INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Big Data Storage and Processing

TP: Apache Spark

The goal of this TP is to install Spark, to run some examples (both in interactive and job submission mode) and finally to write your own applications using Scala.

1. Spark installation

Spark needs the Scala programming language in order to run. Be sure that Scala is installed on your machine before you proceed.

Download the latest version of Spark by visiting http://spark.apache.org/downloads.html. For this TP, we will use Spark 2.4.0, pre-built for Hadoop 2.7 and later: `spark-2.4.0-bin-hadoop2.7.tgz`. Unpack the archive to a directory of your choice and add the following lines to the `~/.bashrc` file and then `source` it.

```
export SPARK_HOME = <your SPARK directory>
export PATH = $PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

To verify your Spark installation, launch the Scala-based Spark shell:

```
$ spark-shell
```

To exit the shell, Ctrl+C.

2. Interactive data analysis

The Spark **shell** provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala or Python, we will use the Scala-based one.

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs. Let's make a new RDD from the text of the README file in the Spark directory. Launch the Spark shell from the Spark directory and type:

```
scala> val textFile = sc.textFile("README.md")
```

Now at this point you should have noticed that we didn't get any "feedback" - nothing was printed besides some log statements (for instance, if the file didn't exist the error wouldn't appear now). That's because Spark is **lazily evaluated**, it won't execute any statements (besides registering them) until you perform an **action**.

RDDs have **actions**, which return **values**, and **transformations**, which return pointers to new **RDDs**. Let's start with a few **actions**:

Number of items in this RDD (i.e. number of lines in the text file):

```
scala> textFile.count()
```

First item in this RDD:

```
scala> textFile.first()
```

Now let's use a **transformation**. We will use the `filter` transformation to return a new RDD with a subset of the items in the file.

```
scala> val linesWithSpark = textFile.filter(line =>
line.contains("Spark"))
```

We can chain together transformations and actions and see how many lines contain "Spark":

```
scala> textFile.filter(line => line.contains("Spark")).count()
```

Now word counting is typically the "Hello World" of big data because doing it is pretty straightforward. Let's go ahead and get word counts for our dataset. As you already know, the common data flow pattern to do this is MapReduce. Spark can implement MapReduce flows easily:

```
scala> val wordCounts = textFile.flatMap(line => line.split(
" ")).map(word => (word, 1)).reduceByKey(_+_)
```

After executing this, you will not find any output because this is not an action, this is a transformation. To display the word counts in our shell, we can use the `collect` action:

```
scala> wordCounts.collect()
```

Yeah, you've just computed WordCount in one line ☺ That's a bit of a mouthful so let's go ahead and break it down. First, we're creating a `flatmap` of all the words. At this point we've got a new RDD basically, of those values. We map each word to the number 1 because it appears once. Once we've done that we reduce it by the key. This basically means that the first value in this tuple is the key and we want to reduce it by some supplied function which in this case just adds up the values (all the individual ones).

Spark also supports pulling data sets into a cluster-wide **in-memory cache**. This is very useful when data is accessed repeatedly, such as when querying a small "hot" dataset or when running an iterative algorithm like PageRank. As a simple example, let's mark our `linesWithSpark` dataset to be cached:

```
scala> linesWithSpark.cache()
```

You can also store the results on **disk**:

```
scala> linesWithSpark.saveAsTextFile("output")
```

You can check the status of your Spark jobs at:

http://localhost:4040/

Exercise 2.1

Find the line with the most words.

Hints

- You can associate to each line its number of words.
- Then you can keep just the one with the biggest count of words.

3. Running standalone applications

You can write a self-contained application using the Spark API in Scala, Java and Python. To build those applications for Spark you also need a specific build tool (**sbt** for Scala, **Maven** for Java). We will illustrate our examples with Scala.

You can edit your Scala based Spark programs in Eclipse (with the Scala plugin), Scala IDE or IntelliJ (with the Scala plugin). In all cases, when you create a new Scala project, you will need to add the Spark jars to the Build Path of your project. These are located in: `$SPARK_HOME/jars`

You can install `sbt` from http://www.scala-sbt.org. Place the `sbt` directory in your `$SCALA_HOME` directory and add the `$SCALA_HOME/<sbt_directory>/bin` to your `PATH` in order to execute it from anywhere.

Create a new Scala project and copy the `WordCount.scala` file from the TP archive to it. Our application depends on the Spark API, so we'll also include an `sbt` configuration file, which explains that Spark is a dependency. Create the `wordcount.sbt` configuration file containing:

```
name := "Word Count"
version := "1.0"
scalaVersion := "2.12.1"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
```

For `sbt` to work correctly, we'll need to layout `WordCount.scala` and `wordcount.sbt` according to a typical directory structure:

```
./wordcount.sbt
./src
./src/main
./src/main/scala
./src/main/scala/WordCount.scala
```

Once that is in place, we can create a JAR package containing the application's code:

```
$ sbt package
```

To run our application, we use the `spark-submit` script:

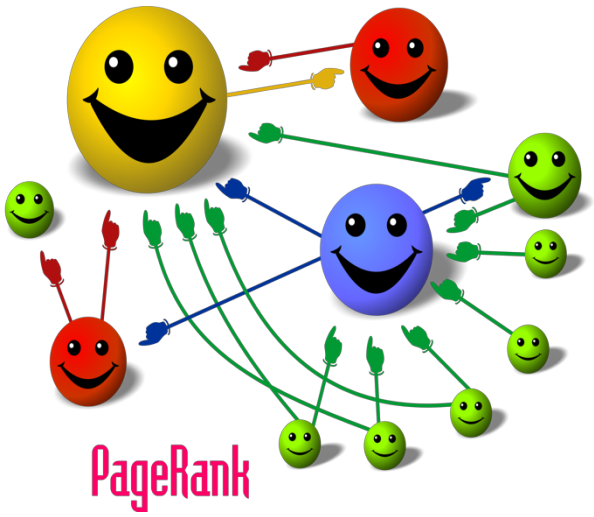
```
$ spark-submit \
--class "MyWordCount" \
--master local[2] \
/path/to/your/word-count_2.11-1.0.jar \
in.txt output
```

Exercise 2.2

Change word count so that you don't count some stop words that you define.

Exercise 3.1 (optional)

PageRank is an algorithm used by Google Search to rank websites in their search engine results. PageRank is a way of measuring the importance of website pages. It works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.



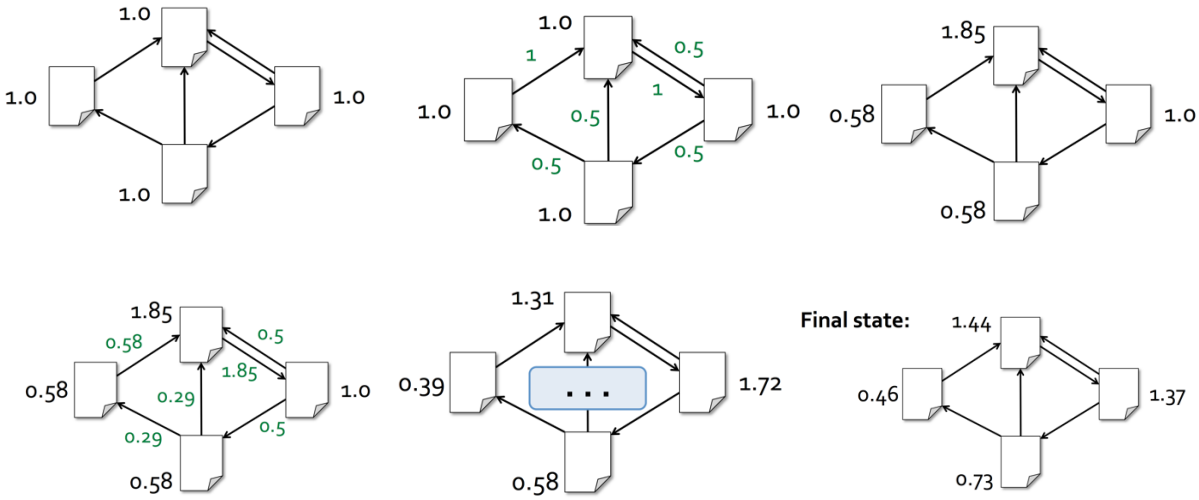
PageRank is a good example of a more complex Spark program since it uses **multiple stages of map & reduce** and therefore it benefits from Spark in-memory caching throughout multiple **iterations** over the same data.

Let's try to write a very simple PageRank program in Spark. The basic idea is to give pages ranks (scores) based on links to them:

- Links from many pages => high rank
- Link from a high-rank pag => high rank

Here is a (very) simple algorithm:

1. Start each page at a rank of 1
2. On each iteration, have page `p` contribute `rankp / |neighborsp|` to its out neighbors
3. Set each page's rank to `0.15 + 0.85 × contribs`
4. Iterate until ranks stabilize.



Fill the `MyPageRank.scala` skeleton to implement this algorithm and run it on Spark. Several existing Spark operations can be useful for this program: `map`, `groupByKey`, `mapValues`, `join`, `flatMap`, etc.