CSCI 650 “Cloud Computing”

The project
“Map Reduce: the average, minimum and maximum of the weighted sum of different classes”

Completed By Andrey Vasilyev

Table of Contents

Introduction 2
Implementation and design consideration of the Map Reduce code. 2
Verification of the results based on DataSetTest file 3
The execution of Map Reduce procedure on big DataSet.txt file. 4
Analysis of Map Reduce execution time 5
Conclusion 7
The Map Reduce code and comments 8

Introduction

The current report provides the description of the Map Reduce procedure and demonstrates how it was implemented based on weighted sum calculation from the DataSet input file. The Mapper functionality collects data from the input file, parses it and calculates weighted sum for every class. The Reducer pulls weighted sum from Mapper and then gets minimum, maximum and average values by grouping values based on the class number (key values). The report starts with the explanation of the code how it was designed and what functionality was utilized. Every single line of the code is commented out. All information about the code can be found at the end of the report in the reference section. In order to know if Map Reduce is implemented correctly and if it produces accurate results it was tested on the sample DataSetTest.txt file. Minimum, maximum and average values are provided after running of Map Reduce on the sample input file. After the verification stage, the Map Reduce is executed 10 times on 2, 4, 6, 8, 10, 12, 14 number of nodes on big DataSet.txt. All statistical information and the average execution time are provided in separate tables and the results are visualized on a graph. The analysis and evaluation of the results are made at the end of the report with the conclusions about the experience and knowledge that was gained from this project.

Implementation and design consideration of the Map Reduce code.

The Map Reduce procedure was implemented based on Map and Reduce classes from Hadoop libraries. The Mapper pulls records from the input file, parses the records and saves “class number”, “weighted sum” pairs in the “key” “value” variables. The Reducer processes “key” “value” pairs, received from the Mapper method, groups the data by class number, calculates average, minimum and maximum values and outputs the results into separate files. The multiple output is implemented by means of MultipleOutputs class from hadoop library. Such implementation provides better representation of the final results and separates minimum, maximum, average values from each other in separate files. The code and the comments with detail explanation are provided at the end of the report in the reference section.

Verification of the results based on DataSetTest file

In order to know if Map Reducer works correctly it was tested on the sample DataSetTest.txt file. The Map Reduce was executed on two nodes and the results, it generated, are provided in the following table. The table contains average, minimum and maximum values grouped by class number.

Class	AVG	MIN	MAX
1	48.29527	22.807	70.476
2	39.411	28.174	50.839
3	49.75538	38.51	67.734
4	45.34261	17.858	79.588
5	52.7759	34.826	74.462
6	46.36025	28.971	58.361
7	52.7516	34.076	76.987
8	55.20414	34.707	77.219
9	54.02844	40.278	66.265
10	57.40345	34.364	75.216

The execution of Map Reduce procedure on big DataSet.txt file.

As it was requested in the assignment the Map Reduce has also been executed on big DataSet.txt input file in order to measure the average running time for different combination of nodes. The Map Reduce was executed 10 times on 2, 4, 6, 8, 10, 12, 14 number of nodes separately. The running time is presented in the tables bellows in milliseconds. The graph, created based on the average running time for every number of nodes, is presented after the tables.

Class	Node 2
1	55983
2	46955
3	46217
4	37100
5	33009
6	37966
7	36995
8	36019
9	49377
10	49376
AVG	42899.7

Class	Node 4
1	34001
2	30967
3	43079
4	34062
5	31981
6	39139
7	38967
8	40983
9	39945
10	39573
AVG	37269.7

Class	Node 6
1	30948
2	39881
3	39843
4	31949
5	31863
6	41486
7	35883
8	45946
9	49110
10	36212
AVG	38312.1

Class	Node 8
1	28964
2	31882
3	35938
4	35956
5	34892
6	32085
7	33015
8	38944
9	33794
10	36945
AVG	34241.5

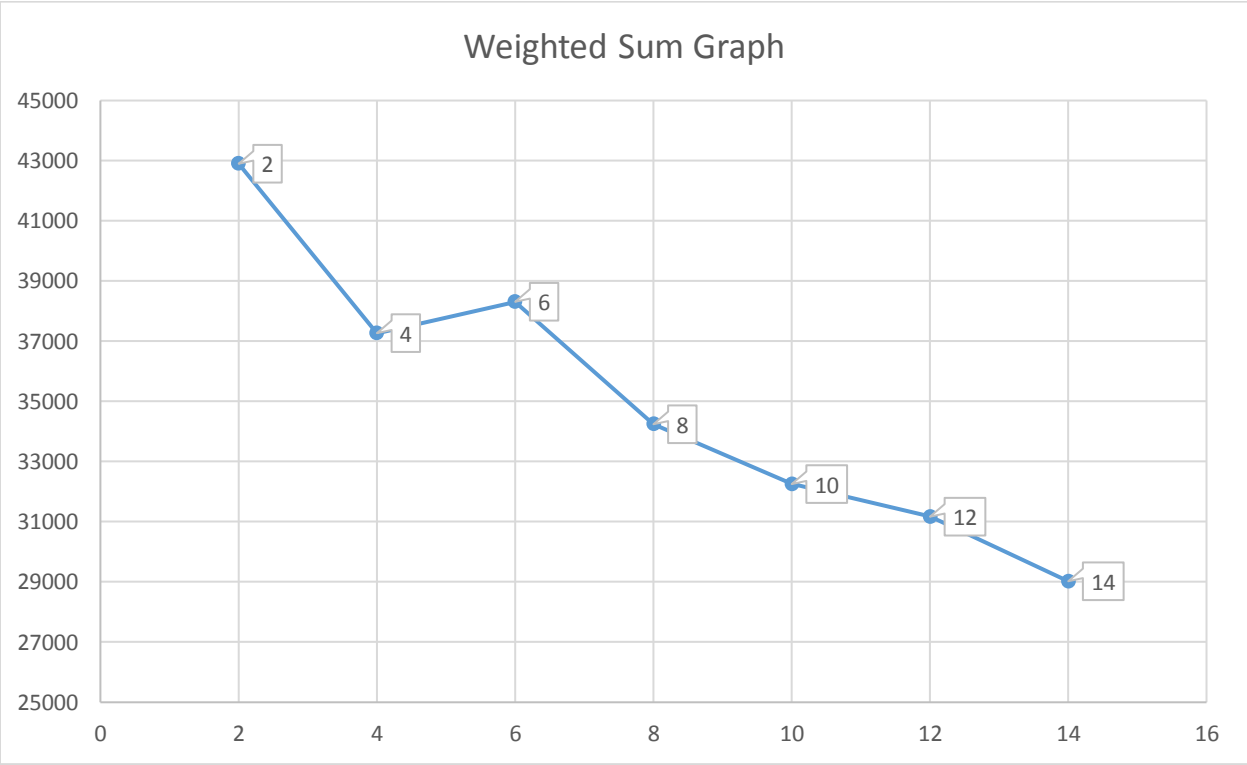
Class	Node 10
1	32946
2	24932
3	33902
4	39094
5	30869
6	36297
7	26866
8	34977
9	29817
10	32829
AVG	32252.9

Class	Node 12
1	27864
2	36928
3	25745
4	30846
5	29762
6	34814
7	29101
8	34875
9	27887
10	33869
AVG	31169.1

Class	Node 14
1	33091
2	30005
3	24961
4	24838
5	25790
6	33025
7	35975
8	31777
9	22940
10	27830
AVG	29023.2

Analysis of Map Reduce execution time

The graph that shows the average execution time is provided below:



By looking at the graph, we can state, that the Map Reduce executes faster and the running time is reduced as more nodes we add. It takes about 42899.7 milliseconds to run Map Reducer for 2 nodes and only 29023.2 milliseconds for 14 nodes. The difference is 13876.5 milliseconds. The graph demonstrates this difference pretty well. However, we can also see the spike on the graph for 6 nodes. The running time is increased when the number of nodes is increased from 4 to 6. Such time increase can be explained by the fact that the difference between the execution time on 4 and 6 nodes is very small and when we add 2 nodes only it doesn't make very big difference for Map Reduce execution standpoint. However, the graph also demonstrates that as more nodes we add as better performance we get and less time the Map Reduce spends on processing of big DataSet file. In spite of constant improvement in running time, we can also notice that graph gets less steep and becomes more gradual after Map Reduce is executed on 10, 12 and 14 nodes. The running time is still reduced but not so much as it was happened when the number of nodes increased from 2 to 4 and from 6 to 8. Such trend can bring additional information for consideration and can help to make certain conclusions on how many nodes would be better to have for the most efficient implementation of Map Reduce. In order to get explanation on what really causes of such changes in the running time I used different online resources and found some interesting facts about it.

1) "Hadoop online tutorial" provides a formula to calculate the number of nodes[1]. The formula is expressed with the following equation:

[1] Hadoop Online Tutorial retrieved from http://hadooptutorial.info/formula-to-calculate-hdfs-nodes-storage/

n = c * r * S / (1-i) / d, where
c - is compression ratio. It depends on the type of compression used (Snappy, LZOP)
r – replication factor (it is usually 3 in production cluster)
s- initial size of data need to be moved to Hadoop. This could be a combination of historical and incremental data.
i - intermediate data factor. It is usually 1/3 or ¼. It is Hadoop's intermediate working space deicated to storing intermediate results of Map Tasks.
d - disk space available per node. Here we also need to consider RAM, bandwidth, CPU configuration of nodes as well.
2) The online article "Sizing and Configuring your Hadoop Cluster" by Khaled Tannir[2] highlights the following major points that have to be taken into consideration during the planning of hadoop clusters and storage nodes:

- The MapReduce layer has two main prerequisites: input datasets must be large enough to fill a data block and split in smaller and independent data chunks (for example, a 10 GB text file can be split into 40,960 blocks of 256 MB each, and each line of text in any data block can be processed independently). The second prerequisite is that it should consider the data locality, which means that the MapReduce code is moved where the data lies, not the opposite.
- The most common practice to size a Hadoop cluster is sizing the cluster based on the amount of storage required. The more data is sent to the system, the more machines are required. Each time when we add a new node to the cluster, we get more computing resources in addition to the new storage capacity.

The online resources provide important points but there are many other factors that has to be taken into consideration during the calculation of number of storage nodes. However, this information is enough for us to make assumptions why the execution time of our Map Reduce speeds up with fewer nodes from 2 to 8 and then slows down when more nodes are added (from 8 to 14).
When we calculate the number of storage nodes we have to take into consideration such factors as physical characteristics of computers, their performance capacity, the number of available clusters and the size of the input data. In our case, since the size of our input DataSet.txt file is not significantly big it is processed more efficiently when we run it on fewer number of nodes, 6 or 8 nodes. However, if we try to add more nodes then we still get improvement in the execution time but we might not have full utilization of all resources that we use. With more nodes Map Reduce has to do more operations when data is distributed between master and slave nodes clusters.

[2] Khaled Tannir "Sizing and Configuring your Hadoop Cluster" retrieved from https://www.packtpub.com/books/content/sizing-and-configuring-your-hadoop-cluster

Conclusion

The project I completed for CSCI 650 “Cloud Computing” gave me a lot of great knowledge and real practical experience with Hadoop Map Reduce procedure. The cloud computing class has been very beneficial for me to know more about distributed systems and concurrent computing that can be scaled on different level from local to global. All theoretical information given in the class would not be so much helpful without applying this knowledge in the practical project. The Hadoop has been very actual topic in the recent years when many different industries has to deal with huge amount of data. Hadoop became the excellent solution to store big amount of data in distributed environment. The current project gave me the chance to execute Map Reduce and check how it works in real environment, how values are matched in the mapper and then reduced based on the key (class) values in the reducer. Beside this, I also learned the syntax, API and libraries in Hadoop. I was able to use all the advantages of hadoop libraries and implement Map Reduce code in Java programming language. After writing and compiling Map Reduce code I gained practical skills to write the software for processing and storing big amount of data in Hadoop environment.

The Map Reduce code and comments

```
import java.util.*;
import java.io.IOException;
import java.lang.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WeightedSum
{
  //Declare and implement the Mapper class
  public static class WSMapper extends Mapper<LongWritable, Text, Text, DoubleWritable>{

    //Declare the variable weightedSum as double writable data type to store weighted sum
    private final static DoubleWritable weightedSum = new DoubleWritable();
    //Declare keyValue variable as Text data type to store the key values.
    //In our case we store class number in the variable keyValue
    private Text keyValue = new Text();

    //Declare and implement the map method that maps keyValue (Class Number) and weighted sum pairs.
    //key - input parameter, LongWritable data type
    //value - input parameter, Text data type
    //context - output parameter, Context data type
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
```

```
//Convert input value parameter from Text data type to String for further processing
String line = value.toString();
//Use StringTokenizer class from Java Library to parse every line of the input file
//The values are pulled from the file based on tab separator
StringTokenizer s = new StringTokenizer(line,"\t");
//declare and initialize variables to calculate weighted sum
double sum=0.0;
int counter=1;
double multiplier=0.0;

//Class (Category) number is saved to the string variable lineNum by means of StringTokenizer function
//the menthod s.nextToken() allows to move cursor line by line
String lineNum = s.nextToken();

//The while loop pulls numbers seperated by a tab symbol
while(s.hasMoreTokens())
{
   //Counter is used to determine the column number and, based on the coumn number, apply appropriate multiplier
   if (counter==1)
   {
      //the multiplier is assigned the value 0.1 if column 1 is parsed
      multiplier=0.1;
   }
   if ((counter==2) || (counter==3) || (counter==5))
   {
      //the multiplier is assigned the value 0.2 if column is 1, 3 or 5 are parsed
      multiplier=0.2;
   }
   if (counter==4)
   {
      //the multiplier is assigned the value 0.3 if column is 4 is parsed
      multiplier=0.3;
   }
   //calculate the weighted sum by multipling the value of multiplier to every single value and add all values together
   sum += multiplier*Double.parseDouble(s.nextToken());
   counter++;
}
//assign class number to the KeyValue variable for the mapper to match key value pairs
```

```
      keyValue.set(lineNum);
      //assign value of the weighted sum to the weightedSum variable
      weightedSum.set(sum);
      //Write key - value pairs to mapper and use the context variable to save this data
      context.write(keyValue, weightedSum);
   }
}

//Declare and implement the Reducer class that reduces all values by class number (line number)
public static class WSReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

   //Declare result variable as DoubleWritable data type to save the results of the reduce procedure
   DoubleWritable result = new DoubleWritable();
   //Declare variable mos as MultipleOutputs Abstract Data Type to output calcualted results into seperate files
   //MultipleOutputs abstract data type is used to store Key - Values pairs as Text and DoubleWritable data types
   //MultipleOutputs ADT outputs data into seperate files for average, minimum and maximum values
   //MultipleOutputs improves readbility and processing time by storing output data in seperate files
   private MultipleOutputs<Text, DoubleWritable> mos;

   //Declare setup method to initialize MoltipuleOutputs mos variable
   @Override
   public void setup(Context context) throws IOException, InterruptedException {
         mos = new MultipleOutputs<Text, DoubleWritable>(context);
   }

   //Declare Reduce method to reduce ke-value pairs by class number and average, minimum, maximum values
   //key - input parameter, Text data type
   //values - input parameter, Iterable values of DoubleWritable data type
   //context - output parameter, Context data type
   public void reduce(Text key, Iterable<DoubleWritable> values, Context context) throws IOException, InterruptedException {

      //declare and initialize variables to store weighted sum, average, minimum and maximum values
      double sum=0.0;
      double avg=0.0;
      double min=100000000.0;
      double max=0.0;
      double tempVal=0.0;
      int counter=0;
```

```
        //iterate through weighted sum values and get minimum, maximum, average values
        for (DoubleWritable val : values) {
           tempVal = val.get();
           if (min > tempVal)
           {
              min = tempVal;
           }
           if(max < tempVal)
           {
              max = tempVal;
           }
           sum +=tempVal;
           counter++;
        }

        //save minimum value into the result variable
        result.set(min);
        //produce output key and result values in the file "min"
        mos.write("min", key, result);
        result.set(max);
        mos.write("max", key, result);
        avg=sum/counter;
        result.set(avg);
        mos.write("avg", key, result);
        //result.set(sum);
        //mos.write("sum", key, result);
    }
 }

 //Main function
public static void main(String args[])throws Exception
{
    //Declare startTime long variable to start timer and measure how many milliseconds it takes to run Map Reduce procedure
    long startTime = System.currentTimeMillis();
    //Instantiate object of Jobconf class to initialize the class Weightedsum for Map Reduce procedure
    JobConf conf = new JobConf(WeightedSum.class);
    //Instantiate object of the Job class and initialize it
```

```
    Job job = new Job(conf, "WeightedSumJob");
    //Add input file path to get input data from DataSet.txt file for map reduce procedure
    FileInputFormat.addInputPath(job, new Path("/user/vasilyev/weightedsum/input"));
    //Add output file path to store files with output data - average, minimum and maximum values
    FileOutputFormat.setOutputPath(job, new Path("/user/vasilyev/weightedsum/output"));
    job.setJarByClass(WeightedSum.class);
    //initiate Mapper and Reduce methods in order to execute Map and Redudce procedure
    job.setMapperClass(WSMapper.class);
    job.setReducerClass(WSReducer.class);
    //Specify input and output format
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    //Specify data types for the key and for the value
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);
    //Specify output files "min", "max" and "avg" for output values minimum, maximum and average respectively
    MultipleOutputs.addNamedOutput(job, "min", TextOutputFormat.class, Text.class, DoubleWritable.class);
    MultipleOutputs.addNamedOutput(job, "max", TextOutputFormat.class, Text.class, DoubleWritable.class);
    MultipleOutputs.addNamedOutput(job, "avg", TextOutputFormat.class, Text.class, DoubleWritable.class);
    //MultipleOutputs.addNamedOutput(job, "sum", TextOutputFormat.class, Text.class, DoubleWritable.class);

    //declare object of the FileSystem class to execute Map Reduce procedure on multiple nodes
    FileSystem fs = FileSystem.get(conf);
    long dataLength = fs.getContentSummary(new Path("/user/vasilyev/weightedsum/input")).getLength();
    //initialize numNodes variable to how many nodes will be used by map reduce procedure
    int numNodes = 2;
    FileInputFormat.setMaxInputSplitSize(job, (long) (dataLength / numNodes));
    job.setNumReduceTasks(numNodes/2);
    job.waitForCompletion(true);
    //determine the end tme of map reduce procedure and save it to the endTime variable
    long endTime = System.currentTimeMillis();
    //output the difference between the beginning and the end time of map reduce procedure in milleseconds
    System.out.println("Time taken to run the program is " + (endTime - startTime) + " milliseconds");
  }
}
```