Big Data Analytics (SOEN 498/691)
Laboratory sessions

Tristan Glatard, Valérie Hayot-Sasson
Department of Computer Science and Software Engineering
Concordia University, Montreal
tristan.glatard@concordia.ca, valeriehayot@gmail.com

March 7, 2017

Contents

Part I

Apache Spark

1 Introduction

Apache Spark is steadily emerging as a replacement of Hadoop MapReduce for the following reasons:

- Spark supports in-memory computing, which is faster than MapReduce's file-based model.
- Spark's programming model is much richer than MapReduce.
- Spark can run on a variety of clusters, including but not limited to Hadoop.

The goal of this session is to install Apache Spark on your computer and go through simple examples to understand its main concepts. Although you won't have to submit anything at the end of this session, the second lab assignment (LA2) will have to be implemented using Spark so it is important that you complete this one to the end. Most of the material in this document is taken from the Apache Spark online documentation:

- Quick Start Guide
- Programming Guide

Feel free to explore this documentation further!

2 Installation

It is assumed that you already have a working Hadoop installation on your computer. Download Apache Spark from there. Choose release 2.1.0, pre-built with user-provided Hadoop. You can also use the command line directly:

```
$ wget http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-without-hadoop.tgz
```

Unpack the release and write the following lines to `conf/spark-env.sh`, as explained here:

(Link to file)

```
#!/usr/bin/env bash
export SPARK_DIST_CLASSPATH=$(hadoop classpath)
```

Add Spark's `bin` directory to your `PATH` environment variables so that the system can find Spark's commands:

```
$ export PATH=$PATH:$PWD/spark-2.1.0-bin-without-hadoop/bin
```

Make sure that the following example runs correctly:

```
run-example SparkPi 10
```

`run-example` is a program located in `$PWD/spark-2.1.0-bin-without-hadoop/bin`. In case the output of the previous command line is `Command not found`, check your `PATH`.

3 Resilient Distributed Datasets (RDD)

Spark programs can be written in Java, Scala, Python and R. Although we use Python in the remainder of this document, feel free to use any other language and find the corresponding commands in the Programming Guide.

An easy way to start using Spark is through `pyspark`, a Spark Python shell started as follows:

```
$ pyspark
```

In the following, commands starting with >>> are typed in pyspark. Spark relies on the concept of Resilient Distributed Datasets (RDDs). RDDs are collections of elements that can be processed in parallel and created from regular data structures or files. For instance, here is how to create a RDD from an array of integers:

```
>>> d = [1,2,3,4]
>>> pd = sc.parallelize(d)
```

Here, sc is the Spark Context object provided by pyspark. In a standalone Spark program, you can create sc as follows:

(Link to file)

In Spark, two types of operations can be performed on RDDs: *transformations* produce another RDD from a given RDD, and *actions* produce a simple value, e.g., an integer, from a given RDD. The map function is an example of transformation used in the following code to add 1 to all the elements in RDD pd:

```
>>> inc=pd.map(lambda x: x+1)
```

Here are a few remarks about this line of code:

1. Spark's map function is very different from MapReduce's map function. Spark's map can be used to implement a MapReduce map function but it is much more general than that.

2. In Python, a *lambda* is an anonymous function and the expression lambda arguments: expression yields a function object. See more details in the Python documentation. Here, a lambda is used to provide the map function with a function that will be applied to all the elements in pd.

3. At this stage, nothing has actually been computed by Spark. That is, inc only contains a reference to the result of the transformation that *will* be applied to pd. The actual value will be computed as late as possible, when it is really needed. This is called *lazy evaluation*.

Let's now use the collect action to return all the elements in the RDD as a Python array:

```
>>> result=inc.collect()
```

The result of your first Spark program should now appear in the shell!

4 WordCount

We will now implement with Spark the classical WordCount example. We will follow the same logic as with MapReduce, i.e., our program will emit a (w,1) pair for every word found in the input text file. Then it will sum the 1s associated with a given word w.

4.1 Step-by-step presentation

First, let's create a simple input text file:

```
$ echo one two three two three three > /tmp/test.txt
$ echo one two three >> /tmp/test.txt
```

Our first step will be to create a RDD from the input text file:

```
>>> lines = sc.textFile(‘‘file:///tmp/test.txt’’)
```

This RDD contains one element for every line in the text file. Now we need to split those lines into words, which we will do using the flatMap transformation. flatMap is used when an element in the input RDD is mapped to an arbitrary number of elements (0 or more) in the output RDD. In our case, every line in the text file will be mapped to all the words in this line:

```
>>> words=lines.flatMap(lambda x: x.split())
```

RDD `words` now contains all the words in the dataset (or it will contain them when the program is evaluated, remember that Spark is lazy). We will now convert these words to `(w,1)` pairs using a `map` transformation:

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

Finally, pairs need to be "reduced", i.e., they need to be grouped by their first elements and the corresponding `1`s must be summed up. This is done using the `reduceByKey` transformation:

```
>>> counts=pairs.reduceByKey(lambda x,y: x+y)
```

The function passed to `reduceByKey` is a binary operator that takes as argument two values from the key-value pairs and returns one. Besides, this function must be commutative and associative. The word counts can now be printed using the `foreach` transformation:

```
>>> def g(x):
... print x
>>> counts.foreach(lambda x: g(str(x[0])+'': ''+str(x[1])))
```

Or they might be saved in a text file using the `saveAsTextFile` action:

```
>>> counts.saveAsTextFile(``file:///tmp/counts'')
```

4.2 Complete program

Here is a complete version of WordCount in Spark:

(Link to file)

To run this program, you will have to update your environment as follows:

```
$ export PYTHONPATH=$PWD/spark-2.1.0-bin-without-hadoop/python
$ sudo pip install py4j
```

The program can be executed as follows:

```
$ ./wordcount.py file:///tmp/test.txt file:///tmp/counts
```

Spark can transparently work with files stored on HDFS. Start your HDFS daemon, upload `test.txt` to HDFS and re-run the WordCount program:

```
$ ./wordcount.py hdfs://localhost:9000/test.txt file:///tmp/counts-hdfs
```

5 Going further

A complete list of transformations and actions available on RDDs is in the Programming Guide. Examples are also available in Spark's Github repository. Try re-implementing in Spark the kmeans clustering algorithm programmed in our previous lab session using MapReduce. A solution is available there.