1. Introduction:

Huge volumes of data are generated in today's data-driven market as a result of algorithms and apps constantly gathering information about individuals, businesses, systems, and organisations. The difficult part is figuring out how to quickly and effectively digest this vast volume of data without losing insightful conclusions.

The MapReduce programming model can help in this situation. MapReduce first appeared as a tool for Google to analyse its search results, but it quickly grew in popularity thanks to its capacity to split and process terabytes of data in parallel, producing quicker results.

MapReduce is a programming model or pattern within the Hadoop framework that is used to access big data stored in the Hadoop File System (HDFS). It is essential to the operation of the Hadoop framework and a core component (Merla and Liang., 2017).

By dividing petabytes of data into smaller chunks and processing them in parallel on Hadoop commodity servers, MapReduce makes concurrent processing easier. In the end, it collects all the information from several servers and gives the application a consolidated output.

For many people, agriculture is their primary source of income. These tools can help to improve this area by providing the means to increase crop productivity or by enhancing seed and crop yields. A thorough analysis in this area could be very beneficial to farmers and the agricultural sector.

The purpose of this framework is to help agriculture officers and researchers to understand and recommend crop, based on evidence from historical data.

2. Dataset:

2.1. Crop Yield Dataset:

Data related to crop production statistics for India are divided by state and district and are extensively available in the dataset. From 1997 to 2023, the dataset encompasses six significant crop seasons: kharif, rabbi, summer, Whole Year, winter, and autumn for a total of 33 crops across 33 states and union territories in India. The information available in the data pertains to the crop yield and annual production in various locations around the nation.

Source: https://www.kaggle.com/datasets/nikhilmahajan29/crop-production-statistics-india

Variable	Description	Type
State	Name of the state	Character
Crop	Name of the crop	Character
Season	Season in which the crop is produced	Character
Production	Total production (Tonnes)	Numeric
Yield	Total yield (Tonnes/Hectares)	Numeric

Table 1: Crop_Yield dataset Table.

2.2. Crop Rainfall Dataset:

The data set comprises detailed data about India's agricultural production statistics obtained from the Area Production Statistics (APS) database of the Indian government. The Ministry of Agriculture and Farmers Welfare maintains the APS, which offers comprehensive information regarding rainfall amount, the temperature required for the crop, and other details in various Indian states and districts.

Source: https://www.kaggle.com/datasets/gokulprasantht/crop-prediction

Variable	Description	Type
STATE	Name of the state the crop grows	Character
SEASON	Season in which the crop is produced	Character
TEMPERATURE	Temperature range	Numeric
RAINFALL	Rainfall amount	Numeric
CROP	Name of the crop	Character

Table 2: Crop_Rainfall dataset Table.

Both datasets are taken from Kaggle and modified according to the requirement and used here.

2.3. Data pre-processing:

The Raw data will contain unknow, noisy and missing values, which makes data analysis errors prevalent. It is essential to pre-process the data. Through this procedure, unprocessed data are transformed into a usable format. The pre-processing approach is used to resolve this problem, and after that, the data are removed for processing.

3. Problem Description and Research question:

The agriculture sector significantly contributes to India’s economy. Consequently, it can be asserted that agriculture is a fundamental aspect of India and characterizes our identity as a nation. Many Indian farmers rely on their intuition to decide which crop to plant during a specific season. Typically, they consider historical factors and traditional farming methods but fail to acknowledge that current weather and rainfall conditions influence crop yield. It can be challenging to hold a farmer responsible for each decision they make because there are countless factors at play and farmers often underestimate or overestimate the fertility of the soil on their farms. Thus follows our research question,

Can we able to obtain information on the maximum and minimum crop yields, average crop production, as well as average temperature and rainfall requirements for six distinct seasons in a particular state? to help policymaker to decide what crop to sow on which season.

Using the Hadoop map-reduce framework with appropriate parameters like rainfalls, temperatures, and other factors such as previous average production and yield for the crops, it is possible to predict the most appropriate crop to be sow during a particular season. Framers can benefit from the dataset by gaining insights into the best crop to grow in their region and making informed decisions on crop management practices.

4. Design Framework:

In this framework, the data flow is as follows:

1. The input for this framework is Crop_Yield.txt and Crop_Rainfall.txt.
2. Mappers get the content from the text file. The filter value in the mapper will filter the content with respect to the 'State' name mentioned by the user.
3. YieldMapper will group the 'Crop' variable in Key and add the label 'Mapper 1:' to the values (Season, Yield, Production) to join the values with the RainfallMapper output.
4. RainfallMapper will group the 'Crop' variable in Key and add the label 'Mapper 2:' to the values (Season, Temperature, Rainfall) to join the values with the YieldMapper output.
5. The (key, value) pair will go to the Partitioner, where it will distribute according to the season. In this dataset we have 6 seasons, so the partition will give 6 key, value pair output for each season with partition number.
6. The partitioner will ensure that all the intermediate key-value pairs with the same key end up in the same partition, and thus the same reducer (Abdelhamid et al., 2020)
7. For each season, the partitioner will assign a different partition number. The partition number is used to determine which reducer will receive the data for further processing.
8. In the shuffle phase of MapReduce, the output of the map tasks is partitioned and sorted by key before being sent to the reduce tasks. This is necessary so that all the values associated with a particular key are grouped together and processed by a single reduce task. The shuffle phase involves the following steps:
   - Partitioning: The Map task results are divided into different sections according to their key values. A different reduce task is assigned for each partition.
   - Sorting: The key-value pairs within each partition are sorted based on the key (Chen et al., 2019).
   - Data transfer: From the map tasks to the reduce tasks, the sorted data is transferred.
9. During the MapReduce's sorting phase, the sorting process occurs based on the key values within the pairs. It is imperative to ensure this because the reduce tasks get the data arranged by the key, and they operate on the data in a sequential pattern. During the sort phase, the data that was produced during the map phase is organized and saved to a temporary file. The reduced tasks go through the sorted data and handle it in a sorted manner.
10. The Reducer will get the output from the Partitioner and joins it by the key and then perform reducer side join for mapper 1 and mapper 2 values.
11. The key is nothing but the crop name. The list of values will have the input from both the datasets. It loops through the values present in the list of values in the reducer, then split the list of values and check whether the value is of Mapper1 or Mapper2.
12. If it is of the Mapper1, it will calculate the minimum and maximum yield and average production of the crop. On the other hand, if the value is of Mapper2, it will calculate the average Rainfall and temperature required for the crop.
13. Finally, it will write the output key-value pair in the output folder in my HDFS.

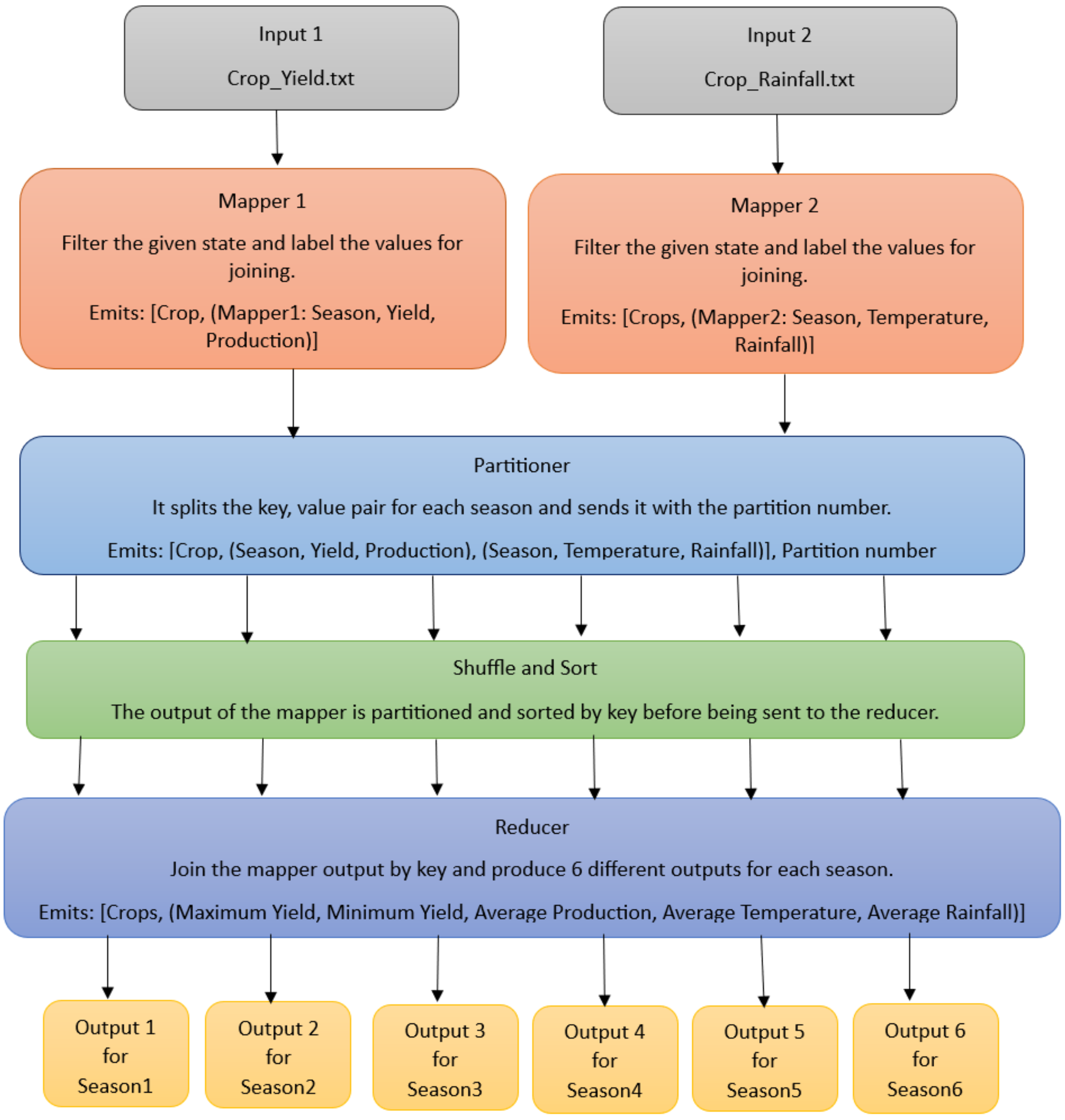

Input 1
Crop_Yield.txt
Input 2
Crop_Rainfall.txt
Mapper 1
Filter the given state and label the values for joining.
Emits: [Crop, (Mapper1: Season, Yield, Production)]
Mapper 2
Filter the given state and label the values for joining.
Emits: [Crops, (Mapper2: Season, Temperature, Rainfall)]
Partitioner
It splits the key, value pair for each season and sends it with the partition number.
Emits: [Crop, (Season, Yield, Production), (Season, Temperature, Rainfall)], Partition number
Shuffle and Sort
The output of the mapper is partitioned and sorted by key before being sent to the reducer.
Reducer
Join the mapper output by key and produce 6 different outputs for each season.
Emits: [Crops, (Maximum Yield, Minimum Yield, Average Production, Average Temperature, Average Rainfall)]
Output 1 for Season1
Output 2 for Season2
Output 3 for Season3
Output 4 for Season4
Output 5 for Season5
Output 6 for Season6

Figure 1: Map Reduce job flowchart.

5. Implementation:

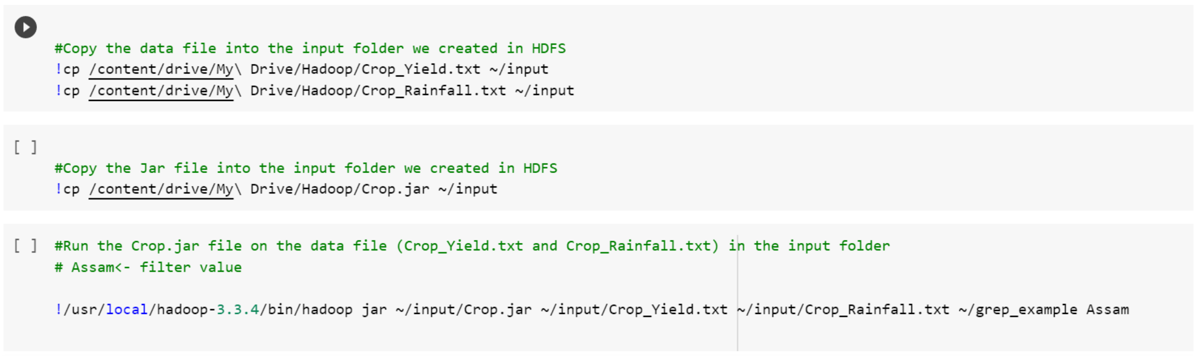
To use the Hadoop MapReduce framework on Google Colab, it's necessary to establish a Hadoop cluster in the cloud and customize its settings so it can function with Colab. Using Java and the Hadoop APIs, we can create MapReduce programmes that can be run on the cluster after it has been configured (Saadoon and Admodisastro, 2022).

Setting up a Hadoop development environment is important for programming MapReduce in Eclipse. This entails installing the Hadoop plugin for Eclipse, configuring the Hadoop configuration files, and importing the required Hadoop libraries. Using Java and the Hadoop APIs, you can write MapReduce programmes that can be executed on a Hadoop cluster once the environment has been configured. For Hadoop programming, Eclipse offers a potent integrated development environment with tools like code completion, debugging, and profiling.

IDE used: Eclipse

Run Environment: Google Colab.

Programming Language Used: Java.



```
#Copy the data file into the input folder we created in HDFS
!cp /content/drive/My\ Drive/Hadoop/Crop_Yield.txt ~/input
!cp /content/drive/My\ Drive/Hadoop/Crop_Rainfall.txt ~/input
```

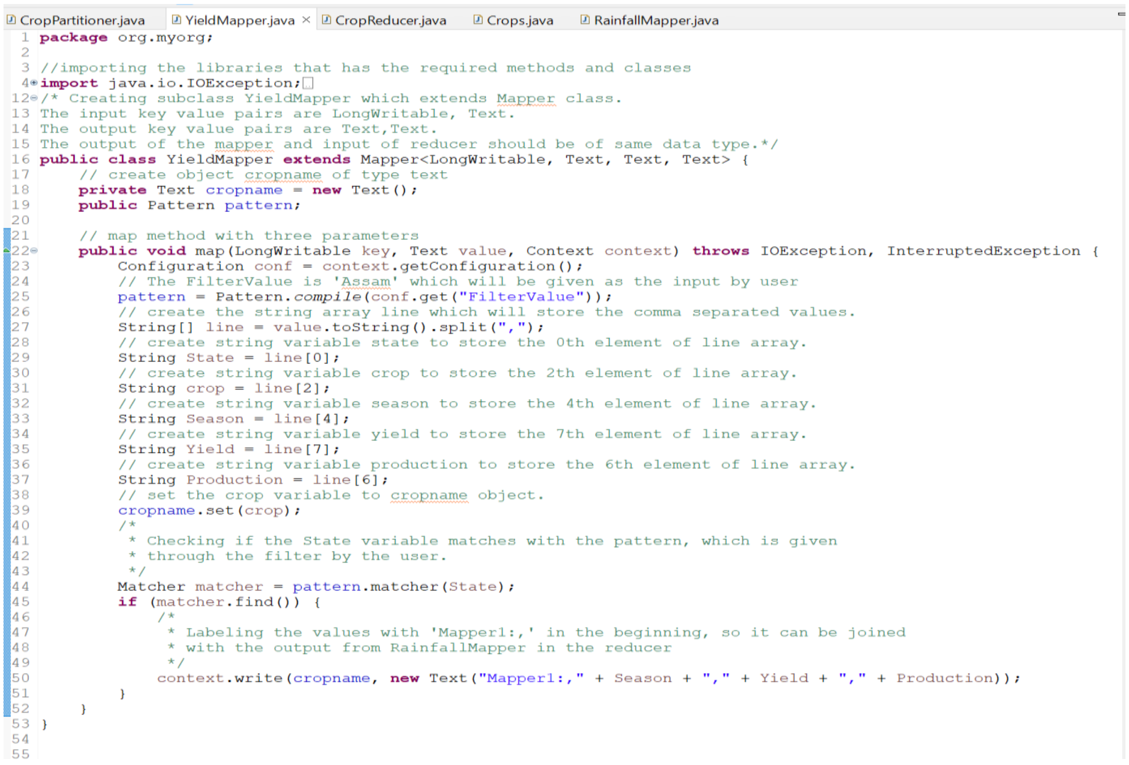
```
[ ]
#Copy the Jar file into the input folder we created in HDFS
!cp /content/drive/My\ Drive/Hadoop/Crop.jar ~/input
```

```
[ ] #Run the Crop.jar file on the data file (Crop_Yield.txt and Crop_Rainfall.txt) in the input folder
    # Assam<- filter value

    !/usr/local/hadoop-3.3.4/bin/hadoop jar ~/input/Crop.jar ~/input/Crop_Yield.txt ~/input/Crop_Rainfall.txt ~/grep_example Assam
```

Figure 2: Implementing in Google Colab.

5.1. Mapper:

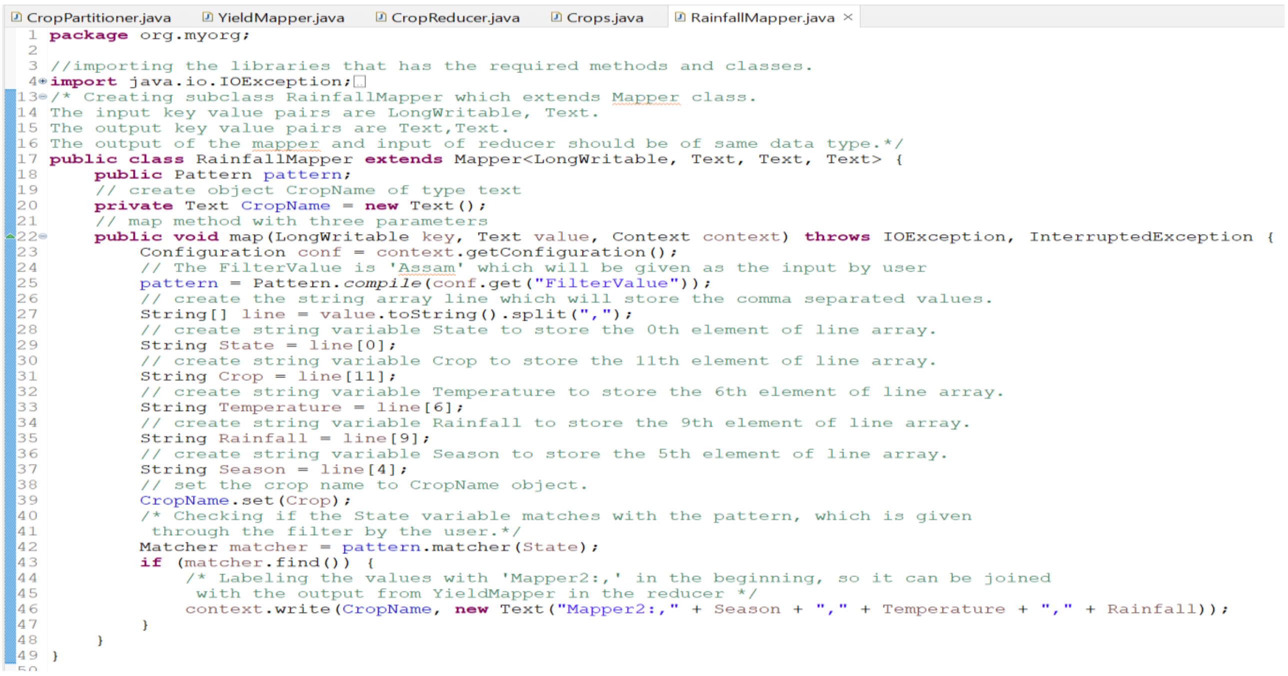
YieldMapper: The Mapper takes in a LongWritable key, a Text value and emits a Text key, Text value pair. The Mapper class analyses an input file that is separated by commas, then breaks it down to identify particular parts such as the yield, season, crop, production, and state. Afterward, an examination is carried out to determine if the State corresponds to a predetermined filtration value that is configured within the Hadoop Configuration entity. If this is the case, the values are identified with the prefix "Mapper1:" and outputted as a pair of Text key and Text value, which consist of the label with value and the crop name. The primary purpose of this class is to serve as the initial mapper in a Hadoop MapReduce function, combining its output later with that of another mapper called RainfallMapper found in the CropReducer class.



```java
package org.myorg;

//importing the libraries that has the required methods and classes
import java.io.IOException;
/* Creating subclass YieldMapper which extends Mapper class.
The input key value pairs are LongWritable, Text.
The output key value pairs are Text,Text.
The output of the mapper and input of reducer should be of same data type.*/
public class YieldMapper extends Mapper<LongWritable, Text, Text, Text> {
    // create object cropname of type text
    private Text cropname = new Text();
    public Pattern pattern;

    // map method with three parameters
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        Configuration conf = context.getConfiguration();
        // The FilterValue is 'Assam' which will be given as the input by user
        pattern = Pattern.compile(conf.get("FilterValue"));
        // create the string array line which will store the comma separated values.
        String[] line = value.toString().split(",");
        // create string variable state to store the 0th element of line array.
        String State = line[0];
        // create string variable crop to store the 2th element of line array.
        String crop = line[2];
        // create string variable season to store the 4th element of line array.
        String Season = line[4];
        // create string variable yield to store the 7th element of line array.
        String Yield = line[7];
        // create string variable production to store the 6th element of line array.
        String Production = line[6];
        // set the crop variable to cropname object.
        cropname.set(crop);
        /*
         * Checking if the State variable matches with the pattern, which is given
         * through the filter by the user.
         */
        Matcher matcher = pattern.matcher(State);
        if (matcher.find()) {
            /*
             * Labeling the values with 'Mapper1:,' in the beginning, so it can be joined
             * with the output from RainfallMapper in the reducer
             */
            context.write(cropname, new Text("Mapper1:," + Season + "," + Yield + "," + Production));
        }
    }
}
```

Figure 3: YieldMapper code snippet

RainfallMapper: The RainfallMapper extends the Hadoop Mapper class, with LongWritable, Text, Text, Text as its input and output data types respectively. The goal of this mapper is to filter out the agricultural data records that match a given State value using a regular expression pattern, and output records with the CropName as the key and a concatenated string of the Season, Temperature, and Rainfall fields as the value with a prefix "Mapper2," added to the beginning of the string.

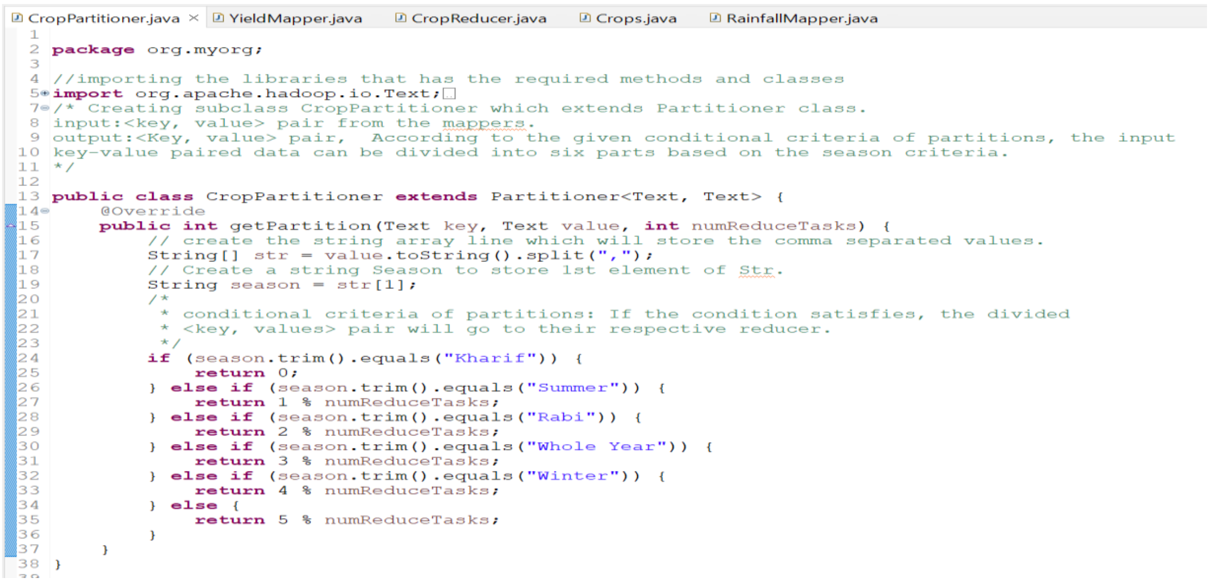


```java
package org.myorg;

//importing the libraries that has the required methods and classes.
import java.io.IOException;
/* Creating subclass RainfallMapper which extends Mapper class.
The input key value pairs are LongWritable, Text.
The output key value pairs are Text,Text.
The output of the mapper and input of reducer should be of same data type.*/
public class RainfallMapper extends Mapper<LongWritable, Text, Text, Text> {
    public Pattern pattern;
    // create object CropName of type text
    private Text CropName = new Text();
    // map method with three parameters
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        Configuration conf = context.getConfiguration();
        // The FilterValue is 'Assam' which will be given as the input by user
        pattern = Pattern.compile(conf.get("FilterValue"));
        // create the string array line which will store the comma separated values.
        String[] line = value.toString().split(",");
        // create string variable State to store the 0th element of line array.
        String State = line[0];
        // create string variable Crop to store the 11th element of line array.
        String Crop = line[11];
        // create string variable Temperature to store the 6th element of line array.
        String Temperature = line[6];
        // create string variable Rainfall to store the 9th element of line array.
        String Rainfall = line[9];
        // create string variable Season to store the 5th element of line array.
        String Season = line[4];
        // set the crop name to CropName object.
        CropName.set(Crop);
        /* Checking if the State variable matches with the pattern, which is given
         through the filter by the user.*/
        Matcher matcher = pattern.matcher(State);
        if (matcher.find()) {
            /* Labeling the values with 'Mapper2:,' in the beginning, so it can be joined
             with the output from YieldMapper in the reducer */
            context.write(CropName, new Text("Mapper2:," + Season + "," + Temperature + "," + Rainfall));
        }
    }
}
```

Figure 4: RainfallMapper code snippet

5.2. Partitioner:

The Partitioner within the MapReduce system manages how the key of the intermediate mapper output is partitioned. The partition is derived by utilizing the key (or a portion of it) through a process called hash function. The number of reduced tasks determines the total quantity of partitions (Wu et al., 2019). In partitioner we are checking the Season from the values given as the input. Depending upon the season, the key values pairs are split into six parts with partition number.

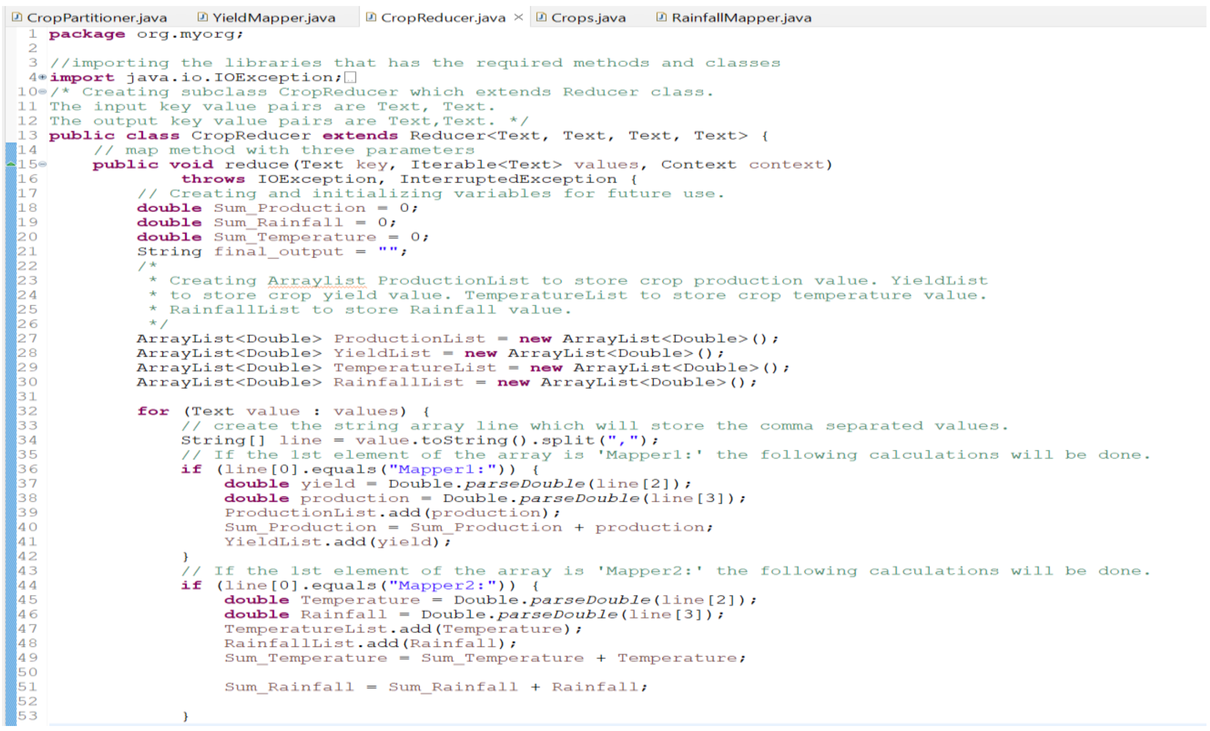


```java

package org.myorg;

//importing the libraries that has the required methods and classes
import org.apache.hadoop.io.Text;
/* Creating subclass CropPartitioner which extends Partitioner class.
input:<key, value> pair from the mappers.
output:<Key, value> pair,  According to the given conditional criteria of partitions, the input
key-value paired data can be divided into six parts based on the season criteria.
*/

public class CropPartitioner extends Partitioner<Text, Text> {
    @Override
    public int getPartition(Text key, Text value, int numReduceTasks) {
        // create the string array line which will store the comma separated values.
        String[] str = value.toString().split(",");
        // Create a string Season to store 1st element of Str.
        String season = str[1];
        /*
         * conditional criteria of partitions: If the condition satisfies, the divided
         * <key, values> pair will go to their respective reducer.
         */
        if (season.trim().equals("Kharif")) {
            return 0;
        } else if (season.trim().equals("Summer")) {
            return 1 % numReduceTasks;
        } else if (season.trim().equals("Rabi")) {
            return 2 % numReduceTasks;
        } else if (season.trim().equals("Whole Year")) {
            return 3 % numReduceTasks;
        } else if (season.trim().equals("Winter")) {
            return 4 % numReduceTasks;
        } else {
            return 5 % numReduceTasks;
        }
    }
}
```

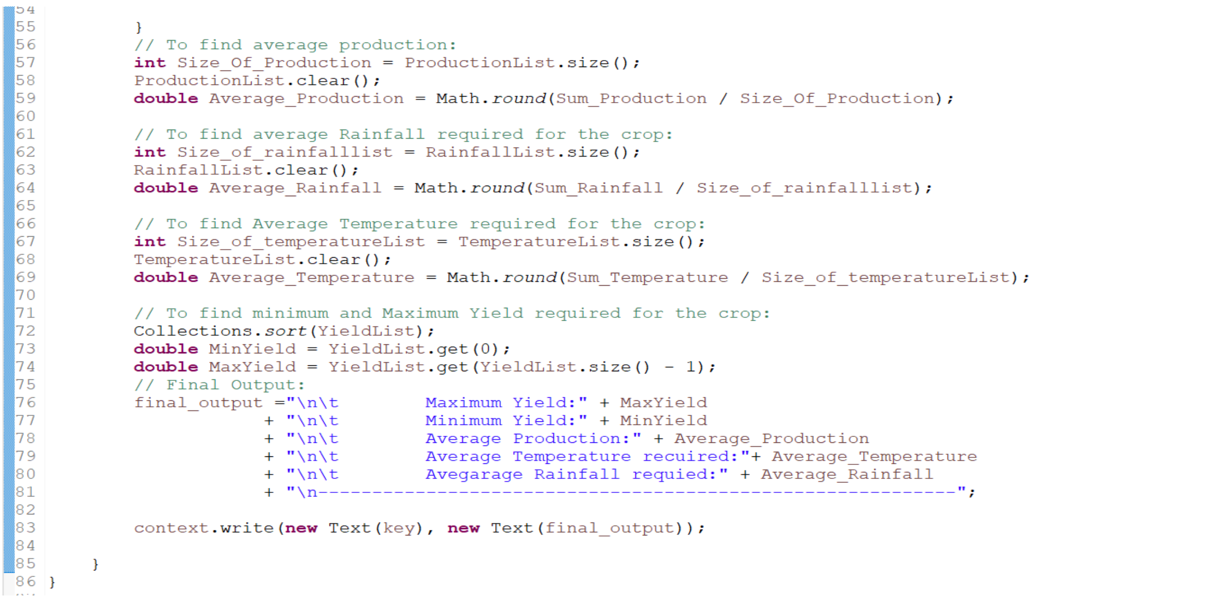
Figure 5: CropPartitioner code snippet

5.3. Reducer:

When it comes to reduce side joins, the role of carrying out the join operation is assigned to the reducer. Compared to the map side join approach, this method is simpler and more feasible to execute because during the sorting and shuffling process, values with the same keys are automatically directed to the same reducer, resulting in organized data (Nallathamby et al., 2021). Following the phase of categorization and rearrangement, a reducer is provided with a list of values and a corresponding key. At present, the final collective result is attained when the reducer associates the key with the existing values in the list.



```java
package org.myorg;

//importing the libraries that has the required methods and classes
import java.io.IOException;
/* Creating subclass CropReducer which extends Reducer class.
The input key value pairs are Text, Text.
The output key value pairs are Text,Text. */
public class CropReducer extends Reducer<Text, Text, Text, Text> {
    // map method with three parameters
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Creating and initializing variables for future use.
        double Sum_Production = 0;
        double Sum_Rainfall = 0;
        double Sum_Temperature = 0;
        String final_output = "";
        /*
         * Creating Arraylist ProductionList to store crop production value. YieldList
         * to store crop yield value. TemperatureList to store crop temperature value.
         * RainfallList to store Rainfall value.
         */
        ArrayList<Double> ProductionList = new ArrayList<Double>();
        ArrayList<Double> YieldList = new ArrayList<Double>();
        ArrayList<Double> TemperatureList = new ArrayList<Double>();
        ArrayList<Double> RainfallList = new ArrayList<Double>();

        for (Text value : values) {
            // create the string array line which will store the comma separated values.
            String[] line = value.toString().split(",");
            // If the 1st element of the array is 'Mapper1:' the following calculations will be done.
            if (line[0].equals("Mapper1:")) {
                double yield = Double.parseDouble(line[2]);
                double production = Double.parseDouble(line[3]);
                ProductionList.add(production);
                Sum_Production = Sum_Production + production;
                YieldList.add(yield);
            }
            // If the 1st element of the array is 'Mapper2:' the following calculations will be done.
            if (line[0].equals("Mapper2:")) {
                double Temperature = Double.parseDouble(line[2]);
                double Rainfall = Double.parseDouble(line[3]);
                TemperatureList.add(Temperature);
                RainfallList.add(Rainfall);
                Sum_Temperature = Sum_Temperature + Temperature;

                Sum_Rainfall = Sum_Rainfall + Rainfall;

            }
```

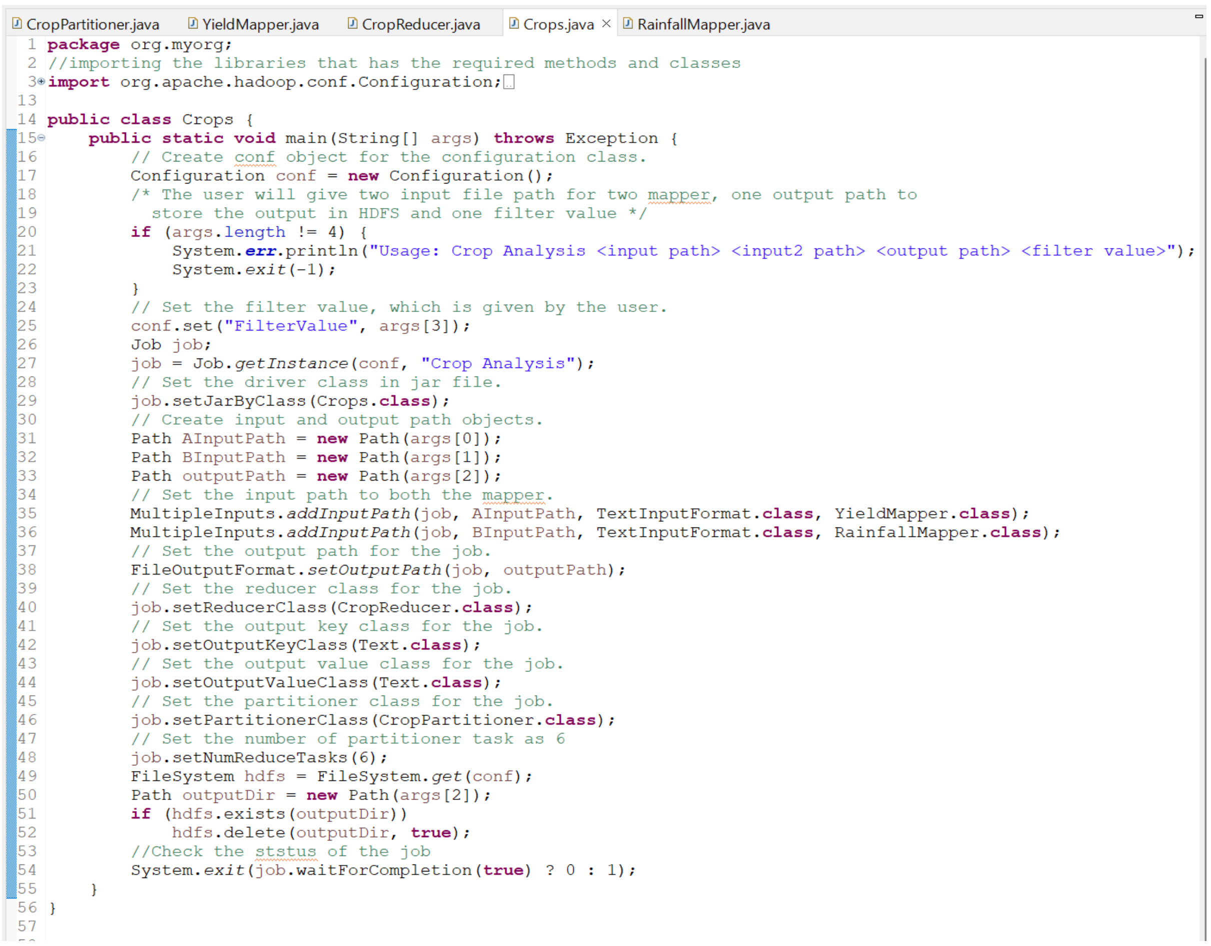


```java

        }
        // To find average production:
        int Size_Of_Production = ProductionList.size();
        ProductionList.clear();
        double Average_Production = Math.round(Sum_Production / Size_Of_Production);

        // To find average Rainfall required for the crop:
        int Size_of_rainfalllist = RainfallList.size();
        RainfallList.clear();
        double Average_Rainfall = Math.round(Sum_Rainfall / Size_of_rainfalllist);

        // To find Average Temperature required for the crop:
        int Size_of_temperatureList = TemperatureList.size();
        TemperatureList.clear();
        double Average_Temperature = Math.round(Sum_Temperature / Size_of_temperatureList);

        // To find minimum and Maximum Yield required for the crop:
        Collections.sort(YieldList);
        double MinYield = YieldList.get(0);
        double MaxYield = YieldList.get(YieldList.size() - 1);
        // Final Output:
        final_output ="\n\t        Maximum Yield:" + MaxYield
                    + "\n\t        Minimum Yield:" + MinYield
                    + "\n\t        Average Production:" + Average_Production
                    + "\n\t        Average Temperature recuired:"+ Average_Temperature
                    + "\n\t        Avegarage Rainfall requied:" + Average_Rainfall
                    + "\n-----------------------------------------------------------";

        context.write(new Text(key), new Text(final_output));

    }
}
```

Figure 6: CropReducer code snippet

5.4. Driver Class:

The user is required to provide four input parameters to the program, namely, two paths for mapper input files, an HDFS output path where the output should be stored, and a filter value. To begin with, the software verifies if the user has entered all the necessary input arguments. It takes the user-provided filter value and assigns it to the Hadoop Configuration object. A new "Crop Analysis" Hadoop Job object is created. With the use of the MultipleInputs class, it establishes the input path for both mappers. The FileOutputFormat class is employed to establish the job's output path. Additionally, it assigns the job's partitioner class and specifies the partitioner tasks to be 6.



```java
package org.myorg;
//importing the libraries that has the required methods and classes
import org.apache.hadoop.conf.Configuration;

public class Crops {
    public static void main(String[] args) throws Exception {
        // Create conf object for the configuration class.
        Configuration conf = new Configuration();
        /* The user will give two input file path for two mapper, one output path to
          store the output in HDFS and one filter value */
        if (args.length != 4) {
            System.err.println("Usage: Crop Analysis <input path> <input2 path> <output path> <filter value>");
            System.exit(-1);
        }
        // Set the filter value, which is given by the user.
        conf.set("FilterValue", args[3]);
        Job job;
        job = Job.getInstance(conf, "Crop Analysis");
        // Set the driver class in jar file.
        job.setJarByClass(Crops.class);
        // Create input and output path objects.
        Path AInputPath = new Path(args[0]);
        Path BInputPath = new Path(args[1]);
        Path outputPath = new Path(args[2]);
        // Set the input path to both the mapper.
        MultipleInputs.addInputPath(job, AInputPath, TextInputFormat.class, YieldMapper.class);
        MultipleInputs.addInputPath(job, BInputPath, TextInputFormat.class, RainfallMapper.class);
        // Set the output path for the job.
        FileOutputFormat.setOutputPath(job, outputPath);
        // Set the reducer class for the job.
        job.setReducerClass(CropReducer.class);
        // Set the output key class for the job.
        job.setOutputKeyClass(Text.class);
        // Set the output value class for the job.
        job.setOutputValueClass(Text.class);
        // Set the partitioner class for the job.
        job.setPartitionerClass(CropPartitioner.class);
        // Set the number of partitioner task as 6
        job.setNumReduceTasks(6);
        FileSystem hdfs = FileSystem.get(conf);
        Path outputDir = new Path(args[2]);
        if (hdfs.exists(outputDir))
            hdfs.delete(outputDir, true);
        //Check the ststus of the job
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 7: Crops code snippet

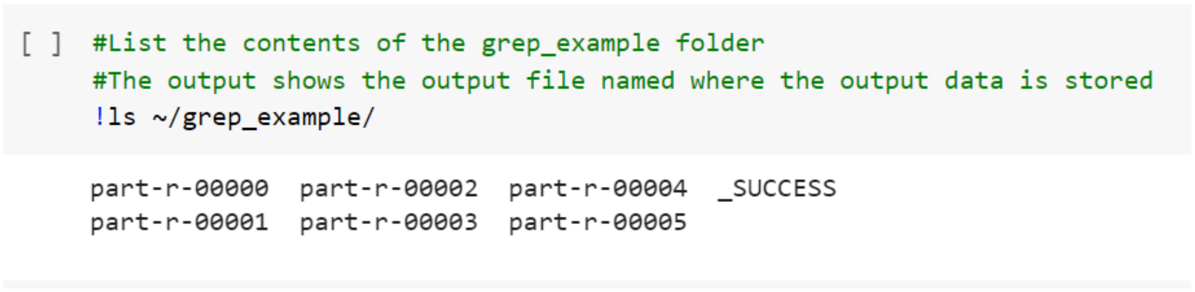
Stage	Input	Output	Input Type	Output Type
Mapper 1	Dataset and Filter value.	Crop, (Mapper1:,Season,Yield,Production)	K: LongWritable, V: Text	K: Text, V: Text
Mapper 2	Dataset and Filter value.	Crop, (Mapper2:,Season,Temperature,Rainfall)	K: LongWritable, V: Text	K: Text, V: Text
Partitioner	Key-Value pair from both the mapper	Segmented key-value pair for each season. Crop,(Mapper1:,Season,Yield,Production) Crop,(Mapper2:,Season,Temperature,Rainfall)	K: Text, V: Text	K: Text, V: Text
Reducer	Segmented Key-Value pair from the partitioner	Segmented key-value pair for each season. Crop, (Maximum of yield, Minimum of yield, Average Production, Average Temperature, Average Rainfall)	K: Text, V: Text	K: Text, V: Text

Table 3: Input and Output parameter Table

6. Result and Evaluation:

We will get six outputs from the reducer, one for each season. Going back to our research question, **can we able to obtain information on the maximum and minimum crop yields, average crop production, as well as average temperature and rainfall requirements for six distinct seasons in a particular state?**

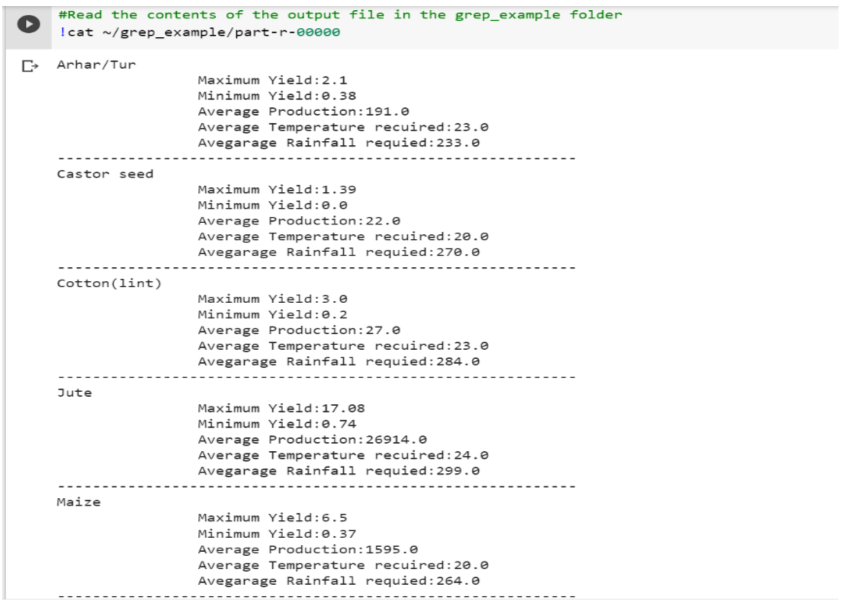
The output data from the reducer given the required information for specific city (in our case, it is Assam) and for six seasons such as Winter, summer, autumn, whole year, kharif and rabi that are needed for the policymaker to make decision to determine which crop to sow on which season and what is the average rainfall and temperature needed for the crop.



```
[ ]  #List the contents of the grep_example folder
     #The output shows the output file named where the output data is stored
     !ls ~/grep_example/

     part-r-00000  part-r-00002  part-r-00004  _SUCCESS
     part-r-00001  part-r-00003  part-r-00005
```

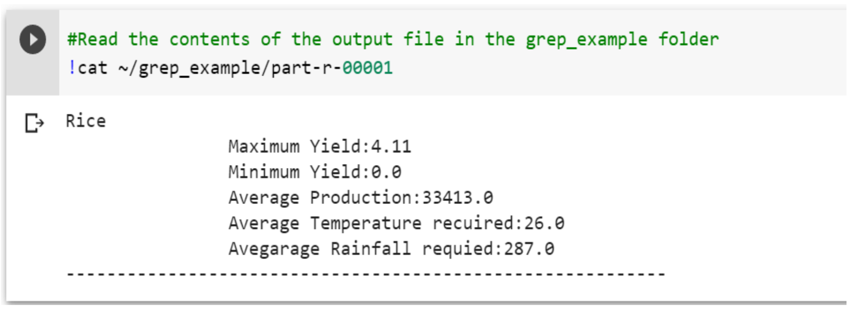
Figure 8: Output file from google colab.

The output for each season is shown below,



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00000

Arhar/Tur
                Maximum Yield:2.1
                Minimum Yield:0.38
                Average Production:191.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:233.0
------------------------------------------------------------
Castor seed
                Maximum Yield:1.39
                Minimum Yield:0.0
                Average Production:22.0
                Average Temperature recuired:20.0
                Avegarage Rainfall requied:270.0
------------------------------------------------------------
Cotton(lint)
                Maximum Yield:3.0
                Minimum Yield:0.2
                Average Production:27.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:284.0
------------------------------------------------------------
Jute
                Maximum Yield:17.08
                Minimum Yield:0.74
                Average Production:26914.0
                Average Temperature recuired:24.0
                Avegarage Rainfall requied:299.0
------------------------------------------------------------
Maize
                Maximum Yield:6.5
                Minimum Yield:0.37
                Average Production:1595.0
                Average Temperature recuired:20.0
                Avegarage Rainfall requied:264.0
------------------------------------------------------------
```

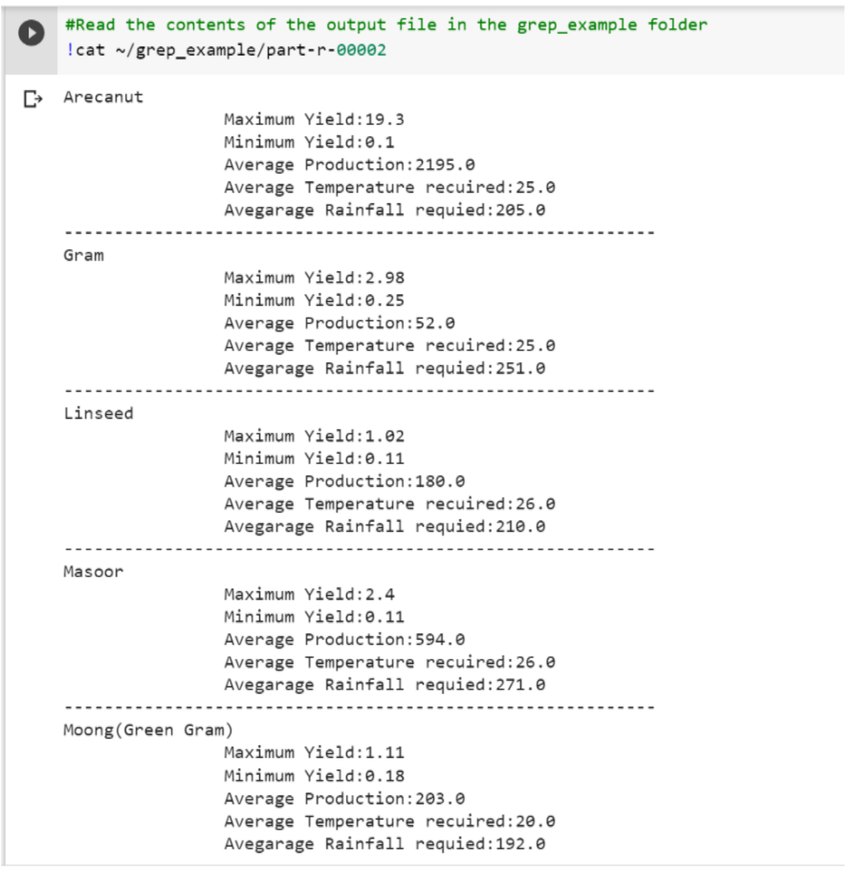
Figure 9: Output for Kharif Season



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00001
```

```
Rice
                Maximum Yield:4.11
                Minimum Yield:0.0
                Average Production:33413.0
                Average Temperature recuired:26.0
                Avegarage Rainfall requied:287.0
-----------------------------------------------------------
```

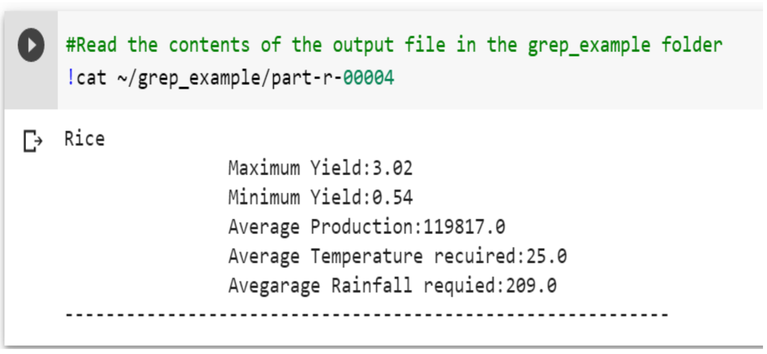
Figure 10: Output for Summer Season



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00002
```

```
Arecanut
                Maximum Yield:19.3
                Minimum Yield:0.1
                Average Production:2195.0
                Average Temperature recuired:25.0
                Avegarage Rainfall requied:205.0
-----------------------------------------------------------
Gram
                Maximum Yield:2.98
                Minimum Yield:0.25
                Average Production:52.0
                Average Temperature recuired:25.0
                Avegarage Rainfall requied:251.0
-----------------------------------------------------------
Linseed
                Maximum Yield:1.02
                Minimum Yield:0.11
                Average Production:180.0
                Average Temperature recuired:26.0
                Avegarage Rainfall requied:210.0
-----------------------------------------------------------
Masoor
                Maximum Yield:2.4
                Minimum Yield:0.11
                Average Production:594.0
                Average Temperature recuired:26.0
                Avegarage Rainfall requied:271.0
-----------------------------------------------------------
Moong(Green Gram)
                Maximum Yield:1.11
                Minimum Yield:0.18
                Average Production:203.0
                Average Temperature recuired:20.0
                Avegarage Rainfall requied:192.0
```

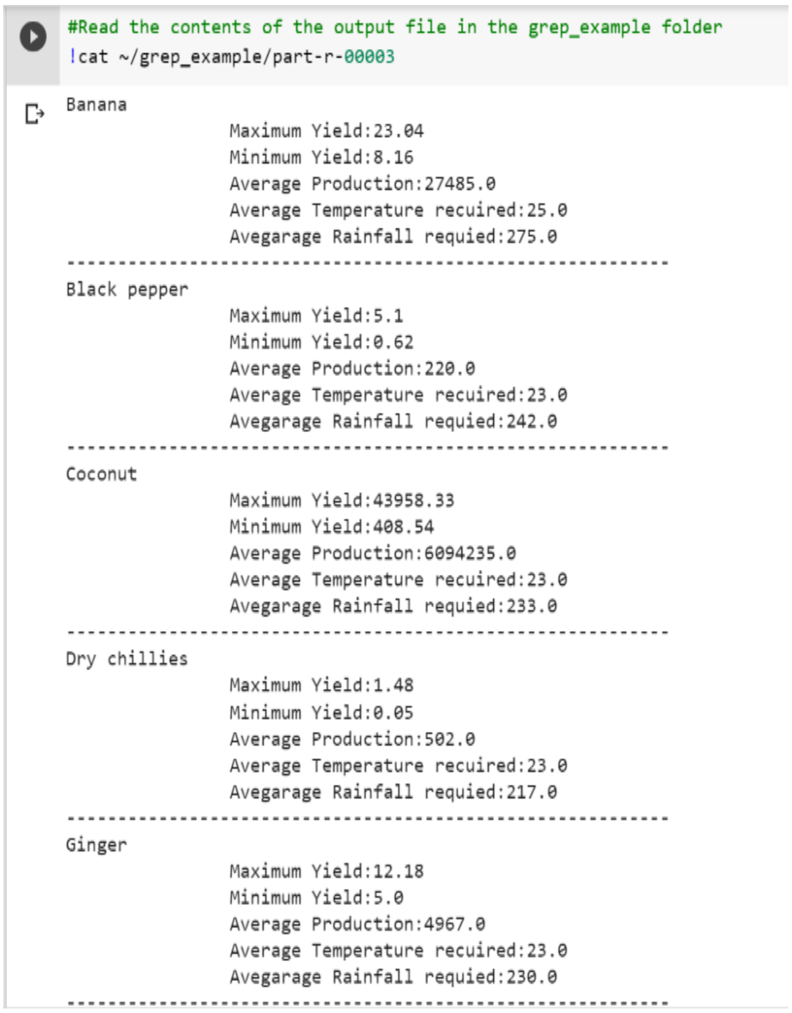
Figure 11: Output for Rabbi season



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00004
```

```
Rice
                Maximum Yield:3.02
                Minimum Yield:0.54
                Average Production:119817.0
                Average Temperature recuired:25.0
                Avegarage Rainfall requied:209.0
-----------------------------------------------------------
```

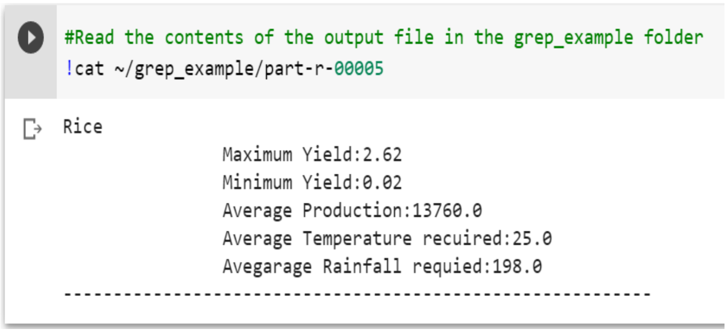
Figure 12: Output for Winter season



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00003
```

```
Banana
                Maximum Yield:23.04
                Minimum Yield:8.16
                Average Production:27485.0
                Average Temperature recuired:25.0
                Avegarage Rainfall requied:275.0
-----------------------------------------------------------
Black pepper
                Maximum Yield:5.1
                Minimum Yield:0.62
                Average Production:220.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:242.0
-----------------------------------------------------------
Coconut
                Maximum Yield:43958.33
                Minimum Yield:408.54
                Average Production:6094235.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:233.0
-----------------------------------------------------------
Dry chillies
                Maximum Yield:1.48
                Minimum Yield:0.05
                Average Production:502.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:217.0
-----------------------------------------------------------
Ginger
                Maximum Yield:12.18
                Minimum Yield:5.0
                Average Production:4967.0
                Average Temperature recuired:23.0
                Avegarage Rainfall requied:230.0
-----------------------------------------------------------
```

Figure 13: Output for Whole Year



```
#Read the contents of the output file in the grep_example folder
!cat ~/grep_example/part-r-00005
```

```
Rice
                Maximum Yield:2.62
                Minimum Yield:0.02
                Average Production:13760.0
                Average Temperature recuired:25.0
                Avegarage Rainfall requied:198.0
-----------------------------------------------------------
```

Figure 14: Output for Autumn Season

By visualising the output data of Kharif season, we can able to see that the sugarcane has the maximum production with value of 55,976 tonnes, so the policy maker will get the idea, that it is preferable to sow sugarcane in that season.

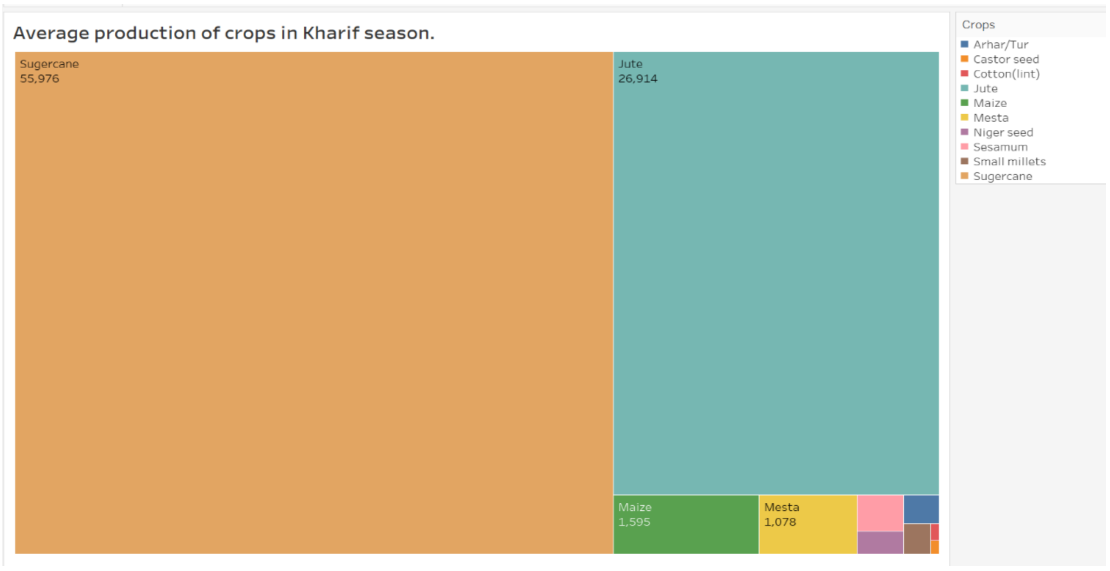


Figure 15: Data Visualisation of crop production for kharif season

The below figure visualises the average temperature for the crops in Kharif season, from this we can see that Jute needs large amount of rainfall and mesta needs least amount of rainfall. All this information will help the policy maker to decide which crop to plan based of the rainfall and temperature. Farmers can also benefit from the dataset by gaining insights into the best crops to grow in their region and making informed decisions on crop management practices.

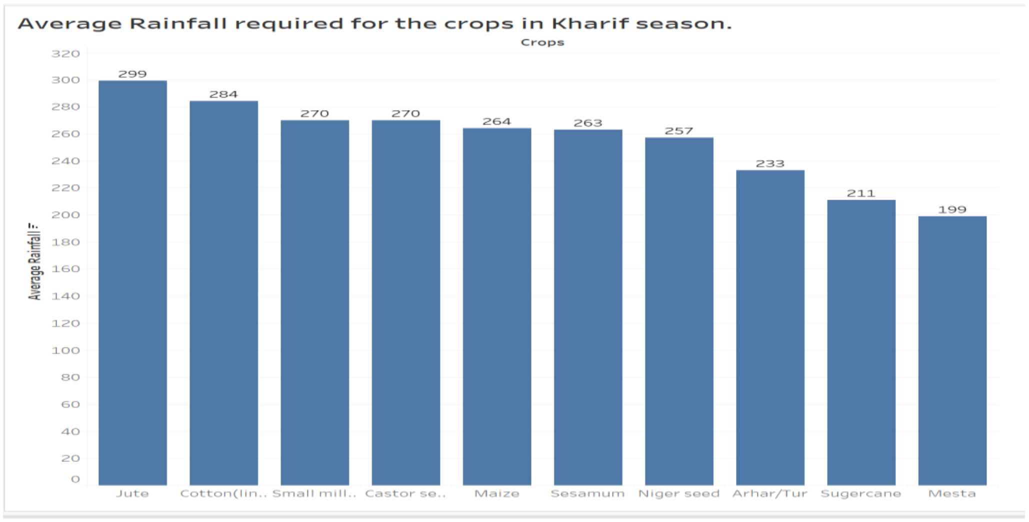


Figure 16: Data visualisation of average Rainfall and Average Temperature for kharif season.

The Data visualization of the maximum and minimum crop, shows that sugarcane have maximum yield of 74.99 tonnes and minimum of 33.59 tonnes in kharif season. Overall, the project offered useful insights for the Indian agriculture industry and showed how well the MapReduce programming model works for analysing enormous amounts of data.

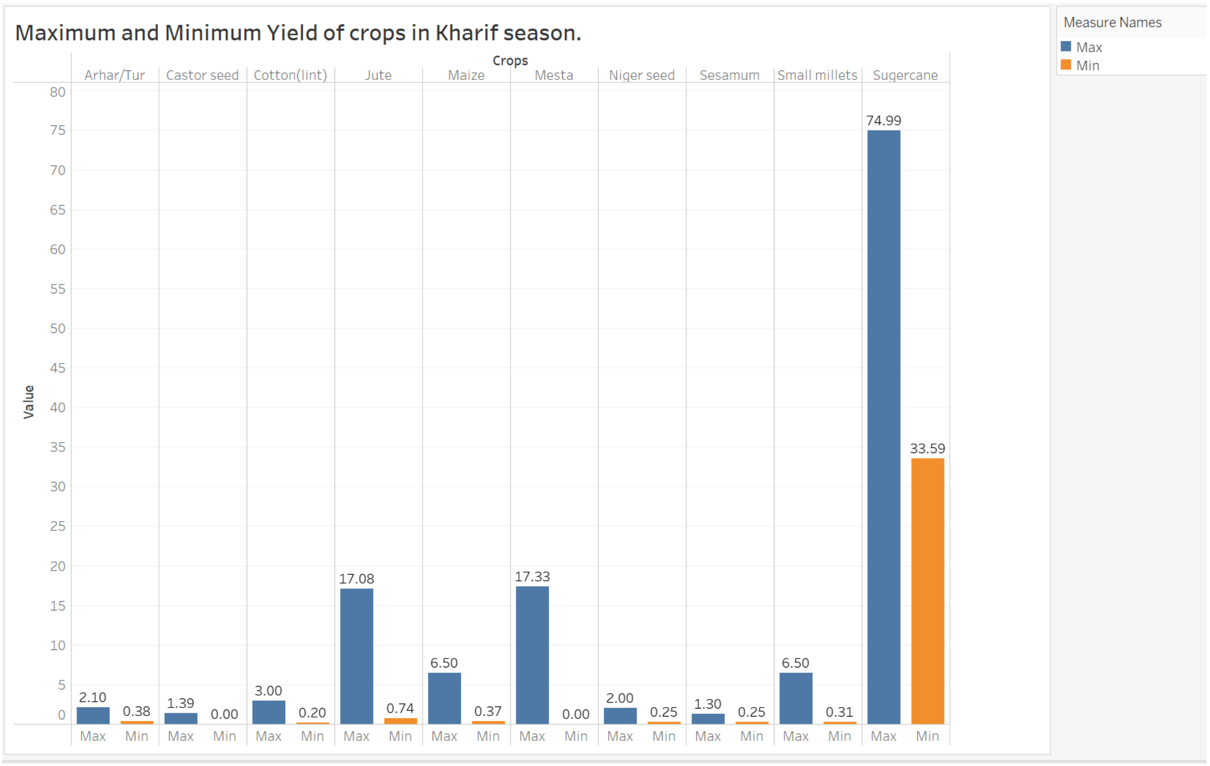
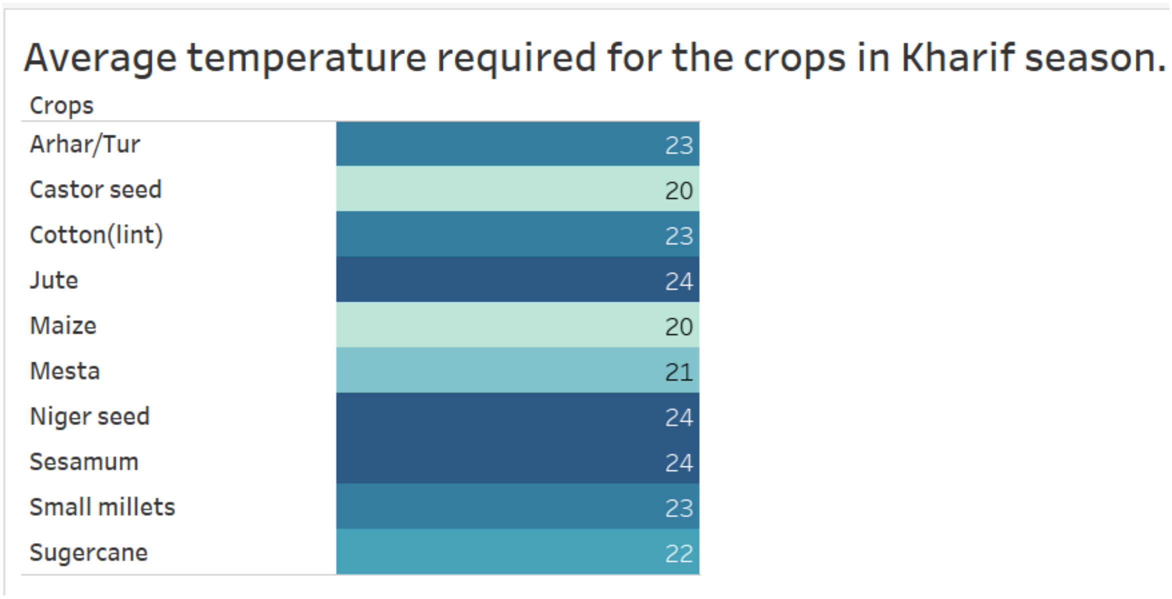


Figure 17: Data visualization of the maximum and minimum yield of crops.



Average temperature required for the crops in Kharif season.

Crops	
Arhar/Tur	23
Castor seed	20
Cotton(lint)	23
Jute	24
Maize	20
Mesta	21
Niger seed	24
Sesamum	24
Small millets	23
Sugercane	22

Figure 18: Data visualization of average temperature required for the crops.

7. Reflection:

The information produced by the MapReducer can be extremely useful to policymakers and farmers in various states when deciding on the appropriate crops to plant.

Farmers can improve their crop yields, mitigate crop failures, and enhance their profitability by utilizing this information. Policymakers can use this information to make knowledgeable choices about the utilization of water as well as other crucial determinants that have an impact on plant development and productivity. This is possible due to the swift and precise processing of extensive data. The policymakers can quickly and effectively analyse enormous volumes of data, producing more accurate recommendations that, in turn, result in more effective and efficient use of agricultural resources, higher crop yields, and better economic consequences.

8. References:

1. Merla, P. and Liang, Y., 2017, December. Data analysis using hadoop MapReduce environment. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 4783-4785). IEEE.

2. Abdelhamid, A.S., Mahmood, A.R., Daghistani, A. and Aref, W.G., 2020, June. Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (pp. 2455-2469).

3. Chen, H., Madaminov, S., Ferdman, M. and Milder, P., 2019, April. Sorting large data sets with FPGA-accelerated samplesort. In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 326-326). IEEE.

4. Saadoon, M. and Admodisastro, N., 2022, May. Self-Configured Workflow Platform for MapReduce Job Execution in Cloud Environment. In 2022 Applied Informatics International Conference (AiIC) (pp. 122-125). IEEE.

5. Nallathamby, R., Rene Robin, C.R. and Miriam, D.H., 2021. Optimizing appointment scheduling for out patients and income analysis for hospitals using big data predictive analytics. Journal of Ambient Intelligence and Humanized Computing, 12, pp.5783-5795.

6. Wu, H., Knottenbelt, W.J. and Wolter, K., 2019. An efficient application partitioning algorithm in mobile environments. IEEE Transactions on Parallel and Distributed Systems, 30(7), pp.1464-1480.