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.*

Analytical Queries in Apache Hive

Problem 1. **12 Points**

In this problem, we will compare SQL and HiveQL based on the IMDB queries we previously already developed for PostgreSQL and Apache Pig. To do so, load the four TSV files from IMDB, namely

- `name.basics.tsv`
- `title.basics.tsv`
- `title.principals.tsv`
- `title.ratings.tsv`

into four *managed tables* in Hive. Next translate the following three analytical queries into HiveQL and report their results.

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

Data Clustering & Bucketized Map-Side Joins in Apache Hive

Problem 2. **12 Points**

In this problem, we will develop a set of HiveQL queries for clustering, bucketizing and running joins over the IMDB data dumps we already considered earlier. Specifically, consider the two files `name.basics.tsv` and `title.principals.tsv` to solve this problem.

(a) First, load the data from the two TSV files into two *external tables* in Hive. Next, insert the data from these two tables into two new *clustered tables* which should each maintain 8 sorted buckets according to the actors' `nconst` column (which is contained in both TSV files).

Verify the resulting clustering of the data by describing both tables in Hive, and by checking the files that Hive created under the storage location of the tables in your file system (as it was shown during the lecture). **4 Points**

(b) Write a HiveQL query to perform a *bucketized Map-side join* to join the two tables based on their common `nconst` column. To enable a bucketized Map-side join in Hive, you need to set the below Hive parameters to `true`:

```
set hive.auto.convert.sortmerge.join=true;
set hive.optimize.bucketmapjoin=true;
set hive.optimize.bucketmapjoin.sortedmerge=true;
```

Verify the usage of the bucketized Map-side join by explaining the query's execution plan in Hive.

Also compare the performance of this query with the case when the input tables are not clustered. **4 Points**

(c) Run the three analytical queries from Problem 1 in HiveQL and compare their runtimes with the case when the input tables are not clustered (i.e., using the *managed tables* from Problem 1 instead). Note that this should not require any rewriting of your queries.

Memory-backed Joins with Apache Hadoop & HBase

Problem 3. **12 Points**

Consider once more the IMDB data dumps and the two TSV files `name.basics.tsv` and `title.principals.tsv` we already used for Problem 2 of this sheet. This time, however, employ Hadoop's MapReduce APIs in combination with HBase in order to join the two files via a *Memory-backed join* (see Slide 62 of Chapter 3 of the lecture slides for a hint).

(a) To do so, first load the smaller file, namely `name.basics.tsv`, into an HBase index by using the `nconst` attribute as rowkey. **4 Points**

(b) Next, implement a `map` function in Hadoop which, for each line of `title.principals.tsv`, looks up the corresponding actor in the previously created HBase index by issuing a `get` operation to HBase for the current `nconst` value. In case a matching actor is found (i.e., the `primaryProfession` should contain the value `actor` or `actress`), emit the actor's name together with the movie id (i.e., the current `tconst` value) to your `reduce` function. **4 Points**

(c) Finally, implement a corresponding `reduce` function that receives the emitted actor names and counts the number of distinct movie ids in which each actor occurred. Emit only those actors to the `part-r-XXXXX` files created by Hadoop which occurred in at least 100 movies. Finally, sort all `part-r-XXXXX` files to find the *top 25 actors with the highest numbers of movies* and once more report your results. **4 Points**