LOG8415
Advanced Concepts of Cloud Computing
MapReduce on AWS

Vahid Majdinasab
Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
`vahid.majdinasab[at]polymtl.ca`

Team 6
Ana Karen López Baltazar (2311301)
Ali Hazime (2270891)
Lucy Nguyen (2089225)
Majdi Saghrouni (1733556)

GitHub: https://github.com/LuuN2122/LOG8415

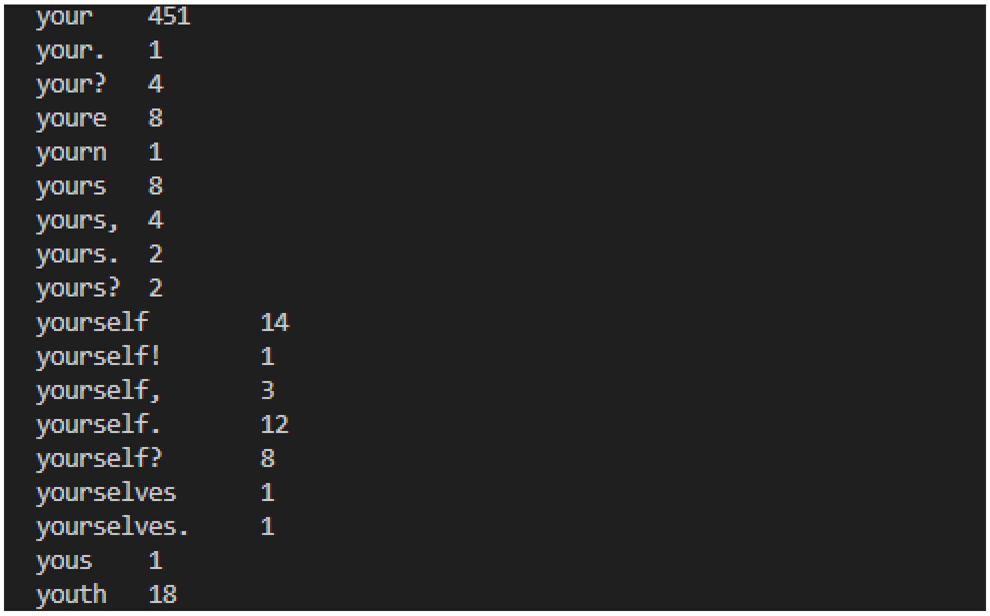
November 18, 2023

1 Experiments with WordCount program

Execution steps:

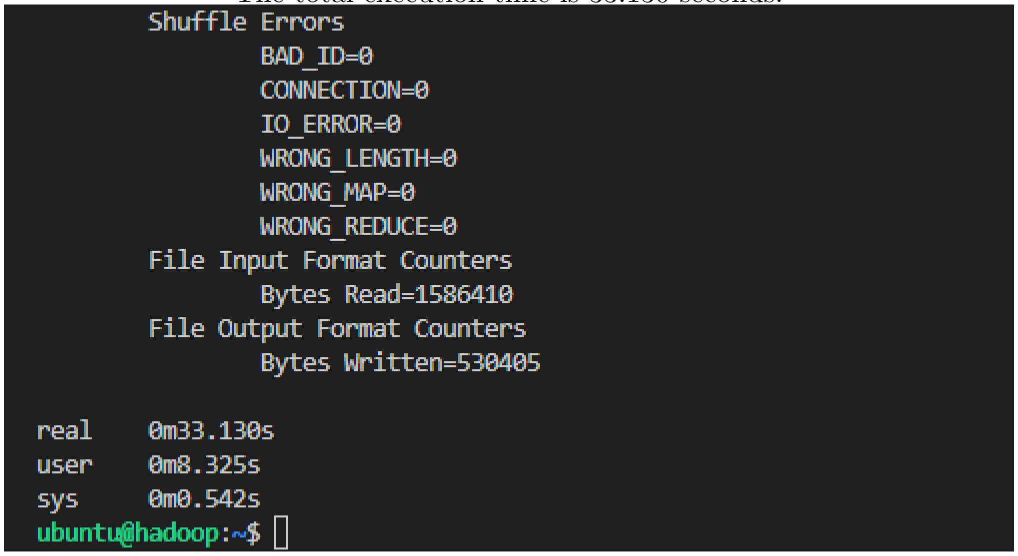
1. On your local machine, run the bash script automation_script.sh to create and configure an ec2 instance
2. After SSH'ing to the ec2 instance for the first time, run the bash script setup.sh to install and configure Hadoop and Spark
3. After the script terminates, the instance will reboot and continue setting up Hadoop and Spark in the background for a few minutes
4. SSH again to the ec2 instance
5. When the terminal seems responsive again, run the following command to update the environment variables:
   - source ~/.bashrc
6. Launch the Hadoop cluster by running the following commands:
   - hdfs namenode -format
   - $HADOOP_HOME/sbin/start-dfs.sh
   - $HADOOP_HOME/sbin/start-yarn.sh
   - $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver
7. Run the following command to create a folder for the inputs:
   - hdfs dfs -mkdir -p WordCount/input
8. Run the following command to add the text file pg4300.txt to the input folder:
   - hdfs dfs -put datasets/pg4300.txt WordCount/input
9. Run the WordCount.java program on the input provided using the following command:
   - time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce /hadoop-mapreduce-examples-3.3.6.jar wordcount WordCount/input WordCount/output
10. When the program terminates, observe the output file WordCount/output/part-r-00000, and the real execution time that will be printed to the screen

Image showing a subset of the output taken from the generated file WordCount/output/part-r-00000:



```
your    451
your.   1
your?   4
youre   8
yourn   1
yours   8
yours,  4
yours.  2
yours?  2
yourself        14
yourself!       1
yourself,       3
yourself.       12
yourself?       8
yourselves      1
yourselves.     1
yous    1
youth   18
```

The total execution time is 33.130 seconds:



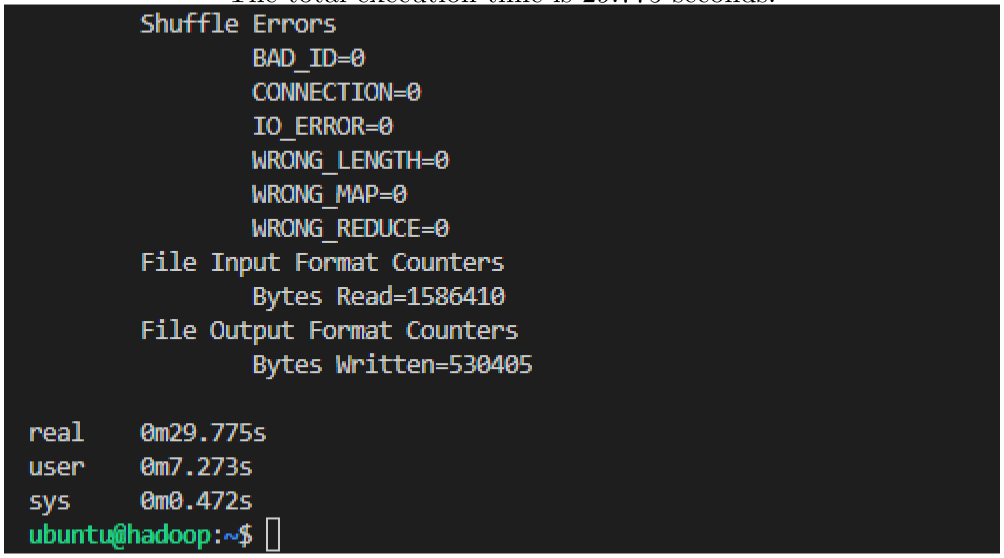
```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1586410
        File Output Format Counters
                Bytes Written=530405

real    0m33.130s
user    0m8.325s
sys     0m0.542s
ubuntu@hadoop:~$
```

2 Performance comparison of Hadoop vs. Linux

Hadoop:

1. If the WordCount/output folder already exists, delete it to execute Hadoop again:
   - hdfs dfs -rm -r WordCount/output
2. Run the WordCount.java program on the input provided using the following command:
   - time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce /hadoop-mapreduce-examples-3.3.6.jar wordcount WordCount/input WordCount/output
3. When the program terminates, observe the real execution time that will be printed to the screen

The total execution time is 29.775 seconds:

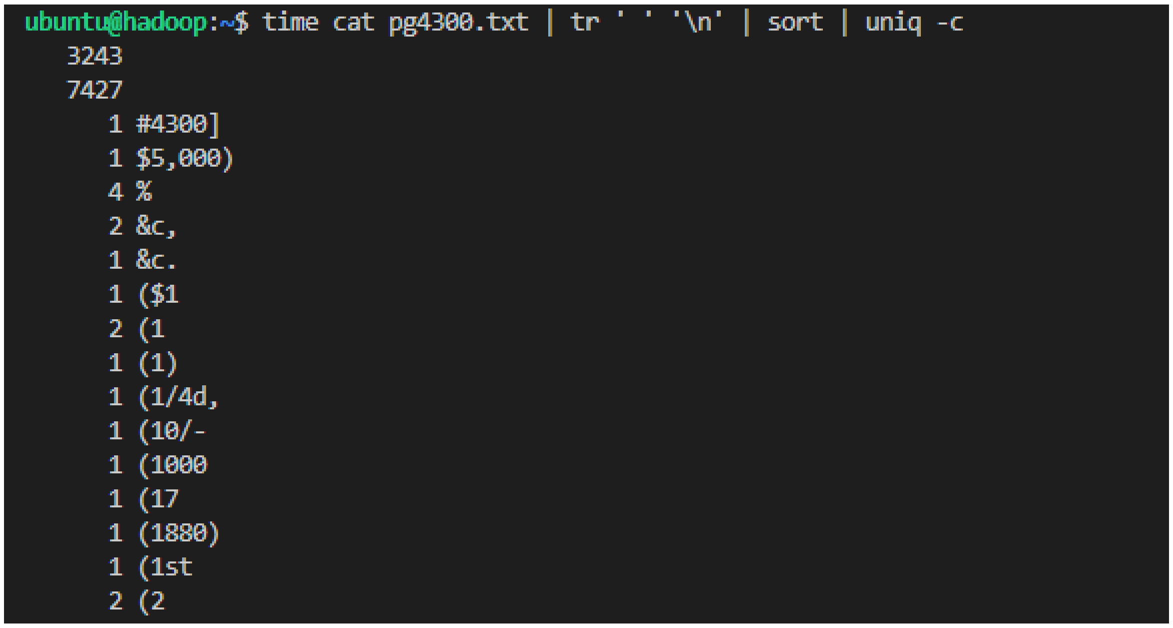


```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1586410
        File Output Format Counters
                Bytes Written=530405

real    0m29.775s
user    0m7.273s
sys     0m0.472s
ubuntu@hadoop:~$
```

Linux:

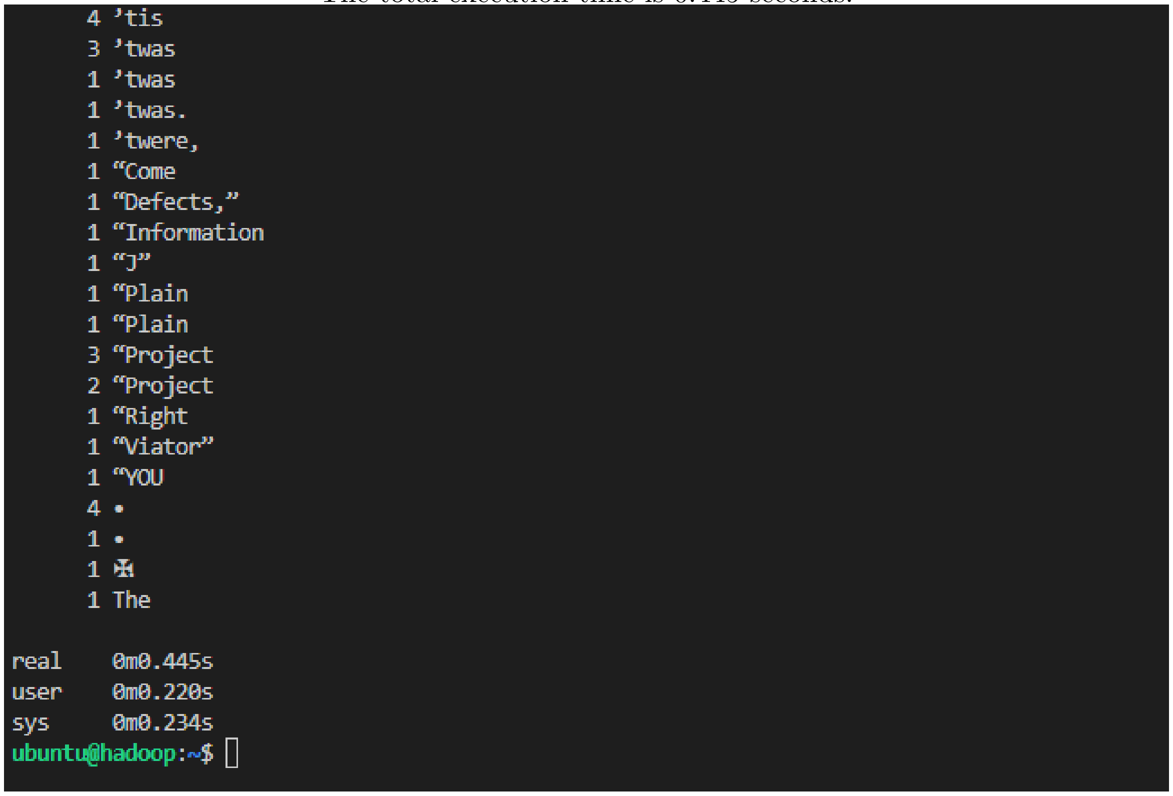
1. Run the following bash command to execute the linux version of the WordCount program:
   - time cat pg4300.txt | tr ' ' '\n' | sort | uniq -c
2. When the program terminates, observe the real execution time that will be printed to the screen at the end after showing the ouput

Image showing an execution of the linux WordCount program:



```
ubuntu@hadoop:~$ time cat pg4300.txt | tr ' ' '\n' | sort | uniq -c
   3243
   7427
      1 #4300]
      1 $5,000)
      4 %
      2 &c,
      1 &c.
      1 ($1
      2 (1
      1 (1)
      1 (1/4d,
      1 (10/-
      1 (1000
      1 (17
      1 (1880)
      1 (1st
      2 (2
```

The total execution time is 0.445 seconds:



```
      4 ’tis
      3 ’twas
      1 ’twas
      1 ’twas.
      1 ’twere,
      1 “Come
      1 “Defects,”
      1 “Information
      1 “J”
      1 “Plain
      1 “Plain
      3 “Project
      2 “Project
      1 “Right
      1 “Viator”
      1 “YOU
      4 •
      1 •
      1 ✠
      1 The

real    0m0.445s
user    0m0.220s
sys     0m0.234s
ubuntu@hadoop:~$ 
```

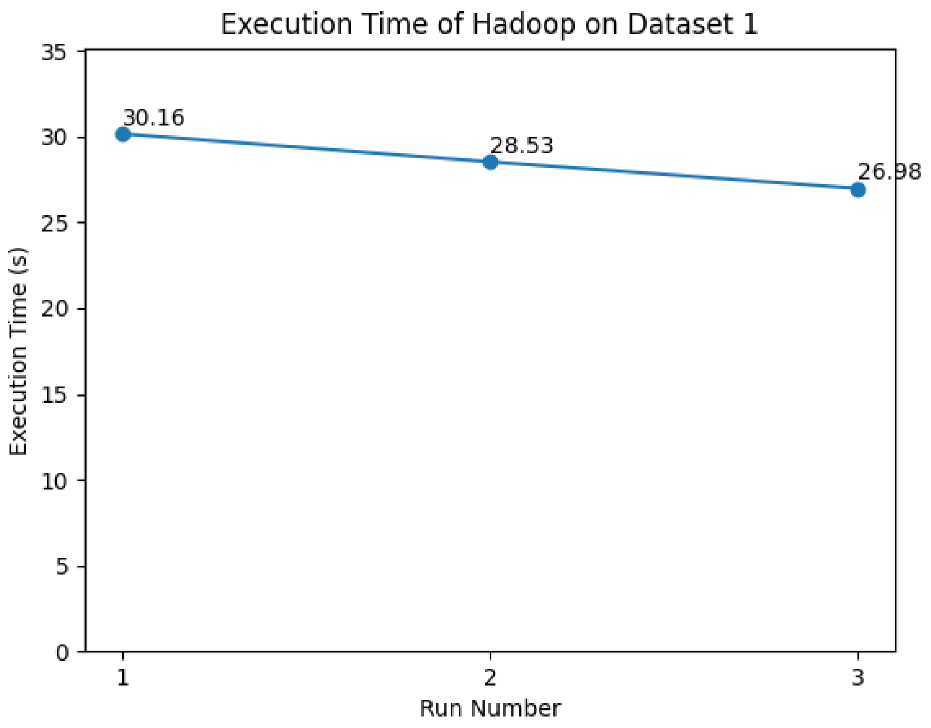
3 Performance comparison of Hadoop vs. Spark on AWS

Execution steps:

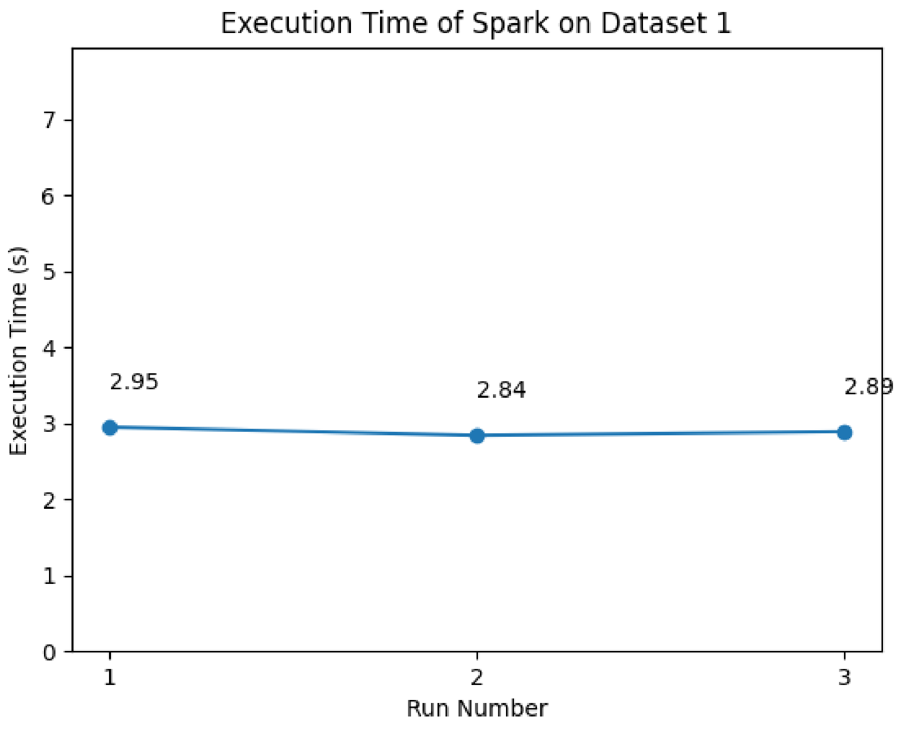
1. On your local machine, run the bash script automation_script.sh to create and configure an ec2 instance
2. After SSH'ing to the ec2 instance for the first time, run the bash script setup.sh to install and configure Hadoop and Spark
3. After the script terminates, the instance will reboot and continue setting up Hadoop and Spark in the background for a few minutes
4. SSH again to the ec2 instance
5. When the terminal seems responsive again, run the following command to update the environment variables:
   - source ~/.bashrc
6. Run the bash script time_comparison.sh to perform the WordCount program on Hadoop and Spark machines 3 times on each dataset. The script will plot all the execution times and report the average times using matplotlib.
7. Finally, run the following command on your local machine to send the graphs folder that was created to your local machine:
   - scp -i ms_kp_pem.pem -r ubuntu@ec2-xx-xx-xx-xx.compute-1.amazonaws.com: /home/ubuntu/graphs ~/Downloads/

Results:

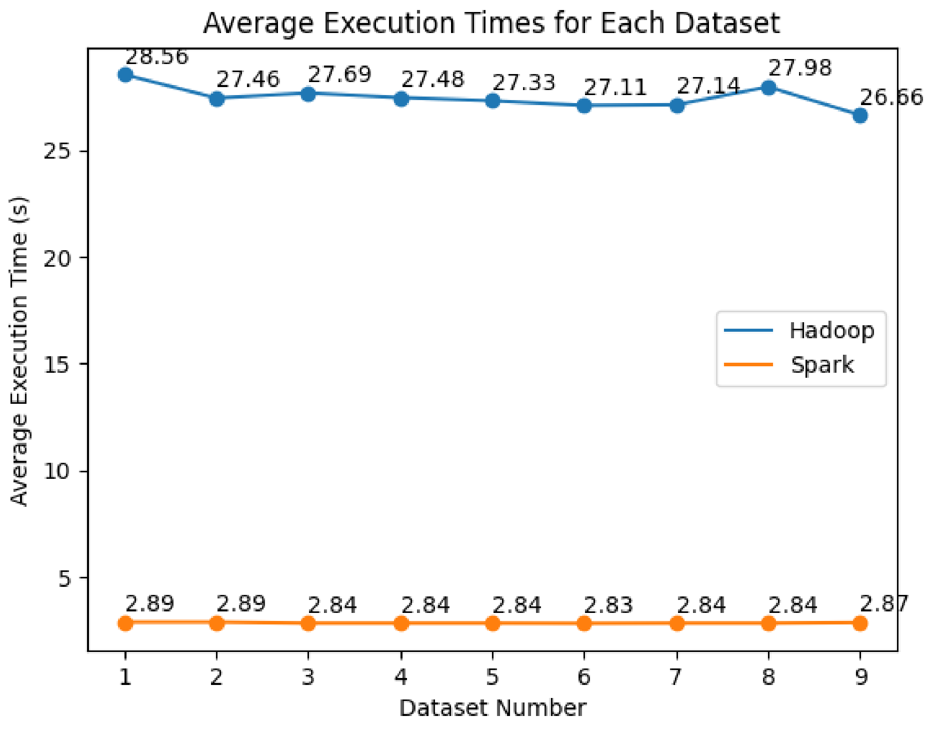
Here is an example of a graph representing the execution time of Hadoop on dataset 1:



Here is an example of a graph representing the execution time of Spark on dataset 1:



Here is the graph comparing the average execution time of Hadoop and Spark on all datasets:



Discussion: When analyzing the average execution times in the previous graph, we clearly see that the execution time on a Spark machine is much smaller than that on a Hadoop machine for all datasets. Therefore, the data collected supports the initial hypothesis that Spark should outperform Hadoop MapReduce.

4 Social network problem

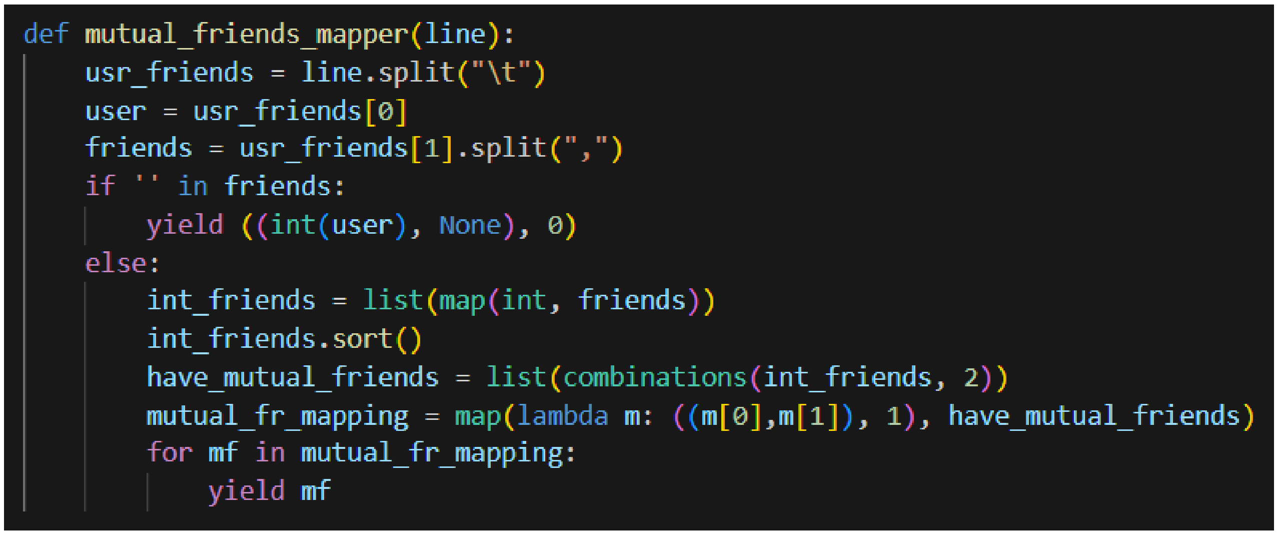
MapReduce to solve the social network problem
The input data was in this format: <User><TAB><Friends>
The output is needed to be in the following format <User><TAB><Recommendations>
To achieve that, we used the MapReduce paradigm in the following manner:

1. Map phase:
   For each line of the input data, we emit 2 sets of key-value pairs:
   - For the direct friends of a user, ((<User>, <Friend>), -1) was emitted where the key is (<User>, <Friend>) and the value is -1.
   - For the friends list, we run all the possible combinations of 2. The emitted keys represent the users who have the current user as a mutual friend. So the key is (<Friend1>, <Friend2>) and the value is 1 referring to the single mutual friend.
   - For those who have no friends, the key-value pair emitted was like ((<User>, `None`), 0).
2. Reduce phase:
   After the map phase, we have 3 different values in the emitted key-value pairs:
   - 1: users who are not direct friends but have a common one.
   - 0: users who have no friends.
   - -1: users who are direct friends.

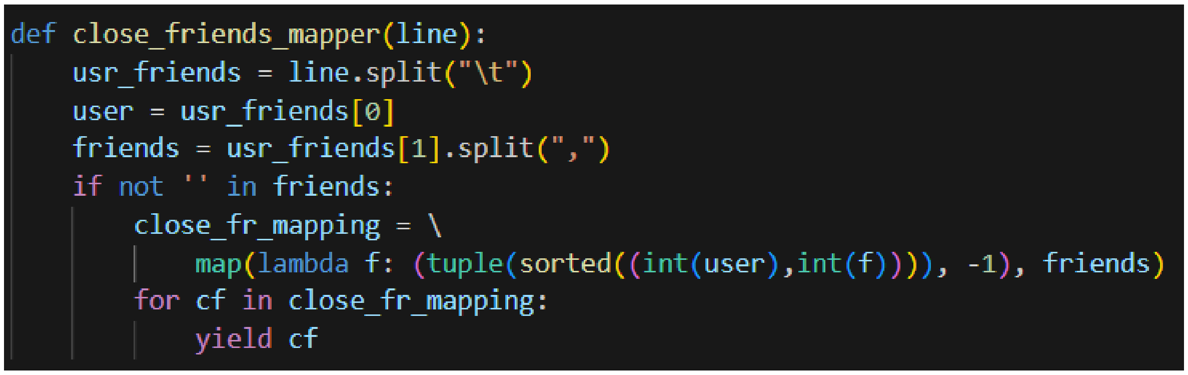
   In the reduce phase, we sum for each key the positive values indicating the number of mutual friends. If we encounter -1, we don't proceed with the recommendation since the 2 users are already friends. The key will be kept with its negative value (-1) to be dropped later. So at the end of the reduce phase, the key-value pairs will look like:
   - ((<User1>, <User2>), X) where X indicates the number of mutual friends.
   - ((<User1>, <User2>), -1) which means that the 2 users are close friends.
   - ((<User>, `None`), 0) for the users who don't have any friends.

Algorithm

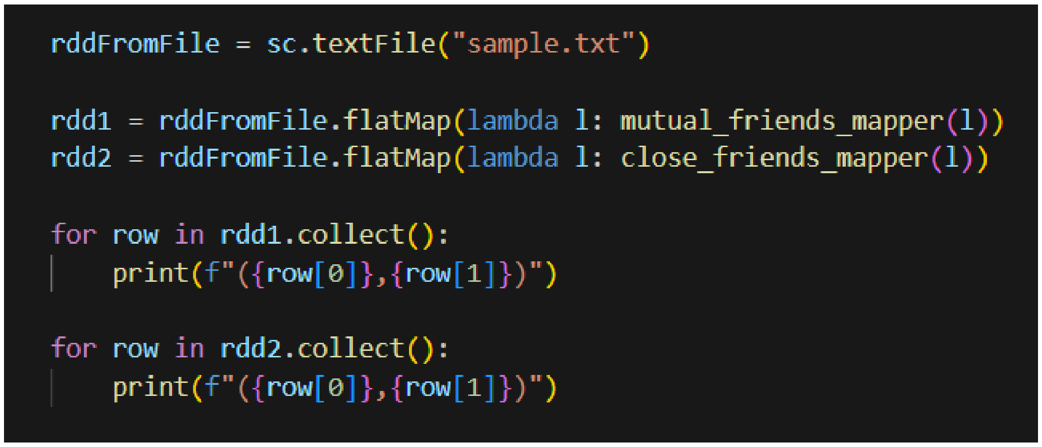
1. We define the first mapper to emit the set of key-value pairs of the users who have a mutual friend and those who have no friends.


```python
def mutual_friends_mapper(line):
    usr_friends = line.split("\t")
    user = usr_friends[0]
    friends = usr_friends[1].split(",")
    if '' in friends:
        yield ((int(user), None), 0)
    else:
        int_friends = list(map(int, friends))
        int_friends.sort()
        have_mutual_friends = list(combinations(int_friends, 2))
        mutual_fr_mapping = map(lambda m: ((m[0],m[1]), 1), have_mutual_friends)
        for mf in mutual_fr_mapping:
            yield mf
```

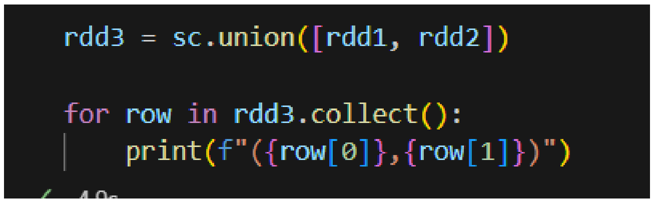
2. The second mapper will emit the set of key-value pairs of the users who are direct friends. We sort the key tuple values in ascending order.


```python
def close_friends_mapper(line):
    usr_friends = line.split("\t")
    user = usr_friends[0]
    friends = usr_friends[1].split(",")
    if not '' in friends:
        close_fr_mapping = \
            map(lambda f: (tuple(sorted((int(user),int(f)))), -1), friends)
        for cf in close_fr_mapping:
            yield cf
```

3. We execute the 2 mappers against each line of the file.

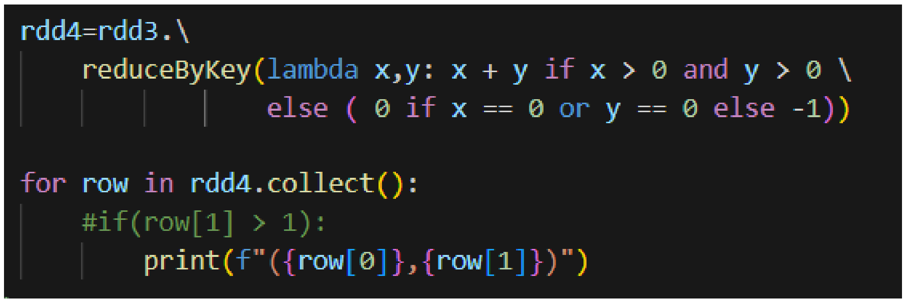

```python
rddFromFile = sc.textFile("sample.txt")

rdd1 = rddFromFile.flatMap(lambda l: mutual_friends_mapper(l))
rdd2 = rddFromFile.flatMap(lambda l: close_friends_mapper(l))

for row in rdd1.collect():
    print(f"({row[0]},{row[1]})")

for row in rdd2.collect():
    print(f"({row[0]},{row[1]})")
```

4. We merge the 2 RDD resulting from the execution of the mappers.



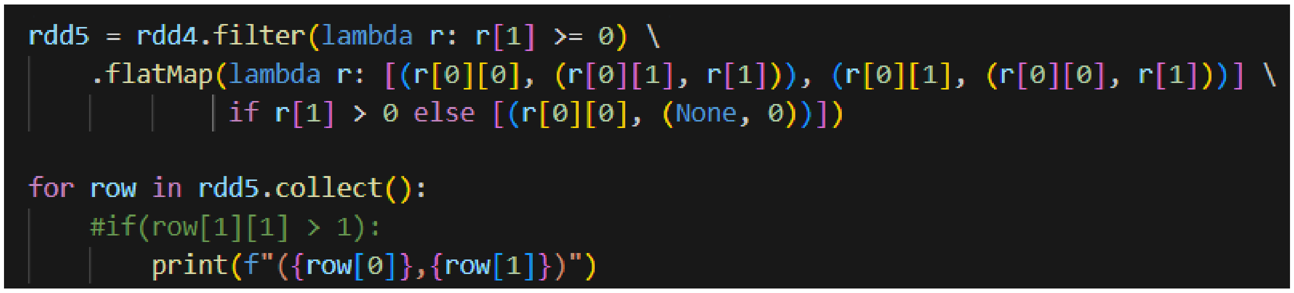
```
rdd3 = sc.union([rdd1, rdd2])

for row in rdd3.collect():
    print(f"({row[0]},{row[1]})")
```

5. We sum for each key the mutual friends if the values are positive. However, if one of these values is zero or negative, we keep it as such for that key.



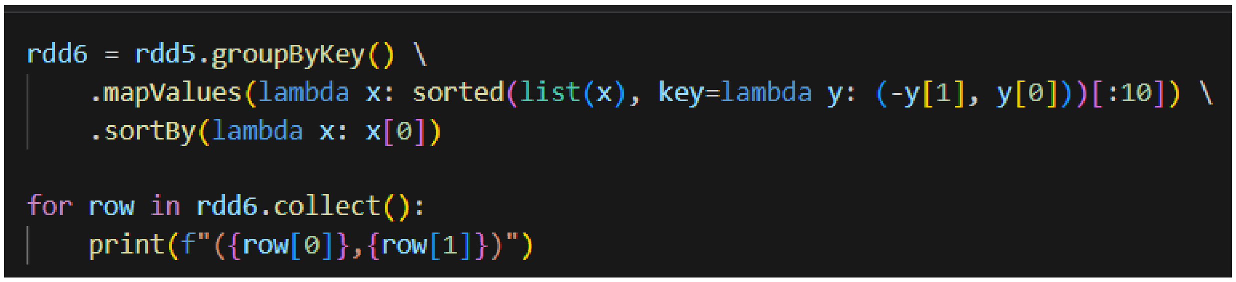
```
rdd4=rdd3.\
    reduceByKey(lambda x,y: x + y if x > 0 and y > 0 \
                else ( 0 if x == 0 or y == 0 else -1))

for row in rdd4.collect():
    #if(row[1] > 1):
        print(f"({row[0]},{row[1]})")
```

6. Drop the key-value pairs of the direct friends and map the elements of the dataset to create 2 new keys from the combination of the users of each key.
   Example: ((U1, U2), X) =>((U1,U2), X) and ((U2,U1), X)



```
rdd5 = rdd4.filter(lambda r: r[1] >= 0) \
    .flatMap(lambda r: [(r[0][0], (r[0][1], r[1])), (r[0][1], (r[0][0], r[1]))] \
            if r[1] > 0 else [(r[0][0], (None, 0))])

for row in rdd5.collect():
    #if(row[1][1] > 1):
        print(f"({row[0]},{row[1]})")
```

7. Group by key and sort the list of values by the number of mutual friends in descending order and then by the `User ID` in ascending order. Finally, we took the top 10 results.



```
rdd6 = rdd5.groupByKey() \
    .mapValues(lambda x: sorted(list(x), key=lambda y: (-y[1], y[0]))[:10]) \
    .sortBy(lambda x: x[0])

for row in rdd6.collect():
    print(f"({row[0]},{row[1]})")
```

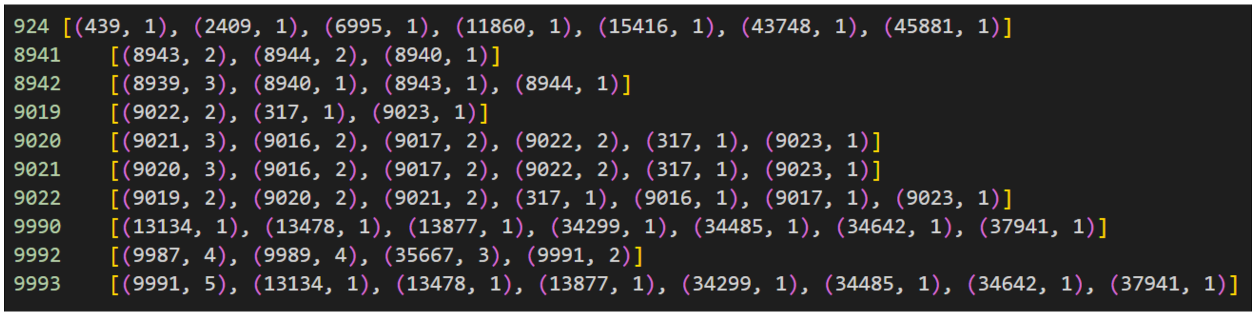
Recommendations
To interpret the results, it is important to understand their format:

(User, [(Recommendation1, X1), (Recommendation2, X2), ...])

Where:

- User is the user for whom recommendations are being made.
- Recommendation1, Recommendation2, etc., are the recommended users (friends).
- X1, X2, etc., are the number of mutual friends with the main user.

Below are the results for specific user IDs:



```
924 [(439, 1), (2409, 1), (6995, 1), (11860, 1), (15416, 1), (43748, 1), (45881, 1)]
8941    [(8943, 2), (8944, 2), (8940, 1)]
8942    [(8939, 3), (8940, 1), (8943, 1), (8944, 1)]
9019    [(9022, 2), (317, 1), (9023, 1)]
9020    [(9021, 3), (9016, 2), (9017, 2), (9022, 2), (317, 1), (9023, 1)]
9021    [(9020, 3), (9016, 2), (9017, 2), (9022, 2), (317, 1), (9023, 1)]
9022    [(9019, 2), (9020, 2), (9021, 2), (317, 1), (9016, 1), (9017, 1), (9023, 1)]
9990    [(13134, 1), (13478, 1), (13877, 1), (34299, 1), (34485, 1), (34642, 1), (37941, 1)]
9992    [(9987, 4), (9989, 4), (35667, 3), (9991, 2)]
9993    [(9991, 5), (13134, 1), (13478, 1), (13877, 1), (34299, 1), (34485, 1), (34642, 1), (37941, 1)]
```

Figure 1: Friend Recommendation Results (Reduced)

924 [(439, 1), (2409, 1), (6995, 1), (11860, 1), (15416, 1), (43748, 1), (45881, 1)]

8941 [(8943, 2), (8944, 2), (8940, 1)]

8942 [(8939, 3), (8940, 1), (8943, 1), (8944, 1)]

9019 [(9022, 2), (317, 1), (9023, 1)]

9020 [(9021, 3), (9016, 2), (9017, 2), (9022, 2), (317, 1), (9023, 1)]

9021 [(9020, 3), (9016, 2), (9017, 2), (9022, 2), (317, 1), (9023, 1)]

9022 [(9019, 2), (9020, 2), (9021, 2), (317, 1), (9016, 1), (9017, 1), (9023, 1)]

9990 [(13134, 1), (13478, 1), (13877, 1), (34299, 1), (34485, 1), (34642, 1), (37941, 1)]

9992 [(9987, 4), (9989, 4), (35667, 3), (9991, 2)]

9993 [(9991, 5), (13134, 1), (13478, 1), (13877, 1), (34299, 1), (34485, 1), (34642, 1), (37941, 1)]

Instructions to run the code
To execute the program, a configured Spark/Hadoop cluster is required. Ensure that the input file soc-LiveJournal1Adj.txt is accessible on the distributed file system. Follow the steps below:

1. Execute the bash script automation_script.sh on your local machine to create and configure an EC2 instance.

2. After the first SSH login to the EC2 instance, run the setup.sh bash script to install and configure Hadoop and Spark.

3. After the script completes, the instance will reboot and continue setting up Hadoop and Spark in the background for a few minutes.

4. Copy the code and input file to the instance using the following commands:

   - sudo scp -o StrictHostKeyChecking=no -i ms_kp_pem.pem friend_u_may_know.py ubuntu@ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/home/ubuntu
   - sudo scp -o StrictHostKeyChecking=no -i ms_kp_pem.pem soc-LiveJournal1Adj.txt ubuntu@ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/home/ubuntu

5. Reconnect to the EC2 instance via SSH.

6. When the terminal seems responsive again, update the environment variables with the command:

   - source ~/.bashrc

7. Launch the Hadoop cluster by running the following commands:

   - hdfs namenode -format
   - $HADOOP_HOME/sbin/start-dfs.sh
   - $HADOOP_HOME/sbin/start-yarn.sh
   - $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver

8. Upload the input file to Hadoop Distributed File System (HDFS):

   - hdfs dfs -put soc-LiveJournal1Adj.txt

9. Execute the Spark application in the cluster:

   - spark-submit friend_u_may_know.py

10. Finally, use the following commands (the first on the instance and the second on your local machine) to copy the part-00000 file with the results:

    - hadoop fs -copyToLocal recommendations ~/
    - sudo scp -i ms_kp_pem.pem -r ubuntu@ec2-xx-xx-xx-xx.compute-1.amazonaws.com: /home/ubuntu/recommendations/part-00000 ~/Downloads

Summary of results
In this work, we explore the WordCount program, revealing a total execution time of 33,130 seconds. Moving on to the performance comparisons, Hadoop showed a total execution time of 29.775 seconds, while Linux came out ahead with just 0.445 seconds. This notable difference between times highlights the efficiency of Linux in this context.

Now, coming to the comparison between Hadoop and Spark on AWS, the average execution time graph showed that Spark consistently performed better on all data sets. On the other hand, focusing our attention on the problem of social networks, specific friend recommendations were generated for users with distinct IDs. For example, user 924 received recommendations such as (439, 1), (2409, 1), and (6995, 1), which indicate potential connections based on mutual friends.

To reproduce and further explore the results in each section, follow the instructions provided to run the code in the specified AWS environment.