Big Data Analytics

Prof. Dr. Martin Theobald, Alessandro Temperoni & Jingjing Xu
Summer Semester 2022 – University of Luxembourg
Exercise Sheet #1 – Due on April 7, 2022 (before the lecture starts)

Please upload the source code of your solutions on or before the due date to the provided Moodle assignment as a single zip file using your group id as the file name. Provide some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`*, and report also the most important results (such as number of results, runtimes, etc.) of your solutions to that problem within this file. Remember that all solutions may be submitted in groups of up to 3 students.*

Modified Word Counting via the Java API of Hadoop **12 Points**

Problem 1. Consider the three Java classes `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWordStripes.java` provided on Moodle to solve this exercise. Set up your local Hadoop environment by following the steps described in the "GettingStarted" guide. Note that in particular the solutions to parts (c) and (d) of this exercise are computationally quite costly and may be measured either by using your local Hadoop installation or via the IRIS HPC cluster (a local Hadoop installation on one of the IRIS nodes, e.g., in your home directory, is needed in this case).

(a) Modify the three Java classes above, such that they split the input lines received by the `map` function into sequences of (i) *lower-case alphabetic characters* (i.e., "words" consisting of chars `a-z` including dashes "-" and under-dashes "_") and (ii) *numeric characters* (i.e., "numbers" consisting of digits `0-9` separated by at most one "`.`"), respectively.

Next, compute the counts of all distinct words and numbers you receive at the `reduce` function like before, but this time write the counts for the words and the counts for the numbers into two different output files on your local filesystem (each sorted in descending order of their counts). **3 points**

See, e.g., `http://www.tutorialspoint.com/java/java_regular_expressions.htm` for a tutorial on using regular expressions in Java.

(b) Take a look at the `HadoopWordPairs.java` and `HadoopWordStripes.java` classes and further modify the `map` functions of these two classes to extract *all pairs of words and/or numbers* within a given distance of m such tokens per line.

That is, modify the `map` functions of these two classes (using two nested loops), such that they find also actual word- and/or number-pairs with a larger distance within each input line (instead of just extracting two consecutive words per line as it is currently the case). **3 points**

(c) Run a first scalability test of the `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWord Stripes.java` classes by running each of the three classes on the `AA`, `AB`, `AC` and `AD` subdirectories of the `Wikipedia-En-41784-Articles.tar.gz` archive.

That is, run the three classes first on the `AA` directory (25% input size) and note the runtime of the programs; next run the programs on the `AA` and `AB` directories (50% input size) and again note the runtime of the programs; continue this until you also have the 75% and 100% measurements. Fix the maximum distance of word pairs extracted by the `HadoopWordPairs.java` and `HadoopWordStripes.java` classes to $m = 5$. **3 points**

(d) Run a second scalability test of the `HadoopWordPairs.java` and `HadoopWordStripes.java` classes by running the two classes on the `AA` subdirectory (25% input size). This time, however, change the maximum distance of word pairs stepwisely from $m = 1$ to 5 and 10. **3 points**

Please discuss the following points in your `Problem_1.txt` file:

- Which observation can you make regarding the runtime of your programs between the steps described in (c) and (d)?
- What is the principal difference between these two scalability tests in terms of "linear scalability"?

Modified Word Counting via the Scala API of Spark — **12 Points**

Problem 2. Consider the `SparkWordCount.scala` example provided on Moodle.

Follow the steps described in the "GettingStarted" guide to compile the Scala class via IntelliJ or SBT (or alternatively follow the steps described in "`run-intro-examples.sh`" to do the same from the command-line shell). Note that the solutions of this exercise may be run either by using your local Spark installation or directly via the IRIS HPC cluster.

(a) Modify the `SparkWordCount.scala` object, such that it extracts (i) *words* (i.e., sequences of the letters "`a-z`" including dashes "`-`" and under-dashes "_") and (ii) *numbers* (i.e., sequences of the digits "`0-9`" separated by at most one "`.`") into two separate RDD objects. Also here, first turn all input strings into lower cases, and then apply suitable regular expressions in Scala to extract these patterns. **3 points**

See, e.g., `http://www.tutorialspoint.com/scala/scala_regular_expressions.htm` for a tutorial on using regular expressions in Scala. Note that you need to make yourself familiar with the more functional programming style of Scala to modify the tokenizer as described above.

(b) Apply your modified `SparkWordCount.scala` object to the complete set of 41,784 Wikipedia articles provided on Moodle in order to find the (i) *top-1,000 most frequent words* and the (ii) *top-1,000 most frequent numbers* contained among all articles, respectively. Sort the words and numbers from the two RDDs in descending order of their counts, and finally store the two lists of sorted numbers and words in two separate files on your local file system. Compare the results with the ones you obtained via Hadoop in Problem 1 and explain possible differences in your `Problem_2.txt` file. **3 points**

(c) Parse the file `stopwords.txt` provided on Moodle into a sequence object (i.e., an `Array` or a `List`) such that each stopword becomes one entry in this sequence, and store the sequence as an immutable variable `val stopwords = ...` in your Spark environment. Refer to the modified parsing function for words you implemented in (a) and filter out all stopwords from the RDD holding the words by referring to `val stopwords` in the *closure* of your filtering function. Finally, compute the word counts of the remaining words as before. **3 points**

(d) Repeat the steps of (c) but this time create a new *broadcast variable* in your Spark environment to store the stopwords. Compute the word counts of the filtered words as before and compare the runtime of this approach to the one of (c). **3 points**

Briefly discuss the results of this exercise in your `Problem_2.txt` file.

Analyzing Live Tweets via the Scala API of Spark — **12 Points**

Problem 3. Consider the `SparkTwitterCollector.scala` example provided on Moodle.

Follow the steps described in the "GettingStarted" guide to compile the Scala class via IntelliJ or SBT (or alternatively follow the steps described in "`run-intro-examples.sh`" to do the same from the command-line shell). Note that the solutions of this exercise may be run either by using your local Spark installation or directly via the IRIS HPC cluster.

(a) Modify the `SparkTwitterCollector.scala` object, such that it extracts only hashtags ("`#`" followed by case-sensitive alpa-numeric sequences of characters and digits) from the `text` fields of the received tweets[1]. You may use a suitable JSON API and/or regular expressions for Scala to extract these texts fields and the respective hashtags. **4 points**

See, e.g., `http://www.tutorialspoint.com/scala/scala_regular_expressions.htm` for a tutorial on using regular expressions in Scala. Note that you need to make yourself familiar with the more functional programming style of Scala to modify the tokenizer as described above.

[1]See: `https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/overview`

(b) Modify the `SparkTwitterCollector.scala` with the RDD transformations of `SparkWordCount.scala` in order to count the frequency of all hashtags from your 1-minute snapshot of Twitter. Write out the top-1,000 most frequent hashtags together with their counts into a file on your local filesystem (again in descending order of their counts). **4 points**

(c) Modify once more the `SparkTwitterCollector.scala` object in order to also extract all *pairs of hashtags* which co-occur within a same tweet. Run the Twitter collector app again for 1 minute, and store the 1,000 most frequent pairs of hashtags (again sorted in descending order of their tweet counts) in a file on your local filesystem. **4 points**

Report the most important results of this exercise in your `Problem_3.txt` file.