Lab 02: MapReduce programming

CSC14118 - Introduction to Big Data - 20KHMT1

The Girls

2023-04-01

Contents

1 Lab 02: MapReduce programming

1.1 Work assignment

Student ID	Full name	Work assignment	%
20127011	Le Tan Dat	Problem 1, 4, report	25%
20127438	Le Nguyen Nguyen Anh	Problem 2, 6, 8, report	25%
20127458	Dang Tien Dat	Problem 3, 7, 10 report	25%
20127627	Nguyen Quoc Thang	Problem 5, 9, report	25%

-> Our team not only consulted the lab requirement file in drive lab 2 but also solved problems

1.2 Explain the code in details.

1.2.1 Problem 1: WordCount

1.2.1.1 *Mapper*

```
public static class WordCountMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable number = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context) throws IOException,
    ↪  InterruptedException {
        StringTokenizer token = new StringTokenizer(value.toString());
        while (token.hasMoreTokens()) {
            word.set(token.nextToken());
            context.write(word, number);
        }
    }
}
```

- The mapper class is a subclass of the Mapper class.
- 2 variables are declared:
  - `number` is a constant variable with value 1 to count the number of words.
  - `word` is a variable of type Text to store the word.
- Idea of the mapper class:
  - The mapper class will read the input file line by line.
  - Then, the mapper class will split the line into words by using the `StringTokenizer` class.
  - After that, the mapper class will loop through the words and write the word and the number 1 to the context.
  - The context will be used to write the output file.

1.2.1.2 *Reducer*

```
public static class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
    ↪  IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

- The reducer class is a subclass of the Reducer class to reduce the output of the mapper class.
- 1 variable is declared:
  - `result` is a variable of type IntWritable to store the number of words is counted.
- Idea of the reducer class:
  - The reducer class will read the output file of the mapper class line by line.
  - Then, the reducer class will split the line into words and the number of words by using the `StringTokenizer` class.
  - After that, initialize the variable `sum` to `0`. This variable is used to count the number of words.

- The reducer class will loop through the words and count the number of words by using the variable `sum`.
- Finally, set the value of the variable `result` to `sum` and write the word and the number of words to the context.

1.2.1.3 Guide to run the program

- Step 1: Create file `WordCount.java` in the folder `src` of the project.
- Step 2: Create file `wordcount.txt` in the folder `data` of the project and then `put` the file to the local HDFS by using the command

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/wordcount.txt /input
  ```

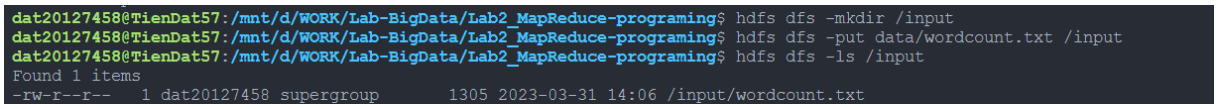


Figure 1.1: Step 2

- Step 3: Compile and run the program by using the command
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main WordCount.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar WordCount*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar WordCount /input /output`
- Step 4: Check the result by using the command

  ```
  hdfs dfs -cat /output/part-r-00000
  ```

```
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src$ cd problem01
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem01$ ls
WordCount.java
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem01$ hadoop com.sun.tools.javac.Main WordCount.java
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem01$ jar cf wc.jar WordCount*.class
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem01$ hadoop jar wc.jar WordCount /input /output
2023-03-31 14:08:07,972 INFO impl.MetricsConfig: Loaded properties from hadoop-metrics2.properties
2023-03-31 14:08:08,170 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2023-03-31 14:08:08,170 INFO impl.MetricsSystemImpl: JobTracker metrics system started
2023-03-31 14:08:08,725 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execu
te your application with ToolRunner to remedy this.
2023-03-31 14:08:08,910 INFO input.FileInputFormat: Total input files to process : 1
2023-03-31 14:08:08,953 INFO mapreduce.JobSubmitter: number of splits:1
2023-03-31 14:08:09,135 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local1220648815_0001
2023-03-31 14:08:09,135 INFO mapreduce.JobSubmitter: Executing with tokens: []
2023-03-31 14:08:09,355 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
2023-03-31 14:08:09,356 INFO mapreduce.Job: Running job: job_local1220648815_0001
2023-03-31 14:08:09,361 INFO mapred.LocalJobRunner: OutputCommitter set in config null
2023-03-31 14:08:09,385 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2023-03-31 14:08:09,385 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cl
eanup failures: false
2023-03-31 14:08:09,387 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
2023-03-31 14:08:09,453 INFO mapred.LocalJobRunner: Waiting for map tasks
2023-03-31 14:08:09,456 INFO mapred.LocalJobRunner: Starting task: attempt_local1220648815_0001_m_000000_0
2023-03-31 14:08:09,518 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2023-03-31 14:08:09,518 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cl
eanup failures: false
2023-03-31 14:08:09,553 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
2023-03-31 14:08:09,563 INFO mapred.MapTask: Processing split: hdfs://localhost:9000/input/wordcount.txt:0+1305
2023-03-31 14:08:09,718 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
2023-03-31 14:08:09,718 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
2023-03-31 14:08:09,718 INFO mapred.MapTask: soft limit at 83886080
2023-03-31 14:08:09,718 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
2023-03-31 14:08:09,718 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
2023-03-31 14:08:09,725 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
2023-03-31 14:08:10,365 INFO mapreduce.Job: Job job_local1220648815_0001 running in uber mode : false
2023-03-31 14:08:10,366 INFO mapreduce.Job:  map 0% reduce 0%
2023-03-31 14:08:10,572 INFO mapred.LocalJobRunner:
2023-03-31 14:08:10,601 INFO mapred.MapTask: Starting flush of map output
```

Figure 1.2: Step 3

1.2.2 Problem 2: WordSizeWordCount Program

1.2.2.1 *Mapper*

```java
public static class WordSizeWordCountMapper extends Mapper<Object, Text, IntWritable, Text> {
    public void map(Object key, Text value, Context context) throws IOException,
    ↪  InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            String curToken = itr.nextToken();
            IntWritable wordSize = new IntWritable(curToken.length());
            context.write(wordSize, new Text(curToken));
        }
    }
}
```

- The mapper class is a subclass of the Mapper class.
- Declared variables:
  - `wordSize` is a variable of type IntWritable to store the length of the word.
  - `curToken` is a variable of type String to store the word.
  - `itr` is a variable of type StringTokenizer to split the line into words.
- Idea of the mapper class:
  - The mapper class will read the input file line by line.

```
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem01$ hadoop fs -cat /output/part-r-00000
In      1
Infinite,       1
Nobody  1
This    1
We      1
When    1
Whether 1
Worry,  1
Years   1
Youth   2
a       11
adventure       1
aerials 2
and     8
appetite        1
appetite,       1
are     4
as      3
at      2
back    1
beauty, 1
being's 1
body    1
bows    1
but     2
by      2
catch   1
center  1
cheeks, 1
cheer,  1
child-like      1
courage 2
covered 1
cynicism        1
deep    1
deserting       1
die     1
down,   1
dust.   1
ease.   1
eighty. 1
emotions;       1
enthusiasm      1
even    1
every   1
exists  1
fear,   1
for     1
```

Figure 1.3: Step 4

- Then, the mapper class will split the line into words by using the `StringTokenizer` class and save the words to the variable `itr`.
- After that, the mapper class will loop through the words and write the `length` of the word and the `word` to the context.
- The context will be used to write the output file.

1.2.2.2 *Combiner*

```
public static class Combiner extends Reducer<IntWritable,Text,IntWritable, Text> {
    private IntWritable result = new IntWritable();

    public void reduce(IntWritable key, Iterable<Text> values,Context context)
      throws IOException,InterruptedException {
      int sum = 0;

      for(Text x: values)
      {
          sum += 1;
      }

      result.set(sum);

      context.write(key, new Text(result.toString()));
    }
  }
```

- The `Combiner` class inherits from `Reducer` class.
- Declared variables:
  - `result` is a variable of type IntWritable to store the length of the word.
  - `sum` is a variable of type String to store the word.
- Idea of the combiner class:
  - The combiner class will read the input file line by line.
  - Then, the mapper class will split the line into words by using the `StringTokenizer` class and save the words to the variable `itr`.
  - After that, the mapper class will loop through the words and write the `length` of the word and the `word` to the context.
  - The context will be used to write the output file.

1.2.2.3 *Reducer*

```
public static class WordSizeWordCountReducer extends Reducer<IntWritable, Text, IntWritable,
↪  IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(IntWritable key, Iterable<Text> values, Context context) throws
    ↪  IOException, InterruptedException {
      int sum = 0;
      for (Text val : values) {
        sum += Integer.parseInt(val.toString());
      }
      result.set(sum);
      context.write(key, result);
    }
}
```

- The `WordSizeWordCountReducer` class is a subclass of the `Reducer` class to reduce the output of the mapper class.
- 1 variable is declared:
  - `result` is a variable of type IntWritable to store the number of words is counted.
- Idea of the reducer class:
  - The reducer class reads the output file of the mapper class line by line.
  - Then, the reducer class will split the line into words and the number of words by using the `StringTokenizer` class.
  - After that, initialize the variable `sum` to `0`. This variable is used to count the number of words.
  - The reducer class will using loop to traversal and count the number of the words with the same length accumulating in `sum`.
  - Finally, set the value of the value of `sum` to `result` and write the tuple <length, number of words> to the context.

1.2.2.4 *Guide to run the program*

- Step 1: Create file WordSizeWordCount.java in the folder src of the project.
- Step 2: Create file wordcount.txt in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/wordcount.txt /input
  ```

- Step 3: Compile and run the program by using the command
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main WordSizeWordCount.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar WordSizeWordCount*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar WordSizeWordCount /input /output`
- Step 4: Check the result by using the command `hdfs dfs -cat /output/part-r-`

```
NguyenAnh20127438@master:~$ hadoop fs -cat /problem2/output/part-r-00000
1       9460
2       40612
3       55193
4       44402
5       33864
6       25875
7       21186
8       14205
9       9520
10      6120
11      3606
12      1970
13      1088
14      507
15      229
16      106
17      75
18      27
19      19
20      10
21      10
22      4
23      1
24      6
25      2
26      3
27      2
28      2
29      1
30      2
31      1
34      3
37      2
39      1
53      1
71      2
00000
```

1.2.3 Problem 3: WeatherData Program

1.2.3.1 *Mapper*

```
public static class MaxTemperatureMapper extends MapReduceBase implements Mapper<LongWritable,
↪  Text, Text, Text> {
    @Override
    public void map(LongWritable arg0, Text Value, OutputCollector<Text, Text> output,
    ↪  Reporter arg3) throws IOException {
        String line = Value.toString();
        String date = line.substring(6, 14);
```

```
        float temp_Max = Float.parseFloat(line.substring(39, 45).trim());
        float temp_Min = Float.parseFloat(line.substring(47, 53).trim());

        if (temp_Max > 40.0) {
        output.collect(new Text("Hot Day " + date),
                new Text(String.valueOf(temp_Max)));
        }
        if (temp_Min < 10) {
        output.collect(new Text("Cold Day " + date),
                new Text(String.valueOf(temp_Min)));
        }
    }
}
```

- The `MaxTemperatureMapper` class is a subclass of the `MapReduceBase` class which implements `Mapper` class.
- Declared variables:
  - `line` is a variable of type String storing Value parameter.
  - `date` is a variable of type String extracting date information from the line.
  - `temp_Max` is a float variable splitting the line to take highest temperature value.
  - `temp_Min` is a float variable splitting the line to take lowest temperature value.
- Idea of the mapper class:
  - The mapper class will read the input file line by line and each line is stored in `line` variable.
  - Then, from date to highest and lowest temperature informations from each line will be splitted and stored.
  - After that, consider to problem conditions and collect those cases into output. #### ***Reducer***

```
public static class MaxTemperatureReducer extends MapReduceBase implements Reducer<Text, Text,
↪  Text, Text> {
     @Override
     public void reduce(Text Key, Iterator<Text> Values, OutputCollector<Text, Text> output,
     ↪  Reporter arg3) throws IOException {
       float max = Float.parseFloat(Values.next().toString());
       while (Values.hasNext()) {
           float temp = Float.parseFloat(Values.next().toString());
           if (temp > max) {
               max = temp;
           }
       }
       output.collect(Key, new Text(String.valueOf(max)));
     }
   }
```

- The `MaxTemperatureReducer` class extends to `MapReduceBase` class which implementing `Reducer` class.
- Declared variables:
  - `max` is a float variable storing current Value parameter.
  - `temp` is a float variable storing temporary value for comparision.
- Idea of the mapper class:
  - The mapper class will read the input file line by line and each line is stored in `line` variable.
  - Then, create a current value while using loop to travesal and continue creating temporary value to compare aim to finding max value.
  - Finally, collect found value into output.

1.2.3.2 *Guide to run the program*

- Step 1: Create file WeatherData.java in the folder src of the project.
- Step 2: Create file weather_dât.txt in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/weather_data.txt /input
  ```

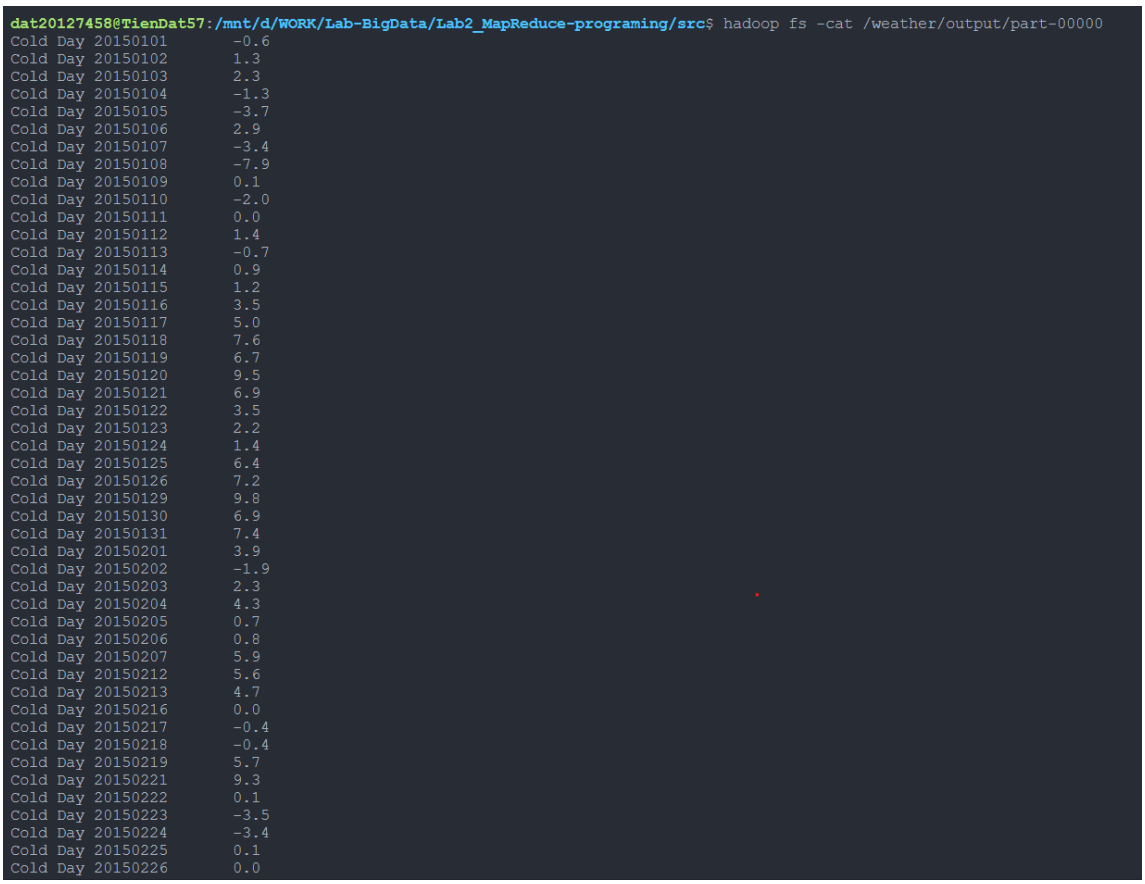
- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main WeatherData.java
    ```

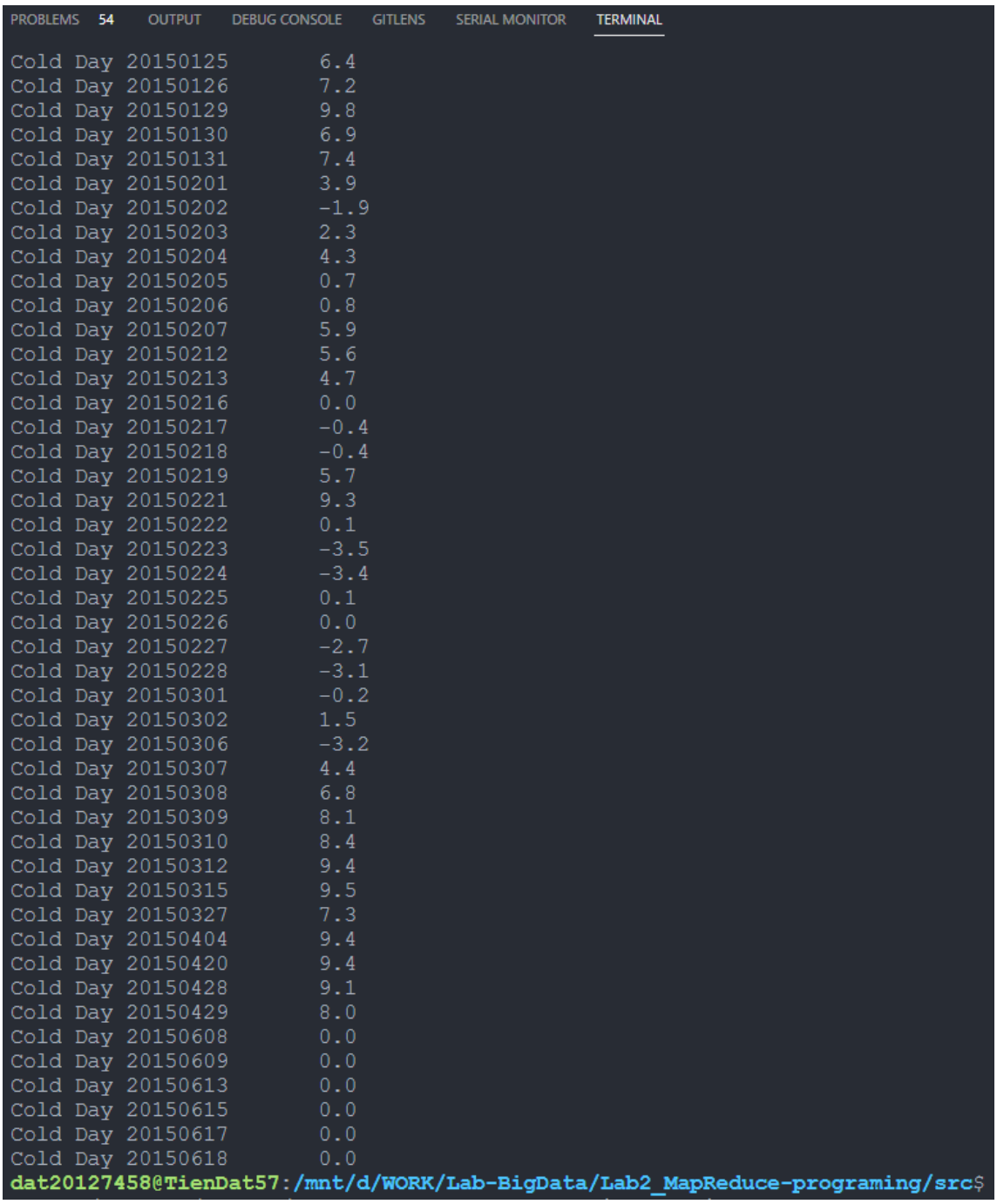
  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar WeatherData*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar WeatherData /input /output`
- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-`


dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src$ hadoop fs -cat /weather/output/part-00000
Cold Day 20150101 -0.6
Cold Day 20150102 1.3
Cold Day 20150103 2.3
Cold Day 20150104 -1.3
Cold Day 20150105 -3.7
Cold Day 20150106 2.9
Cold Day 20150107 -3.4
Cold Day 20150108 -7.9
Cold Day 20150109 0.1
Cold Day 20150110 -2.0
Cold Day 20150111 0.0
Cold Day 20150112 1.4
Cold Day 20150113 -0.7
Cold Day 20150114 0.9
Cold Day 20150115 1.2
Cold Day 20150116 3.5
Cold Day 20150117 5.0
Cold Day 20150118 7.6
Cold Day 20150119 6.7
Cold Day 20150120 9.5
Cold Day 20150121 6.9
Cold Day 20150122 3.5
Cold Day 20150123 2.2
Cold Day 20150124 1.4
Cold Day 20150125 6.4
Cold Day 20150126 7.2
Cold Day 20150129 9.8
Cold Day 20150130 6.9
Cold Day 20150131 7.4
Cold Day 20150201 3.9
Cold Day 20150202 -1.9
Cold Day 20150203 2.3
Cold Day 20150204 4.3
Cold Day 20150205 0.7
Cold Day 20150206 0.8
Cold Day 20150207 5.9
Cold Day 20150212 5.6
Cold Day 20150213 4.7
Cold Day 20150216 0.0
Cold Day 20150217 -0.4
Cold Day 20150218 -0.4
Cold Day 20150219 5.7
Cold Day 20150221 9.3
Cold Day 20150222 0.1
Cold Day 20150223 -3.5
Cold Day 20150224 -3.4
Cold Day 20150225 0.1
Cold Day 20150226 0.0

00000


PROBLEMS 54
OUTPUT
DEBUG CONSOLE
GITLENS
SERIAL MONITOR
TERMINAL
Cold Day 20150125 6.4
Cold Day 20150126 7.2
Cold Day 20150129 9.8
Cold Day 20150130 6.9
Cold Day 20150131 7.4
Cold Day 20150201 3.9
Cold Day 20150202 -1.9
Cold Day 20150203 2.3
Cold Day 20150204 4.3
Cold Day 20150205 0.7
Cold Day 20150206 0.8
Cold Day 20150207 5.9
Cold Day 20150212 5.6
Cold Day 20150213 4.7
Cold Day 20150216 0.0
Cold Day 20150217 -0.4
Cold Day 20150218 -0.4
Cold Day 20150219 5.7
Cold Day 20150221 9.3
Cold Day 20150222 0.1
Cold Day 20150223 -3.5
Cold Day 20150224 -3.4
Cold Day 20150225 0.1
Cold Day 20150226 0.0
Cold Day 20150227 -2.7
Cold Day 20150228 -3.1
Cold Day 20150301 -0.2
Cold Day 20150302 1.5
Cold Day 20150306 -3.2
Cold Day 20150307 4.4
Cold Day 20150308 6.8
Cold Day 20150309 8.1
Cold Day 20150310 8.4
Cold Day 20150312 9.4
Cold Day 20150315 9.5
Cold Day 20150327 7.3
Cold Day 20150404 9.4
Cold Day 20150420 9.4
Cold Day 20150428 9.1
Cold Day 20150429 8.0
Cold Day 20150608 0.0
Cold Day 20150609 0.0
Cold Day 20150613 0.0
Cold Day 20150615 0.0
Cold Day 20150617 0.0
Cold Day 20150618 0.0
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src$

1.2.4 Problem 4: Patent Program

1.2.4.1 *Mapper*

```
public static class Map extends Mapper<LongWritable, Text, Text, Text> {
    private Text _key = new Text();
    private Text _value = new Text();
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
    ↪  InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line," ");
        while (tokenizer.hasMoreTokens()) {
            String jiten= tokenizer.nextToken();
            _key.set(jiten);
            String jiten1= tokenizer.nextToken();
            _value.set(jiten1);
            context.write(_key,_value);
        }
    }
}
```

- The `Map` class is a subclass of the `MapReduceBase` class which implements `Mapper` class.
- Declared variables:
  - `_key` is a Text variable storing Value parameter.
  - `_value` is a Text variable storing Value parameter.
  - `line` is a float variable splitting the line to take highest temperature value.
  - `jiten` is a float variable splitting the line to take lowest temperature value.
  - `jiten1` is a float variable splitting the line to take lowest temperature value.
- Idea of the mapper class:
  - The mapper class will read the input file line by line and each line is stored in `line` variable.
  - Then, from date to highest and lowest temperature informations of each line will be splitted and stored.
  - After that, consider to problem conditions and collect those cases into output.

1.2.4.2 *Reducer*

```
public static class Reduce extends Reducer<Text, Text, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
    ↪  InterruptedException {
```

```
        int sum = 0;
        for(Text x : values){
            sum++;
        }
        context.write(key, new IntWritable(sum));
    }
}
```

- `int sum = 0`: will be used to store the count of values associated with the key.
- `for(Text x : values)`: The Reducer iterates through the values using a for-each loop, and for each value, it increments the sum variable by one.
- `context.write(key, new IntWritable(sum))`: The Reducer writes the key and the sum to the output.

1.2.4.3 *Guide to run the program*

- Step 1: Create file Patent.java in the folder src of the project.
- Step 2: Create file patent.txt in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/patent.txt /input
  ```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main Patent.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar Patent*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar Patent /input /output`

- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`

```
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem04$ hadoop fs -cat /output/part-r-00000
1       13
2       10
3       4
dat20127458@TienDat57:/mnt/d/WORK/Lab-BigData/Lab2_MapReduce-programing/src/problem04$
```

1.2.5 Problem 5: MaxTemp Program

1.2.5.1 *Mapper*

```
public static class MaxTempMapper
  extends Mapper<Object, Text, LongWritable, IntWritable> {
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        String year = itr.nextToken();
        String temp = itr.nextToken();
        context.write(new LongWritable(Integer.parseInt(year)), new
↪   IntWritable(Integer.parseInt(temp)));
    }
}
```

- The MaxTempMapper class is defined as a static class extended from the Hadoop Mapper class, with three generic parameters defined as Object, Text and LongWritable, IntWritable.
- The map() function is overridden to perform input stream analysis, decompose the year and temperature respectively, and generate key/value pairs to write down the output via the Context object.
  - The key is the year, and the value is the temperature.
  - When this Mapper is executed, it parses the input data line by line and generates key/value pairs corresponding to the `year` and `temperature`, then writes them down to the output via the Context object.

1.2.5.2 *Reducer*

```
public static class MaxTempReducer extends Reducer<LongWritable, IntWritable, LongWritable,
↪   IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(LongWritable key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int max = 0;
        for (IntWritable val : values) {
            if (val.get() > max) {
            max = val.get();
            }
        }
        result.set(max);
        context.write(key, result);
    }
}
```

- The reducer has a single reduce() method that takes in a LongWritable key(`year`), an Iterable of IntWritable values(`temparature`), and a Context object.
- Inside the reduce() method, the values are iterated over to find the maximum temperature value. The max temperature value is stored in a local variable “max”. If a new maximum temperature value is found, the variable “max” is updated.
- Finally, the result value is set to the maximum temperature value and is written to the output using the context.write() method with the input key.

1.2.5.3 *Guide to run the program*

- Step 1: Create file MaxTemp.java in the folder src of the project.
- Step 2: Create file patent.txt in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/temparature /input
  ```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main MaxTemp.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar MaxTemp*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar MaxTemp /input /output`
- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`

```
NguyenAnh20127438@master:~$ hadoop fs -cat /problem5/output/part-r-00000
1900    36
1901    48
1902    49
```

1.2.6 Problem 6: AverageSalary Program

1.2.6.1 *Mapper*

```
public static class AverageSalaryMapper
  extends Mapper<Object, Text, Text, FloatWritable> {
    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      String deptId = itr.nextToken();
      String empId = itr.nextToken();
      String salary = itr.nextToken();
      context.write(
        new Text(deptId),
        new FloatWritable(Float.parseFloat(salary))
      );
    }
  }
```

- Mapper class named AverageSalaryMapper which extends the Mapper.
- The map method of this class takes three arguments: Object key, Text value, and Context context. Object key and Text value are the key and value of the input data. Context context is the output of map method.
- The value parameter is tokenized using a StringTokenizer object. The nextToken() method of the StringTokenizer object is called three times to extract the department ID, employee ID, and salary from the input text line.
- The map method writes the output key-value pair to the Context object using the write() method. The Text object representing the department ID is passed as the output key, while the FloatWritable object representing the salary is passed as the output value.

1.2.6.2 *Reducer*

```
  public static class AverageSalaryReducer
    extends Reducer<Text, FloatWritable, Text, FloatWritable> {
    public void reduce(
      Text key,
      Iterable<FloatWritable> values,
      Context context
    ) throws IOException, InterruptedException {
      float sum = 0;
      float count = 0;
      for (FloatWritable val : values) {
        sum += val.get();
```

```
        count += 1;
      }
      context.write(key, new FloatWritable(sum / count));
    }
}
```

- Reducer class named AverageSalaryReducer which extends the Reducer.
- The reduce method of this class takes three arguments: Text key, Iterable values, and Context context. Text key and Iterable values are the key and value of the input data. Context context is the output of reduce method.
- The reduce method iterates over the values and sums up the salary of the department. It also counts the number of employees in the department.
- Then the average salary of the department is calculated by dividing the total salary by the number of employees.
- The reduce method writes the output key-value pair to the Context object using the write() method.

1.2.6.3 *Guide to run the program*

- Step 1: Create file AverageSalary.java in the folder src of the project.
- Step 2: Create file salary in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/salary /input
  ```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main AverageSalary.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar AverageSalary*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar AverageSalary /input /output`

- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`

```
NguyenAnh20127438@master:~$ hadoop fs -cat /problem6/output/part-r-00000
1       54400.0
2       75400.0
3       64000.0
4       89400.0
5       69400.0
6       98400.0
```

1.2.7 Problem 7: De Identify HealthCare Program

1.2.7.1 *Mapper*

```
public static class DeIndentifyMap extends Mapper<Object, Text, NullWritable, Text> {
  public void map(Object key, Text value, Context context) throws IOException,
  ↪  InterruptedException {
      String[] tokens = value.toString().split(",");
      List<Integer> list = new ArrayList<Integer>();
      Collections.addAll(list, encryptCol);
      String result = "";
      for (int i = 0; i < tokens.length; i++) {
        if (list.contains(i + 1)) {
            if (result.length() > 0)
              result += ",";
            result += encrypt(tokens[i], samplekey);
        } else {
            if (result.length() > 0)
              result += ",";
            result += tokens[i];
        }
      }
      context.write(NullWritable.get(), new Text(result.toString()));
  }
}
```

- Mapper class named Map which extends the Mapper.
- The map method of this class takes three arguments: Object key, Text value, and Context context. Object key and Text value are the key and value of the input data. Context context is the output of map method.
- The value parameter is tokenized using a StringTokenizer object. The nextToken() method of the StringTokenizer object is called three times to extract the department ID, employee ID, and salary from the input text line.
- The map method writes the output key-value pair to the Context object using the write() method. The Text object representing the department ID is passed as the output key, while the FloatWritable object representing the salary is passed as the output value.

```
public static String encrypt(String strToEncrypt, byte[] key) {
      try {
         Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
         final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
         cipher.init(Cipher.ENCRYPT_MODE, secretKey);
         final String encryptedString =
         ↪  Base64.encodeBase64String(cipher.doFinal(strToEncrypt.getBytes()));
         return encryptedString.trim();
      } catch (Exception e) {
         logger.error("Error while encrypting", e);
      }
      return null;
   }
```

- The encrypt method takes two parameters: the string to encrypt and the key.
- This uses the AES algorithm to encrypt the data. The Base64 encoding is used to encode the encrypted data.
- Finaly, this returns the encrypted string.

1.2.7.2 *Guide to run the program*

- Step 1: Create file DeIdentifyData.java in the folder src of the project.
- Step 2: Create file salary in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/healthcare.csv /input
  ```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

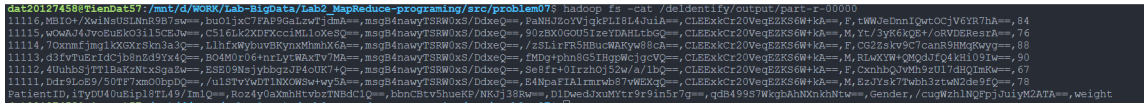
    ```
    hadoop com.sun.tools.javac.Main DeIdentifyData.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar DeIdentifyData*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar DeIdentifyData /input /output`

- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`



1.2.8 Problem 8: Music Track Program

1.2.8.1 *Mapper*

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

    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
      //split the line by | expression
      String[] parts = value.toString().split("[|]");

      // the first ele is the userId and the second ele is the trackId
      context.write(new Text(parts[1]), new Text(parts[0]));
    }
  }
```

- Like the question above, Mapper class named UniqueListenersMapper which extends the Mapper
- Map will split the line by "|" expression. The first element is the userId and the second element is the trackId. Then we write the output as `<trackId, userId>`.

1.2.8.2 *Reducer*

```
public static class UniqueListenersReducer
    extends Reducer<Text, Text, Text, IntWritable> {

    public void reduce(Text key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {
      HashSet<String> set = new HashSet<String>();

      // add all listeners to the set to get all unique listeners
      for (Text val : values) {
        set.add(val.toString());
      }
      context.write(key, new IntWritable(set.size()));
    }
  }
```

- Reducer class named UniqueListenersReducer which extends the Reducer.
- Reduce recieves the output of the map: `<trackId, userId>`. It will add all userIds to the set to get all unique listeners. Then it will write the output as `<trackId, number of unique listeners>`.

1.2.8.3 *Guide to run the program*

- Step 1: Create file UniqueListeners.java in the folder src of the project.
- Step 2: Create file salary in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

  ```
  hdfs dfs -put data/LastFMlog.txt /input
  ```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main UniqueListeners.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar UniqueListeners*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar UniqueListeners /input /output`
- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`

```
NguyenAnh20127438@master:~$ hadoop fs -cat /problem8/output/part-r-00000
222     1
223     1
225     2
```

1.2.9 Problem 9: Telecom Call Data Record Program

1.2.9.1 *Constant*

```
public class CDRCDRConstants {
  public static final int FROM_PHONE_NUMBER = 0;
  public static final int TO_PHONE_NUMBER = 1;
  public static final int CALL_START_TIME = 2;
  public static final int CALL_END_TIME = 3;
  public static final int STD_FLAG = 4;
}
```

1.2.9.2 *Mapper*

```
public class CDRMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  private Text fromPhoneNumber = new Text();
  private IntWritable callDuration = new IntWritable();

  public void map(LongWritable key, Text value, Context context) throws IOException,
  ↪  InterruptedException {
    String[] record = value.toString().split("\\|");

    if (record[Constants.STD_FLAG].equals("1")) {
      int duration = calculateDuration(record[Constants.CALL_START_TIME],
      ↪  record[Constants.CALL_END_TIME]);
      fromPhoneNumber.set(record[Constants.FROM_PHONE_NUMBER]);
      callDuration.set(duration);
      context.write(fromPhoneNumber, callDuration);
    }
  }

  private int calculateDuration(String startTime, String endTime) {
    LocalDateTime startDateTime = LocalDateTime.parse(startTime,
      DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    LocalDateTime endDateTime = LocalDateTime.parse(endTime,
↪   DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Duration duration = Duration.between(startDateTime, endDateTime);
    return (int) duration.toMinutes();
  }
}
```

- The CDRMapper class extends `Mapper` class.
  - Declared variables:
    * `fromPhoneNumber` is a Text variable storing Phone number.
    * `callDuration` is a IntWritable variable storing Call duration.
    * `record` is array of String holding values splitted from line.
  - Auxiliary function:
    * `calculateDuration`: a function return an interger to calculate call duration using `LocalDateTime` and `Duration` libraries.

- Idea of the mapper class:
  - The mapper class will read the input file line by line and each line is stored in `line` variable.
  - Then, from date to highest and lowest temperature informations from each line will be splitted and stored.
  - After that, consider to problem conditions and collect those cases into output.

1.2.9.3 *Reducer*

```
public class CDRReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

- The CDRReducer class extends `Reducer` class.
  - Declared variables:
    - `result` is a IntWritable containing end value to write in context.
    - `sum` is a IntWritable variable storing Call duration.
  - Idea of the reducer class:
    - Initialize `sum` variable and assign it with 0.
    - Then, traversal all elememts through loop and count .
    - After that, consider to problem conditions and collect those cases into output.

1.2.9.4 *Guide to run the program*

- Step 1: Create file CallDataRecord.java in the folder src of the project.
- Step 2: Create file CDRlog in the folder data of the project and then put the file to the local HDFS by using the command.

  ```
  hdfs dfs -mkdir /input
  ```

```
hdfs dfs -put data/CDRlog.txt /input
```

- Step 3: Compile and run the program by using the command.
  - `javac` is a command-line tool that compiles Java source code into Java bytecode.

    ```
    hadoop com.sun.tools.javac.Main CallDataRecord.java
    ```

  - `jar` is a command-line tool that creates a Java archive file (JAR) from a set of Java class files.

    ```
    jar cf wc.jar CallDataRecord*.class
    ```

  - hadoop is a command-line tool that runs a MapReduce job. `hadoop jar wc.jar CallDataRecord /input /output`
- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`

1.2.10 Problem 10: Count Connected Component Program

1.2.10.1 *Mapper*

```
public static class CountConnectedMapper extends Mapper<Object, Text, IntWritable,
↪  IntWritable> {
    private IntWritable _vertex = new IntWritable();
    public void map(Object key, Text value, Context context) throws IOException,
    ↪  InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        _vertex.set(Integer.parseInt(itr.nextToken()));
        while (itr.hasMoreTokens()) {
            context.write(_vertex, new IntWritable(Integer.parseInt(itr.nextToken())));
        }
    }
}
```

- The `CountConnectedMapper` class is a subclass of the `MapReduceBase` class which implements `Mapper` class.
- Declared variables:
  - `_vertex` is a IntWritable variable storing Value parameter.
- Idea of the mapper class:
  - The mapper class will read the input file line by line and each line is stored in `itr` variable.
  - Then, `_vertex` variable will be set to the first value of each line.
  - After that, loop through the rest of the line and write the value to the context.

1.2.10.2 *Reducer*

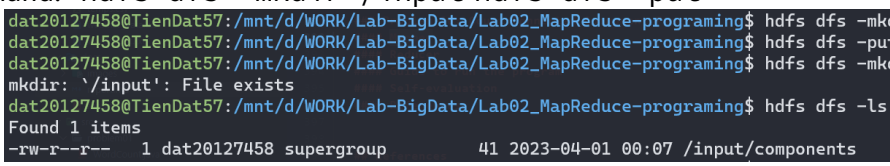
```
public static class ComponentReducer extends Reducer<IntWritable, IntWritable, Text,
↪  IntWritable> {
    private IntWritable _result = new IntWritable();
    private Set<Integer> _visited = new HashSet<Integer>();
    private Stack<Integer> _stack = new Stack<Integer>();
    private int _count = 0;
    public void reduce(IntWritable key, Iterable<IntWritable> values, Context context) throws
    ↪  IOException, InterruptedException {
        if (!_visited.contains(key.get())) {
            _stack.push(key.get());
            while (!_stack.isEmpty()) {
                int v = _stack.pop();
                if (!_visited.contains(v)) {
                    _visited.add(v);
                    for (IntWritable val : values) {
                        _stack.push(val.get());
                    }
                }
            }
            _count++;
        }
        _result.set(_count);
        context.write(new Text("Number of connected component in graph: "), _result);
    }
}
```

- The `ComponentReducer` class is a subclass of the `MapReduceBase` class which implements `Reducer` class.
- Declared variables:
  - `_result` is a IntWritable variable storing number of connected component in graph.
  - `_visited` is a Set variable storing visited vertex.
  - `_stack` is a Stack variable storing vertex.
  - `_count` is a int variable storing number of connected component in graph.
- Idea of the reducer class:
  - The reducer class will read the input file line by line and each line is stored in `values` variable.
  - Then, check if the vertex is visited or not. If not, push the vertex to the stack.
  - After that, loop through the rest of the line and write the value to the context.
  - After that, pop the vertex from the stack and check if the vertex is visited or not. If not, add the vertex to the visited set.
  - After that, loop through the rest of the line and write the value to the context.

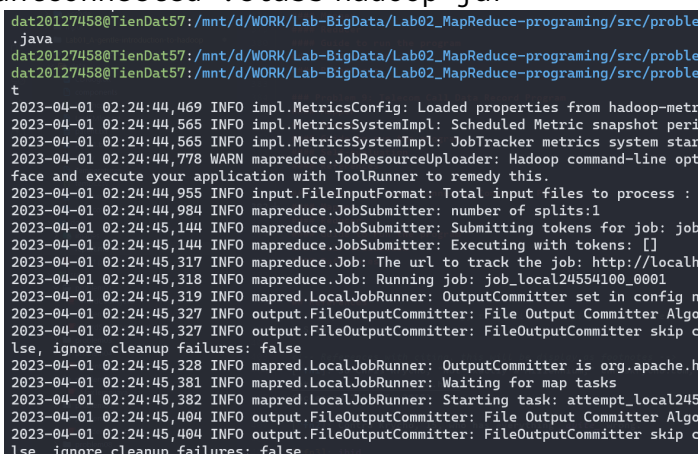
- After that, increase the count by 1.
- Finally, increase the count by 1 and write the result to the context.

1.2.10.3 *Guide to run the program*

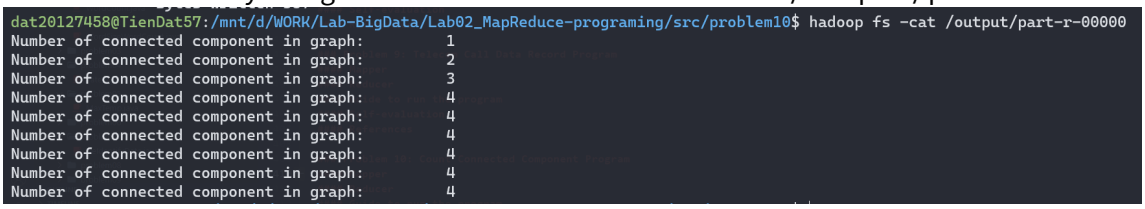
- Step 1: Create file CountConnected.java in the folder src of the project.
- Step 2: Create file graph.txt in the folder data of the project and then put the file to the local HDFS by using the command. `hdfs dfs -mkdir /input` `hdfs dfs -put data/component.txt /input`



- Step 3: Compile and run the program by using the command. `hadoop com.sun.tools.javac.Main CountConnected.java` `jar cf cc.jar CountConnected*.class` `hadoop jar cc.jar CountConnected /input /output`



- Step 4: Check the result by using the command. `hdfs dfs -cat /output/part-r-00000`



1.3 References

- Slide of course.
- LabRequirements.pdf