Polytechnique Montreal
LOG8415 : Advanced Concepts of Cloud Computing
Laboratory 2
MapReduce with Hadoop on AWS

Christian Njon
Dimitry Kamga
Michelle Sepkap Sime
Rui Jie Li

4th November 2022

1 Introduction

MapReduce on Hadoop and Spark using AWS is the subject of the second assignment for the LOG8415E course. The objectives of this assignment are to acquire some skills with large data technologies and learn to integrate issues and methods into the MapReduce paradigm. Four primary sections make up this report. First, we will discuss our Word Count application in Hadoop trials. Second, we compare Hadoop's performance to that of Linux. Third, we compare the performance of Spark and Hadoop on AWS. We wrap up by outlining our algorithm and the MapReduce tasks we used to tackle the social network problem. We present our recommendations for connections based on the algorithm.

2 Hadoop and Spark

2.1 Experiments with Word count Program

Here, we first prepare the lab setting by setting up Hadoop on our computer. We adhered to the assignment's guidelines. Our major goal was to use Hadoop to process a pg4300.txt file. So that the Hadoop Name Node and Data Nodes could share the file, we downloaded it to a local directory and then moved it to the Hadoop Distributed File System (HDFS). The data file pg4300.txt was then moved to the "input" directory we had just created in the Hadoop Distributed File System (HDFS). The wordcount.java program from the Hadoop example directory was then executed. The screen capture of the Hadoop settings on localhost is shown in Figure 1 and Figure 2. The input directory containing the pg4300.txt file is shown in Figure 3.

2.2 Performance comparison of Hadoop vs. Linux

In this part, we compared the word frequency computation capabilities of Hadoop with those of a standard PC running Linux.

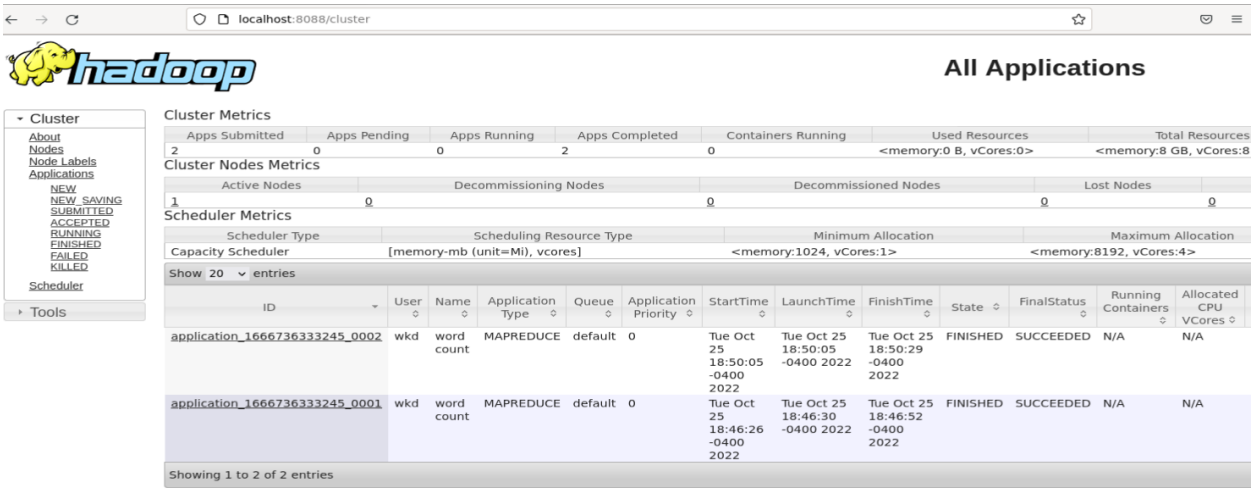

Figure 1: Hadoop Overview GUI - part1

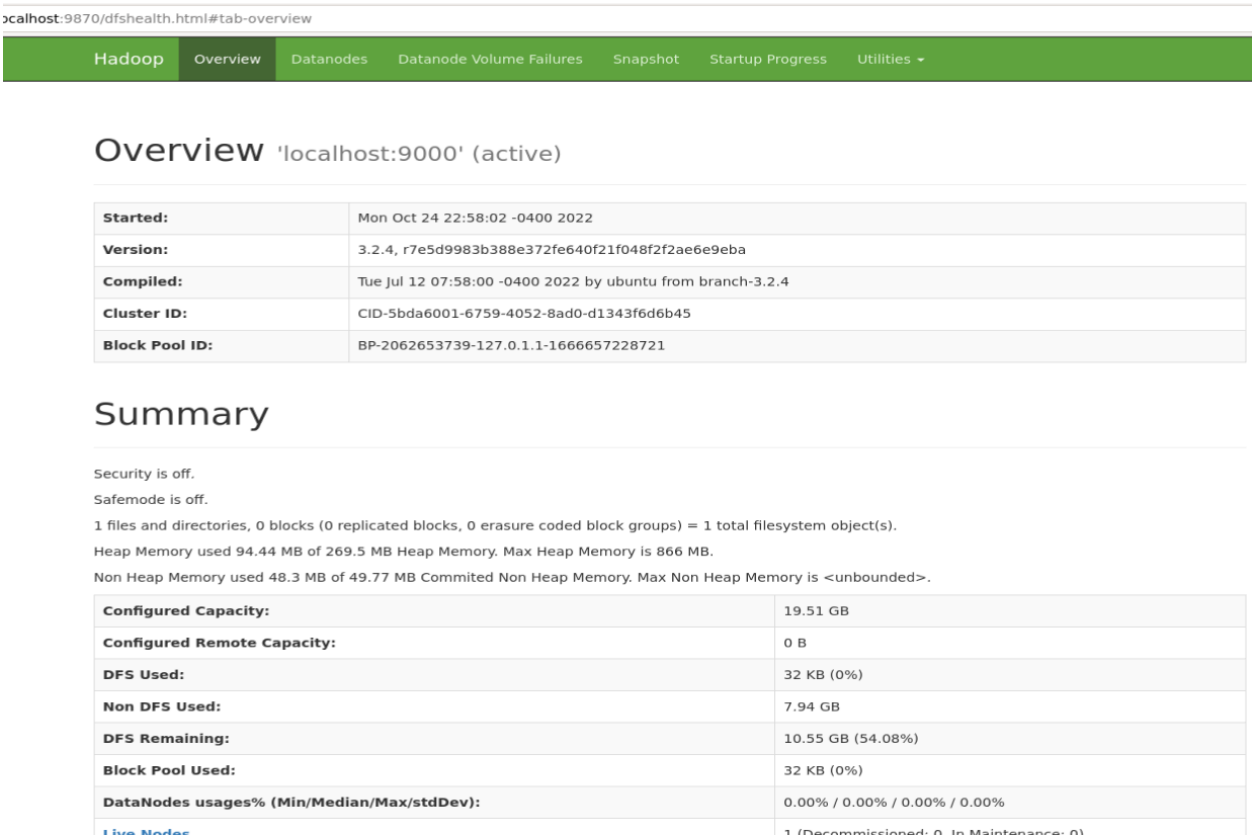

Figure 2: Hadoop Overview GUI - part2

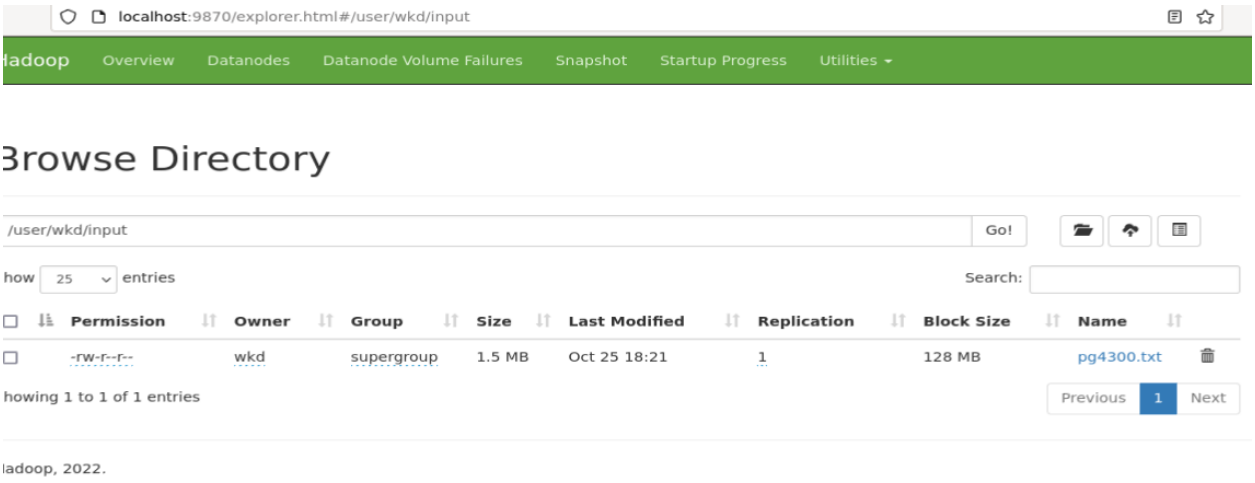

Figure 3: Hadoop directory

First we installed Hadoop and spark binaries. Then we ran the wordcount program with hadoop on a copy of James Joyce's Ulysses book page 4300 available at [1]. The wordcount program just counts how many times each word appears in a file. We also di the same on an AWS M4.Large instance using the command cat ./pg4300.txt |tr ' ' '\n' |sort |uniq -c. Here are the results:

Table 1: Hadoop vs Linux Wordcount

Hadoop	**Linux**
5.961s	0.170s

As we can see, Linux completes the task more quickly than Hadoop. This is expected because Hadoop is acceptable or suitable for more sophisticated tasks than the one we used.

2.3 Performance comparison of Hadoop vs. Spark on AWS

We first set up our infrastructure as follows in order to compare the performances of Hadoop and Spark on AWS. We generated a M4.large linux Ubuntu instance , and we installed Hadoop 3.3.4 and Spark on it. We confirm the installation of all necessary packages. Then, we timed the WordCount program's execution on both Hadoop and Spark machines three times across the entire dataset. We used 9 text files for this comparison, they can be found in the Datasets folders in Lab2, index shown in Figure 4.

Spark was anticipated to be considerably faster than Hadoop since it makes use of random access memory and this is the case indeed. The results are shown in the Figure 5.

This experiment and the last one has been automated through a bash script described in the section 3.

Dataset	Name	index
https://tinyurl.com/4vxdw3pa	buchanj-midwinter-00-t	input1
https://tinyurl.com/kh9excea	carman-farhorizons-00-t	input2
https://tinyurl.com/dybs9bnk	colby-champlain-00-t	input3
https://tinyurl.com/datumz6m	cheyneyp-darkbahama-00-t	input4
https://tinyurl.com/j4j4xdw6	delamare-bumps-00-t	input5
https://tinyurl.com/ym8s5fm4	charlesworth-scene-00-t	input6
https://tinyurl.com/2h6a75nk	delamare-lucy-00-t	input7
https://tinyurl.com/vwvram8	delamare-myfanwy-00-t	input8
https://tinyurl.com/weh83uyn	delamare-penny-00-t	input9

Figure 4: Dataset index

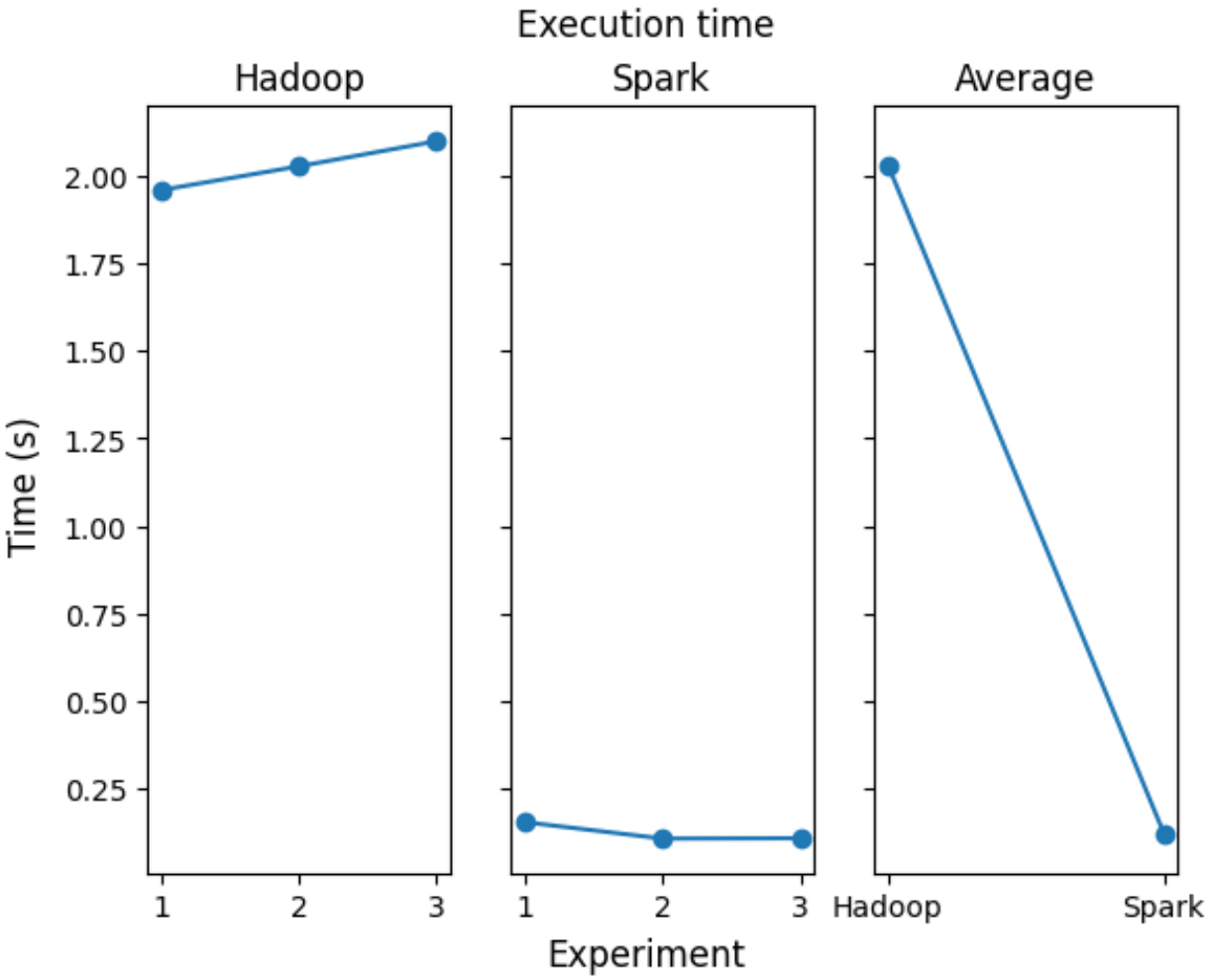

Figure 5: Performance comparison of Hadoop and Spark

3 Instructions to run the code

The entry point of the project is the bash script **run.sh** located at the root level of Lab2 folder. What this script does is simply to schedule all steps that need to be done in order to have the performance results. First, it checks whether the necessary credentials (aws_access_key_id, aws_secret_access_key, aws_session_token) and the region config are set. The check proceeds this way :

- Check if the default values have been set by aws cli by means of configure command,
- If not, check if they are available among the environment variables,
- If they aren't, get them from user input and export them to make them available for upcoming scripts.

Once credentials and minimum config are set, we create and activate a python virtual environment to install dependencies so that user python environment remains unchanged. Then, we deploy and setup infrastructure, it is composed of one M4.Large instance and a security group to allow SSH access. If the setup fails to complete, we teardown already created infrastructure and exit. During the setup we store SSH private key and public IP address for later use.

Infrastructure step completed, we go to the next one which is executing hadoop and spark programs via SSH (using paramiko python library), saving execution time to files (results.txt for hadoop vs linux, hadoop.txt for hadoop performance and spark.txt for spark performance) that are also retrieved by SSH. We save those results in files and plot them. Finally, as soon as we have all we wanted, the infrastructure is destroyed and the virtual environment deactivated.

4 Social media problem

4.1 Input file

The input file has the following format:
userID of a user, followed by a TAB character, followed by the ids of the friends of the user separated by a comma. For example,
1 2,3
3 1,2
means that user 1 is friends with user 2 and 3, and user 3 is friends with user 2 and 1. Each user and their friends is on a different line (separated by a ENTER character).

4.2 Map using Hadoop (Java)

In the map() function, Hadoop automatically splits the input file into lines. The function responsible for this is
`MutualFriends.FriendOfFriendsMapper.map(Object key, Text value, Context context)`
This function receives a value containing a string in the format of `userID+TAB+list of friends separated by a comma`, for example `1TAB2,3,4`, or if the user didn't add anyone, it will look like `1TAB`. The map functions works by

1. Splitting the value by the TAB character, which will result an an array of strings in the format of `[characters before the TAB, characters after the TAB]`. If there are no characters following the TAB, the array will have the format of `[characters before the TAB]`. For example, if the value is `1TAB2,3,4`, the array will be `["1", "2,3,4"]`; if the value is `1TAB`, the array will be `["1"]`

2. if the array resulting from splitting by the TAB character have a length of 1, that means the user did not add any friends yet. The key and the sent to the reducer will be

   (a) key = user ID
   (b) value = user ID + TAB + null

   For example, if the value is "1TAB", the output to the reducer will be

   (a) key = 1
   (b) value = 1TABnull

3. if the array resulting from splitting by the TAB character has a length of more than 1, it means the user added at least some friends. The map function will then create a variable `friendList = user ID + TAB + friends of user ID`. Then, for every user `u` that the user with `user ID` has added (the list can be obtained by splitting the second value of the array by the character ,), the mapper will output to the reducer

   (a) key = u
   (b) value = friendList

   For example, if the array is ["1", "2,3,4"], the mapper will output

   (a) key = 2
   (b) value = 1TAB2,3,4
   (c) key = 3
   (d) value = 1TAB2,3,4
   (e) key = 4
   (f) value = 1TAB2,3,4

   This ensures that the reducer will have every user along with all their friends of friends, and from which friend they come from. For example, if we have

   (a) 1 TAB 2,3,4,5
   (b) 6 TAB 2,3,4,5,7

   For the user 2, the reducer will see

   (a) key = 2; value = 1TAB2,3,4,5
   (b) key = 2; value = 6TAB2,3,4,5,7

   From this, we see that 2 can reach 2,3,4,5 via 1, and 2 can reach 2,3,4,5,7 via 6.

4.3 Reduce with Hadoop (Java)

The reducer collects, for each user, all their friends of friends along with from which friend these come from. For example, if 2 can reach 3 via 1 and 4, then 2 and 3 have two mutual friends, namely 1 and 4. The reduce function basically counts how many times a user ID appears in this lists, and returns the ones that appear most often after removing the ones that the user have already added. If the friend list contains null, the reducer outputs an empty list of suggestions. If not, the following steps will be taken: for example, if we have

- key = 2
- value = 1TAB2,3,4,5

Let `alreadyAdded` be a hashset of hashsets of strings that contains all the users that 2 has already added and `mutualFriendsCount` be a HashMap that uses a HashSet of strings as a key and an integer as value. The steps executed by the reducer will be the following (same steps for each value corresponding to 2):

1. The string `1TAB2,3,4,5` is splitted by tab to get `["1", "2,3,4,5"]`, and the value `"2,3,4,5"` is splitted by `,` to get the array `potentialFriends = [2, 3, 4, 5]`
2. The reducer will add `HashSet[(2)]` and `HashSet[(2,1)]` in `alreadyAdded` since 2 cannot add 2 and 2 has already added 1.
3. For each user `u` in the list `potentialFriends`, a `HashSet h = HashSet([2,u])` will be created. If h is already in `mutualFriendsCount`, it will be added as a key with a value of 1. If it's already added, we will increase its value by 1.

After all the values associated with 2 have been processed, the keys contained in `alreadyAdded` will be removed from `mutualFriendsCount`. The rest will be stored in an array of `Pair` objects, with a `count` attribute that indicates the number of mutual friends and a `uid` attribute that indicates the user ID. The array is then sorted by `Pair.count` (descending order) then by `Pair.uid` (ascending order). The uid of the 10 first Pair objects (or the whole array, if less than 10 friends of friends) will be returned. Example:

- key = 2
- value = 1TAB2,3,4,5
- value = 6TAB4,3,7,5
- value = 8TAB6,3,4,5

After processing all the values, `alreadyAdded` will contain

```
HashSet([
    HashSet([2]),
    HashSet([2, 1]),
    HashSet([2, 6]),
    HashSet([2, 8])
])
```

And `mutualFriendsCount` will contain

```
HashMap([
    HashSet([2   ]): 1,
    HashSet([2, 1]): 1,
    HashSet([2, 3]): 3,
    HashSet([2, 4]): 2,
    HashSet([2, 5]): 3,
    HashSet([2, 6]): 2,
    HashSet([2, 7]): 1,
    HashSet([2, 8]): 1,
])
```

After removing the contents of `alreadyAdded` from `mutualFriendsCount`, `mutualFriendsCount` will contain

```
HashMap([
    HashSet([2, 3]): 3,
    HashSet([2, 4]): 2,
    HashSet([2, 5]): 3,
    HashSet([2, 7]): 1,
])
```

And an array containing

```
array = [
    Pair{uid="3", count= 3},
    Pair{uid="4", count= 2},
    Pair{uid="5", count= 3},
    Pair{uid="7", count= 1},
]
```

will be produced, then sorted first according to count, then according to uid (ex. ”3” comes before ”5” because 3 ¡ 5, even though they both have three mutual friends with 2):

```
array = [
    Pair{uid="3", count= 3},
    Pair{uid="5", count= 3},
    Pair{uid="4", count= 2},
    Pair{uid="7", count= 1},
]
```

The output will be

- key = 2
- value = 3,5,4,7

4.4 Result

For the following users, we have:
924 439,2409,6995,11860,15416,43748,45881
8941 8943,8944,8940
8942 8939,8940,8943,8944
9019 9022,317,9023
9020 9021,9016,9017,9022,317,9023

9021 9020,9016,9017,9022,317,9023
9022 9019,9020,9021,317,9016,9017,9023
9990 13134,13478,13877,34299,34485,34642,37941
9992 9987,9989,35667,9991
9993 9991,13134,13478,13877,34299,34485,34642,37941
All the users have received either 10 recommendations or as many as possible. For example, one of the result is
0 38737,18591,27383,34211,337,352,1532,12143,12561,17880
we can verify manually that 0 and 38737 have 5 friends in common, and 0 and 18591 have 4 mutual friends in common, and so on.

4.5 How to run the program

All the code for running the program is in compile.sh. The only parts that need to be changed manually is the input and the output directory, along with the hadoop path (if necessary). To run compile.sh, just type ./compile.sh in its directory.

4.6 Results on Azure

We also run this program on Azure (this is a screenshot of the content of the output file):

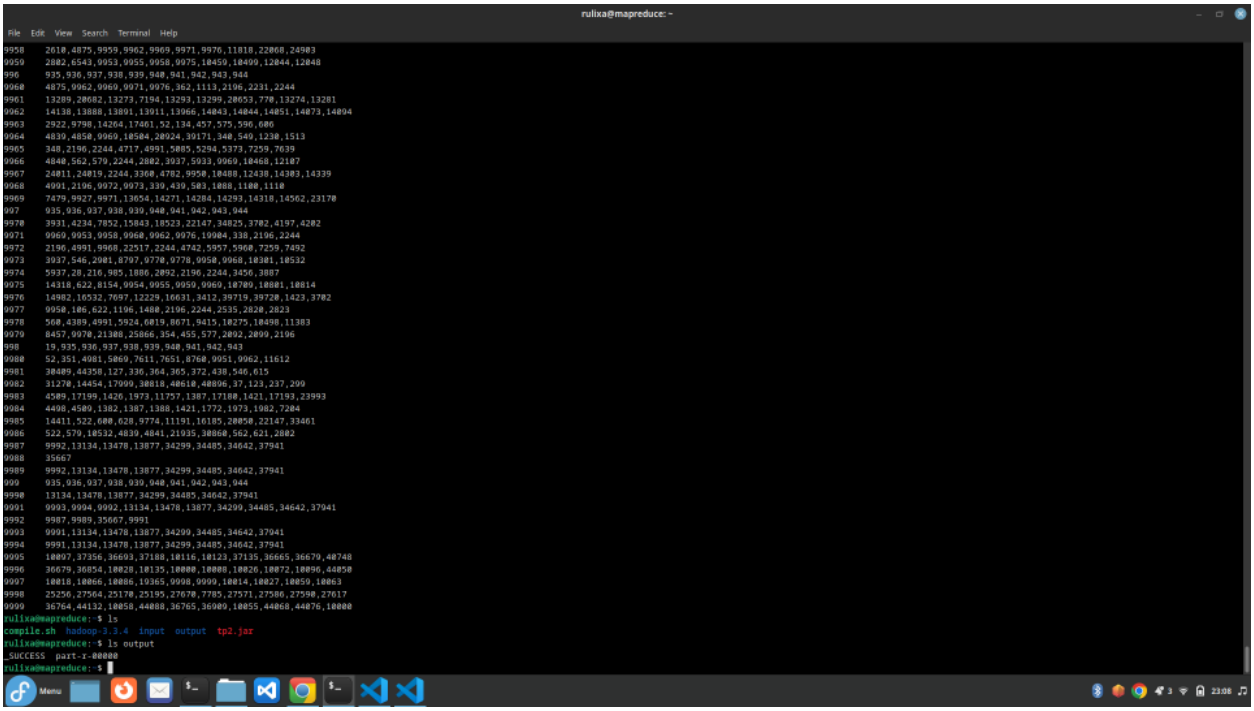

Figure 6: Output of MapReduce from Microsoft Azure

5 References

[1] Gutenberg textual data. http://www.gutenberg.org/cache/epub/4300/pg4300.txt.
[2] Github repo. https://github.com/MichelleSS1/Lab8415