Final Year Project Report

Full Unit – Final Report

Massive scale data analytics with Hadoop

Ruben Curtis

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Hugh Shanahan

Department of Computer Science

Royal Holloway, University of London

April 11, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 17906

Student Name: Ruben Curtis

Date of Submission: 11/04/2024

Table of Contents

Abstract

Hadoop is an open-source software library that allows for the use of multiple computers to perform data analysis on a massive scale set of data. It achieves this using two algorithms. Map and reduce. These algorithms combine to create the MapReduce programming model. The Map algorithm is used to map values to a key and a value pair, with the value being the data that is analysed along with a key which can be modified depending on the requirements of the specific program which Map has been implemented in. It also is where the analysis of the data is typically done, with a condition or conditions specified to filter and sort a given dataset. The Reduce algorithm has the goal of taking a group of Map elements and reduce them into a single element, having the single element as the output. Reduce does not actually modify any data, simply just collates it and returns the data in an organized format. The MapReduce algorithm is designed to operate with datasets up to the size of petabytes [1], which means that the algorithm must be able to process this data efficiently. It achieves this partly through parallelization, where a MapReduce job will have multiple nodes assigned to the task, with each node being given an individual task to complete by the program scheduler, which is all run in parallel on each node. This leads to tasks being completed extremely quickly for the resources given, especially compared to the alternative of tasks being completed in succession.

Hadoop implements the MapReduce algorithm in its framework, along with providing its own file system that can be shared across each node in the Hadoop cluster. The Hadoop Distributed File System (HDFS) allows for an internal file system between nodes in the cluster, making sharing files, such as an output of a MapReduce program, possible. It also is the system used when sending blocks of data to nodes in the cluster when executing a MapReduce job, taking advantage of data locality to improve data analysis efficiency [2]. A significant design choice in Hadoop is assuming that hardware failures are a normal event rather than an exception [3][4], meaning that errors in a single, or multiple nodes can be handled efficiently by the Hadoop system. This a core design choice of the Hadoop HDFS and allows for minimal disruption in the case of errors in the HDFS.

This project aims to provide implementations of several programs using Hadoop MapReduce. This includes filtering datasets by a wordcount, along with filtering using a user input regular expression. Other programs will include implementation of statistical analysis, where a frequency distribution graph will display the data at fixed percentiles. Followed by the implementation of a K-Means algorithm. Finding conclusions using graphed centroids in a 10-dimensional dataset.

Chapter 1: **Introduction**

1.1 MapReduce

MapReduce is an algorithm designed for parallel computation of massive data sets, with its goal to provide massive scale data analysis on various hardware setups, while handling errors in software and hardware in an appropriate fashion [5][6]. The algorithm is split into two distinct parts, each are their own algorithm, but are combined to create MapReduce. The first algorithm is Map. Map has the task of selecting data by given parameters, followed by formatting the data into the form of a key-value pair, with the goal of mapping an individual value to a key. Only filtered values are mapped to a key, as other values that do not satisfy the conditions set by the MapReduce program are discarded. Reduce is the second stage of the MapReduce program, it is dependent on the Map program as its inputs are the key value pair outputs from the Map program. Reduce takes all key value pairs as an input, the program then reduces the given key values into a single output, such as in the case of the first problem, where the Reduce program will sum the occurrences of every individual word.

1.2 Apache Hadoop

Apache Hadoop is an open-source software library that provides an implementation of the MapReduce program [4], it provides the ability to host a Hadoop Distributed File System (HDFS) which is an open-source implementation of the Google File System (GFS) [7] with support for up to thousands of nodes that can be used for massive scale data analytics. Hadoop is used in this project to provide the HDFS and the ability to use MapReduce on multiple nodes. It also provides the ability to create bespoke MapReduce programs using Java through its libraries, which is the primary programming language used in this project.

With Apache Hadoop using up to thousands of nodes, it is likely that failures in a node will occur. The architecture assumes that failures will happen. Accounting for these inevitable failures, Hadoop has a redundant design, providing a level of tolerance given faults in nodes. Using this design, Hadoop will contain a backup of data which it calls a replication on nodes other than the node containing the data, the number of replications can be set by an operator of the Hadoop cluster. This implemented fault tolerance allows for the multi node architecture to function, as failures on a node will not result in data loss or inability to process the data.

Chapter 2: **Project Specification**

This project has four distinct goals, all of which will require the use of Apache Hadoop to achieve. Furthermore, part of this project will require the use of python as a tool to visualise results produced from Hadoop MapReduce operations. These MapReduce algorithms will use a Hadoop cluster configured specifically to handle these algorithms, returning data in the Hadoop Distributed File System (HDFS). This makes Apache Hadoop, a Hadoop cluster, java, and python requirements for this project.

The first problem is taking a dataset of many web pages, followed by finding the number of occurrences of each word in the dataset. This is considered as the “Hello World” of the MapReduce algorithm. This problem would take advantage of the Hadoop distributed file system (HDFS) to provide an input and output location, along with having the HDFS handle files being read and written from multiple different nodes simultaneously. The file output should be in the format of the string, followed by a sum of the total occurrences of the word in the entire dataset.

The second problem becomes more advanced, the problem uses the same dataset as the first problem, where it is a large collection of web pages (i.e. .html files). Using a regular expression given to the program, the program will find all strings that match the given regular expression. A requirement that was added was that the regular expression must be dynamically input by the user, not a hardcoded value. The output of this search must be in .txt format. This must contain the matching string, the file it was found in from the dataset, along with the line number in that file it was found in. This must be made available in an output folder of the HDFS, along with the input being taken from the HDFS.

The third problem is to generate and visualise a frequency distribution graph where given a dataset of numbers, process the data and visualise it where a conclusion or conclusions can be made about the dataset. This also includes finding multiple different order statistics such as a minimum, maximum or any average based on the data itself.

The fourth problem is to implement the K-Means algorithm in the context of a Hadoop MapReduce program. This will include the assignment step of the K-Means being implemented as a mapper, followed by the update step being implemented as a reducer. This algorithm must be used to find an insightful conclusion in a realistic dataset. Followed by an appropriate visualisation of the results of the K-Means algorithm to help find conclusions about the data.

All problems will require a shell script file (.sh) which can be run with the arguments to launch the program, using Hadoop command line arguments to run the programs as opposed to manually using the Hadoop jar command. The script must use the Hadoop installation on the machine that is running it through finding the environment variable HADOOP_HOME.

Chapter 3: **Theory**

3.1 MapReduce

MapReduce as a programming model is an implementation of two distinct algorithms to process and generate results for use in massive scale datasets [8]. Its goal is to provide parallel distributed computation amongst a cluster of machines. A cluster is a variable group of computers that share a common computational goal. A cluster can be viewed as a single large system because of the common goal of the nodes contained in it. The two algorithms are named Map and Reduce respectively.

This programming model allows for the implementation of a variable level of redundancy and fault tolerance. Redundancy is the concept of replicating data across multiple nodes in a cluster, the number of replications can be customized in the configuration of Apache Hadoop clusters. Redundancy allows for a level of fault tolerance in the system entirely. Fault tolerance is a measure of how many errors can occur in both computer software and hardware during operation without either the system operating at a reduced level or not operating at all.

This programming model allows for parallelisation. This is the idea that the same task can be completed across multiple computers concurrently. It achieves this by splitting its single task into multiple of the same, where each individual task can be run separately from one another. Followed by distributing these between different computers in the cluster. This optimises the task as it ensures the available resources are used as efficiently as possible, all at the same time. Removing the risk of a computer in the cluster not being utilized as it is waiting for another task to finish. Parallelisation is an important concept in this project, it is one of the core concepts that allow for programming models such as MapReduce to function.

3.2 K-Means Clustering

K-Means is a NP-HARD problem with the goal being to split a dataset into several clusters, commonly referred to as k, where each data entry is assigned to the cluster with the nearest mean to the data. A cluster centroid is the mean of all the data points assigned to that cluster. The centroid is intended to provide a summarised average of each data point assigned to the given cluster. An additional advantage of K-Means is that it can be used in any number of dimensions of data. For this project, up to 10 dimensions are analysed.

The objective of a K-Means algorithm is to take **n** data points, where each data point is a vector of dimension **i**. Followed by assigning data points to **k** clusters, where a cluster is an aggregate of set **S** = $\{S_1, S_2, \ldots S_k\}$ with the objective to minimise the variances of the values in the set. This is considered a single iteration of K-Means.

K-Means has multiple implementations. The differences in the algorithms include factors such as efficiency and time to achieve convergence. The implementation used in this project is known as Lloyd’s algorithm. This implementation contains two steps. The assignment step and the update step.

The assignment step involves taking a data point **n**, followed by assigning it to the cluster **k** with the nearest mean. This means each data point will be assigned to exactly one cluster. The update step involves recalculating the means of the set of data points assigned to each cluster. This will update the value of the centroid, which will change the behaviour of the assignment step in the next iteration of the algorithm. When the centroid in the update step keeps the same value. It is considered that the algorithm has converged, however does not imply that the optimum centroids have been found. Convergence is more likely the higher the number of iterations the algorithm completes.

There are multiple initialisation methods used in K-Means. For this project, the Forgy method [9] is used to choose the initial centroids. This method will choose **k** number of randomly selected data points, using the selected data points as the centroids in the first iteration. This means that running the algorithm on the same dataset with the same iteration can lead to different results, however differences in these results will be small if the optimum has not been found. Which will typically not affect derived conclusions about the data itself.

Chapter 4: **Technical Analysis**

4.1 WordCount

The first problem was to find the number of occurrences of each word in a dataset. This is considered as the "Hello World" program of MapReduce. This program makes use of a driver class, a mapper class and reducer class. This would all be linked through a package named:

```
main.java.org.FinalYearProject.WordCount
```

Each class is split based on its function in MapReduce as an algorithm, while it is possible to have the same function in a single class compared to splitting driver, map and reduce, it is bad practice to have multiple java classes in the same java file.

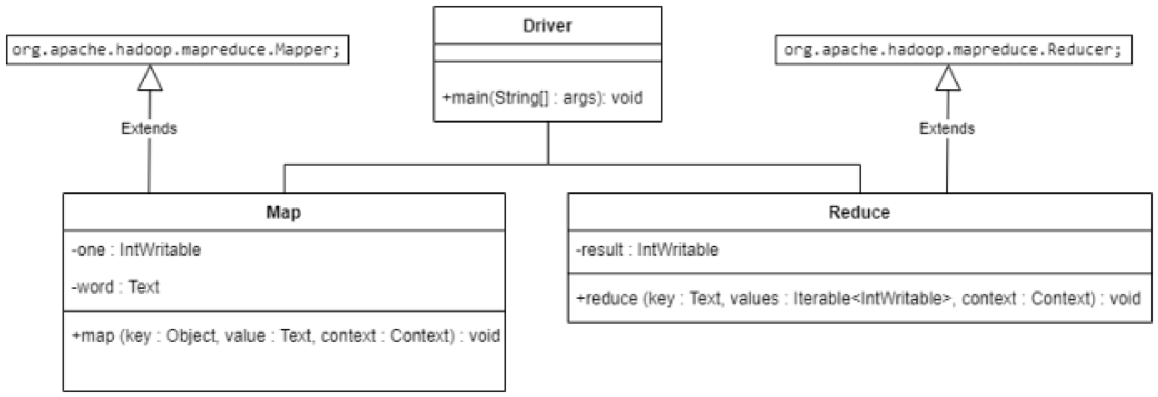

org.apache.hadoop.mapreduce.Mapper;
Driver
+main(String[] : args): void
org.apache.hadoop.mapreduce.Reducer;
Extends
Extends
Map
-one : IntWritable
-word : Text
+map (key : Object, value : Text, context : Context) : void
Reduce
-result : IntWritable
+reduce (key : Text, values : Iterable<IntWritable>, context : Context) : void

Figure 1 **- The UML diagram for WordCount**

4.1.1 Driver Class

This class handles setting the configuration of the program, along with setting the classes in the package to be used by Hadoop. The driver handles launching and exiting the MapReduce job. It handles exiting the job by exiting the program when it has been completed with the following line:

```
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

This will return an exit code of 0 for an error free execution and 1 in cases of errors. Taking advantage of the Hadoop function

job.waitForCompletion(boolean)

sourced from the library:

org.apache.hadoop.mapreduce.job

to determine if Hadoop has successfully completed the job.

This class is also the class selected when running the hadoop jar command, so to select the driver class, it would be:

```
main.java.org.FinalYearProject.WordCount.Driver
```

A .sh script has been included with the program, to be used on Linux Z shells [10] to run the program when given an input and output path. This script is only compatible with Linux Z shells, but the jar is compatible with any Hadoop version 3 installation on any Hadoop supported operating system.

4.1.2 Mapper Class

This class handles the Map operation of the MapReduce job. This Map program does not need to filter the values as it has the goal of counting each string, so this is not required. However, the counting of each string is required. This is done using the StringTokenizer class, from the java.util package. This choice is intended to keep the dependencies of the program at a minimum, with java already being a requirement, this does not change the dependencies of the program while allowing for the feature of assigning each string a unique token.

The Map program begins by creating a StringTokenizer, followed by checking if there is a token available to be analysed. If there is, the following while loop is run:

```
while (itr.hasMoreTokens()) {
     word.set(itr.nextToken());
     context.write(word, one);
}
```

This loop will assign each word a unique token, followed by creating a key value pair, with the key being the word, along with the value being the number 1, which will be used later to sum the count of the words in the reducer.

4.1.3 Reducer Class

The reducer class has the goal of taking the key value pairs taken from the Map class and summing the values given. Therefore, reducing each individual string count into a single sum of that individual string. In the context of this program, the output would be the string, followed by a space, followed by the number of occurrences of the string. The program will also organize the strings alphabetically, this includes special characters in the string, such as “(“. The resulting sum is written to the context of the MapReduce job, meaning that the job can now understand the sum operation, the line:

```
context.write(key, result);
```

communicates with the Driver class to provide the information that was processed in the reduce class. Giving a complete output for the Driver class to use when forming the output. This program has been configured for 4 datanodes, so the number of reduces will be 4. This means that the output will return 4 different text files, all containing the 4 reduces run in parallel on each node.

4.2 Distributed Grep

The second problem to solve was somewhat like the first problem, where the same data is required as an input. However, the program itself was different. Another input was also required, a user input regular expression that would be used to provide the filtering input for this program to filter a dataset into valid regular expressions. This program must find matching strings corresponding to the regular expression, followed by formatting the output file(s). The format must include the line number and the name of the file the matched string was found in. An example of a valid output for the regular expression “[0-9]?[0-9]:[0-9][0-9]” would be “13:27 Linenumber: 5 FileName: webpage.html.

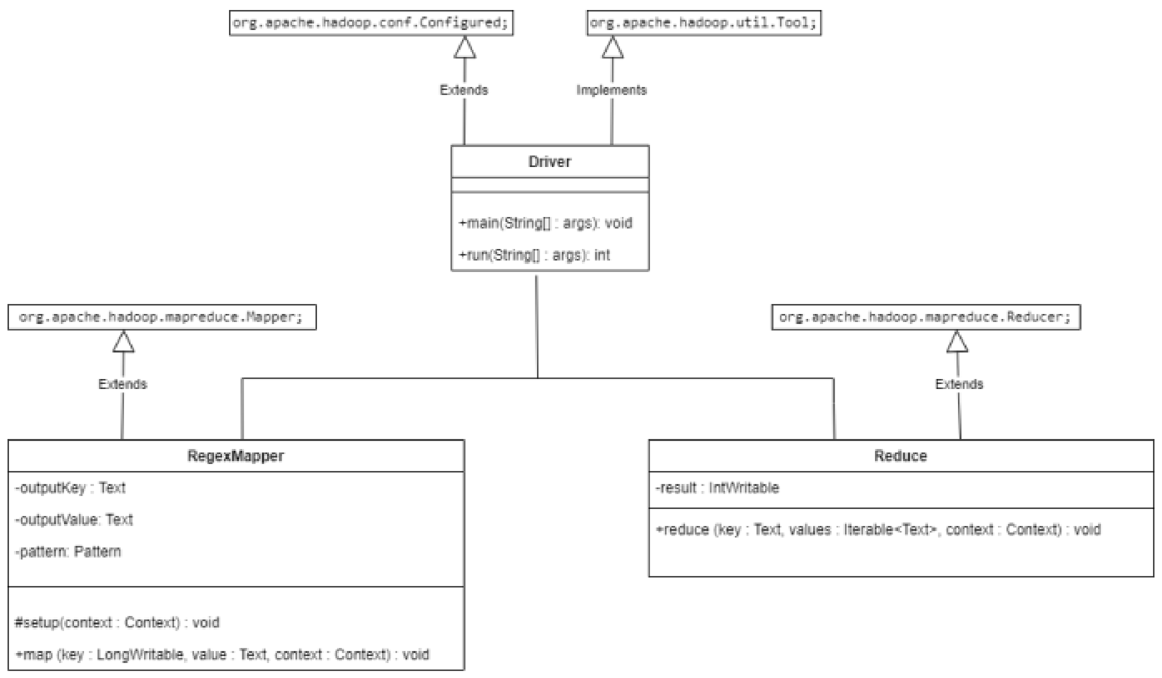

org.apache.hadoop.conf.Configured;
org.apache.hadoop.util.Tool;
Extends
Implements
Driver
+main(String[] : args): void
+run(String[] : args): int
org.apache.hadoop.mapreduce.Mapper;
org.apache.hadoop.mapreduce.Reducer;
Extends
Extends
RegexMapper
-outputKey : Text
-outputValue: Text
-pattern: Pattern
#setup(context : Context) : void
+map (key : LongWritable, value : Text, context : Context) : void
Reduce
-result : IntWritable
+reduce (key : Text, values : Iterable<Text>, context : Context) : void

***Figure 2* - The UML diagram for DistributedGrep**

4.2.1 Driver Class

Driver contains two methods, a typical main method, followed by a method named run, which is an override from the implemented Tool interface. The main method is very small, by the fact that the method simply calls the run method, followed by exiting the program with an exit code returned by the run method. This is intentional design, as it allows for the future modification of the main class, to add extra features to the program easier as the main class has organised code, along with the relevant code being segmented into its own methods (i.e., run).

The run method is where most of the program is run. This method handles the creation, configuration and outputs of the MapReduce algorithm implemented for this program. This method begins with finding the regular expression that a user would like to search for using the InputStreamReader library from java io. This library does not add any required dependencies to this program, keeping the program lightweight while completing the tasks that a user would want the program to do, making using this library a good choice. After taking a user input, only then is the MapReduce job configured and then launched, with the job being configured as a "RegexSearch". The classes are then set, followed by the output key and value classes being set, which in the context of this program is the Text class. The driver class then sets the input and output paths of the job, which are defined in the command line arguments when running the program. The job then begins, in which the driver class waits for it to complete, shown by the line:

```
return regex.waitForCompletion(true) == true ? 0 : 1;
```

in which the driver class will return the exit code given to it by the Hadoop job, which is passed into main, which is then used as the programs exit code. An exit code of 0 indicates successful execution, whereas any other code (such as 1) indicates an error.

4.2.2 Mapper Class

This class handles the selection of items using the user input regular expression as a search term. The way this class will check for a matching regular expression is using the java util regex library, using the Matcher method to check for matches between the line value and the input regular expression. The while loop of the map method is only triggered if the matcher finds a valid value, skipping invalid values. When triggered, it will take the line number and the name of the file the matched string was contained in, followed by creating a value containing the matched string, followed by the line number and file name where the string is located. It then maps this value to a key value pair, as is the case with a typical map algorithm.

A typical output value would be built using this line:

```
outputValue.set(" LineNumber: " + lineNumber + " FileName: "
      + ((FileSplit) context.getInputSplit()).getPath());
```

Which fulfils the requirements regarding the format of the output.

An important method is the setup method, where it overrides the inherited Mapper interface. Setup allows for queries to be made to the configuration of the MapReduce job, which allows for the ability to find the search query, which was input in the driver class, followed by passing the query into a Pattern type class variable, which can be used by the Matcher in the map method.

4.2.3 Reducer Class

This reducer class has small goals. This class simply takes all the filtered values stored on the Hadoop Distributed File System, followed by combining them into a single output. The number of reduce jobs in this program is by default set to 4, as 4 is considered as the assumed number of DataNodes being run in the Hadoop cluster, the appropriate number of reduces would be equal to DataNodes * 0.95, due to the cluster being 4 DataNodes, this calculation was changed to be equal to the number of DataNodes. This method will reduce every output, this is shown by the line:

```
for (Text value : values) {
```

Which will trigger the loop to cover every value that it has been given from the Mapper, again showing that the Mapper class is responsible for filtering the dataset to produce the desired outputs.

4.3 Frequency Distribution

4.3.1 Introduction

This program was designed to provide a frequency distribution with a given dataset, with the given dataset being a large collection of numbers. This includes creating a visual output of the processed dataset, providing an insight into the dataset based on the visualised results. This program contains a MapReduce element to handle calculating the data to be used on the graph. Followed by a Python 3 element to visualise the results.

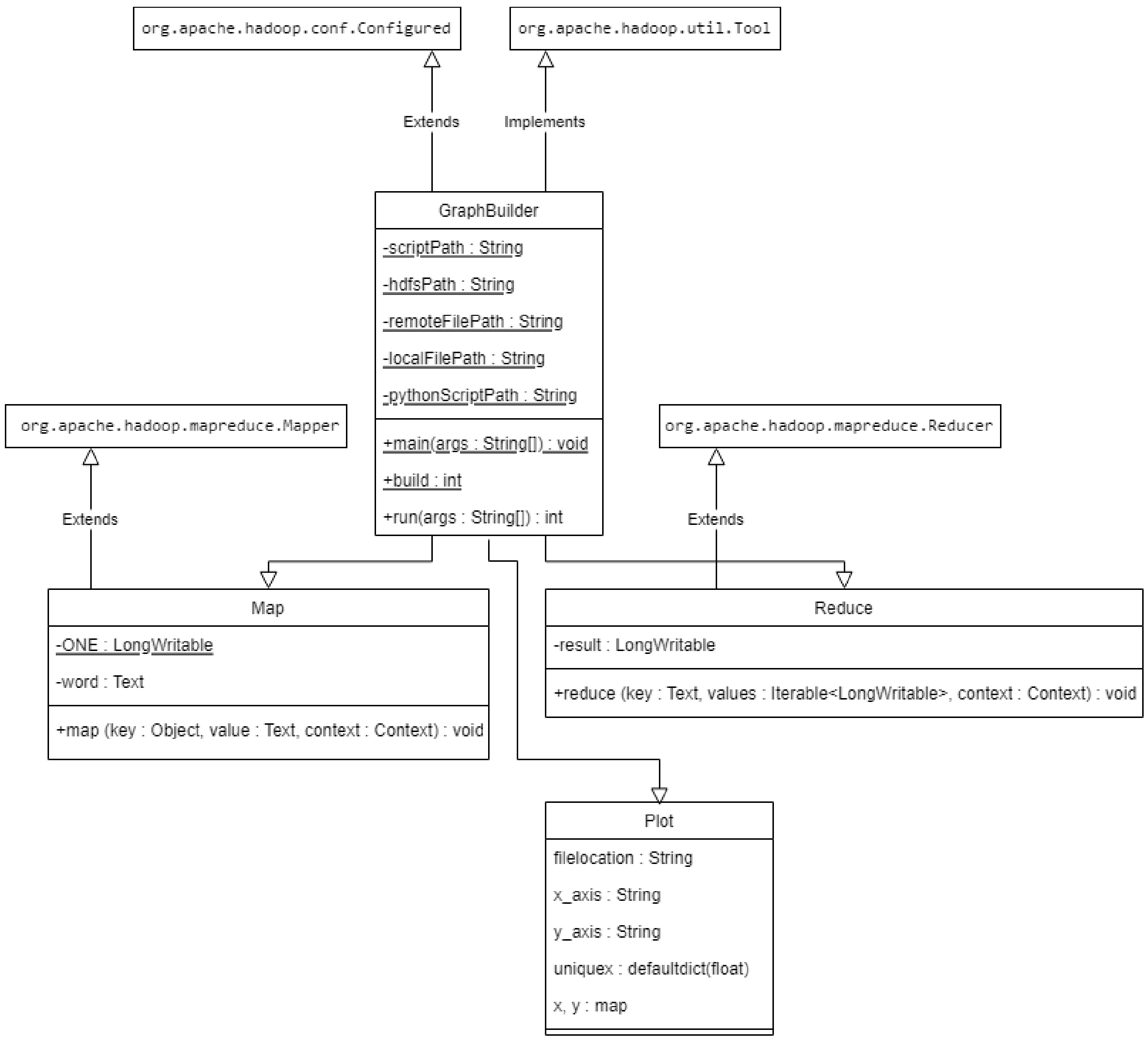

org.apache.hadoop.conf.Configured
org.apache.hadoop.util.Tool
Extends
Implements
GraphBuilder
-scriptPath : String
-hdfsPath : String
-remoteFilePath : String
-localFilePath : String
-pythonScriptPath : String
+main(args : String[]) : void
+build : int
+run(args : String[]) : int
org.apache.hadoop.mapreduce.Mapper
org.apache.hadoop.mapreduce.Reducer
Extends
Extends
Map
-ONE : LongWritable
-word : Text
+map (key : Object, value : Text, context : Context) : void
Reduce
-result : LongWritable
+reduce (key : Text, values : Iterable<LongWritable>, context : Context) : void
Plot
filelocation : String
x_axis : String
y_axis : String
uniquex : defaultdict(float)
x, y : map

Figure 3 - The UML diagram for the Frequency Distribution

The MapReduce element will contain a Driver, Mapper and Reducer class. The goal of the Mapper is to map each value in a dataset to the value one, followed by the value being the key in the key-value pair. The Reducer's goal is to take the mapped values and sum the keys based on the values given. Followed by returning the data in appropriate format.

The Python 3 element requires taking the filtered data, followed by creating an appropriate graph. This graph must show a sum of the occurrences of values in a dataset, ordered by value. With the goal being to find an insightful conclusion from the graph.

4.3.2 GraphBuilder Class

The Driver Class of this program is named GraphBuilder. This class handles the responsibilities of a MapReduce Driver, while also providing the functionality to initiate building a graph of the results. It contains three methods; main, build and run. The main method has the goal of calling the ToolRunner which this class implements to call the run method. Then the build method is called to create the graph.

The main method is the first to be run in the program. The purpose of this method is to ensure the program is run in order. The order being to complete the MapReduce job before graphing the results. This class will throw an exception if the MapReduce job fails, leading to the build method not being called as the program stops. The exit codes that are possible for this program is 0 and 1.

An exit code of 0 is successful. An exit code of 1 is considered an error and will cause the exception to occur.

The run method is the method that handles the MapReduce job. It follows the linear setup for a typical MapReduce job. This includes creating a new Configuration object, followed by configuring its attributes. The key and value pairs are output with:

```
    job.setMapOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);
```

Where the key is assigned as a Text data type. Followed by the value being assigned as a LongWritable data type. Other configuration attributes that are modified in this program include setting the JAR to run this MapReduce job as this class, implemented dynamically using:

```
this.getClass()
```

to ensure that even in cases of the class name being changed, it is ensured the class will reference itself in this configuration.

Continuing with the attributes modified, the Mapper and Reducer class is specified, followed by the number of Reducer tasks being configured. While not part of the job configuration, the file input and output locations are set as the first and second argument respectively when passing arguments when running the program.

After creating the configuration and the job, followed by configuring the MapReduce job, the job object then waits for the Mapper class and the Reducer class to finish execution. Depending on if these classes execute gracefully, the returned number for this method will be 0 if successful, 1 if an error has occurred.

The build method has the purpose of running a Python subprocess to display the graph. It implements this by using class variables defined outside of the method. These variables are intentionally implemented as class variables, so they are at the beginning of the code for this class. This means configuring these variables will be easier as they will be in a more predictable location. These variables will most likely require configuration, as they define exact file locations for use in this class, this includes the path for the Python file, the address of the HDFS, the location of the local data file and the data file stored on the HDFS.

This method begins with defining the HDFS address in a Configuration object. This is achieved through setting the HDFS address under the fs.defaultFS attribute of the Configuration. This is required later in the try-catch statement, where the file in the HDFS is copied to the local system. The action of copying the file is completed with the line:

```
FileUtil.copy(fs, sourcePath, new File(destination.toString()), false,
cfg);
```

Where FileUtil, a library from Hadoop, gives the ability to perform operations to files in the HDFS. Its arguments include a FileSystem object, the location of the file to be copied on the HDFS, the local destination for the file to be copied to. A Boolean to delete the source file, which I have set to false. Followed by the Configuration object being the final argument.

Once the file has been copied onto the machine locally. The method will run a try-catch statement to run the Python file using the java ProcessBuilder library. This statement handles the implementation of a BufferedReader object to read the output of the Python program. This is intended to read errors and text outputs that are returned from the Python program, followed by printing them in the java program. Once this method has been completed, it returns an exit code of 0 or 1, with 0 being a successful execution, 1 being unsuccessful.

4.3.3 Mapper Class

The Mapper is an intentionally short program. Its goal is to take the values passed to it and sum the occurrences of those values in the dataset. This includes adhering to the output key-value classes defined in the GraphBuilder class. It achieves this using a StringTokenizer object to give each data value a unique token. This allows for each unique value to be identified. Using a StringTokenizer allows for iteration through the entire dataset, this is achieved through the while loop:

```
while (itr.hasMoreTokens()) {
```

This allows for manipulation of a single data entry at a time. Selecting the value to be summed is done in the line:

```
word.set(itr.nextToken());
```

Where the value of the word is set as the key in the key value pair for the Mapper. As it is known that the occurrence of a value only gives it a value of one, it can be assumed the value in the key value pair can be set to one with no unexpected results. This is assigned to the context through the line:

```
context.write(word, one);
```

So that this output can be used in the Reducer.

4.3.4 Reducer Class

The Reducer class operates as both a combiner and reducer in the context of a MapReduce algorithm. Taking a key of type Text and a value of type LongWritable. The same types defined in the GraphBuilder class, the class acting as the Driver for the MapReduce program.

This class has the goal of summing the occurrences of keys passed to it from the Mapper. This class contains a single method reduce, extending the Reducer class from the Hadoop MapReduce library. The reduce method sums the values given to the keys using a for loop. The for loop:

```
for (LongWritable value : values) {
        sum += value.get();
      }
```

Will take all the values assigned to the specific key, followed by summing them. After this loop, the result is set as the variable sum. Followed by assigning the key and the result of the sum to the context of the MapReduce job. Achieving the goal of summing the values of all keys, followed by combining the same keys together, summing their respective values together.

4.3.5 Plotting the Graph

Plotting the graph requires the use of matplotlib, a library used with Python. Using the SubProcess from the GraphBuilder class. The Python program Plot.py will be executed under this SubProcess. The program has the functionality to be run independently, given an already processed dataset and will provide the same result.

The program has the goals to open a file to read. Followed by passing unique keys from the file into a dictionary. The key-value nature of MapReduce allows for the data to be parsed through and passed to a map in python using a for loop to iterate through each data value in the text file.

```
for line in lines:
    x, y = map(float, line.strip().split())
    uniquex[x] += y
```

This loop will find each unique key using a map. Followed by appending each unique key to a dictionary of type float.

The dictionary is then applied to the x and y axis of the graph:

```
x_axis = list(set(uniquex.keys()))
y_axis = [uniquex[x] for x in x_axis]
```

Where the X axis denotes the values themselves. The Y axis denoting the occurrences of a value. This is then passed into a matplotlib bar chart using:

```
plt.bar(x_axis, y_axis, label="Count")
```

Which assigns the values to its respective axis.

After assigning the correct data, the program then labels the graph as appropriate. It will then show the graph along with saving the graph as an image in the line:

```
plt.savefig("plot.png")
```

With the string argument given being the name in which the file will be saved as. This program will end once the graph shown has been manually closed by the user.

4.4 K Means

4.4.1 Introduction

This program contains an implementation of the K-Means algorithm using Hadoop MapReduce. This program will take a dataset of up to 10 dimensional numerical vectors, followed by producing a graphical output of the resulting centroids. The goal of this program is to create a working implementation of K-Means using Hadoop MapReduce. Another goal is to create a graphical representation of the results, where conclusions on the data can be made from the output. The graphical representation will be in two dimensions for simplicity, as the complexity in this program originates from the K-Means MapReduce program itself.

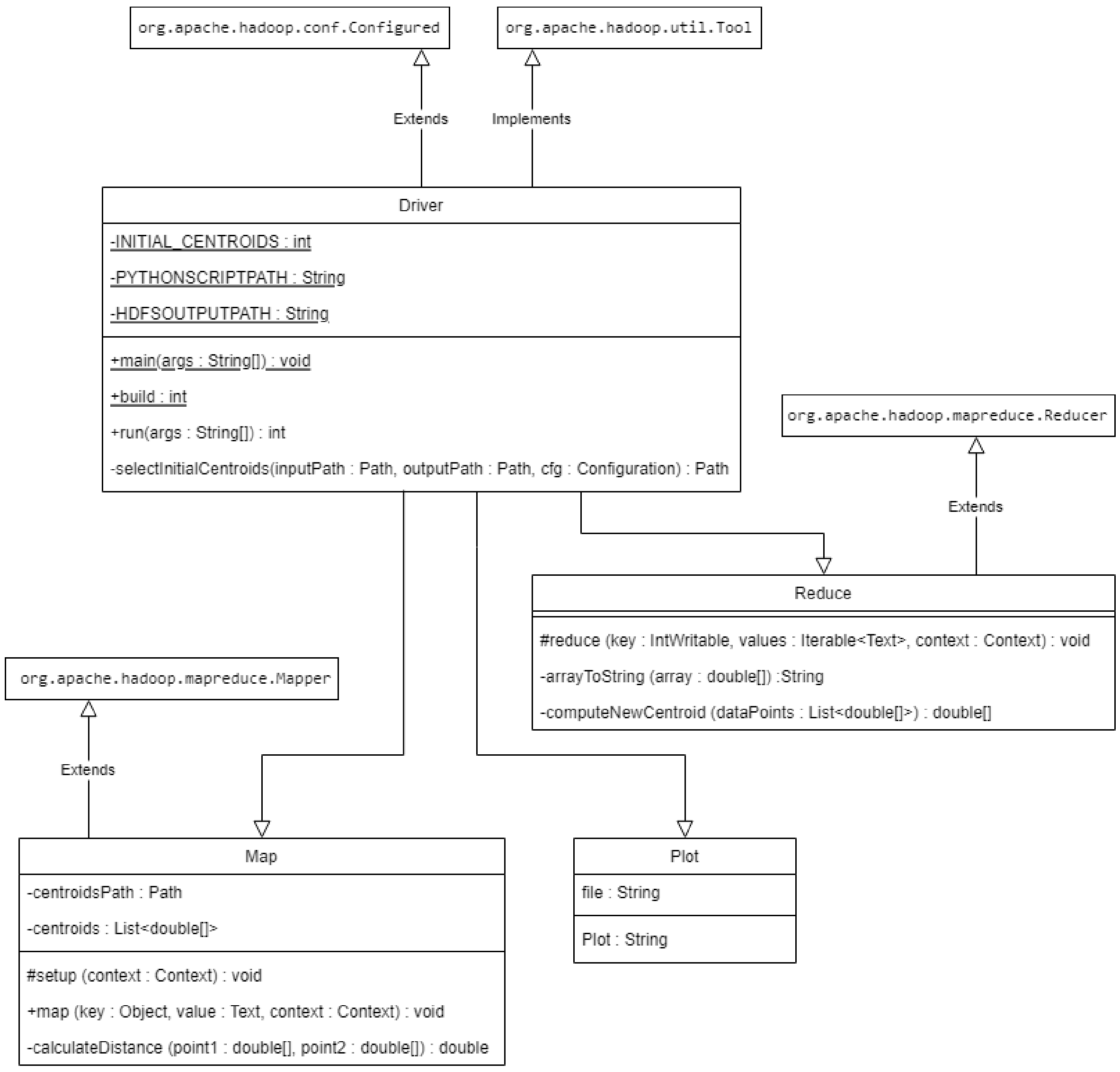

org.apache.hadoop.conf.Configured
org.apache.hadoop.util.Tool
Extends
Implements
Driver
-INITIAL_CENTROIDS : int
-PYTHONSCRIPTPATH : String
-HDFSOUTPUTPATH : String
+main(args : String[]) : void
+build : int
+run(args : String[]) : int
-selectInitialCentroids(inputPath : Path, outputPath : Path, cfg : Configuration) : Path
org.apache.hadoop.mapreduce.Reducer
Extends
Reduce
#reduce (key : IntWritable, values : Iterable<Text>, context : Context) : void
-arrayToString (array : double[]) :String
-computeNewCentroid (dataPoints : List<double[]>) : double[]
org.apache.hadoop.mapreduce.Mapper
Extends
Map
-centroidsPath : Path
-centroids : List<double[]>
#setup (context : Context) : void
+map (key : Object, value : Text, context : Context) : void
-calculateDistance (point1 : double[], point2 : double[]) : double
Plot
file : String
Plot : String

Figure 4 - The UML diagram for K-Means

The structure for this program shares similarities with the previously mentioned Frequency Distribution program. Where there is a MapReduce element and a Python 3 element. The java-based MapReduce element will contain a Driver, Mapper and Reducer class. The Python element will have a single program, using matplotlib to graph data that has already been processed from the MapReduce program that will execute beforehand. The graphing program will have the functionality to run independently from the MapReduce program however, it will be called using a java subprocess in the MapReduce program.

4.4.2 Driver Class

The Driver class in this program acts as the Driver class in a Hadoop MapReduce algorithm. This class is an extension of Configured from its respective Hadoop package, along with the Tool from Hadoop being implemented into this class. This class contains four methods. These methods are named main, run, selectInitalCentroids and build. In order of which method is executed first.

The program runs the main method first. This method handles running the MapReduce program by calling the ToolRunner, this ToolRunner will then use the run method to run the MapReduce program. Once the MapReduce program has completed and returned an exit code. If the exit code is 0 (i.e. successful execution) then main will call the build method. The main method will then exit the program with the exit code given from the MapReduce program.

The method run is the method where Hadoop MapReduce specific operations are performed. This method begins with ensuring the correct number of arguments have been provided when executing the program. These arguments are the input path, the output path, and the number of iterations the program will go through. If the number of arguments is not exactly three, the program will not begin running and an error will be thrown through returning the number 1 back to the method main.

Once the number of arguments is validated. The run method continues to assign the arguments to variables. Followed by calling the selectInitalCentroids method. Passing the input and output directories as arguments to the method. The third argument being the configuration used in Hadoop.

The process of selecting the initial centroids is the Forgy method [9] of initialisation. The source of random selection used is sourced from the package:

```
java.util.random
```

To provide simplicity and keep dependencies of the program at a minimum.

The try statement:

```
    try (FileSystem fs = FileSystem.get(cfg);
        FSDataOutputStream outputStream = fs.create(centroidPath);
        FSDataInputStream inputStream = fs.open(inputPath);
        BufferedReader reader = new BufferedReader(new 
InputStreamReader(inputStream))) {
```

Will attempt to create the output directory for the initial centroids. Followed by attempting to open and read the input given. This statement checks the validity of the arguments given. With the same error thrown if any is encountered.

Upon both validating the arguments and setting up selecting the centroids. The for loop which will run based on the number of centroids that will be created. Defined by the constant class variable INITIAL_CENTROIDS. In every pass of this for loop, a random number is generated between 1 and the length of the file. This random number is used to seek to the line of data, followed by writing this line into the initial_centroids.txt file if it is not empty. As the output of this method is set on the HDFS, only the file path of the initial centroids will be returned.

After the initial centroids have been selected. In the run method, a for loop which will run for however many iterations have been defined as an argument by the user begins. This for loop contains the configuration for the MapReduce job. This includes setting the input and output paths, allowing for output paths to be created dynamically based on the iteration the program is running on the line:

```
FileOutputFormat.setOutputPath(job, new Path(outputPath, "iteration_" + 
(i + 1)));
```

This makes each iteration documented, allowing for the ability to identify if the K-Means program that was run reached convergence based on differences between an iteration and its previous.

The output key for this program was decided to be type IntWritable. The output value was decided to be of type Text. Both types are derived from their Hadoop libraries respectively.

The program then passes the centroid path file location to the HDFS distributed cache. Allowing for efficient reading of the centroids amongst nodes. To end the for loop, the program waits for the MapReduce job to complete, followed by defining the new path for centroids for the next iteration.

After all iterations have completed without error. The main method will call the build method. This method handles the ProcessBuilder object to call the Python 3 program to visualise the results. The method begins with creating the process, using the constants defined for the python script file location, along with the HDFS output file location. The Process then begins, waiting for the process to finish and return an exit value.

If an error occurs in waiting for the process to finish, it is caught followed by printing the error in the java program. An error in the Python program will also cause an error which is handled gracefully in the java program. It requires a successful MapReduce program to execute before this method is called.

4.4.3 Mapper class

This Mapper class has the goal of imitating the assignment step of a K-Means algorithm. This requires that each data point in the dataset is iterated to its closest centroid. It is assumed that the centroids to be compared to for assignment are provided with the centroids calculated initially from Forgy initialisation in the Driver class, or from the Reducer class where the update step occurs in the previous iteration.

This program contains three methods, named setup, map and calculateDistance. The setup is an override from the Mapper class, from its respective Hadoop MapReduce library. The map method is used by the Driver when executing a MapReduce job. The calculateDistance method is called by the map method to find the squared Euclidean distance between two points.

The setup method for this program has the goal of reading the centroids, followed by passing them into an ArrayList to be used in the map method later in the program lifecycle. This method will use the class variable centroids defined as:

```
private List<double[]> centroids = new ArrayList<>();
```

To provide a source across the entire class for a searchable object containing each centroid. Because the file being read contains only the centroids, it is appropriate to iterate through the entire file in the for loop which parses the data into type Double. The data is then added in its entirety to the previously mentioned variable centroids for future use. While this is not directly related to the assignment step of the K-Means algorithm, it is required to begin computation of the assignment step in the other two methods.

The map method has the goal to perform the assignment step of the K-Means algorithm. This involves assigning a data point to the cluster it has the least squared Euclidean distance to. The method begins with splitting the value passed to it and parsing it as type Double. Sharing similarities as to how the setup method parsed the data as type Double.

When calculating the nearest centroid. There is a hard limitation on how far a distance can be searched. This being the maximum value the type of Double can store. This is why a check for ensuring the distance is below this maximum value is implemented with the following:

```
double minDistance = Double.MAX_VALUE;
int nearestCentroidIndex = -1;

for (int i = 0; i < centroids.size(); i++) {
  double distance = calculateDistance(dataPoint, centroids.get(i));
    if (distance < minDistance) {
      minDistance = distance;
      nearestCentroidIndex = i;
  }
}
```

This also doubles as the logic for finding the smallest distance between a data point and the centroids. As a for loop to iterate through each centroid is used. By default, setting the nearest centroid as invalid means in cases where errors can occur means no data is written and passed to the Reducer. In cases where convergence has been reached, it will continue passing the same data to the Reducer as what it began with, writing the same data into a new iteration and allowing for the convergence to be discovered. In summary, there will only be values passed to the Reducer if the Mapper can find a centroid to assign the data value to.

The calculateDistance method provides the calculation of the squared Euclidean distance of two objects of type Double[]. Because it is possible to have multiple Doubles in the Double list, there is a check to ensure the length of the two data points are the same dimension. Allowing for this method to function with N dimensions of Double lists, provided they are of the same length.

Mathematically, the method will calculate the Euclidean distance using the formula point 2 – point 1 to find the difference, followed by squaring the distance, summing the distance for each iteration of the for loop. This is shown in the following:

```
for (int i = 0; i < point1.length; i++) {
        double difference = point2[i] - point1[i];
        sum += difference * difference;
    }
```

A pass of the for loop simulates for each dimension individually. After summing all the dimensions, the sum is square rooted and returned to the map method that called it.

4.4.4 Reducer class

The Reducer class implements the update step of the K-Means algorithm, extending Reducer from the Hadoop MapReduce library to achieve this. This class contains three methods named reduce, arrayToString and computeNewCentroid. The reduce method has the goal of implementing the update step of K-Means. The arrayToString method takes an array and converts it into a String, used when formatting data to be output from the Reducer. The method computeNewCentroid implements the update step where the centroids are recalculated to reflect the mean of its assigned values. All implemented to work with datasets in N dimensions.

The reduce method for this class takes the IntWritable key and Text value pairs output from the previously mentioned Mapper as input. It begins with parsing each datapoint into type double, followed by assigning it to the following class variable:

```
List<double[]> dataPoints = new ArrayList<>();
```

After pushing each value to this variable, the method computeNewCentroid is called from reduce.

The method computeNewCentroid handles the computation of selecting the new centroid, intentionally separated from the reduce method as it has different goals compared to it. The method will first find the dimensions of the centroid to be calculated, using the dimensions of the data points given to it. Following this, it will sum the values of the data points in a nested for loop, iterating through each. Following this, the centroid for that iteration of the for loop will be calculated through dividing the sum by the size of the data points. This is repeated for every dimension of the data. Once complete, a single recomputed centroid is returned.

After returning the new centroid back to the reduce method, the output of the reducer is given as the key, which is unchanged from the mapper, followed by the new centroid. The method arrayToString is used to change the centroid from type double[] to type String.

The method arrayToString uses a StringBuilder object to format the double list into a string, simply iterating through the list, appending commas to the end of each data entry, deleting the trailing comma, then returning it as a type String. This method only exists as the data type Text does not have a constructor for type double[] but does for type String. Justifying the implementation of this method.

4.4.5 Plotting the graph

Creating a graph of the results was implemented using the matplotlib library in Python 3. This implementation includes a single Python file, followed by the build method in the Driver class handling execution from the java program. The Python file can be run independently assuming there is a dataset of results available for use. Therefore, not requiring a MapReduce job to run each time to generate a graph.

The Python program contains a single function named Plot. It requires an argument for the file location as a string, which is defined as the variable named file. The first operation of the function is to read the file passed as an argument to the program. Implemented as a try except clause in case of a file being unable to be found.

After the file being read, a for loop to iterate through each line begins, where each line is split into its individual data points. Because of the dataset being used in this program, there is a check that the number of dimensions is equal or more than 10. Following this, two dimensions are selected as the x and y axis. Appending their entire dimension to the list of their respective axis. An example of these two dimensions can include:

```
x = float(items[7])#Median Income
y = float(items[9])#Median House Value
x_axis.append(x)
y_axis.append(y)
```

Where the 8th dimension is used as the x axis and the 10th dimension is used as the y axis. This is because Python indexing begins at 0, making 0 the first index, 1 being the second and so on.

After building the axis, the graph is then built using the matplotlib library. This includes calling the functions defined under plt.function as plt is what matplotlib has been imported under. The analysis of more dimensions than two in the MapReduce program means any two out of the 10 dimensions in the dataset used can be analysed to draw conclusions from using this graphical representation, with minimal changes in configuration required.

Chapter 5: **Development**

5.1 Introduction

This section is to document the history of this program's development, with the goal of highlighting the successes and challenges of working with Hadoop and by extension MapReduce. This will be done through comparing the project plan created before the beginning of development to the project diary that was created as the development of the project progressed. This provides the ability to assess the development and provide recommendations for current and future projects involving the components of this project.

5.2 Creating the development environment

5.2.1 Creating a Repository

This project makes use of a GitLab repository provided by Royal Holloway, University of London. Available at https://gitlab.cim.rhul.ac.uk/zjac308/PROJECT/-/tree/main . All that was required was to log in using a given username and password provided.

5.2.2 Creating a virtual machine

For this project, development and deployment will be completed on a virtual machine, this is an intentional design choice as for this project, all nodes will be run on a Linux virtual machine hosted on a single powerful PC. This is different from a real-world scenario of Hadoop; however, it simulates the use of commodity hardware in each instance of a Linux virtual machine.

The software used for creating the virtual machines is VirtualBox, an open-source software that allows for the concurrent running of multiple operating systems on a single machine [11]. Once VirtualBox has been installed on the main PC (note: the main PC is running windows 10) the Linux operating systems are then installed in VirtualBox. VirtualBox includes features such as taking snapshots of operating systems, an important feature that will be used when setting up nodes later. The Linux snapshots used comes installed with Java 17, removing the requirement to install Java on each node separately. This is the same for Secure Shell (SSH), this software is also installed on the Linux virtual machines, meaning individual installation is not required. Both Java and SSH are software requirements for Hadoop.

5.2.3 Linux Nodes

Out of all the Linux nodes, one single node has been selected as the node to develop and test programs on, in Hadoop terms, this would be considered as the NameNode. This node has extra software installed compared to its DataNode counterparts. This includes the Eclipse Java IDE, version 2023-09, which is where the programs for this project will be developed. Once in Eclipse, the IDE is connected to the GitLab repository containing the up-to-date project, followed by using Git Pull to update the local repository.

For all nodes, Hadoop 3.3.6 is installed, this is so the nodes can connect to the HDFS and can be commanded to complete MapReduce jobs. To note, the version used for Hadoop is extremely important, as Hadoop has many depreciated properties [12], slightly limiting the compatibility of created programs.

To install Hadoop, the environment variable HADOOP_HOME and the variable PATH must be set with the Hadoop installation directory and the HADOOP_HOME/bin directory respectively. In this project scenario, the variable was set in Z Shell. If the variables were set correctly, the command line should return this with the following command:

```
$ which hadoop
/path/to/Hadoop/installation/bin/Hadoop
```

Where the path is the Hadoop root installation

5.2.4 Configuring Hadoop

Once Hadoop is installed, the Hadoop Distributed File System (HDFS) now needs to be initialised. In this project, the HDFS will be hosted using SSH localhost, as all nodes will be able to connect.

The following commands will launch the HDFS on localhost:

```
$ sudo service ssh restart
[sudo] password for user:

$ ssh localhost
```

Note: if the system cannot localhost without a password (as was the case for my node), generating RSA keys solves this problem I had. The commands [13] used were:

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

After starting the localhost, the NameNode directory must be formatted, this is done with the command:

```
$ bin/hdfs namenode -format
```

The HDFS will now be functional, with the NameNode being available at the localhost on port 9870, Hadoop's given default port.

To test that Hadoop can perform a MapReduce job successfully, the Hadoop jar command can be used to run a .jar file containing a MapReduce program.

Hadoop jar commands are run as the following, this assumes the working directory is the same as the HADOOP_HOME environment variable:

```
bin/hadoop jar <jar file> <package.driver.class> <input file(s)> <output
directory>
```

The condition,

```
<package.driver.class>
```

is the location of the driver class in the jar file. Along with this, both the inputs and output locations must be locations on the HDFS, the output directory must not exist when the command is run, even if the directory is empty, or it will return an error and not attempt a MapReduce job.

To put a directory onto the HDFS, use the following command:

```
hadoop fs -mkdir <New_HDFS_Directory>
```

To transfer a local directory to place in this newly created directory, the command:

hadoop fs -put <Local_Files> <HDFS_Directory>

will copy the selected files to the HDFS.

If a MapReduce job is run without error using the previous command, then the HDFS has been configured correctly. A single node Hadoop cluster has now been configured. Using multiple nodes will be configured later, after the development of both programs. The use of a multi node cluster would be considered as testing for both programs.

5.3 Developing the WordCount application

Developing the WordCount application brought up multiple challenges as it was the first program to use Hadoop. The application was originally prototyped in a single java file containing the driver, mapper and reducer class under a single file. This is obviously bad practice, so the program was refactored to split these three methods into individual classes. Along with this refactor, I decided to create a new project in eclipse to ensure that the project was correctly configured. The first class that was created was the Driver class, as creating the class meant that I would also have to know the names of the Mapper and Reducer classes, which in this case was Map and Reduce respectively. Using IntWritable as an output value would be suitable for this program as it was just a wordcount application, but in the case that a larger dataset may be used, it may be suitable for the type LongWritable to be used. This is because the limits of type int will be -2147483648 to 2147483647. On the surface this seems like a large enough number to hold, however compared to the numerical limit of long, which is -9223372036854775808 to 9223372036854775807,the risk of overflow errors occurring is clearly orders of magnitude less. However, using long will result in more data being used, so it is best to only use LongWritable when it is required.

The driver class was also developed first, as it allowed for easier testing, with the method of testing being to build a jar (using maven package) followed by executing the Hadoop jar command on the HDFS, it would then return an error, providing the necessary feedback to continue developing the class. It is also a reason why the Mapper class was developed before the Reducer class, as it would be selected first by the Driver class, so errors from Mapper would be returned before errors in reducer. Of course, both classes would have to exist before either Mapper or Reducer can be tested fully.

Once driver was completed, development of the Mapper begun. As an extension of the default Hadoop Mapper, this class only needed to configure the filtering and sorting of the given dataset, which is to why it is a small program. The goal was to find an output where given a string, it would map it to a key, followed by the number 1, so it can be summed in the Reducer later. Testing that the correct output had been created was difficult without the existence of a reducer, so priority when testing the Mapper was to ensure that it worked without error (i.e., the errors would only throw when trying to reduce using a non-existent reducer). Testing the outputs of the MapReduce job would be done when both Mapper and Reducer class was developed to a point where it was possible to evaluate the outputs.

The reducer class was the simplest program to develop, with the Mapper being the most challenging due to being unfamiliar with StringTokenizer. Reducer was much simpler, as it was just taking every value, summing it with the following lines:

```
int sum = 0;
for (IntWritable val : values) {
  sum += val.get();
}
result.set(sum);
```

Which was simple as every value was known to be a valid value.

At this point the program has been tested multiple times in a Hadoop single node cluster. The structure of these tests was to pass the program to a Hadoop JAR command, this was very time consuming, as each test required recompiling the JAR, followed by using a command line command. Alternatives such as using the MRUnit libraries were considered, however it was incompatible with version 3.3.6 of Hadoop and was retired in 2016 [14]. So, testing now became even more important as each test took much longer to complete because of the lack of unit tests to provide rapid testing of smaller sections of the program. Testing required the entire implementation to be completed before it could begin, following the waterfall software engineering methodology.

Testing WordCount however was not too difficult, errors that were thrown did not take an excessive amount of time to fix, along with logical errors being rare due to the nature of this specific program having little logical operations.

5.4 Developing the DistributedGrep application

This program was more advanced than the previous, with the goals of this program being to find all matching strings with a given regular expression.

The plan for developing this application would be like the WordCount application, where the driver class will be developed first, as it can be tested using the Hadoop jar command. When testing the original driver for this program, errors were continuously thrown about the use of a Tool being required for the program. After research, this is referencing two libraries that is in org.apache.hadoop.util, Tool and ToolRunner. The Tool interface is what allows for the search query to be added to the MapReduce job, so it was implemented along with job, to allow the program to run correctly. This also changes the structure of the program, as a requirement is that the run method from Tool is overridden as it is now the main method for running the program. This also allows for the main class to stay as clean as possible, making the code much easier to maintain if new features were to be implemented.

To take a user input, the library BufferedReader from java.io was used, as it was a simple tool required to take a user input and can be run without any extra dependencies other than what has already been specified. The use of the command line to take a user input also makes sense, as running the program requires the use of a command line, with or without the use of the given scripts.

The Mapper class was next to be developed in this cycle, with the goal being to take the user input regular expression, filter it with the dataset then create a key value pair where the value is the located string and the required line number and file it was located in. After some research, the method chosen to find matching regular expressions was the java library java.util.regex.Pattern. This library can compare a line given to its Matcher object, to find if it is a valid regular expression. The line:

```
while (m.find()) {
```

returns either true or false depending on if a valid expression was found. This class took the longest to test, due to the more complex filtering in comparison to the WordCount application raising more complex errors. This meant that the importance of having a defined goal with each test case was paramount. An optimisation with test cases would be found, where using a significantly smaller dataset would allow for faster completion of tests, providing feedback significantly faster.

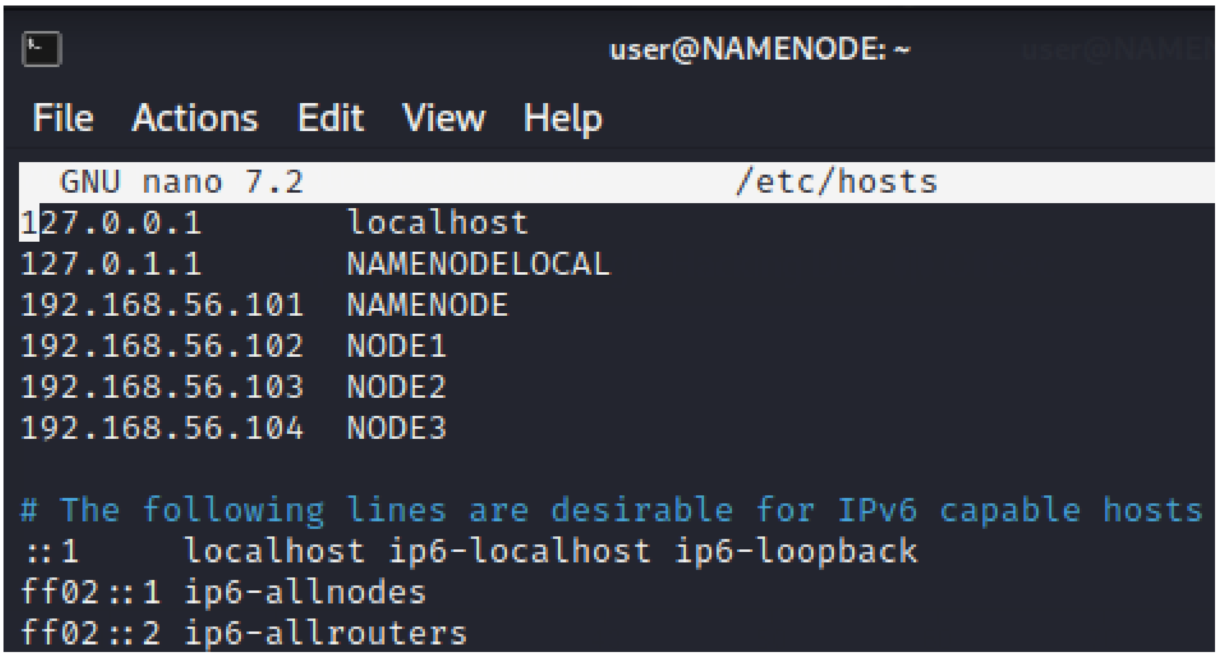
Developing the Reducer class was much less challenging than Mapper, as the filtering, computation and formatting of the outputs was completed within Mapper, this class simply required to combine each output together to create an organised output text file containing the required data.

Overall, ensuring that appropriate TDD was completed for the Mapper class especially was a significant challenge, mostly due to the turnaround time of each test case, however impact on development was only minor. Development was surprisingly smooth considering the wealth of new libraries that had to be used.

5.5 Creating a multi node cluster

Hadoop provides the ability to run MapReduce jobs on a single node cluster. While theoretically there is no issue with running a single node, in real-world scenarios, Hadoop clusters typically consist of multiple nodes, with Hadoop being demonstrated on clusters of up to 2000 nodes [4]. The goal for this implementation was to have 4 datanodes, which would answer to a single master node. The master node would also be one of the datanodes.

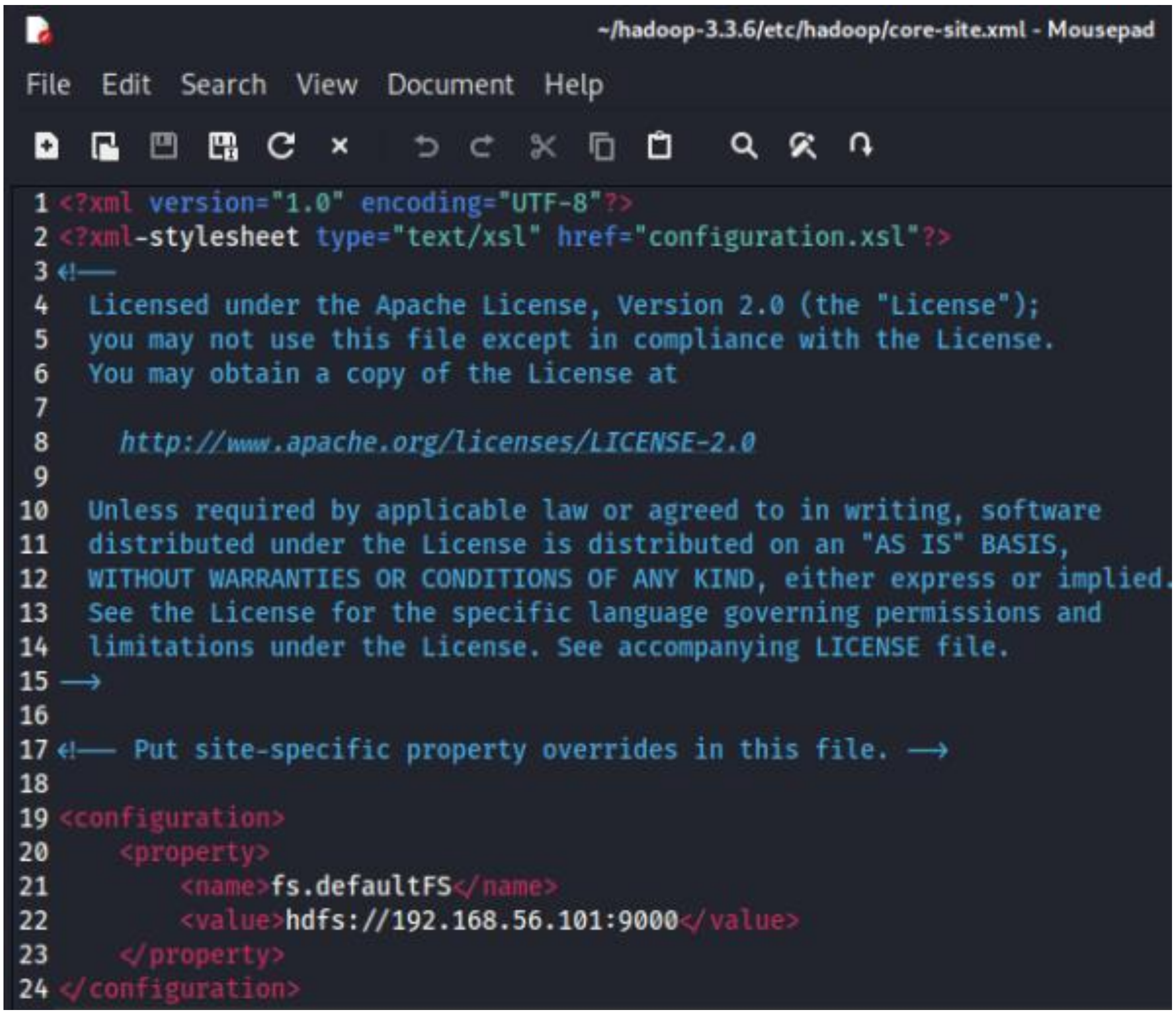
To begin configuration of the cluster, the single node VM was cloned until there were 4 VMs. The requirements for the cluster to function was for each node to have shared ssh keys, allowing each node to transfer data to every other node, effectively building a mesh network. This required the use of a VirtualBox Host-Only Ethernet Adapter to connect each VM. To note, an extra connection would be required to connect the VM to the internet, in the case of using GitLab. Once each VM had its own IP, the file /etc/hosts in each VM had to be modified to include each IP of the network, irrespective of the node being a datanode or namenode.



```
  GNU nano 7.2                     /etc/hosts
127.0.0.1       localhost
127.0.1.1       NAMENODELOCAL
192.168.56.101  NAMENODE
192.168.56.102  NODE1
192.168.56.103  NODE2
192.168.56.104  NODE3

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

***Figure 5* - The file /etc/hosts opened in the nano text editor.**

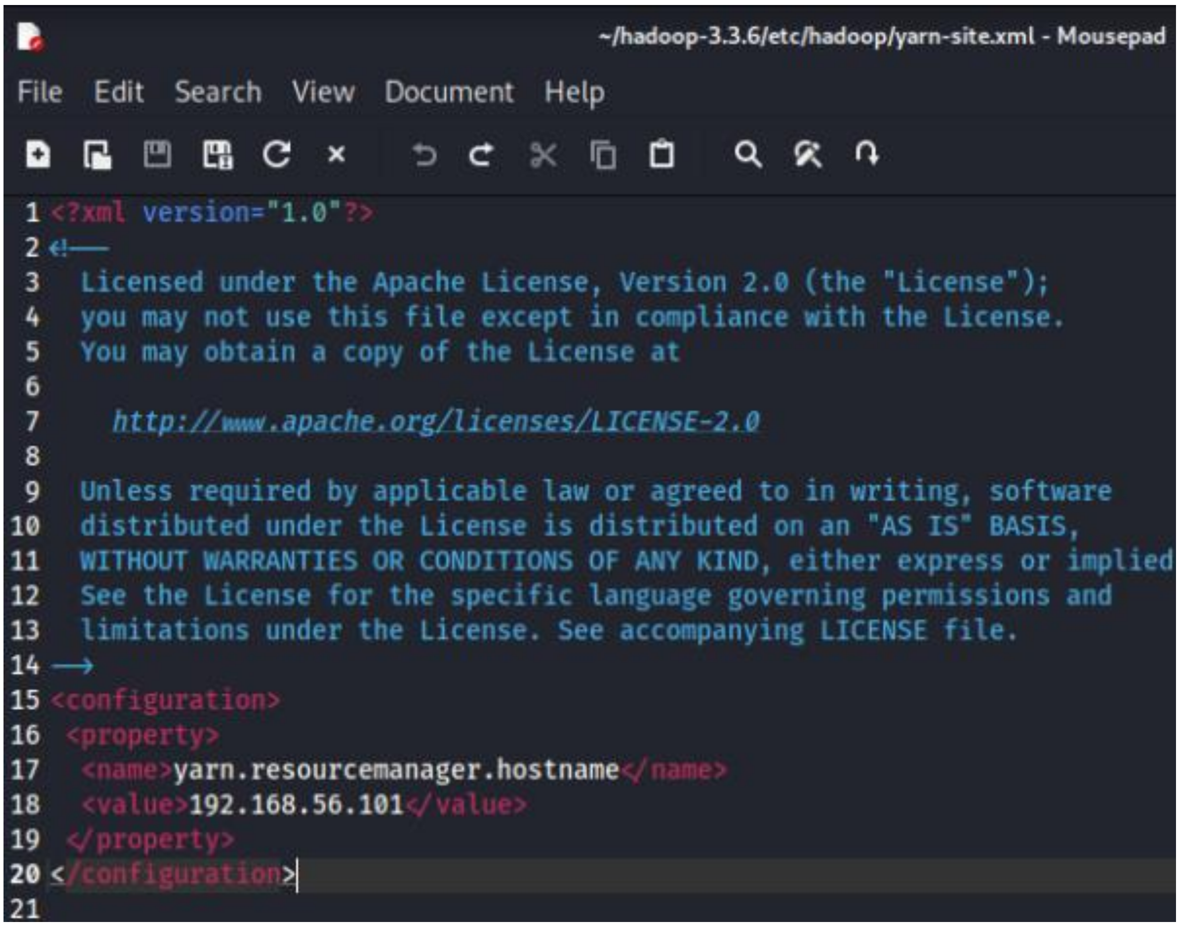
Once this was complete. The Hadoop installation had to be modified to provide datanodes with the correct IP address to connect to. This involved changing the files in Hadoop installations /etc/hadoop directory, namely the files **core-site.xml** and **yarn-site.xml**. The former required setting the default file system IP to the datanode's IP.


~/hadoop-3.3.6/etc/hadoop/core-site.xml - Mousepad
File Edit Search View Document Help

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://192.168.56.101:9000</value>
    </property>
</configuration>
```

***Figure 6* - An example core-site.xml file to be used on each node of the cluster.**

Along with this, the IP of a resource manager must be defined. This would be configured in the **yarn-site.xml** file.


~/hadoop-3.3.6/etc/hadoop/yarn-site.xml - Mousepad
File Edit Search View Document Help

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>
 <property>
  <name>yarn.resourcemanager.hostname</name>
  <value>192.168.56.101</value>
 </property>
</configuration>
```

Figure 7 - A configured yarn-site.xml file for use on every node in the cluster.

Another note is that the hdfs-site.xml contains the following property:

```
<name>dfs.replication</name>
<value>1</value>
```

While not required, it is best to define the number of replications of files between nodes on the HDFS.

Once the file configuration was complete. Sharing ssh keys was completed. Followed by launching the namenode and its datanodes. If successful, the HDFS would show the following information about the datanodes:

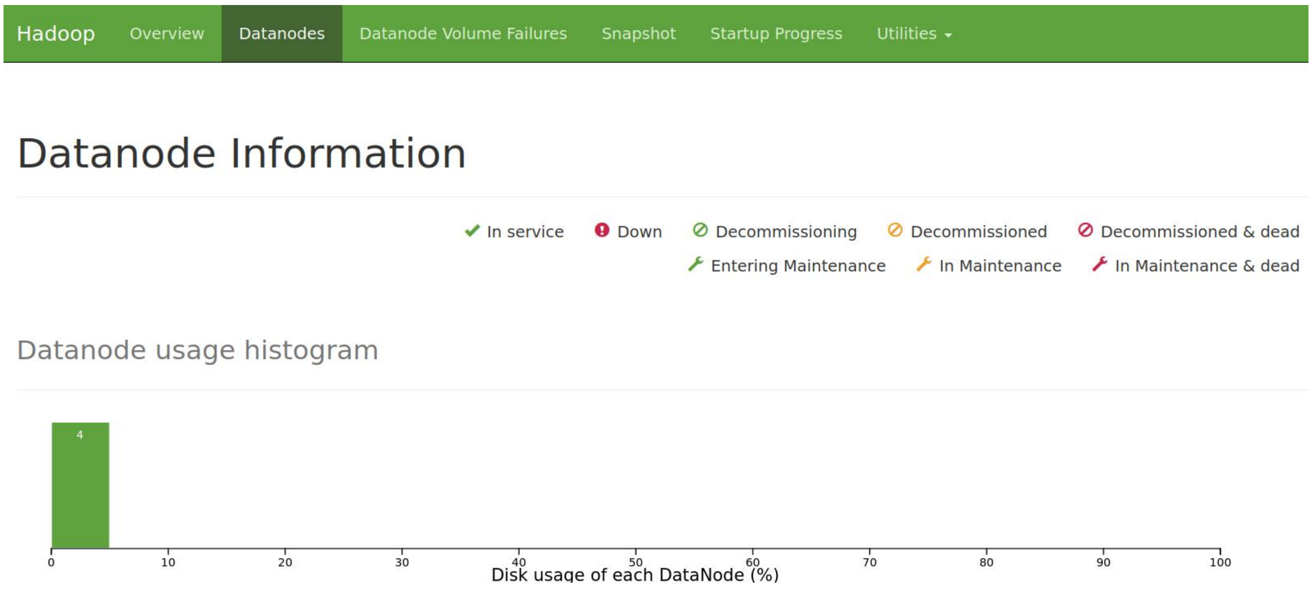
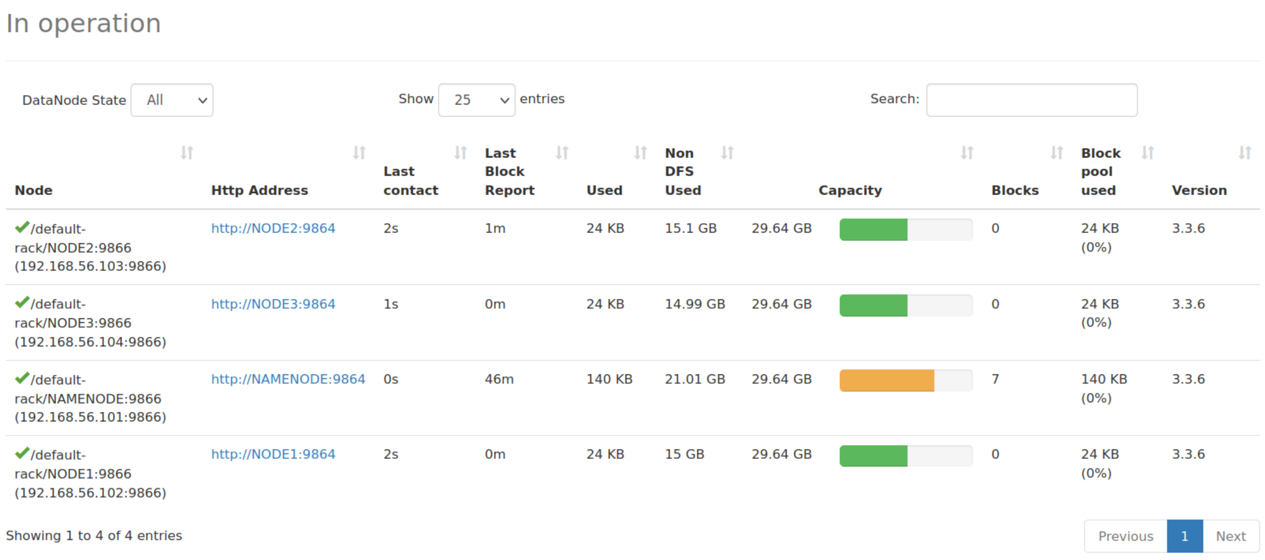


Figure 8 - The URL <HDFS IP>:9870/dfshealth.html showing all nodes as being in service.



In operation

DataNode State All Show 25 entries Search:

Node	Http Address	Last contact	Last Block Report	Used	Non DFS Used	Capacity		Blocks	Block pool used	Version
✓/default-rack/NODE2:9866 (192.168.56.103:9866)	http://NODE2:9864	2s	1m	24 KB	15.1 GB	29.64 GB	🟩	0	24 KB (0%)	3.3.6
✓/default-rack/NODE3:9866 (192.168.56.104:9866)	http://NODE3:9864	1s	0m	24 KB	14.99 GB	29.64 GB	🟩	0	24 KB (0%)	3.3.6
✓/default-rack/NAMENODE:9866 (192.168.56.101:9866)	http://NAMENODE:9864	0s	46m	140 KB	21.01 GB	29.64 GB	🟧	7	140 KB (0%)	3.3.6
✓/default-rack/NODE1:9866 (192.168.56.102:9866)	http://NODE1:9864	2s	0m	24 KB	15 GB	29.64 GB	🟩	0	24 KB (0%)	3.3.6

Showing 1 to 4 of 4 entries

Figure 9 - The list of nodes on the Hadoop cluster, shown as being in service.

Setting this cluster configuration up was easily the most challenging part of this entire project so far, with the nature of configuring a mesh network going far beyond the original Hadoop documentation, such as the use of Secure Shell and VirtualBox host only adapters to simulate a real-world Hadoop cluster. Providing more detailed insight into optimal cluster configuration. To note, all VMs are hosted on a single powerful windows PC. So, while not a perfect recreation of a typical Hadoop cluster, it is remarkably close.

5.6 Developing the Frequency Distribution program

5.6.1 Introduction

This program has the goal of taking a dataset and providing a frequency distribution graph of the dataset. This will include using MapReduce to filter the dataset. It was decided that using python to visualise the graph was suitable due to previous experience using python to visualise graphs. So outlined as significant modules for this program include the MapReduce program and the python program. The MapReduce program can also be broken down into its separate Mapper, Reducer and Driver classes respectively. Breaking this program into 4 distinct subcomponents that can be used to quantify development progress.

When planning this program, development flow was planned so the first objective was to build the MapReduce program, creating the components in the order of Driver, Mapper and then Reducer. Followed by refactoring the Driver to accommodate for the created Mapper and Reducer. An example of changes in this refactor will include changing the configuration to include the created Mapper and Reducer.

5.6.2 Driver

Developing the Driver class involves two phases, the first phase was to create the Driver without the existence of the Mapper and Reducer, this means that a generic Driver class will be made. This includes creating the methods for running a MapReduce program, followed by the methods for graphing the results from the MapReduce operation. This approach was decided as this would make refactoring the code mandatory.

To begin developing this Driver, the goal was to build a general Driver class, developing this was simple as I have made Driver classes previously. The difficulty would be in adapting the Driver class after the Mapper and Reducer have been completed. The first phase of development was quite simple, with the challenges originating from organising large portions of the program. This includes the main method, the driver method (which I named run) and the method to build a graph. At the end of this first phase, the general code structure was outlined. This included the implementation of the logic of the program, with the second phase of developing this class having the goal of providing specifics to the configuration. This would include for example specifying the locations of scripts, HDFS locations and various file paths on the HDFS and locally stored. Check style [15] was implemented from the beginning of development, primarily as implementing check style later in the development process would be a more difficult implementation compared to sticking to a check style from the beginning of development.

The second phase of development would take place after implementing the Mapper, Reducer, and python script to graph the data. This phase turned out to be more difficult than the previous not because of the logic implementation that was required, rather the phase included testing not just the Driver, but the Mapper and Reducer included. This caused this phase to be focused on testing more than developing the program. It was planned to test the MapReduce program to ensure that it worked correctly. After ensuring it worked correctly, then further development and testing of the python program can be completed.

Overall, this class was the most complicated implementation in this entire program. Primarily because testing this class required testing the Mapper and Reducer class by extension. This class also required significantly more refactoring in comparison to the rest of this program because of the two-phase nature of developing this class. This requirement for significant testing and refactoring was required for the developed program to be of a high quality.

5.6.3 Mapper

Developing the Mapper for this program required counting the number of occurrences of a value in a dataset. It was planned to develop this program before the Reducer, primarily because of the data format output from the Mapper would be passed to the Reducer, so it would be appropriate to develop the Mapper first.

Developing the Mapper itself was simple. The challenges in developing the Mapper came from how it would interact with both a Driver and Reducer class. Viewing it as a middle ground between the Driver and Reducer. As an error in the Mapper will only occur when the Driver has executed without error and will not provide information about if the Reducer is erroneous. This made testing the Mapper only possible when the Driver was functional. This blocked testing until the Driver was functional, justifying the decision to implement the Driver first.

The Mapper is a smaller class in comparison to the rest of the program. However, this is an extremely important program, with optimisation being a high priority goal. Overall, this program efficiently completes its task. It also achieves its goal of interacting with the Driver and Reducer correctly and without error.

5.6.4 Reducer

The Reducer class in the Frequency Distribution program has the goal to sum the counted occurrences of a value in a dataset. Not to sum the values themselves, rather sum the occurrences of those values. Implementing this program was the easiest to implement, primarily due to the previously mentioned complexity of the Mapper and Driver class. This program was implemented last in the MapReduce program. Meaning this Reducer would be developed and tested fully before development on the python program to graph the data was implemented.

Implementing the logic of the Reducer was the most difficult part of the implementation. It began with counting the values. This was done through summing each word and its given value from the Mapper. When developing the Mapper, it was decided that counting each word as the key, followed by the value being one in the key value pair was appropriate. Therefore, using the output from the Mapper. The use of a loop to sum the keys was deemed appropriate for the Reducer. Once implemented, returning the results of the sum into the context of the MapReduce algorithm.

Out of the three MapReduce classes in this program, the Reducer was the smoothest program to implement. This is mostly due to most of development in the MapReduce program being completed before beginning development of the Reducer. These previous programs define sections of the program such as data types and formats the Reducer will use, including the use of data types of output from the Mapper. Testing this program was also the smoothest, as it required both the Driver and Mapper to run error free before testing was possible. This made the errors that were thrown during testing small in comparison to the significant errors thrown from the other classes. Overall, in comparison to the other classes, this class was the smoothest to develop and test while still having its challenges.

5.6.5 Graphing in Python

Deciding on the method to graph the results of the MapReduce program was based primarily on what was familiar and/or simplicity to implement. This choice was justified through wanting to focus on developing the MapReduce program to the highest possible standard, as that program is likely to be the significant challenge in implementation. Based on these factors, it was decided that plotting the graph would be completed in Python using the matplotlib library.

This choice allows for an appropriate level of separation between the MapReduce program and the graphing program, so they are viewed as distinct, independent modules. This splits the program from a single MapReduce and graph into two distinct modules. Part of the design choice was to ensure that running the python separately after a MapReduce operation will produce the graph independently given the correct data. This makes the program more flexible, allowing for multiple graphs to be plotted with the only required change being to change the text file being read from in the python program. This also keeps the python program free from any dependencies on a MapReduce program.

Beginning with implementation. This was completed last in the development of this program. This means the entire MapReduce program had been completed when development on this graph program began. This includes the method in the Driver class to execute the python script from inside the Driver. In the method in Driver, it was decided to make a configuration of class variables to pass to the method, which is then used in the process of downloading the MapReduce results locally, followed by passing them to python.

Implementing the build method was quite challenging. One problem was to allow the java program to execute an external python program in of itself. It was found after some time that the incorrect file permissions on the Linux system used were given to both the java and python files respectively. After fixing this, the libraries used for executing the python script are not Apache Hadoop libraries. This is intended as this part of the program was not Hadoop related. The libraries that proved to be useful after testing various others include ProcessBuilder, BufferedReader and InputStreamReader libraries. These libraries are packaged within java, a choice made during development to keep the dependencies at the absolute minimum. Handling the execution of the script within the java program required a lot of research beforehand and some significant testing. However, after the appropriate implementation was found, the logic behind the method itself was smooth.

The python program had specific requirements. Take a given text file containing already filtered data. Followed by graphing this data appropriately. Implementation was possible through the familiarity with matplotlib already. Followed by extracting the data from the file, it was decided to read each summed value followed by assigning it to a unique key in a python dictionary. This would provide the x axis for the graph. This design was chosen as the number of entries on the x axis would be impossible to practically keep the same, therefore it was seen as a requirement to have a dynamic number of entries on the x axis part way. A requirement which was found during development.

Testing this python script was quite simple after implementation. This is because its dependency is on an already filtered dataset. So, if a correct dataset was given when testing, testing the program was typical of a normal small program, bringing with it a high level of efficiency when testing. Overall, developing this python module of the program followed a typical development cycle. Because of the planning to separate the program from its java counterpart, it made developing and testing the program separate from the java. This was important as testing a program with both java and python elements can lead to difficulty testing appropriately. Avoiding this was a significant goal when developing this program which was achieved. This also achieves the goal of ensuring that there are no direct dependencies between the java and python elements of the program. To expand, this means the python can execute given a dataset text file without a MapReduce job running beforehand.

Overall, the implementation of the python program proved to be a different challenge. This challenge was somewhat familiar as I have previous experience using matplotlib. The challenges were present where I was unfamiliar with portions of the code I needed to develop. This includes sections such as filtering into a dictionary given a text file. These sections overall were difficult but were implemented well.

5.7 Developing the K-Means program

5.7.1 Introduction

This program is intended to take advantage of the K-Means algorithm to take a dataset and provide an insight into trends observed through its clusters. It was decided that the program will require both a java element and a python element. The java element will consist of a MapReduce program, where the K-Means calculation is done, followed by the python script plotting the graph. It is an important requirement to note that the python script does not actually calculate any part of the K-Means program, it is only meant to take the already process data and visualise the results.

To begin making a K-Means algorithm, I first had to know what the K-Means algorithm was [9]. This involved extensive research into the algorithm itself. The research required was to include understanding the algorithm to a point where implementation in java would be possible. This is an extremely important part of the development for this program as because of the NP-HARD nature of K-Means clustering, creating an efficient program is a high priority. The decided implementation was to use a standard K-Means algorithm implementation, rather than attempt a more complex (but potentially more efficient) implementation.

K-Means as an algorithm contains two distinct steps, an assignment step, and an update step. Knowing this, it was decided to implement the Mapper as the assignment step in K-Means and the Reducer as the update step [9].

Another important requirement was the requirements of the dataset being used. It was decided that any dataset that consists of only real numbers will be considered a valid data input. So, in other terms, any dataset that contains any letter will be considered invalid and will throw an error when being processed. This also means that the correct error handling processes will be one of the program requirements.

So, to summarise, the requirements that were derived from this include making a K-Means java application for use in Hadoop and its MapReduce framework, followed by using a python script using the matplotlib library to graph the data output from the java program. This java program will require the ability to set an arbitrary number of iterations that the K-Means algorithm will run through, as there can be observed differences between the number of iterations and how close the data is to convergence (assuming it is not already at convergence).

5.7.2 The Mapper Class

This java class requires that, given initial centroids for the iteration, each data point is assigned to the nearest centroid based on its Euclidean distance squared (as per K-Means). This is considered as the assignment step of K-Means, where each data point is assigned to the centroid.

This program began with attempting to even read the data, as the requirements include reading the initial centroids that were generated in the Driver class. It was decided that reading the initial centroids file would be performed using the BufferedReader library in java.io, where the Hadoop filesystem would open the file, then pass it into a BufferedReader object to read the file. After this decision was made, it was also required that the mapper would compare with the initial centroids data to allow for the calculation of the distances to determine the appropriate centroid to assign the datapoint to. It was decided this operation would be performed in the setup for the mapper, overriding the default setup method for a mapper.

After setting up the mapper correctly, the assignment step can now begin. The goal being to replicate the behaviour of the assignment step of the K-Means algorithm. The first implementation led to multiple problems, the first of which being the values being passed to the mapper being parsed incorrectly, fixing this was relatively simple with only a few minor changes to how the values were read to the mapper, along with using the correct data types in the definition of the generic method. However, a rather interesting bug occurred where each data input was assigned to a unique cluster, basically meaning no actual distance was calculated as the centroid was considered the same as the data, effectively having x data points having 0 distance to its unique centroid given. This was a complex logic bug in the program and was one of the longest problems to fix in the entire program. The cause of this bug stemmed from incorrectly reading the data file as the initial centroids, effectively reading the data file twice, once for the initial centroids and once to parse the data points, which is why this occurred. While this seemed like a simple error to fix, it was one of the last bugs to be fixed in this entire program as due to the complex nature of K-Means mathematically, pinning down where the specific issue was located was difficult even with the luxury of stack traces. As the bug was only discovered after creating the mapper, reducer, and driver, it was difficult to even know the bug originated from the mapper at first, but was soon discovered after some searching.

When creating the mapper class, testing was difficult as it was impossible to test unless there was also a driver and reducer class to attempt a MapReduce program, meaning that testing this program required making all three programs, then finding out what has gone wrong in any of the three programs. Errors could span over multiple programs, where the mapper could pass an empty datapoint, throwing an error in the reducer. The fix would therefore have to be done in the mapper; however, the stack trace would identify a line in the reducer as causing the error. This would lead to a lot of complicated debugging as all three files would in theory be tested at the same time.

Overall, developing the mapper class required a significant amount of planning for not just only that class, but also for classes that depend on it, such as the driver and reducer. Even with extensive planning, there was inevitably unforeseen errors that would require fixing, due to the complex nature of K-Means, some of these errors would be much harder to solve than others, as I am quite familiar with Hadoop at this point in development, further exacerbating the differences between Hadoop specific errors and K-Means specific errors.

5.7.3 The Reducer Class

The reducer class for this problem is intended to be modelled around the assignment step for a typical K-Means algorithm, where the centroids are recalculated to identify either convergence if the centroids do not change, or to identify the new centroids for the next iteration of the algorithm. Developing this class began with research into the K-Means algorithm, where understanding the mathematical concepts of the assignment step is required to provide an appropriate implementation. Once this research into the algorithm itself was complete, only then could I begin with the java implementation.

When beginning the java implementation, it was important to note that the mapper will be passing values from its class into the reducer, the values itself being an IntWritable type and a Text type. This was important to factor in as the behaviour of the reducer was heavily dependent on the data types being passed through. In the reducer, there were two significant operations that needed to be completed, the first being to read the values from the mapper, followed by computing the new centroids from the data taken from the mapper. Reading the data was somewhat simple, where each entry of data is read individually and parsed into an ArrayList object where it is stored.

After being stored in the ArrayList object, it was clear that computing the new centroid would be much more complex, it was decided to use a new method where the centroid can be computed outside of the reduce method. While it was not required to compute it outside of the reduce method, I felt as it would be much easier to separate these two operations into different methods as combining them into one can get quite confusing for debugging at the time, along with any modifications that were to be done after the program was completed in the future.

Computing the mathematical step of computing the new centroid was the most difficult part of developing this program, mostly due to the complex mathematical nature of deriving the new centroids. This was exacerbated by the previously mentioned error in the mapper, where each datapoint was assigned to a unique cluster where the centroid was the same data as the data entry itself, which made knowing if the new centroid being correct or not practically impossible until that error was resolved.

Another choice that had to be made during development was to either MapReduce the entire dataset, or potentially minimize the dataset into an N number of dimensions, which in my case would most likely be two dimensions as this data had to be visualised. Inevitably the dataset would have to be filtered down into two dimensions, it was rather the case of reducing the dimensions in the reducer or later in the program life cycle, such as when the data is to be visualised. While it would make sense in terms of optimising the program to only MapReduce the two dimensions being used in the final data output, the level of potential insight into the data would be severely reduced as there would be only two rather than N dimensions of data being processed. It was therefore decided that the program will be able to handle N dimensions of data processing, with the python script handling the data that is graphed. This also makes the program much more adaptable, rather than hard limiting the program to specific datasets.

Accounting for the N in N dimensions was surprisingly simple, as finding the dimensions of the dataset can be completed with .length operators on the given value. Along with the use of arrays, the operators that arrays can use will also provide the ability to process data in N dimensions, the dataset used contains 10 dimensions, so accounting for N dimensions as compared to attempting to hardcode in 10 different operations per dimension is the obvious choice that was made when developing this.

One of the more difficult errors to get past included handling the data types in the generic call for the mapper and reducer class, as using an incorrect data type will throw an error when testing. A significant problem was that it was not possible for data types that can be casted to others normally cannot be casted in Hadoop MapReduce, for example Int and Double. This initially caused a lot of confusion as IntWritable and DoubleWritable strictly cannot be compared in any way. It makes sense now as these data types are custom to Hadoop and will not work as they are implemented as an argument in a generic method, but also there is there is no possible comparison that can be done to the different data types. This caused errors when there were attempts to convert from anything between IntWritable, LongWritable and DoubleWritable, therefore the decision was made to only use an IntWritable data type, with java doubles being used when calculating internally within the reducer. For the reducer specifically, it was important to note that both the input and output values stay the same, with it being an IntWritable and Text input/output. As it was discovered during development that changing the output format from its input was not required to generate an accurate result. While using Int, there was the risk of overflow errors given potentially an incredibly large single input (say anything +/-2147483648) however this risk is so incredibly small that a dataset containing a single value which is out of that number range is likely to not be a realistic dataset and therefore not a dataset to be used in this program.

5.7.4 The Driver Class

The driver class for this program requires the typical features from a MapReduce program, however it will also require a method where a python process is called to build a graph once the K-Means algorithm has been completed. This driver class would also require a way to select the centroids initially, with each iteration selecting a new set of initial centroids. Developing this class was difficult in the sense that it was bringing the entire program together, relying on both the mapper and reducer to function correctly for it to function correctly.

The first goal was to get the typical MapReduce requirements completed; this includes creating the configuration for Hadoop etc. Thankfully my experience with Hadoop already helped enable for smooth development of this section of the program.

The difficult part of this program included handling the initial centroids, where a file containing the initial centroids must be made followed by populating it with centroids. It was decided that the centroids would be selected randomly, as per a random partition in K-Means terms. This was implemented using java.util.random, which was considered the optimal implementation as it does not add an extra dependency, along with being relatively efficient and a competent source of randomness. Reading the initial centroids was difficult as it was based on the number of initial centroids defined, therefore requiring a dynamic loop as part of implementation. This loop would include finding a random entry in the data file, followed by assigning just that entry into a centroid. This took a long time to get working correctly, especially factoring in all the errors from other parts of the program, due to the nature of developing a MapReduce program concurrently with a Driver, Mapper and Reducer.

5.7.5 Plotting the graph in Python

This program required a way to graph the data in two dimensions. This program would also begin development after the completion of the MapReduce implementation of K-Means. So, to contextualise, I already know the data output from the MapReduce program. This being a 10-dimensional output of a list of N centroids in a dataset. It was decided to filter this down to 2 dimensions. This decision was made to keep the implementation of the program simple, as attempting to visualise results in 10 dimensions would be incredibly difficult.

Given these prerequisites, the choice of implementation was decided to use matplotlib in Python 3, sharing similarities to the implementation in the Frequency Distribution program. To begin this program, first the text file must be read. It was decided that due to the required nature for text files to be read from different file locations, using a global variable, followed by passing the variable into the function was deemed the appropriate implementation for determining the text file to be read. The MapReduce program specifies that the file will be named “final_centroids” therefore it is required that the end of the string defining the file location will contain this string. Reading the file itself in Python was simple to implement, as the format has been defined in the MapReduce program.

Handling the data to pass into the graph presented some challenges. All data was in a numerical format, including the use of decimals and can be positive or negative. This made using a float data type in Python the appropriate way to handle such data, while also allowing numerical operations to be performed when matplotlib quantifies its value on a graph. Once the string of data has been read from the file, converting it to a float was simple. This is because the data is comma separated, meaning splitting the data into a list is easily possible, followed by converting into a float and appending the data into a list for use with matplotlib. If the data is in an incorrect format, it is most likely a fault in the MapReduce program, however this Python program must check for it. This led to the use of a try-except statement to account for this. An important optimisation that was found was to check for an error in each individual line, as opposed to the entire dataset given. This way of handling errors means that if an error is present in one line, however the rest of the dataset is formatted correctly, the program will still function as intended, however with the omitted centroid.

Once the data was correctly passed into the matplotlib graph, graphing the data itself was simple. Overall, the challenges in this program originated from handling the data itself, followed by passing it correctly into the graph. The way this program has been implemented also allows for the program to analyse any two dimensions in the dataset of 10. Allowing for more insightful conclusions into the dataset. I believe the goals of this program have been achieved along with allowing for future aspirational goals to be implemented in the future, be it using the same MapReduce program or different. The goal of making the program as modifiable as possible allows for these future goals to be possible.

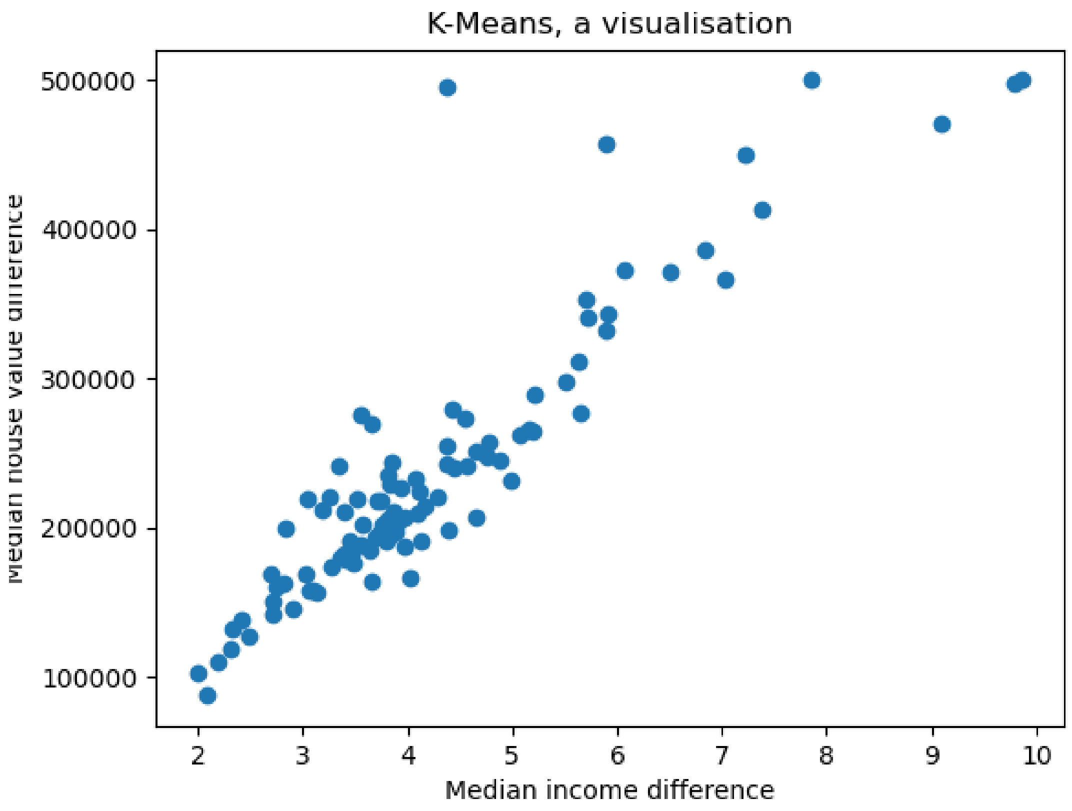


Figure 10 - An example output of the K-Means program.

Chapter 6: **Waterfall and Software Engineering**

6.1 Software Engineering

The concept of software engineering includes all the processes involved in the creation and maintenance of a software application. This includes the design, development, testing and maintenance of the software. Software engineering has multiple approaches, the one used in this project is the waterfall approach. Another software engineering approach considered was using Test Driven Development (TDD). Finding the correct approach is dependent on the factors of the project. This can include factors not related to the software such as how many people are working on the project, for example having a daily scrum meeting with over one hundred people would be unsuitable in a practical scenario. Taking parts of a methodology and not using the other parts can also be suitable. For example, a single developer using agile may implement the idea of sprints into their project development. However, will not have daily scrum meetings as the project is individual.

Taking these factors into account, it was decided to apply the waterfall approach for software engineering for this project. Primarily because of the nature of the five-stage process being applied fitting the best in the context of how a Hadoop MapReduce program is developed.

6.2 Waterfall

The waterfall software engineering method is used as one of many software engineering methodologies which can be used to develop and maintain high quality software. In a high-level context, waterfall follows a five-stage process for software development. These steps in order are:

- Requirements
- Design
- Implementation
- Testing
- Maintenance

This method of development was decided as the software engineering method used for this project. Another method considered for this program was Test Driven Development. This methodology was to implement test cases which double as the intended goals for software, followed by implementing code to pass every test case. This methodology has its advantages in ensuring code is as efficient as possible for solving test cases, an important factor given the context of this project. However, implementation requires an appropriate test case implementation. For MapReduce, the only test case found when researching this project was MRUnit. However, implementing this was not possible, as MRUnit was retired in 2016 [14], so the decision to use a waterfall approach was made.

The requirements stage for this methodology would include quantifying what a successful implementation of the program would need. This included finding what the program was to use in terms of software libraries. In the case of this project, this would be using Apache Hadoop and its associated MapReduce libraries. Other example requirements include making the program run within a certain level of efficiency, quantified by using Big O notation. When assigned this project, most requirements were given by the project booklet, making this stage much more focused on the software implementation itself rather than finding what overall behaviours I wanted the program to achieve.

The second stage, design, covers all significant plans made before creating the program to help in the implementation stage. This can include creating UML diagrams or Gantt charts for theoretical programs to be implemented within a certain timeframe. This stage also requires a significant amount of decision making, as multiple implementations of the same program are possible, this stage attempts to find the optimal implementation given the outlined requirements which have been finalised before this stage begins. In the case of this project, this required several drafts of UML diagrams to organise the overall structure of the program. This led to decisions such as separating significant mathematical operations into separate methods. At the end of this stage, it should be possible for any programmer to read the requirements and design and understand both well enough to create an implementation of the program with minimal differences, as every programmer can implement even the same program differently given a perfect requirements list and design.

Implementation of the program is the most important stage of the waterfall process, as it depends heavily on the success of the previous stages and will heavily influence the rest of the stages. A poor requirements analysis and poor design will make providing a good implementation exponentially difficult. This poor implementation will take a long time to test, followed by future maintenance being difficult as the original implementation was so poor. For this project, implementation took a significant amount of time, as due to the importance of this stage, I wanted to provide the highest quality program I could, with the idea that extra time spent on implementation would lead to time saved in testing, as the number of bugs in the program would be less prevalent and less significant. This would also shift the testing stage from focusing less on syntax errors and more on the quality of the code itself.

The testing stage in waterfall covers all refactors and tests completed in a program that has just been implemented. This stage was the longest and most difficult for this project. Primarily because of how MapReduce works as a program, requiring three different classes to function without error simultaneously when testing proved more of a challenge then originally thought. This stage requires that all implementations must be completed, either for the program as a whole or for a certain section, such as an individual class for example. The goal of testing is to put the program into a state where it can be used by its intended target, which can surprisingly vary the levels of testing required to reach this state. Overall, the program must be in an acceptable state when this stage has completed, fulfilling all the requirements specified.

The final stage, maintenance, involves keeping the completed software up to date. This includes refactoring previously undiscovered errors or updating the program to no longer use depreciated code to name some examples. This process is ongoing after the program has been completed, be it indefinitely or for a given planned length of time corresponding to the lifecycle of the project given. This stage is where this project is at currently. Maintenance for this project does not include implementation of new features, as these features will go through the waterfall methodology from the first step, as it is considered a new program to develop compared to maintaining a current one. The goal of maintenance is to keep changes at a minimum, only changing the program when necessary to maintain or improve its quality.

Overall, the use of waterfall has proven to be a good choice for this project. While it is unfortunate that my original idea to implement Test Driven Development (TDD) did not occur, waterfall has proven to be a suitable choice. When revisiting this project in the future, a goal will be to find a suitable implementation of TDD to provide new features and optimisations for this project.

Chapter 7: **Professional Issues**

7.1.1 Introduction

Using not just Hadoop but any massive scale data analytics implementation can come with some very important ethical dilemmas that must be addressed through acknowledgement and appropriate implementation to combat these issues. This section will be used to document some of the issues that I faced while developing this project, followed by the remedial actions I took to mitigate or eliminate some of the risks presented by a project such as this. Risks in this project include but are not limited to handling the inherently difficult nature of making Hadoop an accessible to execute program, handling the Apache License 2.0 that is the license used by Hadoop for users of their software utilities and handling the ever-changing nature of using an open-source library.

7.1.2 Accessibility

Accessibility can be seen as a very broad topic, in the sense that accessibility can be applied to any individual or group of people or for certain systems running certain Hadoop configurations and Hadoop versions. In the context of this program, accessibility is specific to any operators of a Hadoop cluster, in which they would like to run the programs on. Making this program accessible is a high priority not only for others that are using the program in the future, but also for myself if I would like to modify or create new programs in the future that run with a dependency on these programs.

Accessibility for other cluster operators is an important issue with this program, as clusters can have many different configurations available, both in hardware and software. The hardware components that can be configured include using a single node and a multi node cluster, multiple nodes include using up to thousands of nodes, thankfully Hadoop handles these scenarios quite gracefully from a software standpoint, which means that there are not that many compromises that need to be made to account for individual Hadoop setups. Mitigating this issue was therefore quite simple thanks to the nature of Hadoop.

7.1.3 Licensing

As with using an external software library, even if it is open source, it is used subject to an agreed upon license when programs using the library are developed. The license used for Hadoop is the Apache License 2.0. This license allows me to use Hadoop for private use, along with distribution of programs created using the software libraries. The license has limitations which include but are not limited to Apache not being liable for what occurs when using Hadoop. The permissions that the license gives will be sufficient for the program, not requiring any compromises to develop the program.

Using java included agreeing to the Oracle Technology Network License Agreement for Oracle Java SE, this license is more limiting than the Apache License 2.0, however did not compromise the project either. However, if certain modifications were to be done to this project, such as using the programs for commercial use, there would be the addition of licensing fees for commercial use. It is important to note these conditions and potential future limitations, however in the scope of this project it does not require any compromises to fit within the license.

Finally, using python version 3 in this project means agreeing to the Python Software Foundation Licence. Thankfully the license, like the previously mentioned licenses, does not require any compromises to be made in the context of developing this project, as the license grants permission to use, copy, modify and distribute programs created from python, including the use of python’s included libraries. Making the risk of violating python’s license low, even with significant changes in future implementations of the python in this project.

For this program, I will keep the code private for personal use only. However, if the code was to be released, I would most likely license it as open source. With the only restrictions being the same imposed by the licenses I have agreed to when creating this program.

7.1.4 Future changes to Hadoop

Hadoop is an ever-changing library, with the programs running on Hadoop 3.3.6, one of the risks of using a program that continuously gets updated include ensuring that the code you make will be compatible with future and previous versions of Hadoop. One of the mitigating strategies for making code up to date include not using depreciated code, this is not exclusive for just Hadoop, but also for the java libraries that are used in this program. As this program also requires the use of external libraries such as importing a source of randomness for java. This means that there is an inherent dependency on both the version of Hadoop being used and the version of java being used.

This presents the idea of both backwards and forwards compatibility in programming. A feature of programs that can also be seen as a way for a program to be accessible in the sense that as many versions of software can work with a given program. Backwards compatibility in java thankfully is very simple to implement, with all JREs having the ability to run any of its previous versions through java's own backwards compatibility, meaning that dependencies in java is based on the lowest version on java that can run the program. This is a really good solution to mitigate any issues arising from mismatched versions.

Hadoop handles mismatched versions slightly differently, as the implementation will also differ compared to java. Hadoop is required to be installed on every node regardless of its role, be it a namenode, datanode etc. Hadoop provides backwards and forwards compatibility within minor releases of its libraries, with there being no compatibility between major releases, for example a Hadoop version 2 node cannot work with a Hadoop version 3 HDFS. This means there is an inevitable dependency on the specific Hadoop version, as opposed to java where it is simply having a newer java version than the one specified. This leads to the issue where clusters running Hadoop 2 cannot work with clusters running Hadoop 3 and vice versa, therefore requiring a declaration of what specific version the MapReduce programs will function on. Mitigating this can be done with providing a readme file in which the dependencies of Hadoop are listed, same as java, however more specific in the sense that backwards and forwards compatibility is limited, which is something that a Hadoop cluster operator will likely know, however it is good practice to specify what version(s) are compatible in the readme.

7.1.5 Conclusion

Overall, there are clear consequential factors outside of the programs I create which must be considered before, during, and after development. Not adapting the choices that I make based on these factors can lead to serious consequences outside of the programs themselves. Therefore, understanding and planning around these potential issues should be a high priority, as the potential level of impact from not mitigating or eliminating the risks can be more severe than risks present in the development of the programs themselves.

Chapter 8: **Reflection**

Overall, I believe this project has been mostly successful. This project has provided challenges in both software engineering and mathematics which has required a significant level of research to understand enough to create implementations of these concepts using Hadoop itself. I believe what I have learned from this project is vital for my development as a computer scientist, providing me with an invaluable skill of understanding not just Apache Hadoop as a software but concepts implemented in any form of massive scale data analytics implementation, both computational and mathematical.

This project does allow for the ability to expand on the programs created in the future. For example, expanding on original functionality of the program, say providing the ability to pick and choose types of initial centroid initialisation methods other than Forgy in the K-Means algorithm. This was one of my goals for this project, while not directly examined. Making programs that allowed for convenient future modification would ensure the code I made was of high quality.

Before starting this project, I had no experience using Apache Hadoop, no knowledge of MapReduce as an algorithm and no knowledge of K-Means as an algorithm for clustering data. My knowledge was limited to java, Python and the matplotlib library. This may raise questions as to why I picked this specific project. The answer being the interest I had in data analytics as a field, including its potential for simply ludicrous software and hardware implementations, using the most efficient implementation possible to find solutions to problems thought to be computationally infeasible.

If I were to begin this project again, I would most likely spend more time on research. This is primarily because when implementing the programs, sometimes I would be blocked by a pure lack of knowledge on what I could use from MapReduce libraries. Given the experience I have with Hadoop now, implementing the same project again would have some changes in its approach.

Bibliography

[1] Karloff, H., Suri, S. and Vassilvitskii, S., 2010, January. A model of computation for mapreduce. In Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms (pp. 938-948). Society for Industrial and Applied Mathematics.

[2] Wang, W. and Ying, L., 2014, September. Data locality in MapReduce: A network perspective. In 2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton) (pp. 1110-1117). IEEE.

[3] Borthakur, D., 2007. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 11(2007), p.21.

[4] Apache Software Foundation, 2010. Hadoop, Available at: https://hadoop.apache.org.

[5] Elgendy, N. and Elragal, A., 2014. Big data analytics: a literature review paper. In Advances in Data Mining. Applications and Theoretical Aspects: 14th Industrial Conference, ICDM 2014, St. Petersburg, Russia, July 16-20, 2014. Proceedings 14 (pp. 214-227). Springer International Publishing.

[6] Gustafsson, T. and Hansson, J., 2004, May. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004. (pp. 182-191). IEEE.

[7] Karun, A.K. and Chitharanjan, K., 2013, April. A review on hadoop—HDFS infrastructure extensions. In 2013 IEEE conference on information & communication technologies (pp. 132-137). IEEE.

[8] Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), pp.107-113.

[9] Pena, J.M., Lozano, J.A. and Larranaga, P., 1999. An empirical comparison of four initialization methods for the k-means algorithm. Pattern recognition letters, 20(10), pp.1027-1040.

[10] Siever, E., Weber, A., Figgins, S., Love, R. and Robbins, A., 2005. Linux in a Nutshell. " O'Reilly Media, Inc."

[11] Virtualbox, O.V., 2011. Oracle vm virtualbox. Change, 107, pp.1-287.

[12] Deprecated properties (2023) Apache Hadoop 3.3.6 – Deprecated Properties. Available at: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoopcommon/DeprecatedProperties.html

[13] Setting up a single node cluster. (2023) Hadoop. Available at: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoopcommon/SingleCluster.html (Accessed: 23 November 2023).

[14] Google Java Style Guide (2023) Google java style guide. Available at: https://google.github.io/styleguide/javaguide.html (Accessed: 28 November 2023).

[15] 2016/04/30 - apache mrunit has been retired. (2016) Apache MRUnit TM. Available at: https://mrunit.apache.org/ (Accessed: 28 November 2023).