LOG8415E - Assignment 2

Marie-Lee Brault (2050165)
Philippe Côté-Morneault (1905783)
Samuel Saito-Gagné (1902565)

Monday, November 16, 2020

1 Experiments with WordCount

1.1 Methodology

In order to experiment with a Word Count program we used the datasets specified in the assignment renamed as follows:

- data1.txt (574 KB) (http://goo.gl/9GqADe)
- data2.txt (228 KB) (https://www.gutenberg.org/files/28299/28299-0.txt)
- data3.txt (127 KB) (https://tinyurl.com/y2somksc)
- data4.txt (425 KB) (http://www.gutenberg.org/cache/epub/39850/pg39850.txt)
- data5.txt (74 KB) (http://www.gutenberg.org/cache/epub/8578/pg8578.txt)

Instead of referring to these datasets by their original filename, we will refer to them by their index (data1, data2, etc.). We will also refer to their file size in our analysis, since file size has an impact on WordCount's execution time.

The objective of the experiment is to compare and analyze the performance of using Hadoop and HDFS vs. Spark and S3 vs. Spark and HDFS vs Linux on Master Node of a WordCount program with each of the dataset. In order to test the performance, we perform each of these methods 5 times of each dataset and reported the average execution time.

1.1.1 HDFS

Both Hadoop and Spark used HDFS to perform the WordCount program. HDFS is a distributed file storage system. To setup HDFS on our cluster, we created an input directory to store all input datasets. We then downloaded all input datasets and copied them to HDFS. This is all automated with scripts.

1.1.2 S3

Spark used S3 to read and write data for the WordCount program. We created a bucket on AWS S3 and uploaded all datasets manually before performing the experiments. We then used *boto3* and *pandas* to read and write to the bucket from our cluster within Python scripts. These modules were installed on the cluster using scripts.

1.1.3 Linux command

The Linux command used for WordCount is simply a combination of replacing all spaces with newlines, sorting these lines and counting the occurrences of each lines. This is done by reading and writing the data directly from the cluster's disk. We then use the Linux *time* command to measure the execution time of this pipeline of commands. We have decided to use the *real time* from the output as the total execution time of the WordCount program since it measures from start to finish how much time has elapsed.

1.1.4 Hadoop and HDFS

To perform the WordCount experiment using Hadoop and HDFS, we first setup HDFS as mentioned in a previous section. Then, we created a Python script for the map and reduce jobs that would be executed by Hadoop. These mapper simply outputs each word. The reducer simply counts the occurences of each words and writes it to HDFS. The Linux *time* command is used to measure the execution time of Hadoop's WordCount. We decided not to take any time that came from Hadoop's output as it would not be fair for the Linux command. Hadoop's output provided with various times for each map and reduce job. We feel like it would not be accurate to only take the time taken for the MapReduce jobs and ignore the total execution time that includes any overhead associated with Hadoop. We feel like any overhead by this method should be taken into account, hence we use the *real time* from the *time* command in our results.

1.1.5 Spark and HDFS

The WordCount program written using pyspark reads the data directly from the HDFS of the cluster then performs pre-processing steps: remove the punctuation, lowercase all the words, and split them. Then the map function emits 1 for each word, and the reducer function aggregates the counts according the the work key. The resulting output and then stored directly on HDFS in the output folder. As mentioned earlier, we also used the *time* command to store the execution time in order to account for the overhead.

1.1.6 Spark and S3

The WordCount program written using pyspark reads the data directly from the s3 bucket then performs pre-processing steps: remove the punctuation, low-

ercase all the words, and split them. Then the map function emits 1 for each word, and the reducer function aggregates the counts according the the work key. The resulting output and then stored directly on s3 in a separate folder. As mentioned earlier, we also used the *time* command to store the execution time in order to account for the overhead.

1.2 Results

The raw data from all executions can be found in the appendix of the present document. Table 1 presents the average execution time for each dataset. Since we ran the WordCount program five times on each dataset, this table only contains the average execution time. This table also includes the average execution time of all datasets for each method.

Table 1: Average execution time

Method	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average time per method (s)
Linux	0.218	0.0866	0.0438	0.1672	0.0248	0.1081
Hadoop	41.2598	41.7624	41.1296	41.6154	41.3564	41.4247
Spark HDFS	25.0682	25.0354	25.2984	25.2042	25.6422	25.2497
Spark S3	26.1634	26.3968	25.9646	26.3128	25.6336	26.0942

Figure 1 presents the average execution time of all datasets for each method in seconds. Figures 2 to 6 present the average execution time for individual datasets for each method. As a reminded, the execution time is defined as the output of the *time* command and not what is provided by Spark's or Hadoop's output.

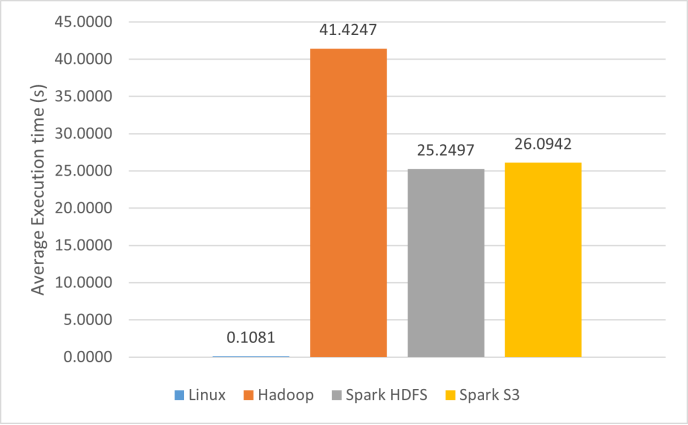

Figure 1: Average execution time for all datasets

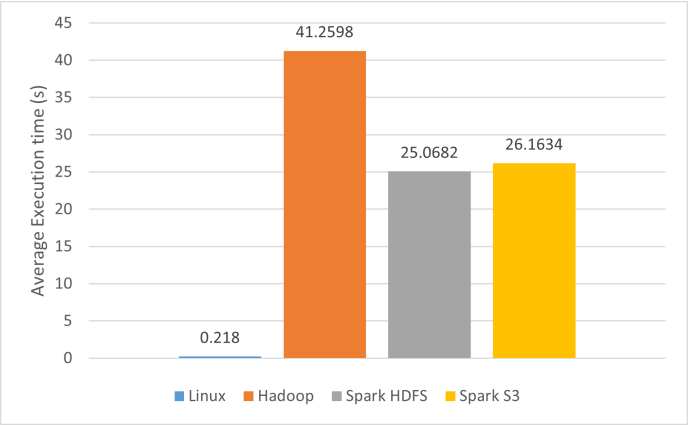

Figure 2: Average execution time for dataset 1 (574 KB)

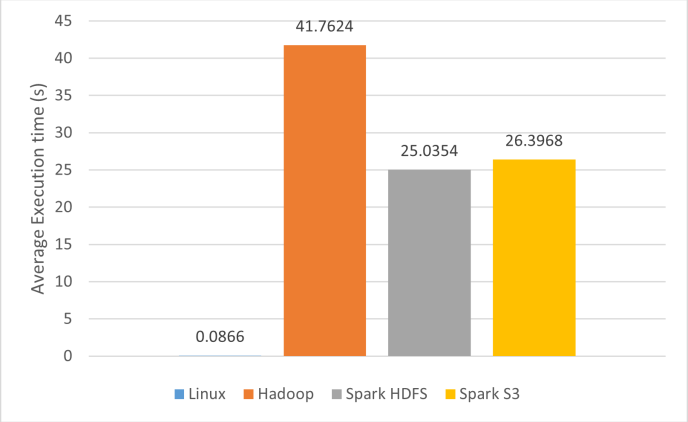

Figure 3: Average execution time for dataset 2 (228 KB)

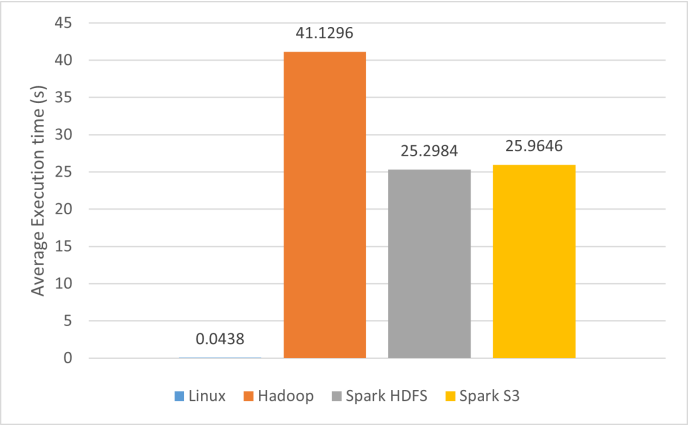

Figure 4: Average execution time for dataset 3 (127 KB)

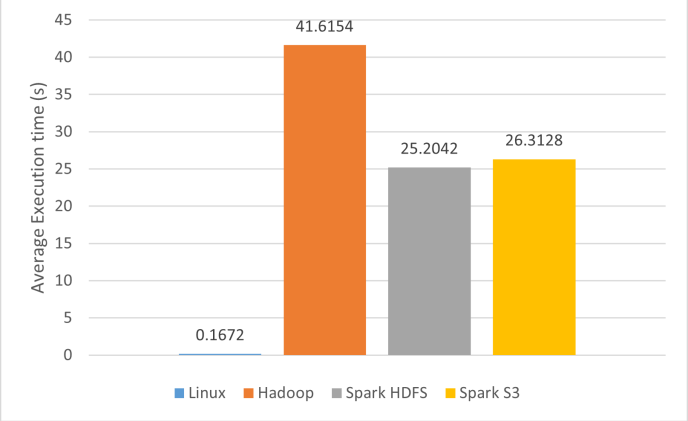

Figure 5: Average execution time for dataset 4 (425 KB)

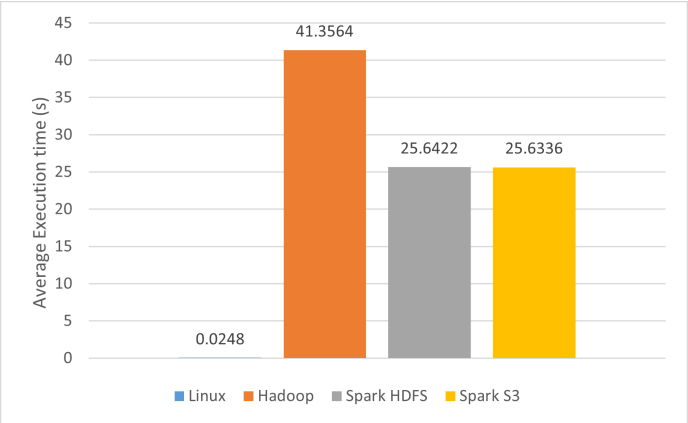

Figure 6: Average execution time for dataset 5 (74 KB)

1.3 Performance comparison of Hadoop vs Linux vs Spark

These results are very surprising at first. The execution time of the Linux command is roughly 200 times less than of Spark's. We could expect Spark to perform the best since it stores intermediate results in-memory and uses the MapReduce paradigm, yet the Linux command's performance is incredibly better.

These results actually have a very simply explanation. The largest dataset was only 574 KB. This is in no way Big Data. MapReduce is usually appropriate for dealing with large datasets. The MapReduce paradigm is all about doing parallel computations on different nodes. The Linux command is simply executing a chain of commands in serial. The MapReduce paradigm is very efficient when we are dealing with a lot of data since the Linux command's efficiency is lacking, hence the Linux command's performance drops when adding more data. However, MapReduce's smart design comes with the cost of initial overhead. There is a significant amount of time required to create various map and reduce jobs, schedule them, send them on different nodes, handle faults and manage all jobs. This overhead becomes insignificant when you have a lot of data to process because the actual computations in the MapReduce jobs become much more demanding than the MapReduce setup.

Another reason that could explain these results is that the Linux command was executed locally on a master node when the other methods where distributed across nodes. We were not able to verify this, but our assumption is that running the Linux command through SSH on the master node only executed it on the master node. Since the data is stored locally on the master node, this would imply that the master node did not have to use network resources to execute the WordCount program. All other methods were distributed across nodes, which

requires network usage and adds latency. Being able to perform the program locally for such a small amount of data gave the Linux command an advantage over other methods.

We also noticed that Spark was about twice as fast as Hadoop. This could be easily explained by the fact that Spark processes data in-memory whereas Hadoop has to store intermediate results on HDFS. Performance comparisons between Hadoop and Spark have been made many times in the literature and Spark's performance usually outperforms Hadoop by a large margin.

Unfortunately, our datasets were too small to test our hypothesis that Hadoop and Spark would start to perform better than the Linux command when increasing the input size. For future work, it would be interesting to have various datasets that have a wide range of sizes. This would allow us to compare the performance of each method according to specific dataset sizes.

1.4 Experiments to answer BI questions

We queried the dataset provided in the assignment previously stored in s3 by using SparkSQL commands to answer the following business intelligence questions. We made multiple assumptions during this exercise.

1.4.1 Assumptions

- A distinct member is represented by a unique member_id. When referring to the number of members, we counted only distinct members in order to count the real number of members and not the number of transactions.
- For question 4, a single member_id's "vip" attribute can be both "True" and "False" (e.g. member_id = 100011). Hence, the number of VIP members is the number of VIP minus duplicate member_ids to keep only a distinct list. (The results would be 1303, if we queried directly from the distinct members query from question 1).
- The country code for Canada is "CA".
- For question 5, a single member_id's "gender" attribute can be both "Female" and "Male" (e.g. member_id = 100011). Hence, the number of Canadian female members who purchased a product from zone 7 is the number of Canadian females who purchased from zone 7 minus duplicate member_ids to keep only a distinct list. (The results would be 35, if we queried directly from the distinct members query from question 1).
- For question 6, the maximum amount of price means finding the product_id of the transaction with the highest amount.
- The same product_id can change its amount in different transactions.

1.4.2 Results BI questions

1. How many distinct members are included in the data set?
   2500 distinct members

2. When did the first purchase happen?
   The first purchase happened on **2007-10-12**

3. What are the different card types used in purchasing?
   There are 16 different card types. The different card types are the following: **jcb, visa-electron, maestro, diners-club-international, solo, americanexpress, diners-club-enroute, china-unionpay, bankcard, switch, mastercard, instapayment, visa, diners-club-carte-blanche, diners-club-us-ca and laser**

4. How many VIP members do we have in our data set?
   2486 VIP distinct members

5. How many Canadian female members purchased an item from Zone7?
   322 distinct Canadian female members

6. Which item has the maximum amount of price?
   The item with the maximum price has **a product_id of 330092 and a price of $9999.87**.

7. What are the top 3 countries from which people made purchases?
   The top 3 countries from which people made purchases are **China (CN)** with 15 026 purchases, **India (ID)** with 9 039 purchases and **Russia (RU)** with 4 789 purchases.

8. What are the relevant patterns for each column of the dataset?
   Refer to next section

9. There are three rows in the dataset that can be identified as anomaly.
   Refer to next section

1.4.3 Anomaly Detection

For the anomaly detection, we have used the following methodology:

1. First, for each column of attributes, use simple MAP and REDUCE functions to crawl through the data and get general information: length of value, type of data (`string`, `boolean`...).

2. From the results of the previous step and by querying some example rows, this gives us enough information to create recognize some patterns and do some basic regexes accordingly.

3. Finally, with those regexes, we can filter the rows of data to find the anomalies for each column. Additionally, if applicable, we can also detect any outliers regarding the column's specific logic with `regexp_extract`.

From this methodology the following patterns (regexes) have been observed:

1. ***member_id*:** The regex "`^\d{6}$`" has been used to represent a number of 6 digits. We noticed that all IDs start with "10" so technically, the regex could be "`^\10d{4}$`" for this dataset but we decided to keep the pattern more general.

2. ***date*:** The regex "`^(\d\d\d\d)-(\d\d)-(\d\d)$`" has been used for the date, with the groups representing the year, month and day, respectively. Additionnaly, we made sure that the values are valid: the day value should be between 1 and 31, the month should be between 1 and 12 and the year should not be greater than 2020. With more information, we could also add a lower bound for the year, but we found that the lowest year was 2007. We did not verify if the number of days in the month was valid, as we did not think it was necessary to find the anomaly.

3. ***country*:** We used the regex "`^[A-Z][A-Z]$`" to validate the country, as all codes are made of 2 capital letters from what we have observed.

4. ***gender*:** For the gender, we simply checked if the value was in "`['Female', 'Male']`".

5. ***ip_address*:**The regex "`^(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})$`" was used to represent the IP address. With `regexp_extract`, we checked the validity of each of the 4 groups to see if the value fit in 0 to 255. It would have been possible to forego the use of `regexp_extract` and put the validation in the regex as well.

6. ***amount*:** The regex "`^\$(0|[1-9]\d*)\.(\d{2})$`" was used to represent the amount. Basically, the amount always starts with the dollar symbol, followed by the dollars and the cents (with a precision of 2) separated with a dot. We also made sure to detect an invalid amount with trailing zeros (e.g. "0123.99"), but it did not seem to matter for this dataset.

7. ***vip*:** For VIP, the value is either "`['true', 'false']`" to represent a boolean.

8. ***product_id*:** The regex "`^(\d{3})(\d{3})$`" was used for the product ID. Just like the member ID, we noticed it was made of 6 digits. We noticed that the ID always started with "330" (result of the first capture group). Again, we kept the pattern more general, but we could have used the regex "`^330(\d{3})$`" instead.

9. ***card_type*:** The regex "`^[a-z]+(?:-[a-z]+)*$`" was used for the card type. We did not have much information on this column and the possible values were really diversified, from a length of 3 to 25. However, we noticed that all the values were lowercase and sometimes had hyphens, so we deduced that the pattern could be represented with the regex above. We could also have simply checked if the card type was one of the 16 types

present in the dataset, but that would have assumed that no other card types existed and would detect the new ones as anomalies, so we rejected that idea.

10. ***serial*:** For the serial number, we used the regex "`^\d{3}-\d{2}-\d{4}$`". From our experiments, we observed that the serial number is made of 3 groups of 3, 2 and 4 numbers separated by hyphens.

11. ***zone*:** The regex "`^zone[1-7]$`" was used for the zone. Every value in the column starts with "zone" and ends with a number from 1 to 7. We assumed the zone represents the continent so we kept the range from 1 to 7, but it can easily be modified to fit higher values.

Note: every regex starts with `^` and ends with `$` to make sure the entire cell value matches.

Then, by filtering the dataframe, we found 3 rows that did not respect those patterns (anomalies highlighted):

Table 2: Rows with anomalies detected

member_id	date	country	gender	ip_address	amount	vip	product_id	card_type	serial	zone
100379	**2011-06-1**	MN	Female	5.142.225.113	$1053.09	false	330113	laser	177-29-7433	zone2
102169	2016-05-10	CA	Female	**164.176.777.46**	$7338.41	true	330082	jcb	417-94-9025	zone6
100216	2015-02-01	CN	Male	151.210.200.198	$6835.26	false	330090	jcb	**425-18-99000**	zone7

1. The first row in table 2 is an anomaly because it does not fit the date pattern. Even though the year and the month have the correct format, the day is invalid: it says "1" instead of "01". The day should indeed have a leading "0" as it is single digit number and the pattern expects 2 digits for the day (and a length of 10 overall instead of 9). The correct format for this value would be `2011-06-01`.

2. The second row does not fit the same pattern as the others because of its IP address. All other rows in the dataset have a valid IP address, which means all 4 values have to be between 0 and 255, as mentioned earlier. However, this particular row has an abnormally large number in the third group of 777. Obviously, it is way higher than 255, unlike the rows. A valid IP address would be `164.176.77.46`, for example.

3. Finally, the last row differs from the others because of the "serial" column. It is not obvious at first sight, but the serial number is 1 digit too long, as the last number should be of 4 digits instead of 5. Because of this, it does not respect the pattern unlike the other rows and we can flag this row as an anomaly. Removing the trailing "0" would make it a valid serial number: `425-18-9900`. In this case, using the "end" character in the regex (`$`) was crucial to find the anomaly, as it would not have been detected without it.

A Raw data

Table 3: Raw execution times

Method	Dataset	Iteration 1 (s)	Iteration 2 (s)	Iteration 3 (s)	Iteration 4 (s)	Iteration 5 (s)	Average (s)	Average time per method (s)
Linux	1	0.212	0.227	0.212	0.212	0.227	0.218	0.1081
	2	0.084	0.094	0.084	0.087	0.084	0.0866	
	3	0.046	0.041	0.046	0.045	0.041	0.0438	
	4	0.164	0.165	0.169	0.17	0.168	0.1672	
	5	0.024	0.024	0.026	0.026	0.024	0.0248	
Hadoop	1	40.691	41.228	41.816	40.827	41.737	41.2598	41.4247
	2	41.757	40.382	42.737	41.773	42.163	41.7624	
	3	39.758	40.74	40.685	42.7	41.765	41.1296	
	4	39.63	40.388	42.781	42.619	42.659	41.6154	
	5	42.871	43.798	39.783	40.661	39.669	41.3564	
Spark HDFS	1	24.678	25.385	24.675	24.465	26.138	25.0682	25.2497
	2	24.992	25.402	25.044	25.018	24.721	25.0354	
	3	25.245	26.246	24.81	24.983	25.208	25.2984	
	4	25.079	25.265	25.16	24.884	25.633	25.2042	
	5	25.585	25.76	25.729	25.425	25.712	25.6422	
Spark S3	1	25.922	26.446	26.513	26.094	25.842	26.1634	26.0942
	2	26.316	26.36	25.355	27.282	26.671	26.3968	
	3	25.867	26.329	26.346	25.779	25.502	25.9646	
	4	25.886	26.072	27.361	25.594	26.651	26.3128	
	5	25.254	25.282	25.84	25.571	26.221	25.6336	