This utility can be used to provide input to your MapReduce framework; for instance, by applying it to the output of GraphGen. Each file will be considered as input for a different map function. (The idea is that this imitates the distributed file system underlying real implementations of MapReduce.)

Compile FileSplitter using

```
javac FileSplitter.java
```

Run FileSplitter using, for example,

```
java FileSplitter graph.txt 100
```

What to Submit

Submit the following via the Student Portal:

- Your source code;
- Instructions describing how to compile and run your code;
- Short descriptions of how you solved each of the problems above (Exercises 1–5);
- **Small** but non-trivial test cases to demonstrate that your code runs Exercises 1–5 correctly; and
- A short breakdown of how many hours you have spent on each exercise and on preparing the short descriptions.

All textual answers (e.g., instructions, descriptions, performance evaluation) must be submitted in a single file in PDF format.

Evaluation Criteria

The most important aspect of the assignment is your ability to phrase solutions to problems in terms of 1) the MapReduce abstraction and 2) core Spark operations. Less important are the surrounding bits of code needed to load your data, output your results correctly, and so forth.

Each question is worth 6 points (3 points for the MapReduce solution, 3 points for the Spark solution) based on the following criteria:

- degree to which your solution matches the specification;
- clarity/elegance of your solution;
- quality and appropriateness of test data; and
- clarity, consiceness and completeness of the report.

Plagiarism and Cheating

You are requested to complete this assignment together with your group partner. As usual, if you consult external sources (i.e., sources outside your group), you are required to cite these appropriately. Failing to do so is plagiarism. Plagiarism and cheating are serious academic offenses and will be dealt with according to departmental policy.