Exploring Science Fiction & Fantasy Q/A

Big Data MA120 Documentation

1. ABSTRACT

This paper is the documentation of a project developed for the final assignment of the module Big Data MA120 at Høyskolen Kristiania in 2018. The Assignment consisted of a collection of tasks, that required the development of Hadoop MapReduce jobs. These tasks revolved around a dataset extracted from the stack exchange science fiction forum. This paper outlines the complete development process and explains the design decisions we took along the way.

2. INTRODUCTION

This project was part of the final assignment of the module Big Data MA120 at Høyskolen Kristiania. We used the Hadoop MapReduce framework[1] to analyze a large collection of data from the forum "StackExchange"[2] regarding the topic "Science Fiction". The analysis was mainly focused on users and blog posts. This documentation includes a description of all the MapReduce-Jobs we created to fulfill the tasks outlined in the assignment. We also compare alternative approaches and outline why we have chosen to build the jobs the way we did.

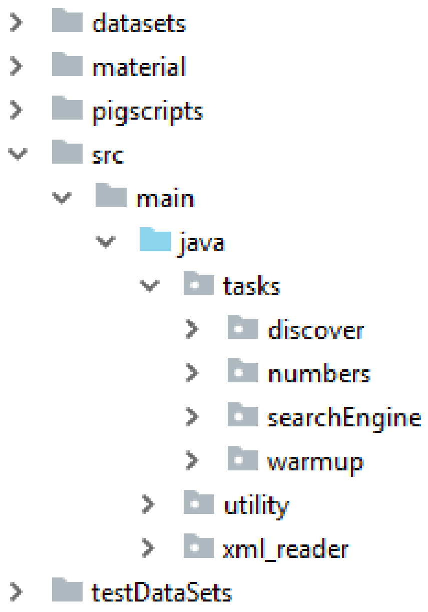
We assume the reader of this paper has an in depth understanding of Hadoop and the MapReduce pipeline and a base understanding of HDFS.

In Section 3 we will showcase how we set up our project environment, how we structured it and how individual tasks can be executed. In Section 4 we will explain how we handled the parsing of data within this project. Section 5 showcases the actual tasks and our solutions. Our struggles during this project are outlined in section 6 And in section 7 we draw a final conclusion and highlight what we have learned.

3. PROJECT STRUCTURE

In this project, we have used a wide variety of tools both for code writing and for test running. We will describe and talk about them in this section.

For the main part of the code, we have used Java language version 8.[3] It comes with all the basic libraries and data structures used for software development. We also have used the Hadoop library version 2.7.3. For the pig script, we used the Apache Pig[4] scripting language and we issued the scripts using "Grunt". We have also written a Python script, that can optionally be used to ease the execution of the MapReduce Jobs we created. The main IDE we used to develop the project has been IntelliJ[5] although we also have worked with Microsoft Visual Studio[6] for the script development and dataset visualization.



```
> datasets
> material
> pigscripts
v src
    v main
        v java
            v tasks
                > discover
                > numbers
                > searchEngine
                > warmup
            > utility
            > xml_reader
> testDataSets
```

Figure 1 The project structure

We organized the top folders of the project as follows: one folder named *pigscripts* containing the pig script of the task "5.1.E.Pig Top 10"; another folder for all the java source code of the project named *src*; a folder called *material* is used to hold the project assignment; two folders to hold both the official dataset and a test dataset we created to run faster tests while developing.

All the Java-related code resides inside the *src* folder. It contains three different subfolders each of them holding a different part of the project. The *xml_reader* folder contains all the classes related to the parsing of XML files. The *utility* folder contains all the helper classes we wrote ourselves and used throughout the tasks. Possibly the most important folder here would be the *tasks* folder, as it holds the code for all the solutions we have developed. We have structured the folder in the following pattern: each task has its own package, containing packages for each of its respective sub-tasks. For example:

tasks-> discover-> answers

[1] Apache, "Apache Hadoop."

[2] "Science Fiction & Fantasy Stack Exchange."

[3] Oracle-Gruppe, "Java SE Development Kit 8."

[4] Apache, "Apache Pig."

[5] Jet Brains, "IntelliJ IDEA."

[6] Microsoft, "Visual Studio-IDE."

To execute all the test runs while developing the project we have used a Docker[7] container based on the Docker image provided by Naimdjon Takhirov[8]. The container builds a Linux system with all the required dependencies to run HDFS. As the MapReduce jobs need an HDFS file system to operate, all the tasks have to be executed through CLI commands that queue the job for execution. These commands specify the JAR file location, as well as the input and output paths of the data. For the pig script, we have used a command line interpreter to translate the commands from the machine to the HDFS environment.

4. PARSING

In order for data to be usable for certain processes, it has to be "parsed" into a format that is readable for that specific process. This Section explains, how we parsed the StackExchange dataset to make it readable for our Program.

4.1 XML

The First Challenge of this project was, getting the dataset into our MapReduce- Jobs. The entire dataset consists of XML files. Hadoop does not natively support XML as an input format. We used the XML reader template by Naimdjon Takhirov[9] and modified it slightly to fit this specific dataset. The XML reader cuts the XML files into splits that are then passed on to the Hadoop Map function. Each Split consists of a string that contains multiple rows of the original XML document. We wrote a Java class that converts this input split into a NodeList[10]. A NodeList maps every element of an XML file to a node. This node can then be addressed like a regular Java object. Attributes can be called or changed, or it can be iterated over.

4.2 Text

Another problem arose when trying to read the values of the individual node attributes. The text stored in the original XML contained tags:

"<p>Post Text</p>"

We wrote the Java class "TextParser" to remove all tags from the input text. In addition to this, the TextParser also removes all contractions from the text. For example: " it's " ->"it is"

[7] Docker Inc., "Docker."

[8] Takhirov, "Docker Image."

This was, in the tasks related to counting words, contractions will not be counted as an individual word, but will instead count towards their word origins. Furthermore, our TextParser removes all special characters and breaks from the input string. Thusly "end." and "end" would count as the same word.

5. TASKS

In this section, we will describe the actual tasks we had to do for this assignment. We will describe how and why we solved them the way we did.

5.1 Warm Up

A. Word Count

In this task, we were asked to count how many times each word appeared on the bodies of posted questions.

The Map class went through all posts and filtered out the questions. Then it parsed the text of the body of each question using out TextParser. Lastly, it split the text up into words. Each word was used as a key and was given the value "1" and then passed on to the reducer. Every entry that now arrived in the reducer, consisted of a word as the key and a list with a length equal to the number of appearances of that word as the value. In the reducer, we just wrote the word with the number of its corresponding appearances to context.

B. Unique words

Here we had to find all the "unique" words in the question titles. Unique, in this context means all the words that appear just once in a title.

To achieve this, we used the same Map class as in the previous task "5.1.A. Word Count". The only difference was in the Reduce function. We simply checked if a given word had more than one value associated with it. If this was the case, we knew that this word had appeared multiple times. Every word that had exactly one value, was then written to context.

C. More Than 10

In this task, we were asked to count the number of questions that had more than ten words in their title.

Here, we used a different approach than on the task before "5.1.B. Unique Words". Instead of relying on the reducer, we did all the work in the map stage. After splitting the title up into its individual words, we checked if the resulting array had more than ten entries. If this was the case, we passed the Id on to the reducer. In the reduce function, we now only had to write all incoming keys to context.

Another option would have been to pass the question id as a key and the array of words as a value to the reducer. Then we would have had to filter and count the words in the reducer. This, however, would have led to a bigger data output from the mapper and could thusly have cause problems when

[9] Takhirov, "Hadoop Xml Reader Template."

[10] Oracle-Gruppe, "NodeList (Java Platform SE 7)."

dealing with large data sets. Like slower execution time or a memory overflow.

D. Stopwords

In this task, we needed to print a list of all the words that appear in the titles of questions of the dataset. We needed to exclude a predefined list of "stop words"[11]. Stop words are words that do not contain any inherent information on their own. For Example: "And", "Or", "will".

We chose to solve this task by implementing two mappers. One for the file containing all the stopwords and one for the list of blog posts. Both mappers behaved in a similar way. They split up their respective files into words and used these words as the key to pass on to the mapper. The value for the words was a fixed string in both mappers: "popularword" in the mapper of the blog posts and "stopword" in the mapper of the stopword list. Both mappers fed the same reducer. This reducer now only had to check, if the list of values for a given word contained the string "stopword". If this was the case, we did not write that word to context.

We chose to solve this task the way we did because this approach takes full advantage of the MapReduce pipeline instead of relying on large data sets being stored in Memory. In addition to this, using two mappers is easily parallelizable, which is highly advantages for distributed applications.

E. Pig top 10

For this task, a pig script was required. We had to output the top ten list of words using the output of "5.1.D. Stop Words".

This is the only pig script that we had to write for the project. It was a completely new scripting language for both of us, thusly this task turned out to be the most difficult one for us. In addition to this, the working environment "grunt"[12] was a tool we had never used before and was something we had to get used to.

Our Pig Script read the data from a text file and used the "GROUP BY" construction to group all recurring words together. Then the scripts mapped the size of each group to its respective word. Then all these maps where "ORDER BY DESC", which meant that the words with the largest appearance number appeared first in the list. Finally, we printed out the first ten entries using the "LIMIT" construction.

F. Tags

In this exercise, we were asked to create a dictionary containing all the tags that are unique throughout the dataset.

We understood "unique" in this context as, a tag that had only been used throughout all the posts once. So we decided to go through all posts and check their "tags" attribute. In the map function, we removed the angled brackets from the attribute values ("<" and ">"). Then we split the value by spaces to get a list of all the tags used for an individual post. Lastly, we used the tags themselves as the keys and the number "1" as the values. In the reduce function we now only had to check which list of values had the length of one and write those keys to context.

5.2 Discover

A. Counting

In this task we needed to count all the unique users in the dataset.

We used the "Id" attribute of the user entries as our key and a single "1" as the value in the Map task. In the reducer we checked if the list of values for each key only contained one entry, if this was the case, the user Id had to be unique. We counted every key that only had one value in a global variable. This variable was written to context in the cleanup function.

By definition, the Id technically has to be unique and thusly we could just count the keys without checking the length of the values. But to be absolutely sure, as we do not know the reliability of this data, we decided to approach this task the way we did.

B. Unique Users

Here we had to output all unique users in the dataset.

We approached this task very similarly to "5.2.A Counting". With the addition of using the display name of a given user as the value in the Map function, instead of using a single "1". We did this, so that we could display the actual name of the users in our output. In the Reduce, we again checked if a given key was unique. If it was, we wrote the user Id and its corresponding name to context.

C. Top Users

Here we needed to output the ten users with the highest reputation attribute.

Firstly, we retrieved all user Ids and their respective reputation in the map function. Now came the challenging part. How do we find the ten users with the highest reputation? We decided to use a SortComparator in between the map and reduce phases, to sort all users by their reputation. A SortComparator will sort all entries created in the map function by their key and ensure that they arrive in the reducer in a specified order. We used LongWritable.DecreasingComparator[13], which sorts all keys from highest to lowest. We used reputation as the key and the Id as the value. Now we knew that the first ten values arriving in the reduce function would be the ones with the highest reputation. We kept track of how many times we had written to context in a global variable and stopped once we had written ten times.

[11] Takhirov, "List of Stopwords."

[12] Apache, "Grunt."

[13] Apache, "DecreasingComparator (Hadoop 1.2.1 API)."

When first approaching this task, we used a global sorted list in the reducer, to keep track of all Ids and reputations. Then we planned on outputting the first ten values in that list in the cleanup function. We abandoned this approach because we were afraid that the global list could get unmanageably long and cause a memory overflow with large datasets.

D. Top Questions

In this task, we had to output the ten questions with the highest reputation attribute.

We approached this task very similarly to "5.2.C Top Users". In the Map function, we filtered the post by their "PostTypeId", so that we had a list of all the questions. Then we used the reputation as the key and the question Id and Body as the value. We decided to add the body to the value because we were curious what the actual questions would be. Then we sorted all questions using the LongWritable.DecreasingComparator and wrote the first ten values to context in the reduce function.

E. Favorite Questions

Here we had to output the ten questions with the highest FavoriteCount attribute.

We solved this in exactly the same manner as "5.2.D Top Questions". The only difference is, that we used FavoriteCount as the key in the Map function and not reputation.

F. Average Answers

For this task, we had to find the average amount of answers to questions in the dataset.

Luckily every question in the dataset had an attribute keeping track of the number of answers that were given to that question. In the map function, we sorted through all the posts, limiting them to only the questions. Then we used the string "answerCount" as a fixed key for every value set and added the actual answer count of every question to that fixed key. The reduce function was now only called once, with a list of the answer counts of every question. We added all the values together and divided them by the number of values:

(total answer count) / (number of questions).

Finally, we wrote the result to context in the cleanup function.

G. Countries

Here we had to output a list that maps the country of origin to the number of users that are from that country.

Users may enter their location in the field "location" when registering for the forum. The problem with this task was, that there is no limitation on what users can enter into that field. This means that there is a lot of useless data in the dataset, that we had to filter out. We wrote the CountryValidator class, that uses the list of countries registered in ISO[14] to check if a string is the name or abbreviation of any of those countries. In the Map function, we retrieved the locations of users, removed all special characters and broke the string up into single words. Then we checked if any of those words was a valid country by the definition of our CountryValidator class. If this was the case, we used the country as the key and the user Id as the value. In the reduce function, we just counted the number of values for each country and wrote that final result in combination with its associated key to context.

This approach had one fundamental flaw. We can only identify countries, that consist of a single word. "United States", for example, would never register as a valid country of origin. Due to the time constraints on this assignment, we decided that our approach is sufficient because this problem only affects a small number of countries.

H. Names

In this task, we needed to find the ten most popular usernames.

One problem with this was, that the only "name" the dataset of users included, was the "displayName". This name had to be unique, thusly counting the times it is used is pointless, because every displayname can only be used once. We decided to cut the displayName by spaces, to not necessarily find the most popular name, but the most popular word in a username. This way we could at least gather some information from this task.
In the Map function, we took the words within each displayName as key and the number "1" as value. Now another problem arose. We could not simply find the most popular entries by sorting by key, as we did before. We essentially needed to do two things: find out how often a given word in a displayName was used and which ten of those results were the biggest. We solved this challenge by using a treeset[15], a treelike structure that is always sorted. Every new object added to the structure will be added in a sorted manner.

Whenever a new entry came into the reduce function, we checked how many times that word was used by looking at the length of the list of values. Then we added a tuple of "word" and "appearances" to the treeset. Lastly, we made sure that the treeset was only ten entries long. If it was not, we removed the smallest entries until it was. By always keeping track of the size of the treeset, we limited the risk of a memory overflow with large datasets. In the cleanup function, we simply wrote the entire treeset to context.

One thing to note about this task is, that we build our own class to represent the tuples of "word" and "appearances" in the treeset. We did this to have full control over the way the comparator and thusly the ordering of treeset entries

[14] ISO, "ISO 3166 Country Codes."

[15] Oracle-Gruppe, "TreeSet (Java Platform SE 7)."

behaves. When two tuples of this kind are compared to each other, the only value that is taken into account is the number of appearances and not the word itself.

I. Answers

Here we had to find the number of questions that have at least one answer.

In the map function, we simply checked the "AnswerCount" attribute for each question. If it was larger or equal to one, we wrote the number "1" as the value to a fixed key called "answer". Reduce would now only be called once. In this call, we checked the length of the list of values and wrote it to context.

5.3 Numbers

A. Bigram

By definition, a bigram[16] is a pair of any consecutive words that can be found in a given text. "Big data", "Fast car" or "Having fun" are examples of this construction.

In this task, we were asked to find the most common bigram in all the question titles of the dataset. Therefore, the output of the MapReduce task will contain a single pair of words (the Bigram) and an integer number representing the number of times the construction had been found in the dataset.

The Map class was in charge of filtering all the input rows and selecting only the titles. It also parsed the text from an XML structure to an array containing only the words of the titles of questions. After this pre-processing step, it looped through the array, forming a bigram for every pair of consecutive words. The Reduce class was one of the challenges of this task. The implementation to only hold a single "top" bigram could have been trivial, but we wanted to be able to output multiple bigrams, in case two or more bigrams were tied for being the most popular one. To achieve this, we used List[17] of Pairs[18] holding the bigram itself and its appearance count. We used a set of if-statements to make sure that this list would always contain the most popular bigram(s).

B. Trigram

By definition, a trigram[19] is a triplet of any three consecutive words that appear in a text.

This task is quite similar to "5.3.A. Bigram". The only difference being, that we had to add a third variable in the map function to be able to keep track of any set of three consecutive words. The Reduce class is exactly the same as in the previous exercise.

C. Combiner

In this task, we were asked to add a Combiner[20] to the MapReduce pipeline used in the task "5.1.A Word Count". This means, that both the Map and Reduce classes are exactly the same and only the driver has been modified.

The main function of a Combiner is to summarize the map output records with the same key. This means that the Combiner is reducing the amount of work the Reducer has to do, by "compressing" its input data and joining all the values for the same key. This has multiple implications. The most obvious one would be the efficiency that it provides to the pipeline. It helps segregating data into multiple groups that can be directly processed by the reducer, decreasing the number of resources needed to operate the reduce task.

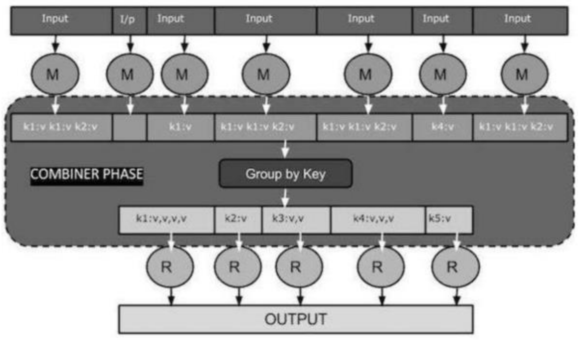


Figure 2 Graphic of Combiner

On the other hand, the use of a Combiner must match the needs of the data processing. There might be some applications in which the joining of the data before the reduce task is not desired. Therefore, the use of a Combiner in the MapReduce pipeline might not be suited for all cases.

D. Useless

In this task, we were asked to count the number of times the word "useless" appears in all the posted questions.

The Map class was in charge of parsing the XML input to an array of words which we then filtered to only contain the word "useless". We used the word "useless" as a fixed key and added the number "1" for each time it appeared in each post. The reducer was only called once and wrote the length of the list of values it received to context.

[16] Oxford Dictionaries, "Bigram: Definition."

[17] Oracle-Gruppe, "List (Java Platform SE 8)."

[18] Oracle-Gruppe, "Pair (Java SE 10 & JDK 10)."

[19] Oxford Dictionaries, "Trigram: Definition."

[20] Google LLC, "Combiner in Mapreduce."

5.4 Search Engine: Title index

In this task we had to create an index of all the words in questions, question titles and answers and link them to the question that they appeared in / are related to. A simplified example of this would be:

Input

```
<posts>
<row Id="10" Title="Cannot obtain secure database connection".../>
<row Id="25" Title="How can I secure my database connection?".../>
[...]
</posts>
```

Result

can	25
cannot	10
connection	10,25
database	10,25
[...]	

In the Map function, we firstly differentiated between questions and answers. If the Post was a question, we used all words in the body and all words in the title as keys and mapped them to the Id of the question. If the Post was an answer, we only used the words of the body and mapped those to the "ParentId", which is the Id of the question this answer was given to. Every entry that now arrived in the reduce, was a word, mapped to all the ids of questions and question titles it appeared in and all the Ids of the questions that it was answered too. The only thing left to do in the reduce was to remove duplicate entries in the list of values. We did this by adding every value in the list to a HashSet[21]. A HashSet can only contain a specific value once. When trying to add a value a second time, the HashSet just ignores the new entry. After the list was converted to a HashSet, we converted it to a string and separated each Id by a comma. Finally, we wrote the word as the key and the string containing all the Ids as the value to context.

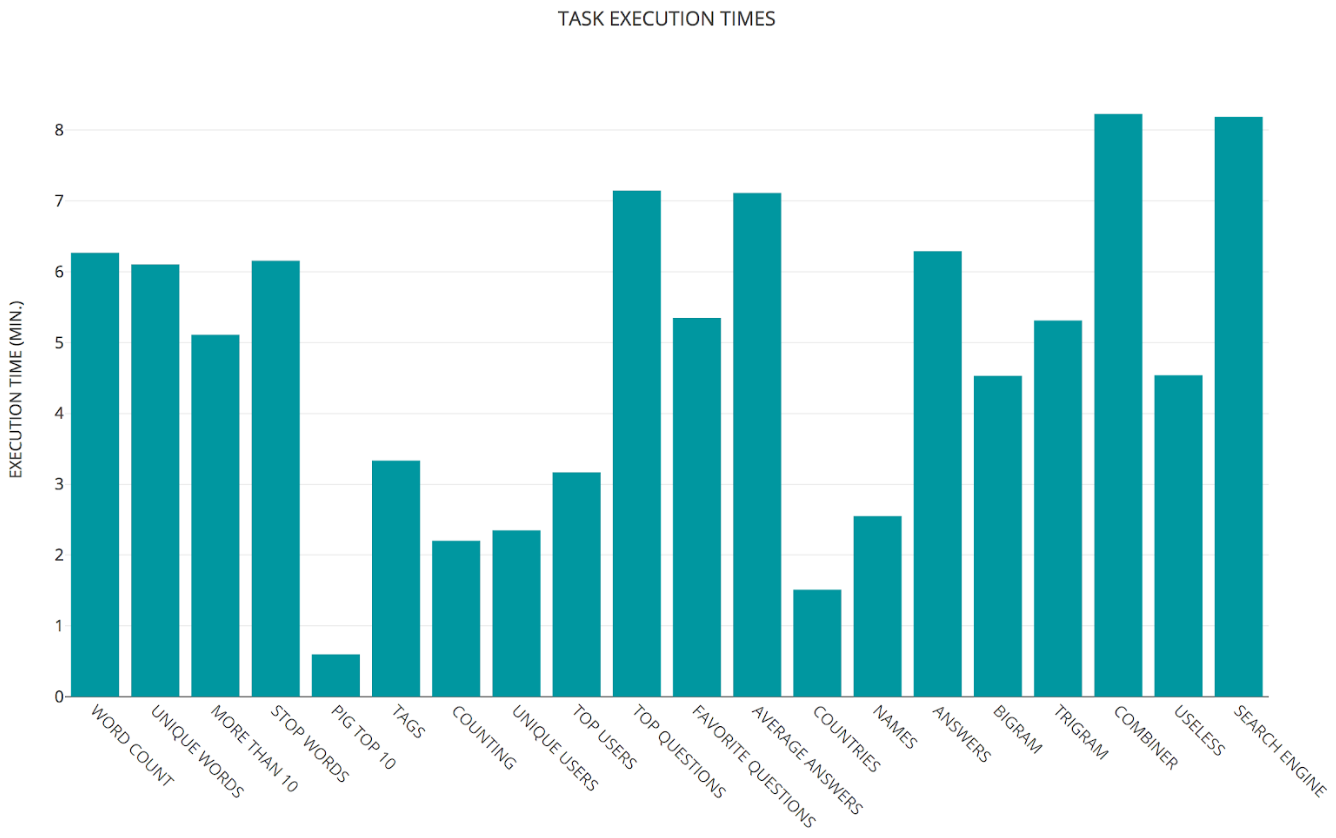


Figure 3 Execution Times

[21] Oracle-Gruppe, "HashSet (Java Platform SE 7)."

5.5 Execution times

To get a clear view of the efficiency of our jobs we decided to run all the jobs, one by one, and measure their execution times. The Result of these executions can be seen in figure 3. To run the jobs, we used the execution environment outlined in section “3. Project Structure”. We captured the execution times directly from the Docker container using the Linux command Time[22]. All executions were performed on a single node, on a 2016 MacBook Pro running a 2 GHz Intel Core i5 with 8Gb of memory.

6. CHALLENGES

In this section, we will showcase some of the challenges we have faced during the development of our solution.

The first problem we had was related to HDFS and docker. While working on this project, we always had to keep track of three different file systems and this could get very confusing. In addition to that, we found the HDFS interface very counterintuitive. Having to write “fs” before every command and not being able to change the current directory was something we had to get used to.

Another problem we had was related to the parsing of text and the conversion of contractions. Our approach to contractions works fine taking the limited amount of time for this project into account. But at the moment we do not have a reliable way to deal with apostrophes. The way apostrophes affect the words they relate to differ strongly from case to case. We hardcoded the most common contractions as special cases but did not nearly cover all of them.

One of the main problems we had was related to the development of the pig script. The environment used to execute the script was very different from the one we were using with the MapReduce tasks. In the beginning, we were not sure how to actually send both the script and the data to the HDFS file system. And once we did get it there we had problems executing it. Finally, we ended up executing it command by command instead of using the script file as we found this approach to be marginally faster.

7. CONCLUSION

The availability of Big Data, low-cost commodity hardware, and the improvements in information management and analytic-software have produced a unique moment in the history of data analysis. The convergence of these trends means that we have the capability to analyze astonishingly large datasets quickly and cost-efficiently. These capabilities represent a genuine leap forward and a clear opportunity to seize enormous gains in terms of efficiency, productivity, revenue, and profitability.

A large amount of software frameworks related to data analysis exist nowadays. We used Hadoop MapReduce as our main entry point to the Big Data world. MapReduce introduces a completely new paradigm to interact with large amounts of data, making it suitable in a wide range of situations. The scalability factor is one of the main strong suits of the framework, allowing to store and distribute large data sets across many servers. Hadoop's simplified model allows programmers to develop MapReduce programs that can handle tasks with more ease and efficiency.

During the development of this project, we used a single machine to run all the tasks. This means that we were not using the whole parallelization power that MapReduce offers. This framework full potential unfolds when all the computations are done in parallel, distributed among a network of nodes. We would have liked to run the tasks we developed in such a distributed system, to see those advantages firsthand. Despite this, the results we gathered from our local executions are still interesting enough to extract conclusions from.

The first thing we have noticed when looking at the results was the execution times. There is great variance between the execution times of all the tasks. This was expected and can be attributed to the differences in dataset sizes, ranging from a few KB to hundreds of MB. Another thing we noticed is related to the task which uses the combiner, specifically the task “5.3.C Combiner”. Normally, the combiner would help to improve the whole system efficiency, summarizing the output of the map stage into different sets of data grouped by their key. This is especially useful in a distributed system, where minimizing the data transfer between nodes executing the map stage and the nodes executing the reduce stage is very desirable. With this in mind, we were shocked to see that the actual execution time with the combiner class was approximately 33% higher than the same task without it. We think this is mainly due to the fact that we are working on a single node. We are merely adding an additional step to the execution pipeline, which takes additional time. We are not taking advantage of the Combiner, since there is no data transfer between nodes. That is one of the main reasons we would have liked to run the jobs in a clustered system. We would have been able to compare the results and see if we are right with our thoughts.

This project gave us the opportunity to get a baseline understanding of what “Big Data” actually is. We have achieved an in depth understanding of the MapReduce pipeline and of the Hadoop framework. We will take the knowledge we gathered in this assignment into our future projects and use it to properly manage and analyze large datasets.

[22] “Time - Linux Man Page.”

8. REFERENCES

Apache. "Apache Hadoop." Accessed October 19, 2018. http://hadoop.apache.org/.

Oxford Dictionaries. "Bigram: Definition." Accessed October 19, 2018. https://en.oxforddictionaries.com/definition/bigram.

Google LLC. "Combiner in Mapreduce." Accessed October 19, 2018. http://hadooptutorial.info/combiner-in-mapreduce/.

Docker Inc. "Docker." Docker. Accessed October 19, 2018. https://www.docker.com/.

Apache. "Grunt." Accessed October 19, 2018. https://gruntjs.com/.

Oracle-Gruppe. "HashSet (Java Platform SE 7)." Accessed October 19, 2018. https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html.

Jet Brains. "IntelliJ IDEA." Accessed October 19, 2018. https://www.jetbrains.com/idea/.

ISO. "ISO 3166 Country Codes." Accessed October 19, 2018. https://www.iso.org/iso-3166-country-codes.html.

Oracle-Gruppe. "Java SE Development Kit 8." Accessed October 19, 2018. https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

———. "List (Java Platform SE 8)." Accessed October 19, 2018. https://docs.oracle.com/javase/8/docs/api/java/util/List.html.

Takhirov, Naimdjon. "List of Stopwords," n.d. https://raw.githubusercontent.com/naimdjon/stopwords/master/stopwords.txt.

Apache. "DecreasingComparator (Hadoop 1.2.1 API)." Accessed October 19, 2018. https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/io/LongWritable.DecreasingComparator.html.

Oracle-Gruppe. "NodeList (Java Platform SE 7)." Accessed October 19, 2018. https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/NodeList.html.

———. "Pair (Java SE 10 & JDK 10)." Accessed October 19, 2018. https://docs.oracle.com/javase/10/docs/api/javafx/util/Pair.html.

"Science Fiction & Fantasy Stack Exchange." Science Fiction & Fantasy Stack Exchange. Accessed October 17, 2018. https://scifi.stackexchange.com/.

Takhirov, Naimdjon. "Docker Image," n.d. https://hub.docker.com/r/kristiania/hadoop/.

———. "Hadoop Xml Reader Template." Accessed October 17, 2018. https://kristiania.instructure.com/courses/529/files/75631?module_item_id=15340.

"Time - Linux Man Page." Accessed October 19, 2018. https://linux.die.net/man/1/time.

Oracle-Gruppe. "TreeSet (Java Platform SE 7)." Accessed October 19, 2018. https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html.

Oxford Dictionaries. "Trigram: Definition." Oxford Dictionaries | English. Accessed October 19, 2018. https://en.oxforddictionaries.com/definition/trigram.

Microsoft. "Visual Studio-IDE." Accessed October 19, 2018. https://visualstudio.microsoft.com/de/.

Apache. "Apache Pig." Accessed October 19, 2018. https://pig.apache.org/.