DBMS FINAL PROJECT

MapReduce

Abstract

We have used MapReduce to work on the weather datasets that were provided to obtain and analyze the weather information of a state in the US.

Anvaya B Narappa & Manoj Nagarajappa

an001@ucr.edu , mnaga024@ucr.edu

Table of Contents

Introduction 2
Team 2
Contribution 2
Tools Used 2
Project 3
Datasets: 3
Data File Construction: 3
Mean Calculation: 7
Find the Highest and Lowest Average: 10
Sorting the Output: 13
Execution 16
Results 16
References 16

Introduction

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. We have created a local Hadoop cluster with one DataNode and NameNode with yarn framework deployed. This configuration is used to map the datasets provided and then Reduce the mapped data to required output as stated in the problem statement given for the project.

Team

Our Team consists of 2 members, namely:

1) Manoj Nagarajappa
   - Student-ID: 862396051
   - Email-ID: mnaga024@ucr.edu

2) Anvaya B Narappa
   - Student-ID: 862392401
   - Email-ID: an001@ucr.edu

Contribution

Although this was a team effort, Individual efforts are as stated: -

The report was done together, by splitting the classes that were written. We used the references mentioned in the References Section to understand some of the important concepts related to MapReduce.

Manoj: - InputFileconstruction.class, OutputSort.class, ReadMe, Report.

Anvaya: - MeanTemppMonthpState.class, MonthHiLo.class, mapreduceRev.cmd script, Report.

Tools Used

- Hadoop 3.2.1
- MapReduce Java
- IntelliJ Editor
- Windows 11, Ubuntu 20.04.5
- Github
- Powershell Scripting

Project

In this section we explain the problem statement and how we chose to solve it.

Before getting into the problems statement and the solutions, we look into the Datasets Provided.

Datasets:

1) We are provided with WeatherLocations.csv which contains the meta data of every weather station across the world. The fields of interest here are **USAF** which is the station-ID of the station, **CTRY** which indicates the country in which the station is located and **ST** which signifies the state in which the station is located in that country.
2) The recordings dataset (here 2006.txt 2007.txt 2008.txt and 2009.txt) provide the temperature recording by each station every day in the respective year indicated by the name of the file respectively. The fields of interest in the dataset are **STN---:** the stationID, **YEARMODA:** the date of recording in the format YYYYMMDD and **TEMP:** the temperature recorded followed the number of recordings for that day.

Data File Construction:

The problem statements ask us to use MapReduce to analyze the data with respect to the State-Name of the station ID that are located in the US. Hence, we need to first prepare a file using the csv file and datasets, to obtain a file where the station ID is mapped with the State Name. These include all the stations in the US only.

We have done in the following way:

ClassName: InputFileConstruction

In the following snippet, we use Map1 to map the values of csv file, the condition here is to eliminate any line without “US” and the header line. The output of this mapper will be of the form

{stationID, D STStationID }, the D here is a dummy value used to make the classification easier at the reducer side.

```java
1 usage
public class InputFileConstruction {
    //To map the csv file with key value pairs of stationID and state-name
    1 usage
    public static class Map1 extends Mapper <Object, Text, Text, Text>
    {

        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException
        {
            Text newKey = new Text();
            Text newValue = new Text();
            String line = value.toString();
            String[] lines = line.split( regex: ",");
            if (lines.length > 6)
            {
//                  For the Line is valid: there should be a state value, the country should be US and should not be the header
                if (lines[3].equals("US") && lines[4].length()>0 && !(lines[0].equals("USAF")))
                {
                    newKey.set(lines[0]);
                    String valuesState = String.format("%s,%s", lines[4], lines[0]);
                    newValue.set("D " + valuesState);
                    context.write(newKey, newValue);
                }
            }
        }
    }
```

The next Mapper Map2 here is used to map the values in the recording datasets. We obtain the key value pairs as {StationID, T DateAvgTempNumberofReadings}. Here T is a dummy value given for easier classification at the reducer side. We have also given the precision that we only get the 2 digits after the decimal point, same as the datasets.

```java
    public static class Map2 extends Mapper <Object, Text, Text, Text>
    {
        //           This mapper is used to map the data in the format of stationID| date AvgTemp Numberof Readings
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException
        {
            Text newKey = new Text();
            Text newValue = new Text();
            String line = value.toString();
            //           if line is valid and not header
            if (!(line.length() == 0) && !(line.substring(0, 6).equals("STN---"))) {
                String station_id = line.substring(0, 6);
                newKey.set(station_id);
                String date = line.substring(14, 22);
                float average_temperature = Float.parseFloat(line.substring(26, 30).trim());
                float number_of_reading = Float.parseFloat(line.substring(31, 33).trim());
                String valuesStations = String.format("%s,%.2f,%.2f", date, average_temperature, number_of_reading);
                newValue.set("T " + valuesStations);
                context.write(newKey,newValue);
            }
        }
    }
```

We have also created 4 mappers for one for each dataset. These mappers stay the same.

At the Reducer side, Red1 we read the each lines that are written to context from the mappers and classify it accordingly.

```
// The reduce takess the input from the above two mapper and outputs the key value pairs with StaionID and weatherData
1 usage
public static class Red1 extends Reducer<Text, Text, Text, Text> {


    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        ArrayList<Text> weatherList = new ArrayList<Text>();
        ArrayList<Text> stationList = new ArrayList<Text>();

        weatherList.clear();
        stationList.clear();
        for (Text t : values) {
            String tmp = t.toString();
            if (tmp.charAt(0) == 'T') {
                stationList.add(new Text(tmp.substring( beginIndex: 2)));
            } else if (tmp.charAt(0) == 'D') {
                weatherList.add(new Text(tmp.substring( beginIndex: 2)));
            }
        }

        if (!stationList.isEmpty() && !weatherList.isEmpty()) {
            for (Text A : stationList) {
                for (Text B : weatherList) {
                    context.write(A, B);
                }
            }
        }
    }
}
```

The Main function is as follows; we have used **MultipleInputs.addInputPath()** so that we can provide multiple datasets at once to the Mappers.

```
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf,  jobName: "Input_File_Construction");
        job.setJarByClass(InputFileConstruction.class);
        job.setReducerClass(Red1.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class, Map1.class);
        MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class, Map2.class);
        MultipleInputs.addInputPath(job, new Path(args[2]), TextInputFormat.class, Map3.class);
        MultipleInputs.addInputPath(job, new Path(args[3]), TextInputFormat.class, Map4.class);
        MultipleInputs.addInputPath(job, new Path(args[4]), TextInputFormat.class, Map5.class);
        Path outputPath = new Path(args[5]);

        FileOutputFormat.setOutputPath(job, outputPath);
        outputPath.getFileSystem(conf).delete(outputPath);
        System.exit(job.waitForCompletion( verbose: true) ? 0 : 1);
    }
}
```

The output of the MapReduce job is obtained as shown in the snippet below. This the input file that is constructed. This file will be used further MapReduce jobs.

```
20080314,65.20,15.00    CA,690150
20080101,43.20,24.00    CA,690150
20080101,43.20,24.00    CA,690150
20080102,44.50,24.00    CA,690150
20080102,44.50,24.00    CA,690150
20080103,44.50,24.00    CA,690150
20080103,44.50,24.00    CA,690150
20080104,45.40,20.00    CA,690150
20080104,45.40,20.00    CA,690150
20080105,56.40,23.00    CA,690150
20080105,56.40,23.00    CA,690150
20080106,47.70,15.00    CA,690150
20080106,47.70,15.00    CA,690150
20080107,48.10,18.00    CA,690150
20080107,48.10,18.00    CA,690150
20080108,42.30,24.00    CA,690150
20080108,42.30,24.00    CA,690150
20080109,45.50,22.00    CA,690150
20080109,45.50,22.00    CA,690150
20080110,47.30,22.00    CA,690150
20080110,47.30,22.00    CA,690150
20080111,49.80,23.00    CA,690150
20080111,49.80,23.00    CA,690150
20080112,51.30,23.00    CA,690150
20080112,51.30,23.00    CA,690150
20080113,50.80,19.00    CA,690150
20080113,50.80,19.00    CA,690150
20080114,50.00,21.00    CA,690150
20080114,50.00,21.00    CA,690150
20080115,46.00,24.00    CA,690150
20080115,46.00,24.00    CA,690150
20080116,47.40,16.00    CA,690150
20080116,47.40,16.00    CA,690150
```

Time Taken:

Total time spent by all map tasks (ms)=123691

Total time spent by all reduce tasks (ms)=23470

Mean Calculation:

The problem statement given is:

For stations within the United States, group the stations by state. For each state with readings.

a. find the average temperature recorded for each month (ignoring year).

We chose to solve this using One Mapper and One reducer using the Data file that was constructed previously.

ClassName: MeanTemppMonthpState

In this we need to calculate the Mean Temperature per Month per State.

The snippet is of the Mapper used (Map1). Reading each line of the input file and mapping the output as {StateNameMonth, AverageTemp} this will pass to the reducer.

```
public class MeanTemppMonthpState {
    1 usage
    public static class Map1 extends Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException
        {
            Text stateAndMonth = new Text();
            Text avgTemp = new Text();
            String line = value.toString();
            line = line.replaceAll( regex: "\\s+",  replacement: ",");
            String[] part = line.split( regex: ",");
// Loading each part of the output lines into variables
            String date = part[0];
            String temperature = part[1];
            String readings = part[2];

            String state = part[3];
            String station_id = part[4];
            String year = date.substring(0,4);
            String month = date.substring(4,6);
            int monthInt = Integer.parseInt(month);
//            To display the name of the month
            String date_with_month = year + "_" +Month.of(monthInt);
            stateAndMonth.set(state + date_with_month);

            float temp = Float.parseFloat(temperature);
            float numberOfReadings = Float.parseFloat(readings);
            float product = temp * numberOfReadings;
//            Setting Precision
            String values = String.format("%.2f,%.2f", product, numberOfReadings);
            avgTemp.set(values);
            context.write(stateAndMonth, avgTemp);
        }
    }
```

At the Reducer, these values are reduced to display the following output. We Iterate over the average temperature provided as value in the context per station for each month.

```
1 usage
public static class Red1 extends Reducer<Text, Text, Text, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException
    {
        float total_temp = 0;
        float total_read = 0;
        for (Text t : values) {
            String tmp = t.toString();
            String[] product_with_readings = tmp.split( regex: ",");
            float product_temp = Float.parseFloat(product_with_readings[0]);
            float n_readings = Float.parseFloat(product_with_readings[1]);
            total_read += n_readings;
            total_temp += product_temp;
        }
        total_temp = total_temp / total_read;
        String total_temp_sum_value = String.format("%.2f", total_temp);
        context.write(key, new Text(total_temp_sum_value));
    }
}
```

The Main Function is simple one with 2 arguments, the input and the output file.

```
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Job job = new Job(conf, jobName: "Calculating_MeanTemp_perMonth_perState");
    job.setJarByClass(MeanTemppMonthpState.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    job.setMapperClass(Map1.class);
    job.setReducerClass(Red1.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    Path out = new Path(args[1]);
    out.getFileSystem(conf).delete(out);
    job.waitForCompletion( verbose: true);
}
}
```

The output of the job is as follows:

```
CA2006_AUGUST     71.03
CA2006_DECEMBER   49.08
CA2006_FEBRUARY   52.06
CA2006_JANUARY    50.51
CA2006_JULY 74.52
CA2006_JUNE 70.87
CA2006_MARCH      49.92
CA2006_MAY  64.29
CA2006_NOVEMBER   56.37
CA2006_OCTOBER    60.94
CA2006_SEPTEMBER  67.98
CA2008_APRIL      57.75
CA2008_AUGUST     72.72
CA2008_DECEMBER   47.26
CA2008_FEBRUARY   49.98
CA2008_JANUARY    47.49
CA2008_JULY 71.57
CA2008_JUNE 68.91
CA2008_MARCH      54.27
CA2008_MAY  62.27
CA2008_NOVEMBER   57.44
CA2008_OCTOBER    63.53
CA2008_SEPTEMBER  69.38
CA2009_APRIL      57.10
CA2009_FEBRUARY   50.25
CA2009_JANUARY    51.04
CA2009_JULY 69.26
CA2009_JUNE 65.86
CA2009_MARCH      53.15
CA2009_MAY  64.76
CO2006_APRIL      46.88
CO2006_AUGUST     64.75
CO2006_DECEMBER   25.29
```

Time Taken:
Total time spent by all map tasks (ms)=13367
Total time spent by all reduce tasks (ms)=6011

Find the Highest and Lowest Average:

Find the months with the highest and lowest averages for that state. Order the states by the difference between the highest and lowest month average, ascending.
For each state, return:

a) *The state abbreviation, e.g., "CA"*
b) *The average temperature and name of the highest month, e.g., "90, July"*
c) *The average temperature and name of the lowest month, e.g., "50, January"*
d) *The difference between the two (from 2.b and 2.c), e.g., "40"*

For this problem, we have used the output of the previous MapReduce job as input.

ClassName: MonthHiLo

We have used one Mapper (Map1) and One Reducer (Red1).

The mapper is used to read the output file of the previous MapReduce job and split the input into State name and date and temperature on that day.

As shown Below:

```
public static class Map1 extends Mapper<Object, Text, Text, MinMaxTemperature> {


    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException
    {
        Text stateID = new Text();
        MinMaxTemperature valueOut = new MinMaxTemperature();

        String line = value.toString();
        line = line.replaceAll( regex: "\\s+",  replacement: ",");
        String[] part = line.split( regex: ",");

        String state = part[0].substring(0,2);
        String date = part[0].substring( beginIndex: 7);
        String temperature = part[1];

        stateID = new Text(state);
        String minTemDate = date + "," + temperature;
        String maxTemDate = date + "," + temperature;
        valueOut.minTemDate = minTemDate;
        valueOut.maxTemDate = maxTemDate;
        valueOut.difference = 0;
        context.write(stateID, valueOut);
    }
}
```

The reducer here uses the lines in the context to record the maximum temperature and the minimum temperature, calculates the difference and outputs the state name as key and the Month with highest and lowest temperature and the difference between them.

```
            if (resultminTemperature == 0 || minTemperature < resultminTemperature)
            {
                result.minTemDate = minlocaldate + "," + minTemperature;
            }

            if (resultmaxTemperature == 0 || maxTemperature > resultmaxTemperature)
            {
                result.maxTemDate = maxlocaldate + "," + maxTemperature;
            }

            if(result.minTemDate.length() > 0){
                String[] final_result_date_with_min_temp = result.minTemDate.split( regex: ",");
                if(final_result_date_with_min_temp.length > 0)
                {
                    finalresultminTemperature = Float.parseFloat(final_result_date_with_min_temp[1]);
                }

            }

            if(result.maxTemDate.length() > 0){
                String[] final_result_date_with_max_temp = result.maxTemDate.split( regex: ",");
                if(final_result_date_with_max_temp.length > 0)
                {
                    finalresultmaxTemperature = Float.parseFloat(final_result_date_with_max_temp[1]);
                }

            }
            float difference = finalresultmaxTemperature - finalresultminTemperature;
            result.difference = difference;
        }
        context.write(key, result);
```

The main function of the job is:

```
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    @SuppressWarnings("deprecation")
    Job job = new Job(conf, jobName: "Minmax_temperature");
    job.setJarByClass(MonthHiLo.class);

    job.setMapperClass(Map1.class);
    job.setCombinerClass(Red1.class);
    job.setReducerClass(Red1.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(MinMaxTemperature.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion( verbose: true) ? 0 : 1);
}
}
```

The Output of the map reduce job is as follows:

```
CA    DECEMBER,47.26,JULY,74.52,27.26
CO    JANUARY,19.95,JULY,69.03,49.08
CT    JANUARY,24.01,JULY,74.48,50.47
DC    JANUARY,32.23,JULY,80.26,48.03
DE    JANUARY,30.79,JULY,78.16,47.37
FL    JANUARY,59.05,AUGUST,82.45,23.4
GA    JANUARY,45.35,AUGUST,81.02,35.67
HI    FEBRUARY,68.54,AUGUST,78.4,9.86
IA    JANUARY,17.42,JULY,76.31,58.89
ID    JANUARY,21.62,JULY,73.17,51.55
IL    JANUARY,22.14,JULY,76.9,54.76
IN    JANUARY,21.38,JULY,76.17,54.79
KS    JANUARY,30.67,JULY,81.66,50.99
KY    JANUARY,31.17,AUGUST,77.42,46.25
LA    JANUARY,49.96,JULY,84.62,34.66
MA    JANUARY,23.29,JULY,73.35,50.06
MD    JANUARY,30.82,AUGUST,78.29,47.47
ME    JANUARY,13.08,JULY,68.68,55.6
MI    JANUARY,15.78,JULY,71.8,56.02
MN    JANUARY,9.51,JULY,74.08,64.57
MO    JANUARY,28.1,JULY,79.83,51.73
MS    JANUARY,45.54,AUGUST,82.84,37.3
MT    DECEMBER,18.53,JULY,74.43,55.9
NC    JANUARY,40.58,AUGUST,79.0,38.42
ND    DECEMBER,10.13,JULY,74.57,64.44
NE    JANUARY,21.9,JULY,78.13,56.23
NH    JANUARY,15.69,JULY,70.71,55.02
NJ    JANUARY,27.1,JULY,76.83,49.73
```

Time Taken:

Total time spent by all map tasks (ms)=4749

Total time spent by all reduce tasks (ms)=5379

Sorting the Output:

Order the states by the difference, ascending.

a) *Each row of your output should contain: The state abbreviation, the average temperature and name of the highest month, the average temperature and name of the lowest month and the difference between the two.*

ClassName: OutputSort

Here we use the output of the previous job and sort the input.

The method sorts the data that we obtained from executing the jar file to get the minimum and maximum difference for various states. To achieve this, we have made use of two jobs. The first job, the output of the previous execution is fed as the input, and the mapper in this class outputs the temperature difference as key and the entire row as value for each row.

```
public class OutputSort{
    public static class Map1 extends Mapper<Object, Text, FloatWritable, Text> {

        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException
        {
            FloatWritable difference = new FloatWritable();
            String line = value.toString();
            line = line.replaceAll( regex: "\\s+",  replacement: ",");
            String[] part = line.split( regex: ",");

            float diff = Float.parseFloat(part[5]);
            difference.set(diff);
            context.write(difference, value);
        }
    }
```

```java
public static class Red1 extends Reducer<FloatWritable, Text, Text, NullWritable> {
    public void reduce(FloatWritable key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException
    {
        for (Text t : values) {
            context.write(t, NullWritable.get());
        }
    }
}
```

The output is stored in an intermediate .lst file before the next job processes it. The subsequent job takes it as the input and sorts it based on the key to obtain the rows sorted by the keys, hence we get the output of the state's sorted by the differences in temperatures. The sorting of the data is handled by the TotalOrderPartitioner class as shown below.

We have made use of the SequenceFileInputFormat class before sending the input to the next job which packs the small file chunks into a single file which improves the file processing in hadoop file system.

```java
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Path inputPath = new Path(args[0]);
    Path partitionFile = new Path( pathString: args[1] + "_stage1.lst");
    Path outputStage = new Path( pathString: args[1] + "_stage2");
    Path outputOrder = new Path(args[1]);

    double sampling_rate = 0.001;

    FileSystem.get(new Configuration()).delete(partitionFile, b: true);
    FileSystem.get(new Configuration()).delete(outputStage, b: true);
    FileSystem.get(new Configuration()).delete(outputOrder, b: true);

    Job sampleJob = new Job(conf, jobName: "Staging");
    sampleJob.setJarByClass(OutputSort.class);

    sampleJob.setMapperClass(Map1.class);
    sampleJob.setNumReduceTasks(0);

    sampleJob.setOutputKeyClass(FloatWritable.class);
    sampleJob.setOutputValueClass(Text.class);

    FileInputFormat.addInputPath(sampleJob, inputPath);

    sampleJob.setOutputFormatClass(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setOutputPath(sampleJob, outputStage);

    int code = sampleJob.waitForCompletion( verbose: true) ? 0 : 1;
    if(code == 0){
        Job orderJob = new Job(conf, jobName: "Sorting the Output");
        orderJob.setJarByClass(OutputSort.class);
        orderJob.setMapperClass(Mapper.class);
        orderJob.setReducerClass(Red1.class);
        orderJob.setNumReduceTasks(1);
        orderJob.setPartitionerClass(TotalOrderPartitioner.class);
        TotalOrderPartitioner.setPartitionFile(orderJob.getConfiguration(), partitionFile);

        orderJob.setOutputKeyClass(FloatWritable.class);
        orderJob.setOutputValueClass(Text.class);
        orderJob.setInputFormatClass(SequenceFileInputFormat.class);
        SequenceFileInputFormat.setInputPaths(orderJob, outputStage);

        TextOutputFormat.setOutputPath(orderJob, outputOrder);
        orderJob.getConfiguration().set("mapreduce.textoutputformat.separator", "");
        InputSampler.writePartitionFile(orderJob, new InputSampler.RandomSampler( freq: .001, numSamples: 10000, maxSplitsSampled: 1));
        code = orderJob.waitForCompletion( verbose: true) ? 0 : 2;
    }
    FileSystem.get(new Configuration()).delete(partitionFile, b: false);
    FileSystem.get(new Configuration()).delete(outputStage, b: true);
    System.exit(code);
}
```

The output of the job is as follows: -

```
CA    DECEMBER,47.26,JULY,74.52,27.259998
LA    JANUARY,49.96,JULY,84.62,34.660004
WA    DECEMBER,33.11,JULY,68.29,35.18
OR    JANUARY,33.87,JULY,69.27,35.399998
GA    JANUARY,45.35,AUGUST,81.02,35.67
SC    JANUARY,44.39,JULY,80.7,36.309998
AL    JANUARY,44.87,AUGUST,81.67,36.8
TX    JANUARY,47.73,JULY,84.64,36.91
MS    JANUARY,45.54,AUGUST,82.84,37.299995
NC    JANUARY,40.58,AUGUST,79.0,38.42
AZ    JANUARY,43.71,JULY,82.15,38.440002
AK    JANUARY,16.95,JULY,57.87,40.92
TN    JANUARY,36.25,AUGUST,79.06,42.809998
NM    JANUARY,33.1,JULY,76.84,43.739998
AR    JANUARY,37.78,JULY,81.68,43.9
VA    JANUARY,34.09,AUGUST,78.02,43.929996
WV    JANUARY,28.17,AUGUST,73.02,44.85
KY    JANUARY,31.17,AUGUST,77.42,46.25
DE    JANUARY,30.79,JULY,78.16,47.370003
MD    JANUARY,30.82,AUGUST,78.29,47.47
OK    JANUARY,37.53,JULY,85.01,47.480003
RI    JANUARY,26.11,JULY,73.86,47.75
DC    JANUARY,32.23,JULY,80.26,48.030003
CO    JANUARY,19.95,JULY,69.03,49.079998
NJ    JANUARY,27.1,JULY,76.83,49.730003
MA    JANUARY,23.29,JULY,73.35,50.059998
```

Time Taken:

Total time spent by all map tasks (ms)=5078

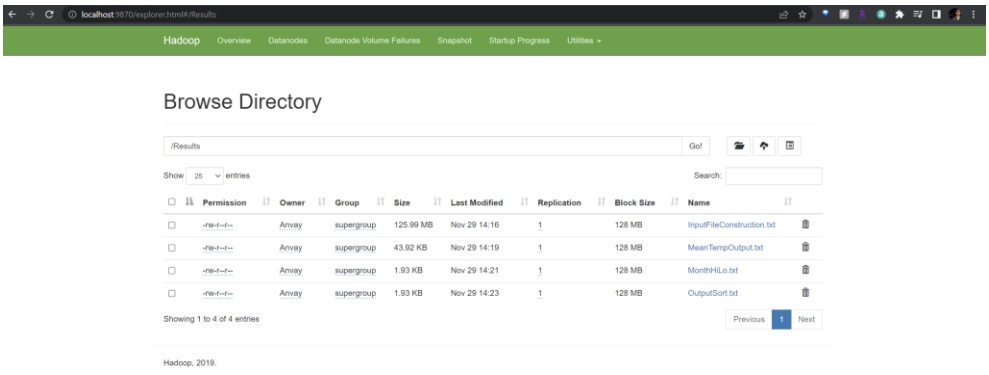
Total time spent by all reduce tasks (ms)=5459

Execution

To execute the jar files, use the mapReduceScriptRev.cmd provided in the files or execute the step1-step6 as mentioned in the README file. For further information on the execution refer the ReadMe file.

Results

After passing the input through multiple Mappers and Reducers we obtain the required results. The files are stored in the Results folder in the Hadoop cluster. As shown in the image below:



Google Drive Link for the Results:
https://drive.google.com/drive/folders/1jK59KibTqBv6X7M61zSdsZ896JgG3m4c?usp=sharing

References

GitHub Link for Our Project: https://github.com/anvayabn/DBMS_UCR_2022_FinalProj

For installing Hadoop 3.2.1 in Ubuntu 20.04.5: https://medium.com/@festusmorumbasi/installing-hadoop-on-ubuntu-20-04-4610b6e0391e

Sequence File Input Format: https://data-flair.training/forums/topic/what-is-sequencefileinputformat-in-hadoop-mapreduce/

Total Order Partitioner:

https://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/mapreduce/lib/partition/TotalOrderPartitioner.html