High Performance Computational Analysis of The National Archives Energy Consumption

Author: Hawra Nawrozzadeh

Date: 25th April 2022

Table of Contents

1 Introduction

Since the introduction of mainframe computers in the 1950s, large amounts of data have been collected and stored in databases (Lee, 2017). With advances in technology and the development of the internet over the years, which plays a massive role in our modern daily lives, there has been a huge growth in data, and changes in the way data is accessed and analysed (Taylor-Sakyi, 2016; Lee, 2017) (*Figure 1*). Data are individual items that come in many forms such as numbers, words and symbols that do not carry any specific meaning (Vaughan, 2019). Although data may describe things about everything and anything, such as events, transactions, and entities, data by itself is not very informative. In order for data to be of any use, the information it contains must be processed and transformed into useable content (Vaughan, 2019).

Big Data are enormous structured and unstructured datasets, which are too complex for processing by traditional data mining techniques (Taylor-Sakyi, 2016; Lee, 2017). There are 4 types of rich data, also known as the 4 V's, which make up Big Data and hold the key to understanding how Big Data is measured. The Volume of data is the size of the dataset that is collected for data processing and analysis, and is measured in zettabytes, commonly due to large use of mobile devices and social networks. The Variety of data is the layout of the data generated either by humans or by machines and comes in numerous formats from many categories and sources such as e-mails, the web and sensors. The Velocity of data is the speed or the rate at which data is generated and transformed. Finally, the Veracity of data refers to the accuracy and the quality of the data. The quality of the data does not always guarantee it's accuracy, this depends more on the trustworthiness, source, type and processing of the data (Taylor-Sakyi, 2016; Lee, 2017).

With the advancement in technology at our fingertips, data is generated every single minute and growing in real time (Taylor-Sakyi, 2016; Lee, 2017). The figure below shows how data is generated by the internet every minute of the day (Lewis, 2021).

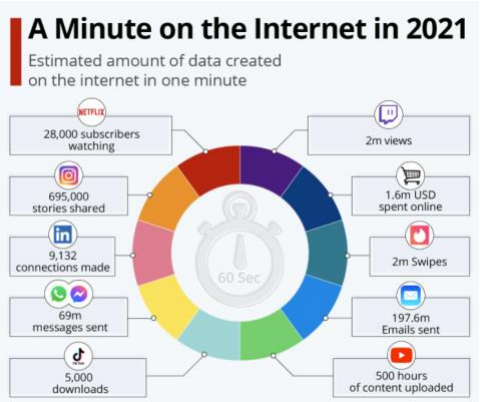

Figure 1: *Statistics showing the amount of data produced every minute by the internet in 2021, extracted from Statista (https://www.statista.com/) (Lewis, 2021).*

However, traditional data mining is no longer adequate to deal with such high volume of data, needed to extract useful and important information from the raw data, in order to make sense of them and make them usable (Lee, 2017). Additionally, with Big Data being generated every single minute, simple computational processing and storage units are not sufficient to handle the large amounts of data being generated by computer systems of large organisations and industries (Lee, 2017). Therefore, techniques such as High-Performance Computing (HPC) have been introduced and developed to address these issues (Taylor-Sakyi, 2016; Lee, 2017). HPC, also known as supercomputing, is a group of computers, called clusters, which consist of operating systems and processors with high storage and network capabilities (Hager and Wellein, 2010). The computer power is aggregated to deliver higher performance and solve large data problems, enabling time and cost-effective analysis of high volumes of data (Hager and Wellein, 2010). Many organisations such Amazon, IBM and Microsoft use the Apache Hadoop (https://hadoop.apache.org/) open-source framework (BDAN, 2022), one of the first HPC frameworks created and developed, to utilise their Big Data Storage system to distribute the task of solving large data problems throughout their system (Taylor-Sakyi, 2016; Lee, 2017; PHOENIXNAP, 2020).

The infrastructure of Apache Hadoop works by combining multiple computers to work in parallel simultaneously to store and process large volumes of data efficiently and quickly (PHOENIXNAP, 2020). The framework consists of 3 fundamental components, to help work on Big Data: Hadoop Distributed File System (HDFS), Yet Another Resource Negotiator (YARN), and MapReduce (PHOENIXNAP, 2020) *(see figure 4)*.

HDFS is a distributed file system used to store the data in order to capitalise the information (Shvachko *et al.*, 2010; PHOENIXNAP, 2020). It is an efficient, fault-tolerant designed storage system that makes multiple copies of the data and stores them across individual data blocks or nodes of multiple systems. As a result, even if one or several nodes fail, the rest of the system will not be affected (Shvachko *et al.*, 2010; PHOENIXNAP, 2020).

Once the data has been successfully stored, it needs to be processed (PHOENIXNAP, 2020). This is performed through MapReduce (Mapper and Reducer), which is a framework that enables several computers to perform parallel computing of the data. During the Mapper Phase, the raw data, which has been split into smaller components, is pre-processed and generated into intermediate key-value pairs. These pairs are then shuffled and sorted into groups and are then passed through the Reducer Phase to become aggregated and generate final key-value pairs *(see Figure 2)*. The figure below is an example of this process used to count the total number of a sequence of repeated words (Kat, 2016) *(see Figure 3)*.

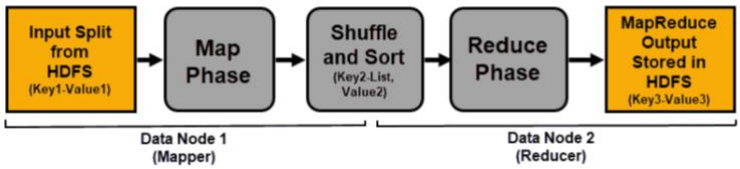

Figure 2: *The general workflow of MapReduce, where the Map Phase is responsible for pre-processing data and the Reduce Phase is responsible for performing aggregation function (PHOENIXNAP, 2020).*

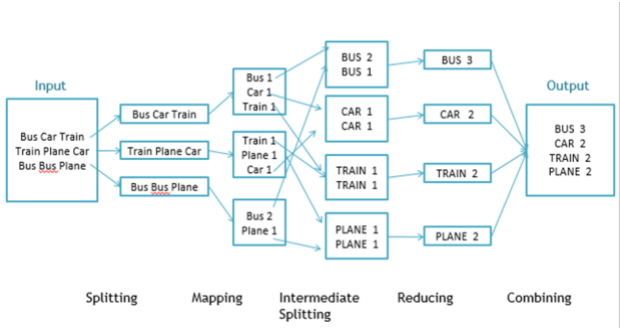

Figure 3: *MapReduce Chaining for a word count task (Kat, 2016).*

YARN is a cluster management source which monitors the Hadoop cluster system by processing job requests and resources. It is the backbone to ensuring the whole Hadoop system runs smoothly (PHOENIXNAP, 2020).

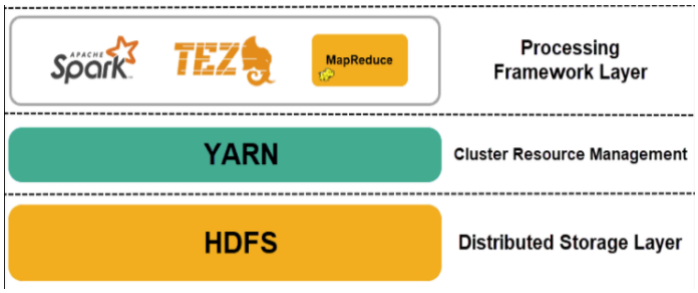

Figure 4: *The Architecture of Hadoop, extracted from (PHOENIXNAP, 2020). There are also other tools similar to MapReduce such as Apache Spark.*

Hadoop's HDFS ability to cope with large data and fast recovery from hardware failure makes it ideal for Batch Processing of large data (Shvachko *et al.*, 2010). This is particularly beneficial and cost-effective for large corporations and industries that need to deal with large real time data and keep historical records, such as payroll and billing systems, printing labels and any group of transactions collected over a period of time (Vaghasiya, 2018). Batch processing is used in almost every field and industry (Thakurdesai, 2016; Vaghasiya, 2018).

This report will provide an in-depth design and implementation of Apache Hadoop's Framework and will implement MapReduce jobs to answer the following research question: ***"Given the real-time dataset on electricity consumption of the National Archives' main building in Kew, has the electricity consumption per day, increased or decreased over the last decade?"***

The data used to answer this question has been extracted from the UK Government's Database (https://data.gov.uk/) and shows real-time data of the amount of electricity used in the Kew Building, the main UK Building site for the National Archive, in 24 hours of each day from 2010 to 2019 (https://data.gov.uk/dataset/da9a88d6-6535-4c7f-8d54-a93a50b2f177/the-national-archives-energy-consumption). The metadata of this dataset can be seen in the table below.

Name of Attribute	Description	Data Type
Location	Location of National Archive (All of the same location; Kew Building, the Main Building)	Character
Utility	The type of energy (all electricity)	Character
Units	Kilowatt per Hour (kWh)	Character
Date	Date/Month/Year (from January 2010 to November 2019)	Character
Time (00:00-23:30)	Energy consumption used at every half an hour time interval (for each day)	Character
Totals	Total energy consumption (electricity) used for each day	Numerical (Double)

Following a quick data clean by mainly removing all empty rows and headers, this dataset contained 3405 observations and 53 attributes.

In order to answer the research question, two areas need to be addressed and analysed through batch processing:

1) What is the total average electricity consumed in the building over the period?
2) Does the electricity consumption vary between seasonal summer and winter months?

2 Design and Implementation

VirtualBox Machine software (https://www.virtualbox.org/) was used to perform the Hadoop's cluster environment for the MapReduce jobs. Java Programming Language was also used, as it is the main language that MapReduce utilise to manipulate and perform their jobs (Verma and Pandey, 2016).

In order to answer the first part, '***What is the total average electricity consumption over the period?'***, a MapReduce Chaining technique was performed. MapReduce Chaining is a very useful way of creating a sequence of MapReduce jobs. This process involves chaining two MapReduce jobs together, meaning the output of the first Mapper and Reducer, is further processed by the next chained Mapper and Reducer to give the final output (Nguyen, 2020). In this case, the first Mapper and Reducer would find and produce an output of the total sum of the electricity consumption for each year. This output is then used for the second Mapper and Reducer to find the average electricity consumption of all the years. *Figure 5* illustrates the workflow design for this MapReduce Chaining, showing the input and output data types for each mapper and reducer.

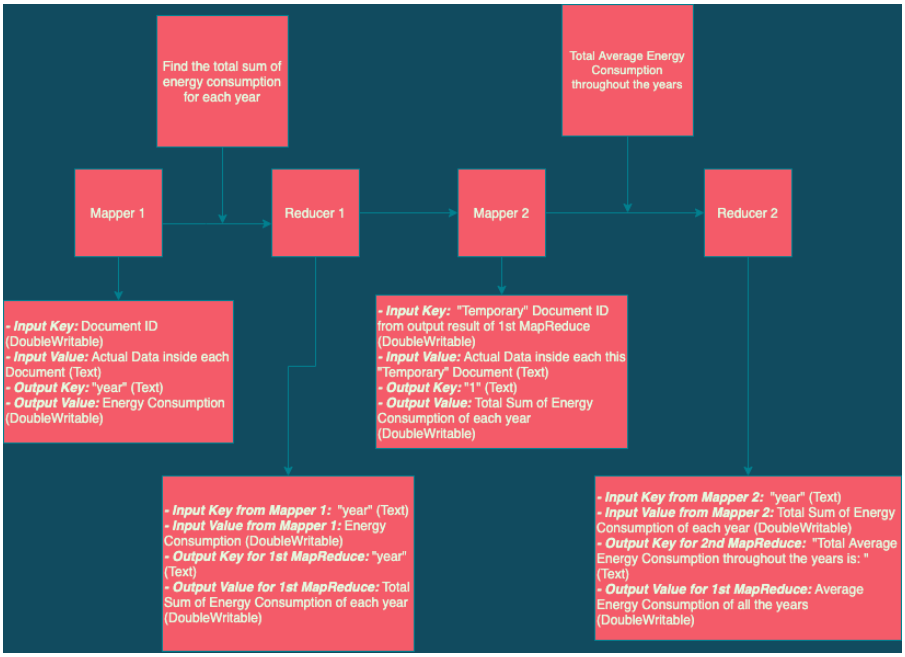

Figure 5: The workflow design and plan out to perform the MapReduce chaining job to answer '***What is the total average electricity consumption over the period?'*** *(In the plan, energy refers to electricity). Created using Diagrams.net (https://drawio-app.com)*

Using Eclipse, an Integrated Development Environment (IDE) software for Java (https://www.eclipse.org/downloads/), within the Virtual Machine, four Java classes were produced to perform this MapReduce Chaining job: the 'TotalAverageEnergyConsumption.java', the 'Mapper1.java' and 'Reducer1.java', 'Mapper2.java' and 'Reducer2.java'.

The ‘TotalAverageEnergyConsumption.java’ is the main driver class that calls the Mapper and Reducer classes and ensures that the Hadoop’s job runs smoothly on the Hadoop cluster (Verma and Pandey, 2016; Nguyen, 2020). As this MapReduce Chaining involves two MapReduce jobs, this driver class will ensure that both jobs are set to their Mapper1 and Reducer1 and Mapper2 and Reducer2 java classes, respectively. This class also ensures the input and output file paths are set accordingly when running the Hadoop’s cluster environment for these two chained jobs (Verma and Pandey, 2016; Nguyen, 2020). The input file path for the first MapReduce is the file path that contains data file (‘EnergyConsumptionModified1.csv’), and the output file path is the ‘temporary’ output path, where the first part of the results, being the total sum of electricity consumption for each year, is written and stored into the ‘temporary’ file (Nguyen, 2020). This ‘temporary’ file path becomes the input file path for the second MapReduce job. The output file path will be the final output result of the whole MapReduce Chaining, where it will contain a value of the total average electricity consumption throughout the years. This driver class also ensures that the configuration parameters are accessible and run smoothly when performing the MapReduce Chaining job within the Hadoop cluster environment (Nguyen, 2020) *(see Figure 6)*.

```java
package org.myorg;

//Importing the required packages or libraries for this driver; it includes classes and methods
import org.apache.hadoop.conf.Configuration;

public class TotalAverageEnergyConsumption{

    public static void main(String[] args) throws Exception
    {
        /* 1st MapReduce Job */
        // creating a new object from the "Configuration" class
        // this "Configuration" allows this driver class to access the configuration parameters needed to perform the Hadoop's MapReduce Chaining Jobs
        Configuration conf1 = new Configuration();

        // Create an conditional 'if' statement to check and make sure the number of arguments is not greater than 3
        // the arguments are the Driver Class (jar file), followed by the input and output path when running the MapReduce Job within the cluster environment
        if (args.length != 3)
        {
            System.err.println("Usage: TotalAverageEnergyConsumption <input path> <output path>");
            // If the argument is greater than 3, than need to exist and stop Hadoop's cluster system from further performing the first MapReduce Job
            // of the MapReduce Chaining
            System.exit(-1);
        }

        // Creating an object from the "Job" class
        // This object is created to be used and represent the Hadoop's MapReduce first Job that will be executed (to the second MapReduce Job)
        // This is an important stage as it ensures that this MapReduce Job will be executed, allowing users make adjustments or changes if needed
        Job job1;
        job1=Job.getInstance(conf1, "Total Sum Energy Consumption For each Year");
        // Directing and allocating the Jar file to be the driver class
        job1.setJarByClass(TotalAverageEnergyConsumption.class);

        //Calling and Setting up the Mapper1 and Reducer1 class (of the first MapReduce Job)
        job1.setMapperClass(Mapper1.class);
        job1.setReducerClass(Reducer1.class);

        // Setting the Key and Value classes for the output from this Job
        // The key is Text as it is String of each year
        job1.setOutputKeyClass(Text.class);
        // The output is DoubleWritable as it is the energy consumption (the sum for each year)
        job1.setOutputValueClass(DoubleWritable.class);

        // Setting up the input and output file path for the Hadoop's first MapReduce Job
        // remember that 1st argument is the driver class, that why input and output path is index with 1 and 2, respectively
        // However, the output path needs to be passed to the second MapReduce Job, therefore will add an file path called "temp", short for temporarily file path
        FileInputFormat.addInputPath(job1, new Path(args[1]));
        FileOutputFormat.setOutputPath(job1, new Path(args[1]+"/temp"));

        // Delete output if exists
        // This code below ensures that output folder (for the temporarily file) within HDFS does not exist,
        // and if it does exist, then it will be deleted in the HDFS system.
        // Either way, this functions ensures that new output folder (for the temporarily file path) is created within HDFS.
        FileSystem hdfs = FileSystem.get(conf1);
        Path outputDir = new Path(args[1]+"/temp");
        if (hdfs.exists(outputDir))
            hdfs.delete(outputDir, true);
        // Completion of the 1st MapReduce becomes true when it finished and then we move onto the next chained MapReduce job
        job1.waitForCompletion(true);

        /* 2nd MapReduce Job */
        //Again, creating another new object from the "Configuration" class for configuration of this last part of the Hadoop's job
        Configuration conf2 = new Configuration();

        // Again, creating an object from the "Job" class
        // This object is created to be used and represent the Hadoop's MapReduce second (and last) Job that will be executed (from the first MapReduce Job)
        Job job2;
        job2=Job.getInstance(conf2, "Total Average Energy Consumption");
        // Directing and allocating the Jar file to be the driver class
        job2.setJarByClass(TotalEnergyConsumption.class);


        //Calling and Setting up the Mapper1 and Reducer1 class (of the second MapReduce Job)
        job2.setMapperClass(Mapper2.class);
        job2.setReducerClass(Reducer2.class);

        // Setting the Key and Value classes for the output from this Job
        // The key is Text as it is Strings of "1"
        job2.setOutputKeyClass(Text.class);
        // The output is DoubleWritable as it is the energy consumption (the total average of all the years)
        job2.setOutputValueClass(DoubleWritable.class);

        // the input path is the output path and temporarily file from the first MapReduce Job (hence the index being 1 plus temp file)
        // The output file is final output from this MapReduce Chaining, hence end with the last index (2)
        FileInputFormat.addInputPath(job2, new Path(args[1]+"/temp"));
        FileOutputFormat.setOutputPath(job2, new Path(args[2]));

        // repeat the same process to check if there is a existing output file within HDFS
        hdfs = FileSystem.get(conf2);
        outputDir = new Path(args[2]);
        if (hdfs.exists(outputDir))
            hdfs.delete(outputDir, true);

        // Completion of the 2nd MapReduce becomes true when it finished and therefore the MapReduce Chaining is completed successfully.
        System.exit(job2.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 6: *The Driver Class for the MapReduce Chaining*

For Hadoop's cluster to run the MapReduce jobs, the key-value pairs that are passed in and out of the Mapper and Reducer classes need to be in the form of Hadoop's Object data type. *Table 2* show the Hadoop's data types for key-value pairs, in equivalence with Java's Data types (Siva, 2014).

Hadoop's Data Type	Java Data Type
IntWritable	Integer
FloatWritable	Float
LongWritable	Long
ShortWritable	Short
DoubleWritable	Double
Text	String

In the first MapReduce job of the MapReduce Chaining, the first Mapper class ('Mapper1.java') involves extracting and pre-processing the important information from the dataset. This class initially takes in the input key and value pairs, the document ID (from the input path from the driver class), and the actual data inside each passing document ID, respectively. The Hadoop's data type of this input key-value pair is LongWritable and Text respectively *(Figure 5)*. This class then extracts the "years" values from the date attribute and sets this as the output key for this first Mapper class. This output key is Text as the "years" extracted are a String data type, for example, "2011", "2012" and so forth. This class will also extract the electricty consumption values from the totals attribute and will be set as the output value for this Mapper class, being DoubleWritable, as it consists of numerical data type values *(Figure 5)*. The output key-value pair is passed to the first Reducer (Reducer1.java) *(see figure 7 to see the code)*.

Mapper1.java ×

```
package org.myorg;

//Importing the required packages or libraries for the first Mapper class; it includes classes and methods
import java.io.IOException;

//Using 'extend' to create a subclass called "Mapper1" from the Mapper parent class
//Passing 4 parameters within this class
//1st and 2nd Parameter is the Input key and value datatype for the first Mapper class.
    // The input key is LongWritable as it is the document ID
    // The input value is Text as it is the actual data inside each passing document ID
//3rd and 4th Parameter is output key and value datatypes from the first Mapper class
    // The output key is Text as this will be "year"
    // The output value is DoubleWritable as this the energy consumption value
public class Mapper1 extends Mapper <LongWritable, Text, Text, DoubleWritable>
{
    //Text is equivalent to String in Java --> Use Hadoop's Object, Text, in order to use the Hadoop's objects and run the Mapper accordingly
    //Creating new Text Object called Year, which will be passed as the output Key of this Mapper class
    private Text Year = new Text();
    // Same concept here; using Hadoop's Object, DoubleWritable, which is equivalent to Java's Object, Double
    // Creating new DoubleWritbale Object called totalEnergy, which will be passed as the output Value of this Mapper Class
    private static DoubleWritable totalEnegry = new DoubleWritable(0);

    // Overriding the map's original behaviour within the parent class
    @Override
    // Passing 3 parameters into this new map method
        // key, value, Context in order to execute and emit the context to the Reducer class
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException
    {
        // "line" String array is created
        // where it separates the values in the attributes within the dataset by ","
        // Each values is stored in 1 array element
        String[] line = value.toString().split(",");

        // Create a String called "year" to extract the year "values" from the forth column (Data Column); the 3 element of line
        // Using substring extract the year "values" with index starting from 6
        String year = line[3].substring(6);

        // Setting this new object, "year" to the Key Text that will be passed and emit from the context output of this first mapper to the first reducer
        Year.set(year);

        // Will do the same for total energy
        // Create a "total" double variable that extracts the total energy values from the 53th column (52th from the line object) and parse is as Double
        double total = Double.parseDouble(line[52].trim());

        // Setting this new total variable to the Value DoubleWritable that will be passes
        // and emit from the context output of this first mapper to the first reducer
        totalEnegry.set(total);

        // Lets now emit the Key and Value and Pass them to the first Reducer class
        context.write(Year, totalEnegry);
    }
}
```

Figure 7: *The Mapper1.java of the first MapReduce job for the MapReduce Chaining*

The ‘Reducer1.java’ class takes in the output results from the Mapper1.java class as the Reducer’s input key-value pair *(Figure 5)*. This Reducer takes in all the content from their input key-value pair and performs aggregation and group-by function to extract the total sum of electricity consumption for each year. Therefore, the output key of this Reducer will be all of the duplicated years, aggregated and reduced to each year with their respective total sum of electricity consumption, which will be the Reducer’s output value *(see figure 8)*. This temporary result of the first MapReduce job, consisting of two attributes, each year and the total sum of electricity consumption, is chained to the next MapReduce to be further processed and aggregated *(Figure 5)*.

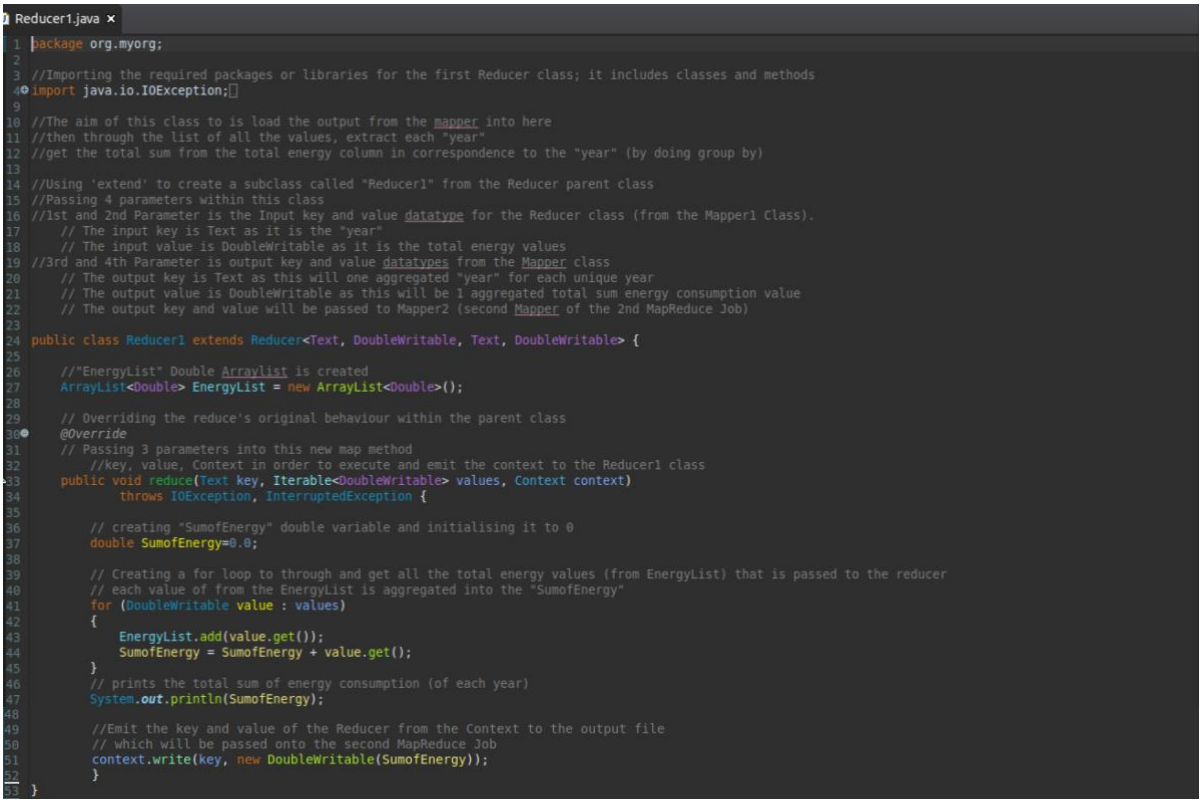

Figure 8: *The Reducer1.java of the first MapReduce job for the MapReduce Chaining*

In the second MapReduce job of the MapReduce Chaining, the whole process is repeated, with the difference being that the document ID is now the temporary file used. Moreover, the output key-value pair of the second Mapper class is a repeated string of “1” and total sum of electricity consumption (for each year), respectively *(Figure 5)*. This output is then passed to the second Reducer class, to be further aggregated to formulate the final output result *(see Figure 9 and 10)*. *Figure 11* shows a small abstract of this MapReduce Chaining.

*Mapper2.java ×

```java
package org.myorg;

//Importing the required packages or libraries for the second Mapper class; it includes classes and methods
import java.io.IOException;

//Using 'extend' to create a subclass called "Mapper1" from the Mapper parent class
//Passing 4 parameters within this class
//1st and 2nd Parameter is the Input key and value datatype for the first Mapper class.
    // The input key is LongWritable as it is the document ID (of the temporarily output file from 1st MapReduce Job)
    // The input value is Text as it is the actual data inside each passing document ID
//3rd and 4th Parameter is output key and value datatypes from the first Mapper class
    // The output key is Text as this will be "1"
    // The output value is DoubleWritable as this the energy consumption value
public class Mapper2 extends Mapper <LongWritable, Text, Text, DoubleWritable>
{
    //Text is equivalent to String in Java --> Use Hadoop's Object, Text, in order to use the Hadoop's objects and run the Mapper accordingly
    //Creating new Text Object called Year, which will be passed as the output Key of this Mapper class
    private Text Ones = new Text();
    private static DoubleWritable totalEnegry = new DoubleWritable(0);

    // Passing 3 parameters into this new map method
    // key, value, Context in order to execute and emit the context to the Reducer class
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException
    {
        // "line" String array is created
        // where it separates the values in the attributes within temporarily file's data,
        // separated by tab, as this structure format from the output temporarily file from the 1st MapReduce Job
        // Each values is stored in 1 array element
        String[] line = value.toString().split("\\t");

        // Creating String object to "1"
        // as there is no unique keys, all the keys should be same in order for the 2nd reducer can only aggregate the values
        String ones = "1";
        // Setting this new object, "ones" to the Key Text that will be passed and emit from the context output of this second mapper to the second reducer
        Ones.set(ones);

        // Will do the same for total energy
        // Create a "total" double variable that extracts the total sum energy values from the 2nd column (2nd from the line object) and parse is as Double
        double total = Double.parseDouble(line[1]);

        // Setting this new total variable to the Value DoubleWritable that will be passes
        // and emit from the context output of this second mapper to the second reducer
        totalEnegry.set(total);

        // Lets now emit the Key and Value and Pass them to the second Reducer class
        context.write(Ones, totalEnegry);
    }
}
```

Figure 9: *The Mapper2.java of the second MapReduce job for the MapReduce Chaining*

*Reducer2.java ×

```java
package org.myorg;

//Importing the required packages or libraries for the second Reducer class; it includes classes and methods
import java.io.IOException;

//The aim of this class to is load the output from the second mapper into here
//then through the list of all the values, extract each "year"
//get the total sum from the total energy column in correspondence to the "year" (by doing group by)
// Then go through the list of all the values, extracting all "1"s  and
// aggregate and get total average of energy consumption of all the years (from the total sum of energy consumption, the second column)

//Using 'extend' to create a subclass called "Reducer2" from the Reducer parent class
//Passing 4 parameters within this class
//1st and 2nd Parameter is the Input key and value datatype for the Reducer class (from the Mapper2 Class).
    // The input key is Text as it is the "ones"
    // The input value is DoubleWritable as it is the total sum energy consumption values
//3rd and 4th Parameter is output key and value datatypes from the Mapper class
    // The output key is Text as this will be a message
    // The output value is DoubleWritable as this will be one aggregated value showing the total average of energy consumption value (of all the years)
    // The output key and value will be the final output of this MapReduce Chaining
public class Reducer2 extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

    //Creating new Text Object called message, which will be passed as the output Key of this Reducer2 class
    private Text message = new Text();
    //"EnergyList" Double Arraylist is created
    ArrayList<Double> EnergyList = new ArrayList<Double>();

    // Overriding the reduce's original behaviour within the parent class
    @Override
    // Passing 3 parameters into this new map method
        //key, value, Context in order to execute and emit the context to the Reducer2 class
    public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
            throws IOException, InterruptedException
    {
        // creating "SumofEnergy" double variable and initialising it to 0
        double SumofTotalEnergy=0.0;

        // Creating a for loop to through and get all the total energy values (from EnergyList) that is passed to the reducer
        // each value of from the EnergyList is aggregated into the "SumofTotalEnergy"
        for (DoubleWritable value : values)
        {
            EnergyList.add(value.get());
            SumofTotalEnergy = SumofTotalEnergy + value.get();

        }

        // get the size number of energy from the arraylist
        int size = EnergyList.size();

        //the the average of total sum of energy is calculated (for all the values)
        double mean = SumofTotalEnergy/size;

        // Creating and setting a text message for the Key output
        String txt = "The total average electricity consumption over the period is: ";
        message.set(txt);


        //Emit the key and value of the Reducer2 from the Context to the output file
        context.write(message, new DoubleWritable(mean));
    }
}
```

Figure 10: *The Reducer2.java of the second MapReduce job for the MapReduce Chaining*

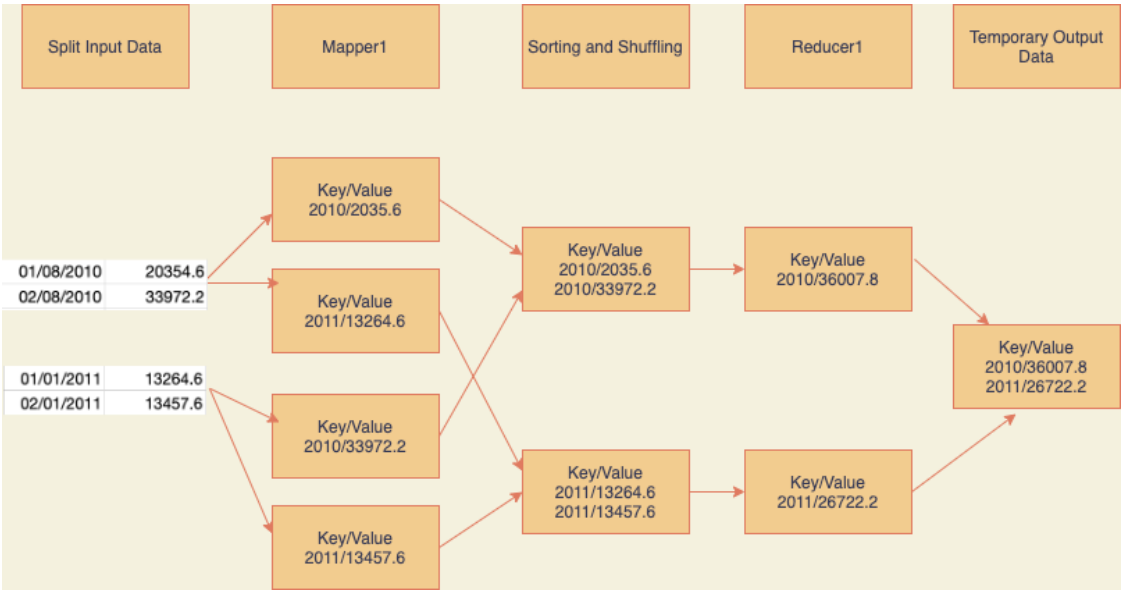

Figure 11: *A small abstract of the MapReduce Chaining, showing the first two observations for 2010 and 2011. Created using Diagrams.net (https://drawio-app.com).*

To run this MapReduce Chaining job within the Hadoop cluster and obtain the final results *(see figure 12)*, the following command lines were executed on the terminal within the VirtualBox Machine (Verma and Pandey, 2016):

1) The following command was used to remove the temporary data from the Hadoop directory

```
(base) hadoop@hadoop-VirtualBox:~$ rm -rf /tmp/hadoop-hadoop/*
```

2) Then the HDFS file system was formatted

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs namenode -format
```

3) The HDFS and YARN was initiated

```
(base) hadoop@hadoop-VirtualBox:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [hadoop-VirtualBox]
(base) hadoop@hadoop-VirtualBox:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
```

4) JPS command was used to check that all the elements of YARN and HDFS were present

```
(base) hadoop@hadoop-VirtualBox:~$ jps
1937 org.eclipse.equinox.launcher_1.5.0.v20180512-1130.jar
3397 DataNode
3958 NodeManager
3255 NameNode
3627 SecondaryNameNode
3804 ResourceManager
4287 Jps
```

5) A new input directory was created within the HDFS directories

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -mkdir /input
```

6) The input data file was copied into this new input directory of the HDFS

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -put Downloads/EnergyConsumptionModified1.csv /input
```

7) The following command was used to execute the MapReduce Chaining job on the Hadoop

```
(base) hadoop@hadoop-VirtualBox:~$ hadoop jar Downloads/TotalAverageEnergyConsumption.jar TotalAverageEnergyConsumption /input/ output
```

8) The output files of the result were then copied from HDFS to the local system to be accessed

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -get output output
```

```
1 The total average electricity consumption over the period is:        6246353.16
```

Figure 12: *The output result from the MapReduce Chaining, answering the first part of the question, 'What is the total average electricity consumption over the period?'*

In order to answer the second part, '***Does the electricity consumption vary between seasonal summer and winter months?'***, a single MapReduce job was performed (Verma and Pandey, 2016). This involved producing 3 java classes: 'SeasonalMonths.java', the driver class, 'SeasonalMonthMapper.java' and 'SeasonalReducer.java', the Mapper and Reducer class, respectively. The workflow for this job is illustrated in figure 13, again showing data types for the different passing key-values pairs for the classes (Verma and Pandey, 2016).

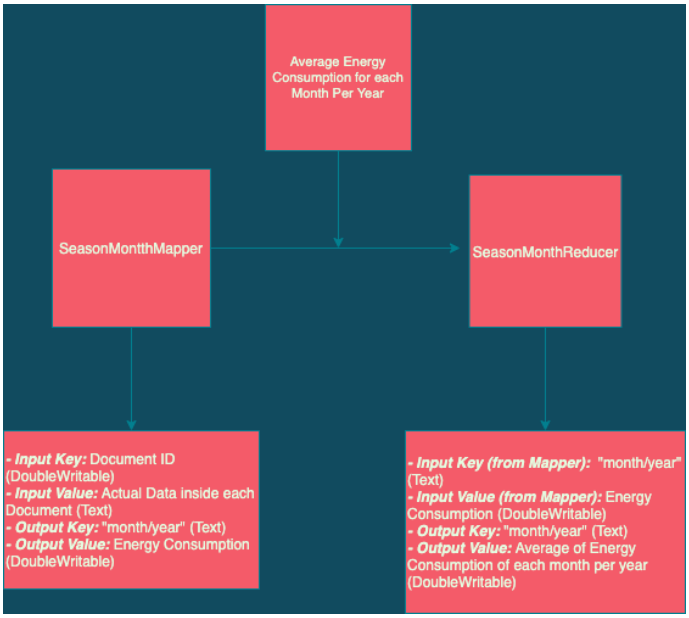

Figure 13: The workflow design and plan to perform the MapReduce job to answer '***Does the electricity consumption vary between seasonal summer and winter months?'*** *(In the plan, energy refers to electricity). Created using Diagrams.net (https://drawio-app.com)*

Within the driver class, a similar process to the driver class from MapReduce Chaining job occurs. However, this driver class calls one Mapper and Reducer class, ensuring that the Hadoop's job runs smoothly on the

Hadoop cluster for this single MapReduce job (Verma and Pandey, 2016). Again, it sets the Mapper and Reducer to their respective java classes and also ensures the input and output file paths are set accordingly when running the Hadoop's cluster environment for this job. The input file path contains the data file ('EnergyConsumptionModified1.csv') and the output file path contains the final output results for this job. Similarly, this driver class also ensures that the configuration parameters are accessible and run smoothly when performing this job within the Hadoop's cluster environment (Verma and Pandey, 2016) *(see Figure 14)*.

SeasonalMonths.java

```java
package org.myorg;

//Importing the required packages or libraries for this driver; it includes classes and methods
import org.apache.hadoop.conf.Configuration;

public class SeasonalMonths

{
    public static void main(String[] args) throws Exception
    {
        // creating a new object from the "Configuration" class
        // this "Configuration" allows this driver class to access the configuration parameters needed to perform the Hadoop's MapReduce Job
        Configuration conf = new Configuration();

        // Create an conditional 'if' statement to check and make sure the number of arguments is not greater than 3
        // the arguments are the Driver Class (jar file), followed by the input and output path when running the MapReduce Job within the cluster environment
        if (args.length != 3)
        {
            System.err.println("Usage: SeasonalMonths <input path> <output path>");
            // If the argument is greater than 3, than need to exist and stop Hadoop's cluster system from further performing the MapReduce Job
            System.exit(-1);
        }

        // Creating an object from the "Job" class
        // This object is created to be used and represent the Hadoop's Job that will be executed
        // This is an important stage as it ensures that MapReduce Job will be executed, allowing users make adjustments or changes if needed
        Job job;
        job=Job.getInstance(conf, "Average of Each Seasonal Month");
        // Directing and allocating the Jar file to be the driver class
        job.setJarByClass(SeasonalMonths.class);

        // Setting up the input and output file path for the Hadoop's Job
        // remember that 1st argument is the driver class, that why input and output path is index with 1 and 2, respectively
        FileInputFormat.addInputPath(job, new Path(args[1]));
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

        //Calling and Setting up the Mapper and Reducer class
        job.setMapperClass(SeasonalMonthsMapper.class);
        job.setReducerClass(SeasonalMonthsReducer.class);

        // Setting the Key and Value classes for the output from this Job
        // The key is Text as it is String of month per year ("month/year")
        job.setOutputKeyClass(Text.class);
        // The output is DoubleWritable as it is the energy consumption
        job.setOutputValueClass(DoubleWritable.class);

        // Delete output if exists
        // This code below ensures that output folder within HDFS does not exist, and if it does exist, then it will be deleted in the HDFS system.
        // Either way, this functions ensures that new output folder is created within HDFS.
        FileSystem hdfs = FileSystem.get(conf);
        Path outputDir = new Path(args[2]);
        if (hdfs.exists(outputDir))
            hdfs.delete(outputDir, true);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 14: *The Driver class for the single MapReduce job.*

In the 'SeasonalMonthMapper.java', similar processes to the 'Mapper1.java' from the previous job, occur. The only difference between the two classes is that the output key of this Mapper is each month per year ("month/year") *(see Figure 15)*. The output key-value pair is passed to the 'SeasonalReducer.java', where this Reducer performs aggregation and group-by function to produce the final output file of the average electricity consumption of each month per year *(see Figure 16)*.

SeasonalMonthsMapper.java

```java
package org.myorg;

// Importing the required packages or libraries for the Mapper class; it includes classes and methods
import java.io.IOException;

// Using 'extend' to create a subclass called "SeasonalMonthsMapper" from the Mapper parent class
// Passing 4 parameters within this class
//1st and 2nd Parameter is the Input key and value datatype for the Mapper class.
    // The input key is LongWritable as it is the document ID
    // The input value is Text as it is the actual data inside each passing document ID
//3rd and 4th Parameter is output key and value datatypes from the Mapper class
    // The output key is Text as this will be "month/year"
    // The output value is DoubleWritable as this the energy consumption value
public class SeasonalMonthsMapper extends Mapper<LongWritable, Text, Text, DoubleWritable>
{
    //Text is equivalent to String in Java --> Use Hadoop's Object, Text, in order to use the Hadoop's objects and run the Mapper accordingly
    //Creating new Text Object called MonthYear, which will be passed as the output Key of this Mapper class
    private Text MonthYear = new Text();
    // Same concept here; using Hadoop's Object, DoubleWritable, which is equivalent to Java's Object, Double
    // Creating new DoubleWritbale Object called totalEnergy, which will be passed as the output Value of this Mapper Class
    private static DoubleWritable totalEnegry = new DoubleWritable(0);

    // Overriding the map's original behaviour within the parent class
    @Override
    // Passing 3 parameters into this new map method
        //key, value, Context in order to execute and emit the context to the Reducer class
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException
    {

        // "line" String array is created
        // where it separates the values in the attributes within the dataset by ","
        // Each values is stored in 1 array element
        String[] line = value.toString().split(",");

        // Create a String called "monthyear" to extract the month and year "values" from the forth column (Data Column); the 3 element of line
        // Using substring extract the month and year "values" with index starting from 3
        String monthyear = line[3].substring(3);

        // Setting this new object, "monthyear" to the Key Text that will be passed and emit from the context output of this mapper to the reducer
        MonthYear.set(monthyear);

        // Will do the same for total energy
        // Create a "total" double variable that extracts the total energy values from the 53th column (52th from the line object) and parse is as Double
        double total = Double.parseDouble(line[52].trim());

        // Setting this new total variable to the Value DoubleWritable that will be passes and emit from the context output of this mapper to the reducer
        totalEnegry.set(total);

        // setting the Key and Values according to the Hadoop's object data type as the Java data type is too big for Hadoop's functions and methods

        // Lets now emit the Key and Value and Pass them to the Reducer class
        context.write(MonthYear, totalEnegry);
    }
}
```

Figure 15: *The Mapper class for the single MapReduce job.*

SeasonalMonthsReducer.java

```java
package org.myorg;

//Importing the required packages or libraries for the first Reducer class; it includes classes and methods
import java.io.IOException;

//The aim of this class to is load the output from the mapper into here
// then through the list of all the values, extract the "month/year"
// get the average from the total column in correspondence to the "month/year" (by doing group by)

//Using 'extend' to create a subclass called "SeasonalMonthsReducer" from the Reducer parent class
//Passing 4 parameters within this class
//1st and 2nd Parameter is the Input key and value datatype for the Reducer class (from the Mapper Class).
    // The input key is Text as it is the "month/year"
    // The input value is DoubleWritable as it is the total energy values
//3rd and 4th Parameter is output key and value datatypes from the Mapper class
    // The output key is Text as this will 1 aggregated "month/year"
    // The output value is DoubleWritable as this will be 1 aggregated average energy consumption value
public class SeasonalMonthsReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

    //"EnergyList" Double Arraylist is created
    ArrayList<Double> EnergyList = new ArrayList<Double>();

    // Overriding the reduce's original behaviour within the parent class
    @Override
    // Passing 3 parameters into this new map method
        //key, value, Context in order to execute and emit the context to the Reducer class
    public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
            throws IOException, InterruptedException {

        // creating "SumofEnergy" double variable and initialising it to 0
        double SumofEnergy=0.0;

        // Creating a for loop to through and get all the total energy values (from EnergyList) that is passed to the reducer
        // each value of from the EnergyList is aggregated into the "SumofEnergy"
        for (DoubleWritable value : values)
        {
            EnergyList.add(value.get());
            SumofEnergy = SumofEnergy + value.get();
        }

        // get the size number of energy from the arraylist
        int size = EnergyList.size();

        //the the average energy energy is calculated (for each unique "month/year")
        double mean = SumofEnergy/size;

        //Emit the key and value of the Reducer1 from the Context to the output file
        context.write(key, new DoubleWritable(mean));
        }
}
```

Figure 16: *The Reducer class for the single MapReduce job.*

The same command lines were executed on the terminal to run this job within the Hadoop cluster to obtain the final results (Verma and Pandey, 2016) (table shown in the appendix), which has been visualised in *Figure 18 and 19. Figure 17* shows a small abstract of this single MapReduce job (Verma and Pandey, 2016).

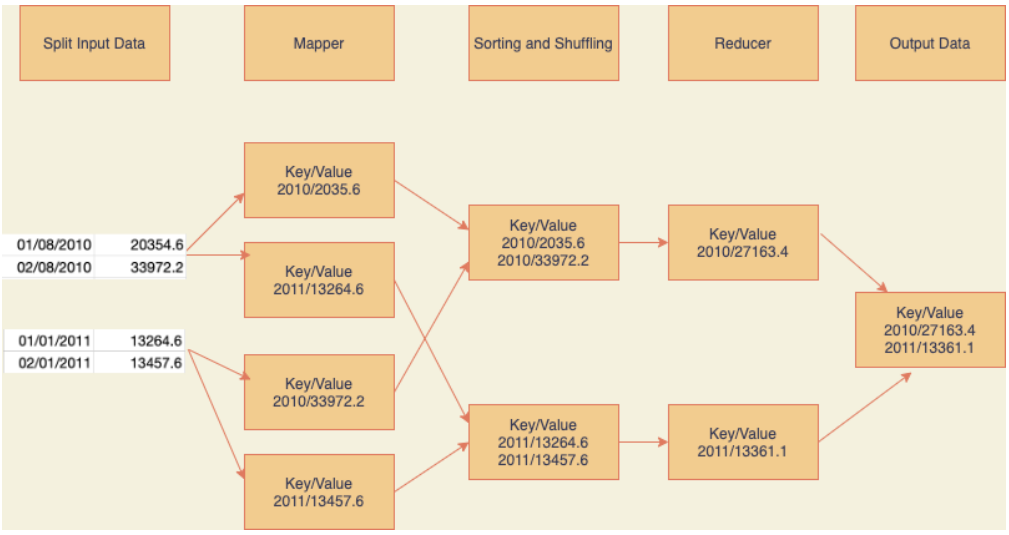

Figure 17: *A small abstract of the single MapReduce, showing the first two observations for 2010 and 2011. Created using Diagrams.net (https://drawio-app.com).*

3 Results and Evaluations

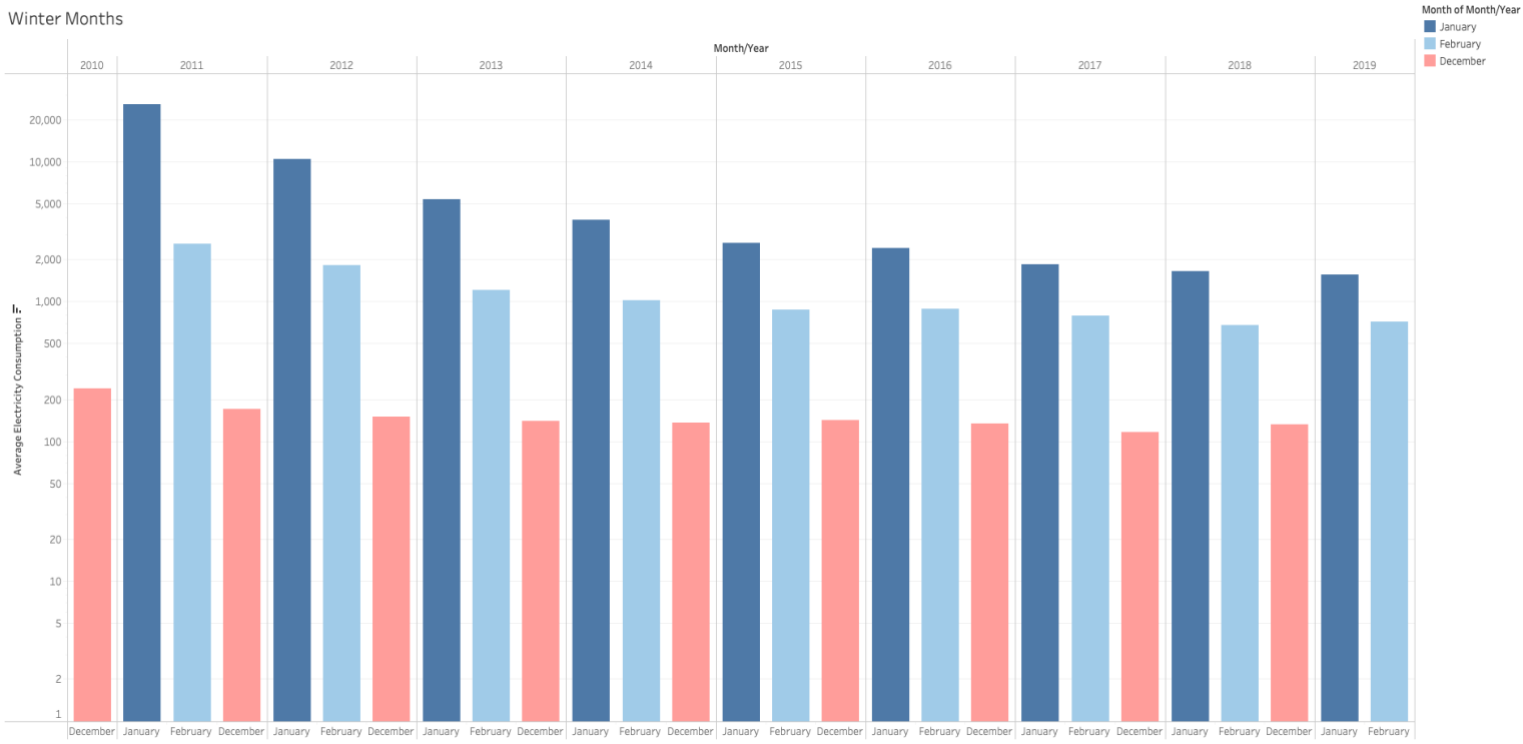

Figure 18: *The average electricity consumption over the winter seasons. This figure was created using Tableau (https://www.tableau.com/en-gb).*

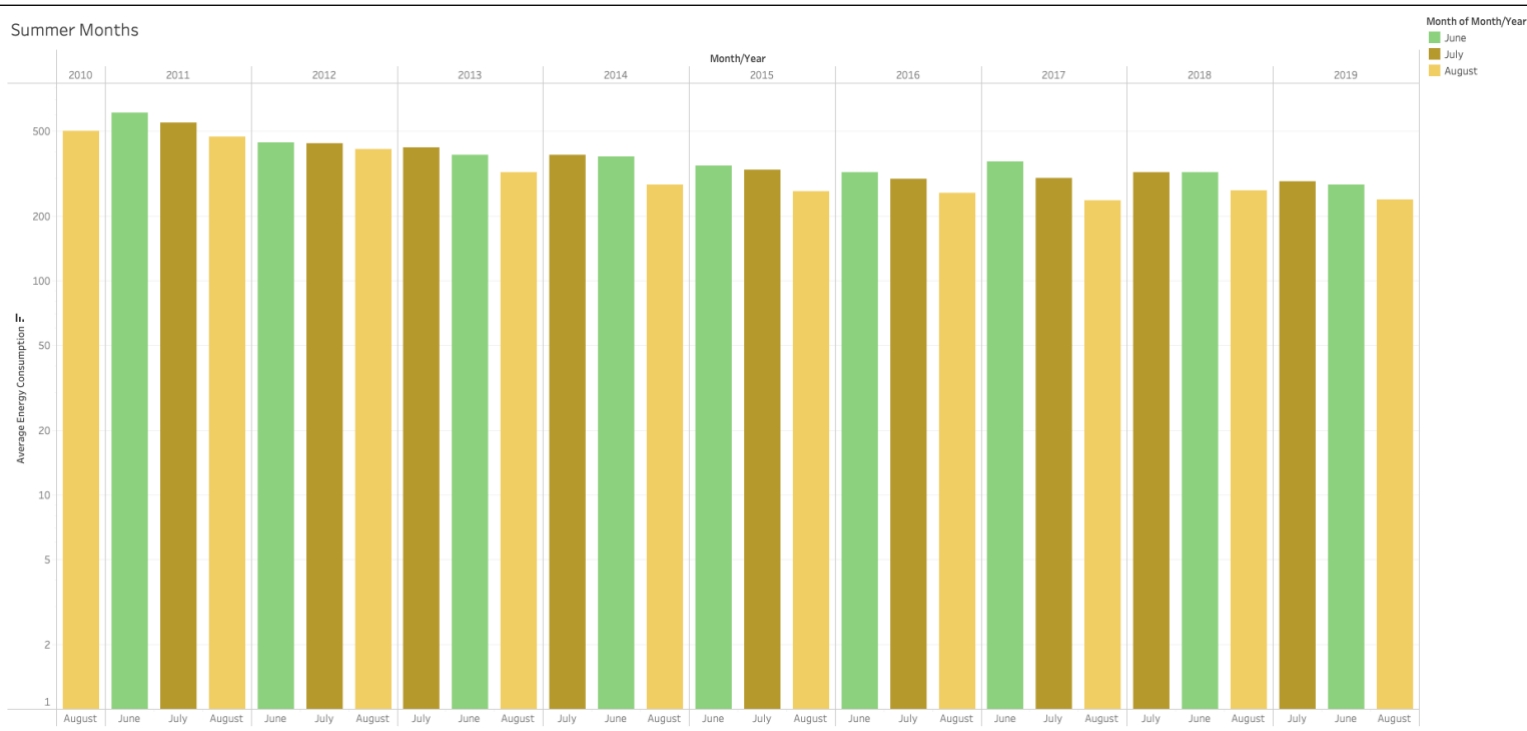

Figure 19: *The average electricity consumption over the summer seasons. This figure was created using Tableau (https://www.tableau.com/en-gb).*

The results show that, during the winter season, the UK National Archive building has the most electricity consumption during January and the least consumption in December. In January 2011, the average consumption was almost 26,000 kWh per day as opposed to an average of under 5,000 kWh per day in December of the same year. During the summer season, the electricity consumption varies each year. For instance, in 2011, the average consumption was highest in June, with a usage of just over 600 kWh per day, whereas, in 2013 the average consumption was highest in July, with a usage of just 420 kWh per day. Additionally, the figures show that considerably more electricity was consumed in the winter months compared to the summer months. Moreover, the changes in electricity usage during both seasonal months highlights that consumption has slowly decreased over the last decade. On average, the total electricity consumption by the building has been 6,246,353.16 kWh over the last 10 years.

Increase in the electricity consumption during the winter months could be attributed to the need to heat the building due to cold weather, in particular, January 2011 showed a very high consumption. During this period, the UK suffered extreme cold weather with significant snow events (MetOffice, 2011). The slow decrease in consumption over the last 10 years during both winter and summer seasons may be due to climate change with milder winters and more modern and energy efficient appliances being utilised.

To conclude, Apache Hadoop is a very useful tool in batch processing of large real-time data, saving computational time and costs. This technique is particularly beneficial in obtaining information in an efficient, user-friendly manner, when there are large amounts of complex data involved.

4 Reference

BDAN (2022) ‘Top 12 Hadoop Technology Companies’, *Big Data Analytics News*, 12 January. Available at: https://bigdataanalyticsnews.com/top-12-hadoop-technology-companies/ (Accessed: 24 April 2022).

Hager, G. and Wellein, G. (2010) *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press.

Kat, S. (2016) *Word Count Program With MapReduce and Java - DZone Big Data, dzone.com*. Available at: https://dzone.com/articles/word-count-hello-word-program-in-mapreduce (Accessed: 25 April 2022).

Lee, I. (2017) ‘Big data: Dimensions, evolution, impacts, and challenges’, *Business Horizons*, 60(3), pp. 293–303. doi:10.1016/j.bushor.2017.01.004.

Lewis, L. (2021) *Infographic: A Minute on the Internet in 2021, Statista Infographics*. Available at: https://www.statista.com/chart/25443/estimated-amount-of-data-created-on-the-internet-in-one-minute/ (Accessed: 24 April 2022).

MetOffice (2011) *UK Snow 2010, Met Office*. Available at: https://www.metoffice.gov.uk/weather/learn-about/weather/case-studies/uk-snow-2010 (Accessed: 25 April 2022).

Nguyen, T. (2020) *Chaining Multiple MapReduce Jobs with Hadoop/ Java, Medium*. Available at: https://towardsdatascience.com/chaining-multiple-mapreduce-jobs-with-hadoop-java-832a326cbfa7 (Accessed: 25 April 2022).

PHOENIXNAP (2020) *Apache Hadoop Architecture Explained (In-Depth Overview), Knowledge Base by phoenixNAP*. Available at: https://phoenixnap.com/kb/apache-hadoop-architecture-explained (Accessed: 24 April 2022).

Shvachko, K. *et al.* (2010) ‘The Hadoop Distributed File System’, in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10. doi:10.1109/MSST.2010.5496972.

Siva (2014) ‘Hadoop Data Types’, *Hadoop Online Tutorials*, 22 April. Available at: http://hadooptutorial.info/hadoop-data-types/ (Accessed: 25 April 2022).

Taylor-Sakyi, K. (2016) ‘Big Data: Understanding Big Data’, *arXiv:1601.04602 [cs]* [Preprint]. Available at: http://arxiv.org/abs/1601.04602 (Accessed: 6 April 2022).

Thakurdesai, S.M. (2016) ‘Green Computing using Hadoop : An Energy Efficient Technique’, *Asian Journal For Convergence In Technology (AJCT) ISSN -2350-1146*, 2. Available at: https://asianssr.org/index.php/ajct/article/view/197 (Accessed: 25 April 2022).

Vaghasiya, D. (2018) *Batch Processing vs Real Time Processing - Comparison, DataFlair*. Available at: https://data-flair.training/blogs/batch-processing-vs-real-time-processing/ (Accessed: 25 April 2022).

Vaughan, J. (2019) *What is Data? - Definition from WhatIs.com, SearchDataManagement*. Available at: https://www.techtarget.com/searchdatamanagement/definition/data (Accessed: 24 April 2022).

Verma, C. and Pandey, R. (2016) ‘An implementation approach of big data computation by mapping Java classes to MapReduce’, in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*. *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 282–287.

5 Appendix

The table of results from the single MapReduce job can be found in the attached ‘files’ zip file understand the name ‘MapReduce_SeasonalMonths.’ The java files for all MapReduce jobs can be found in the within this zip file as well. Lastly, the data can be found within the Data folder in this zip file as well.