MA_DS-5: NoSQL Databases & Cloud Computing (Winter Term 2023/24)
Prof. Dr. Martin Theobald
Exercise Sheet 3: Due on November 30 (before the lecture starts)

UNIVERSITÉ DU LUXEMBOURG

Please upload your solutions to the designated Moodle assignment on or before the due date as a single zip file using your group id as the file name. Include some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`. *All solutions may be submitted in groups of up to 3 students.*

Inverted Indexing & Query Processing in MapReduce

Problem 1. **12 Points**

(a) Implement a basic form of an *Inverted Index* (based on the pseudo code provided on Slide #64 of Chapter 3 from the lecture slides) in the Java APIs of Apache Hadoop. Consider the following points:

- Use the basic `WordCount.java` example as a template for your inverted-index implementation.
- Turn all input lines into lower cases and split the lines using

  ```
  value.toString().split("[^\\w']+");
  ```

  in order to obtain reasonable word tokens in your `map` method.
- Enable the `job.setCombinerClass(...)` option in the driver function (i.e., the `main` method in the Java class) to enable a local aggregation.
- Consider the `Wikipedia-En-41784-Articles.tar.gz` archive from Moodle as input to your inverted index. You may assume that each line in the `wiki_*.1lineperdoc` files corresponds to exactly one Wikipedia article, where the `<doc id="X"...>` opening tag denotes an article with id `X` and the substring between the `<doc ...>` opening and `</doc>` closing tags contains the entire text content of that article.
- Extract all article ids `X` and split the text content of each Wikipedia article into reasonable word tokens as described above.
- The output of your inverted index (at the `reduce` method) should be emitted by the reducers into one or more `part-r-XXXXX` files in your Hadoop output directory. The format of these files should be as follows:

  ```
  token<tab>postings
  ```

  where each `token` is a word token extracted by your tokenizer, and `postings` is a sorted list of `article-id`, `score` pairs of Wikipedia articles in which the token occurs, while `score` is the frequency of the token in that article. Note that all postings under a same `token` should be sorted in increasing order of `article-id`.

5 Points

(b) Implement a simple form of a search engine in the Java APIs of Apache Hadoop by implementing a *Reduce-Side Join* over the `part-r-XXXXX` files you implemented in (a). Also here, consider the following points:

- This time, use the `WordCount2.java` example to see how command-line parameters can be passed to the `Mapper` and `Reducer` classes, respectively.
- Specifically, pass a user-provided list of keywords (i.e., the "search strings") from the command-line to the `Mapper` class.
- Implement a `map` method which reads one line from each `part-r-XXXXX` file at a time and extracts the corresponding `token`, `postings` fields from each such line.
- For each line that contains a `token` which occurs in the provided list of keywords (stored in the `Mapper` class), extract the `article-id`, `score` pairs of the corresponding `postings` list and emit the `article-id` as the key and the `score` as the value of the `map` method.
- In the `reduce` method, make sure that you then sum up all the `score` values received under a same `article-id` as key, and finally emit this `article-id` together with its summed up `score` values as one result of your join operation. The format of the final `part-r-XXXXX` files should be as follows:

  ```
  article-id<tab>sum_of_scores
  ```

5 Points

(c) Repeat steps (a) and (b) of this exercise by using the *Secondary Sorting* optimization (based on the pseudo code provided on Slide #65 of Chapter 3 from the lectures slides) in the Java APIs of Apache Hadoop. How does this optimization affect the usage of a `Combiner` and additional `Partitioner` class? What would be the best join technique for this MapReduce setup?

Also here, the format of the final `part-r-XXXXX` files should be as follows:

`article-id<tab>sum_of_scores` **2 Points**

Analytical Queries in MapReduce

Problem 2. **12 Points**

Consider again the TSV files we downloaded from IMDB for the first exercise sheet. This time, implement the first three analytical queries from Problem 3 (a)–(c) of Exercise Sheet #1 in MapReduce:

(a) Select the *most popular directors* based on how many movies they directed, stop after the top 25 directors with the most movies. **4 Points**

(b) Select the *top 25 pairs of actors* that occur together in a same movie, ordered by the number of movies in which they co-occur. **4 Points**

(c) Find *frequent 2-itemsets of actors or directors* who occurred together in at least 4 movies. **4 Points**

Note that you need to implement a suitable function to parse the various fields from each line of a TSV file into separate strings inside your `map` methods in Java. Splitting each line by `[\t]+` should suffice for our purpose. Also make sure to sort the output of your `reduce` methods in a proper way (e.g., by using an external sorting step as shown in the last exercise sheet). You do not need to consider the A-Priori optimization for solving (c).

Analytical Queries in Apache Pig

Problem 3. **12 Points**

Also here, consider the TSV files we downloaded from IMDB for the first exercise sheet. This time, implement the first three analytical queries from Problem 3 (a)–(c) of Exercise Sheet #1 in Apache Pig by using the Pig Latin operators presented in the lecture slides:

(a) Select the *most popular directors* based on how many movies they directed, stop after the top 25 directors with the most movies. **4 Points**

(b) Select the *top 25 pairs of actors* that occur together in a same movie, ordered by the number of movies in which they co-occur. **4 Points**

(c) Find *frequent 2-itemsets of actors or directors* who occurred together in at least 4 movies. **4 Points**

Also here, you do not need to consider the A-Priori optimization for solving (c).