Big Data Analytics

Prof. Giacomo di Tollo et al.

Summer Semester 2024 – University of Luxembourg

Exercise Sheet #2 – Due on April 8th, 2024 (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 solution to part (c) of this exercise is 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 Java classes above, such that the `map` function emits only the following types of tokens to the reducers:

(1) *words* consisting only of sequences of lower-case letters `a-z`, with a length of at least 5 letters and at most 25 letters; and

(2) *numbers* consisting only of sequences of digits `0-9`, with a length of at least 2 digits and at most 12 digits.

That is, first turn all input strings (i.e., each line of a text file) into lower cases and then apply the above filters, e.g., via corresponding regular expressions. You may remove (i.e., split the input strings at) any form of punctuation or commas between the words and numbers. See, e.g., `http://www.tutorialspoint.com/java/java_regular_expressions.htm` for a tutorial on using regular expressions in Java. **2 Points**

(b) Next, analyze the Wikipedia collection provided on Moodle as follows. Focus only on the `AA` subdirectory to solve the following points.

(1) Find all individual words (not numbers) with a count of exactly 1,000.

(2) Find all word pairs (not numbers) with a count of exactly 1,000 (using a distance of $m = 1$ for immediate neighbours of words in the mappers).

(3) Find the top-100 most frequent individual words (not numbers).

(4) Find the top-100 most frequent pairs of tokens, where the first token is a number and the second token is a word (again using a distance of $m = 1$ for immediate neighbours of numbers and words in the mappers).

Try to combine the above tasks (1)–(4) into as few MapReduce jobs as possible. Note that you may read the output files produced by one MapReduce job as input to another MapReduce job. No further processing outside the `map` and `reduce` functions is allowed (i.e., you in particular need to find a way to filter the top-100 results directly in MapReduce)!

You may choose if you prefer either the `HadoopWordPairs.java` or the `HadoopWordStripes.java` example as basis for computing the frequencies of the pairs (one is sufficient). **8 Points**

(c) Continue your analysis by step-by-step, including also the remaining `AB`, `AC`, ..., `AK` subdirectories of the provided collection of 41,784 Wikipedia articles. That is, include one more subdirectory at each step and measure the runtime of your MapReduce jobs until completion. Report your detailed hardware setup and the runtimes in your `Problem_1.txt` file. Which conclusions can you draw with respect to the scalability of your solution? **2 Points**

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

Problem 2. Repeat the above steps for the `SparkWordCount.scala` example provided on Moodle. You may directly extend the provided tokenizer to also extract immediate pairs of tokens, and you may use any of the RDD *transformations* and *actions* provided in the lecture slides. Also here, try to minimize the amount of Spark transformations you need in order to complete the given tasks. Consider using the IRIS HPC cluster with the provided launcher scripts in particular for the larger jobs. Report your detailed hardware setup and the runtime in your `Problem_2.txt` file.

Using DataFrames using Spark SQL **12 Points**

Problem 3. Consider the Scala script `DataFrame-shell.scala` together with the `titanic.csv` dataset provided on Moodle to solve this problem.

(a) Implement the following query using only built-in DataFrame operations. Refer to the DataFrame API documentation[1] for the list of operations that can be performed on a DataFrame. **3 Points**

Query: *"Find the distinct last names of all passengers whose age is greater than the average age of all passengers on the Titanic."*

(b) Implement the above query by issuing only SQL queries from within your Spark session via `spark.sql("...")`. **3 Points**

(c) Create a user-defined function called `AgeAverage` that would consider all `null` entries for the passengers' ages as the static integer value "23" while calculating the average age. Use this function in the SQL query that you have framed in (b) and observe how the output rows vary. Refer to the "User-Defined Aggregate Functions (UDAFs)" section in the SQL programming guidelines[2] for an example. **3 Points**

(d) Assume we wish to investigate the following (statistical) hypothesis:

Hypothesis: *"The average age of passengers who survived the sinking of the Titanic is not very different from (i.e., neither much below nor much above) the average age of passengers who did not survive the sinking of the Titanic."*

How would you verify the above hypothesis? Please provide an implementation in Spark SQL. You may but you do not have to use a *statistical test* (such as the Student's t-test) to either accept or reject this hypothesis; a simple numeric comparison will also be sufficient. Summarize your results in your `Problem_3.txt` file. **3 Points**

[1] See: `https://spark.apache.org/docs/3.2.1/sql-getting-started.html`
[2] See: `https://spark.apache.org/docs/3.2.1/sql-ref-functions-udf-aggregate.html`