LOG8415
Advanced Concepts of Cloud Computing

William Bourret (2005931), Vlad Drelciuc (1941366),
Charles Fakih (2012512), Simon Tran (1961278)

Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada

1 Project Setup

Our entire source code for this assignment can be found here:
https://github.com/vdrelciuc/LOG8415/tree/master/tp2

For this assignment, we decided to use Microsoft Azure to host our VM instance. The instance has the following attributes:

- Instance type: Standard D2s v3
- vCPUs: 2
- RAM: 8 GiB
- OS: Ubuntu Server 20.04

Although we haven't automized the intialization of the instance, we wrote the following Bash scripts to help with the benchmarking part of this assignment:

- Hadoop setup: /WordCountExperiments/hadoop_setup.sh
- Comparing the performance of Hadoop vs Linux: /WordCountExperiments/hadoop_vs_linux.sh
- Spark setup: /HadoopSparkComparison/spark_setup.sh
- Comparing the performance of Hadoop vs Spark: /HadoopSparkComparison/hadoop_vs_spark.sh

To run any of the scripts above, simply use:

```
sh /path/to/script.sh
```

- To solve the Social Network Problem, use /SocialNetworkProblem/run.sh

```
sh run.sh [ip] [path-to-pem] [path-to-dataset]
```

The [path-to-dataset] is optional and will default to the path of the default dataset (src/main/resources/dataset.txt)
For example, using 20.124.97.80 as the ip and ubuntu_v2.pem as the pem file, we can run the script like this:

```
sh run.sh 20.124.97.80 ubuntu_v2.pem
```

The script will automatically upload the source files to the VM, compile the project, start it and download the results in a output folder

2 Experiments with WordCount

We started our experiment by installing Hadoop on our Azure VM instance following the steps described in /WordCountExperiments/hadoop_setup.sh. We then downloaded a copy of Ulysses and ran the example implementation of WordCount that comes with Hadoop while timing its execution time.

3 Hadoop vs Linux on AWS

Next, we compared the execution time of Hadoop's example WordCount implementation versus counting the word frequency of a text with Linux, using Linux commands and pipes. The detailed step-by-step instructions to recreate our experiment are avaiable in /WordCountExperiments/hadoop_vs_linux.sh. Here are the results:

Table 1: Comparison between Hadoop and Linux execution time (real)

	Hadoop time (s)	Linux time (s)
pg4300.txt	4.517	1.136

The results show that running Hadoop for this specific task takes significantly longer than the native Linux commands. This is probably due to the fact that the Linux command processes the text file natively and line by line. In contrast, Hadoop only supports processing data in the HDFS format, so it will actually translate the text file to HDFS first, then do all the processing on disk. All this overhead leads to poor performance versus Linux for this specific workload.

4 Hadoop vs Spark on AWS

The following step was to compare the execution time of Hadoop's example WordCount on 9 different input files (3 times each) versus the one of Spark's example JavaWordCount implementation. For this, we first installed Spark 2.0.0 (see /HadoopSparkComparison/spark_setup.sh), then ran the following commands: /HadoopSparkComparison/hadoop_vs_spark.sh. Here are the results:

Table 2: Comparison between Hadoop and Spark average execution time (real)

	Hadoop time (s)	Spark time (s)
buchanj-midwinter-00-t.txt	2.167	9.914
carman-farhorizons-00-t.txt	2.725	9.312
charlesworth-scene-00-t.txt	3.126	9.770
cheyneyp-darkbahama-00-t.txt	3.154	9.861
colby-champlain-00-t.txt	2.743	9.491
delamare-bumps-00-t.txt	2.737	9.235
delamare-lucy-00-t.txt	2.739	9.337
delamare-myfanwy-00-t.txt	2.681	9.343
delamare-penny-00-t.txt	2.755	9.254
AVERAGE	2.758	9.502

The results show that running Spark for these workloads took longer than Hadoop. This was surprising given the fact that Spark runs in-memory while Hadoop runs on disk, and therefore that we would have expected Spark's jobs to run faster. An explanation for this is that although Spark is very fast at PROCESSING the data in memory, it has a long startup and shutdown routine, which end up making the overall EXECUTION longer compared to Hadoop. We should note that since the PROCESSING is faster for Spark compared to Hadoop, Spark would scale better on a larger dataset where Hadoop's operations on disk would become a bottleneck.

5 The Social Networking Problem - MapReduce Jobs

For the Social Networking Problem, we are tasked with recommending a list of 10 users for every user in the list according to the number of mutual friends both possess. In our current context, it goes without saying that MapReduce is a good fit for solving the problem since we are potentially dealing with a lot of data in operations that are easily parallelizable.

Hence, the Mapper goes through every user to map its relationship with other users. Crucially, it identifies the friends of every user, as well as the potential friends who happen to be the second-degree friends. The idea is that second-degree friends are either friends, in which case the friendship will be established later in the Map operation, or potential recommendations with mutual friends. This is helped by the fact that second-degree friends would appear a number of times equal to the number of mutual friends with the user initially considered.

The Reducer, though, collects every user and its relationships with other users. Having a list of all a user's friends, the potential friends that are not in that list are considered candidates for the desired recommendations. The potential friends are kept in a hashmap that links the potential friend to the number of mutual friends with the user initially considered, by incrementing that value the number of times the said potential friend appears.

6 The Social Networking Problem - Our Algorithm

Our solution for the Social Networking Problem begins by mapping the user's relationships. For each user, we iterate over their friends and create a friendship relation for each of them. In order to represent a relation between a user and a friend/potential friend, we created a Friendship class. This class takes a user's friend's id (or potential friend's id) as well as an integer representing the type of relationship: -1 for an actual friendship and 1 for a potential friendship. Next, for the friend we are currently iterating, we create a potential friendship relation between the current friend and the user's other friends.

In the reducer, we have a list of strings containing the ids of the user's friends (friendsTracker) and a hash map (recommendations) containing a key-value pairs of the potential friends ids (key) and the number of mutual friends with the user (value). For each of the user's relationships, if it's a friendship, we add it to the list. If not, then we verify if that id is already in the friends list. If so, we simply increment the value of the number of mutual friends for that potential friend, or create a key-pair value and add it the recommendations hash map if it is not in the list. Next, we sort the the hash map from the biggest number of mutual friends to the lowest number of mutual friends and pick the first ten.

7 The Social Networking Problem - Recommendations for Specific Users

After running the script, the results are available under: SocialNetworkProblem/output/dataset.txt/part-00000.

Using those results, we found - amongst others - the following recommendations:

- 924: 45881,6995,43748,439,11860,2409,15416
- 8941: 8943,8944,8940
- 8942: 8939,8940,8943,8944
- 9019: 9022,9023,317
- 9020: 9021,9016,9017,9022,9023,317
- 9021: 9020,9016,9017,9022,9023,317
- 9022: 9019,9021,9020,9023,9016,9017,317
- 9990: 34642,34299,34485
- 9992: 9987,9989,35667,9991
- 9993: 9991,34642,34299,34485,13478,37941,13134