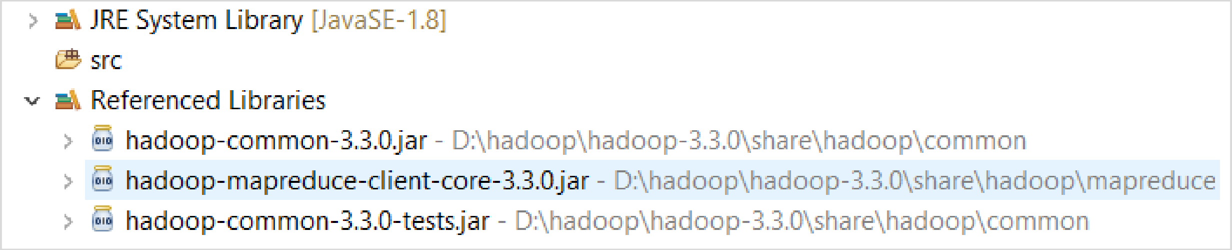
WEATHERDATA PROGRAM

Step 1: Include the necessary library like the picture below:



```
> JRE System Library [JavaSE-1.8]
  src
v Referenced Libraries
  > hadoop-common-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common
  > hadoop-mapreduce-client-core-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\mapreduce
  > hadoop-common-3.3.0-tests.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common
```

Figure 3.1: Include necessary jar library from hadoop.

Step 2: Prepare data and source code for run map reduce on Hadoop:

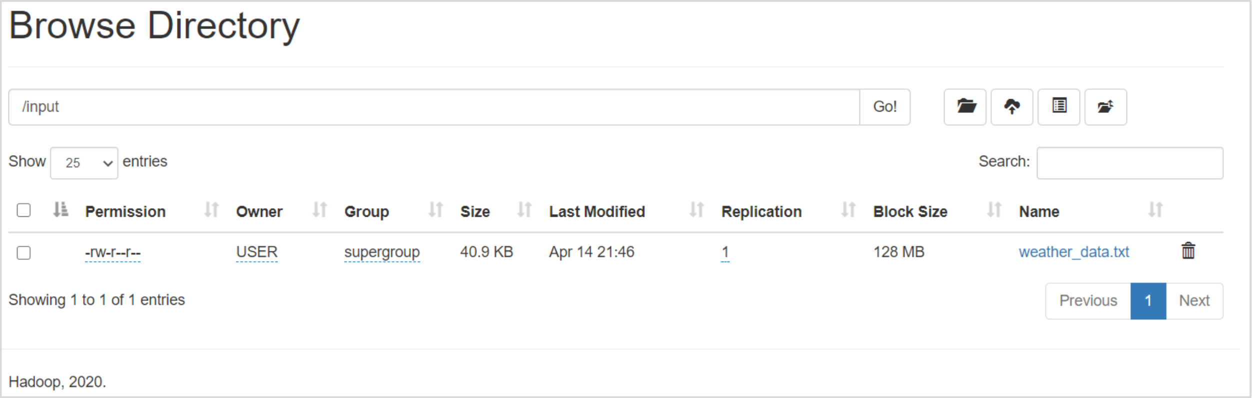
Step 2.1: Create the folder lab02 contain the source code file *Weather.java* and the given *weather_data.txt.*

Run dfs and yarn services as administrator by the following command

```
%HADOOP_HOME%\sbin\start-dfs.cmd
%HADOOP_HOME%\sbin\start-yarn.cmd
```

Then creating the *input* folder in dfs then upload *weather_data.txt* to this folder.

```
Hadoop fs –mkdir /input
hdfs dfs –copyFromLocal weather_data.txt /input
```



Browse Directory

/input

Show 25 entries

Search:

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
☐	-rw-r--r--	USER	supergroup	40.9 KB	Apr 14 21:46	1	128 MB	weather_data.txt

Showing 1 to 1 of 1 entries

Hadoop, 2020.

Figure 3.2: The result after uploading the data file into input folder on dfs.

Step 2.2: Prepare Source code: **self-implement**.

Explain the source code:

- First include the library for program such as:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
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;
public class Weather {
 public static class WeatherMapper extends Mapper<Object,
                    Text,
                    Text,
                    FloatWritable> {
  public void map(Object key, Text value, Context context)
   throws IOException, InterruptedException {
   String line = value.toString();
   String tmp = line.substring(6, 14);
   String date = tmp.substring(6, 8) + '-' + tmp.substring(4, 6) +'-' + tmp.substring(0, 4);
   float maxtmp = Float.parseFloat( line.substring(39, 45));
   float mintmp = Float.parseFloat(line.substring(47, 53));
   if (mintmp < 10.0){
    context.write(new Text( date + " Cold Day"), new FloatWritable(mintmp));
   }
   else if (maxtmp > 40.0){
    context.write(new Text( date + " Hot Day"), new FloatWritable(maxtmp));
   }
  }
 }
 public static class WeatherReducer
    extends Reducer<Text,FloatWritable,Text,FloatWritable> {
   public void reduce(Text key, FloatWritable values, Context context)
    throws IOException, InterruptedException {
    context.write(key, new FloatWritable(values.get()));
  }
}
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Weather");
    job.setJar("Weather.jar");
    job.setJarByClass(Weather.class);
    job.setMapperClass(WeatherMapper.class);
    job.setCombinerClass(WeatherReducer.class);
    job.setReducerClass(WeatherReducer.class);
   job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FloatWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

- Define class Weather contains 2 classes and 1 main function:
    - WeatherMapper: contain the map phase, which prepare the data for reduce phase, in particularly, extracting the ***minimum temperature*** and **maximum temperature** of each day, and extracting the ***date*** then processing to the date in *weather_data.txt* into the output form.
        - We consider the input key type is the object, the input value is each line in the weather data file.
        - The output key is the date and status of that day (Cold Day or Hot Day), it’s datatype is *Text*.
        - The output value is the temperature that made that day is cold or hot. Its datatype is *FloatWritable*.
        - In the map function, first we convert assign the *line* to be the converted String of values, which means the data of one line. The *tmp* variable is the raw date in the data file, which located in the index 6 to 14 in a line so we use *substring function* to get this data then we use *tmp* for making the new form in the date in *date* variable.

```
    String line = value.toString();
    String tmp = line.substring(6, 14);
    String date = tmp.substring(6, 8) + '-' + tmp.substring(4, 6) +'-' + tmp.substring(0, 4);
```

    - The minimum temperature is save at index 47 to 53 in a line and the maxmimum temperature is save at index 39 to 45 in a line. Thus we use the mentioned substring function to get this data then convert to the float datatype by the Float.parseFloat function, next save them in maxtmp and mintmp variable.

```
                float maxtmp = Float.parseFloat(line.substring(39, 45));
                float mintmp = Float.parseFloat(line.substring(47, 53));
```

    - We check the minimum temperature - *mintmp*, if it was lower than 10.0°C, we would determind the status of that day is “Cold Day”. Additionally, we also check the maximum temperature – *maxtmp*, if it was higher than 40.0°C, we would detetmind the “Hot Day” ones. After determined the status of that day, we write into the output collector – in this version of hadoop, it means the *Context class*. Using *write function* to write the result of map phase saved in *context* variable.

```
    if (mintmp < 10.0){
      context.write(new Text( date + " Cold Day"), new FloatWritable(mintmp));
    }
    else if (maxtmp > 40.0){
      context.write(new Text( date + " Hot Day"), new FloatWritable(maxtmp));
    }
```

    - The result of map phase including the output key and output value. As explained we have some instance for clear, such as:

      e.g: (“11-04-2021 Cold Day”, 1.5)

      (“12-09-2020 Hot Day”, 41.0)

- WeatherReducer: because we need the consistency between 2 phase and each each has only one maximum and minimum temperature so in the reduce phase, we only write the key – the date and status and the value – the temperature.

    - The input key and input values from map phase is *Text* and *FloatWritable*, because we don’t process anything so the input and the output is the same datatype.

    - We only write the data again into the context variable to finish and return to the output. Using *get function* to get the value from FloatWritable variable – values.

```
  public static class WeatherReducer
```

```
    extends Reducer<Text,FloatWritable,Text,FloatWritable> {
   public void reduce(Text key, FloatWritable values, Context context)
    throws IOException, InterruptedException {
     context.write(key, new FloatWritable(values.get()));
 }
}
```

- Main function: we set up the config of the map reduce, setJar and setJarByClass whole purpose to idenfy the Weather.jar and Weather.class for run the job avoid the related java.lang errors. Then we set up class for the Mapper, Combiner and Reducer of MapReduce by the setMapperClass, setCombinerClass, setReducerClass function.
    - Set up output key is the Text class and output values is the temperature is the FloatWritable class.
    - The file input and output are pass to arg, when we run the .exe and pass arguments including input path and output path into the command.

```
      Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "Weather");
  job.setJar("Weather.jar");
  job.setJarByClass(Weather.class);
  job.setMapperClass(WeatherMapper.class);
  job.setCombinerClass(WeatherReducer.class);
  job.setReducerClass(WeatherReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(FloatWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
```

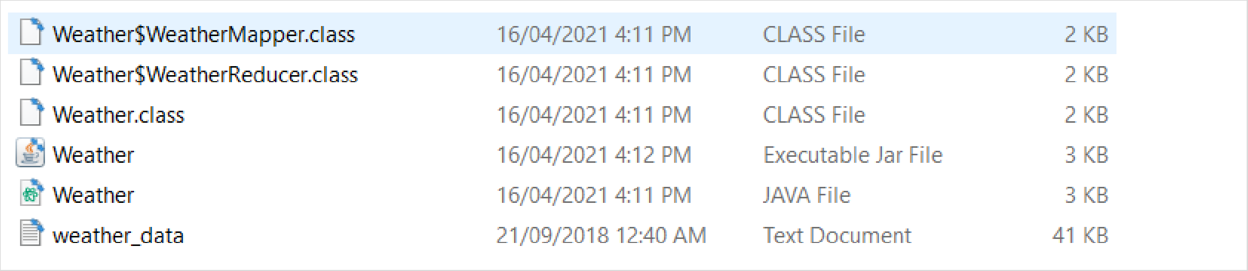
Step 3: Combine to the jar and executable file by using the following command.

Create file Weather.jar and Weather*.class:

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

Then create the executable file Weather.exe:

```
jar cf Weather.jar Weather*.class
```



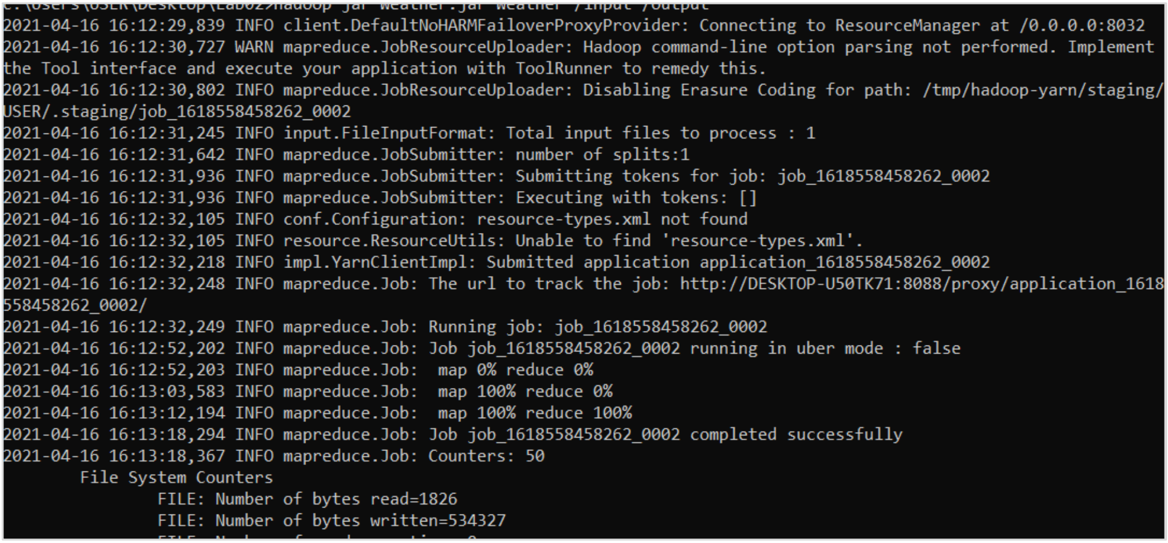
Weather$WeatherMapper.class	16/04/2021 4:11 PM	CLASS File	2 KB
Weather$WeatherReducer.class	16/04/2021 4:11 PM	CLASS File	2 KB
Weather.class	16/04/2021 4:11 PM	CLASS File	2 KB
Weather	16/04/2021 4:12 PM	Executable Jar File	3 KB
Weather	16/04/2021 4:11 PM	JAVA File	3 KB
weather_data	21/09/2018 12:40 AM	Text Document	41 KB

Figure 3.3: The files after run the previous commands.

Step 4: Run the map reduce and print the result on the screen.

To run the map reduce we use this command: this command will run the Weather.exe with the input data is the input folder and the result returning are saving into the file *part-r-00000* the output folder.

```
hadoop jar Weather.jar Weather /input /output
```

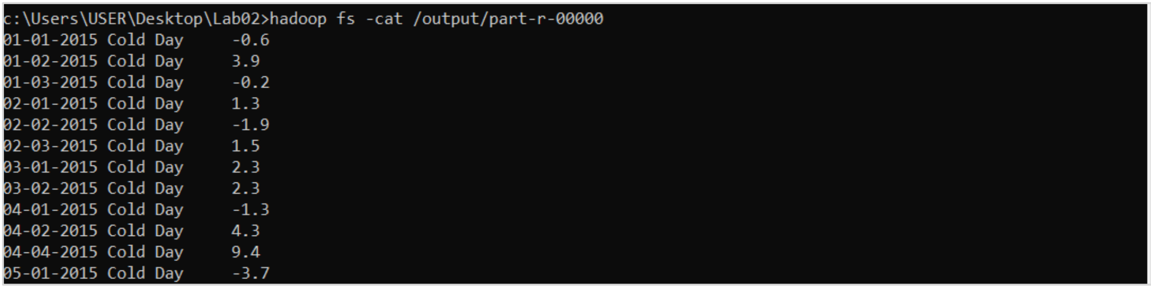


```
c:\Users\USER\Desktop\Lab02>hadoop jar Weather.jar Weather /input /output
2021-04-16 16:12:29,839 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2021-04-16 16:12:30,727 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement
the Tool interface and execute your application with ToolRunner to remedy this.
2021-04-16 16:12:30,802 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/
USER/.staging/job_1618558458262_0002
2021-04-16 16:12:31,245 INFO input.FileInputFormat: Total input files to process : 1
2021-04-16 16:12:31,642 INFO mapreduce.JobSubmitter: number of splits:1
2021-04-16 16:12:31,936 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1618558458262_0002
2021-04-16 16:12:31,936 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-04-16 16:12:32,105 INFO conf.Configuration: resource-types.xml not found
2021-04-16 16:12:32,105 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-04-16 16:12:32,218 INFO impl.YarnClientImpl: Submitted application application_1618558458262_0002
2021-04-16 16:12:32,248 INFO mapreduce.Job: The url to track the job: http://DESKTOP-U50TK71:8088/proxy/application_1618
558458262_0002/
2021-04-16 16:12:32,249 INFO mapreduce.Job: Running job: job_1618558458262_0002
2021-04-16 16:12:52,202 INFO mapreduce.Job: Job job_1618558458262_0002 running in uber mode : false
2021-04-16 16:12:52,203 INFO mapreduce.Job:  map 0% reduce 0%
2021-04-16 16:13:03,583 INFO mapreduce.Job:  map 100% reduce 0%
2021-04-16 16:13:12,194 INFO mapreduce.Job:  map 100% reduce 100%
2021-04-16 16:13:18,294 INFO mapreduce.Job: Job job_1618558458262_0002 completed successfully
2021-04-16 16:13:18,367 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=1826
                FILE: Number of bytes written=534327
```

Figure 3.4: Runing map reduce process.

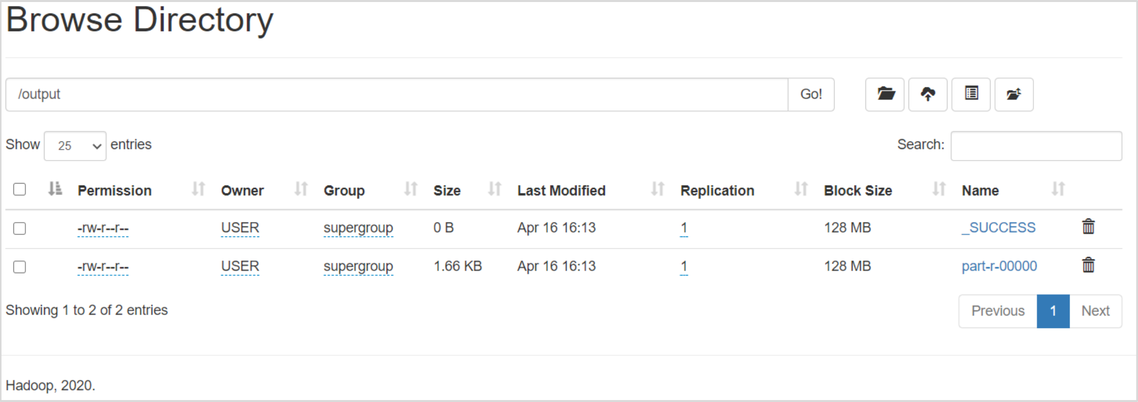
After run successfully, we run the following command to watch the result on the command screen.

```
hadoop fs –cat /output/part-r-00000
```



```
c:\Users\USER\Desktop\Lab02>hadoop fs -cat /output/part-r-00000
01-01-2015 Cold Day     -0.6
01-02-2015 Cold Day     3.9
01-03-2015 Cold Day     -0.2
02-01-2015 Cold Day     1.3
02-02-2015 Cold Day     -1.9
02-03-2015 Cold Day     1.5
03-01-2015 Cold Day     2.3
03-02-2015 Cold Day     2.3
04-01-2015 Cold Day     -1.3
04-02-2015 Cold Day     4.3
04-04-2015 Cold Day     9.4
05-01-2015 Cold Day     -3.7
```

Figure 3.5: A part of result saved in part-r-00000 file in the output folder on dfs.



Browse Directory

/output

Show 25 entries

Search:

□	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
□	-rw-r--r--	USER	supergroup	0 B	Apr 16 16:13	1	128 MB	_SUCCESS
□	-rw-r--r--	USER	supergroup	1.66 KB	Apr 16 16:13	1	128 MB	part-r-00000

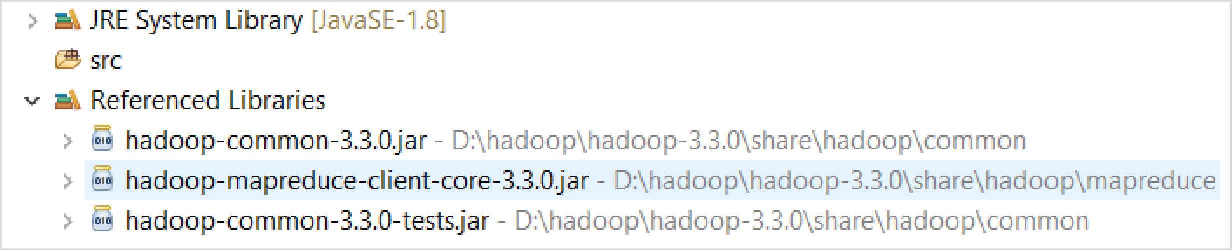
Showing 1 to 2 of 2 entries

Hadoop, 2020.

Figure 3.6: The destination of the result file on dfs.

AVERAGESALARY PROGRAM

Step 1: Include the necessary library like the picture below:



JRE System Library [JavaSE-1.8]
src
Referenced Libraries
hadoop-common-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common
hadoop-mapreduce-client-core-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\mapreduce
hadoop-common-3.3.0-tests.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common

Figure 6.1: Include necessary jar library from hadoop.

Step 2: Prepare data and source code for run map reduce on Hadoop:

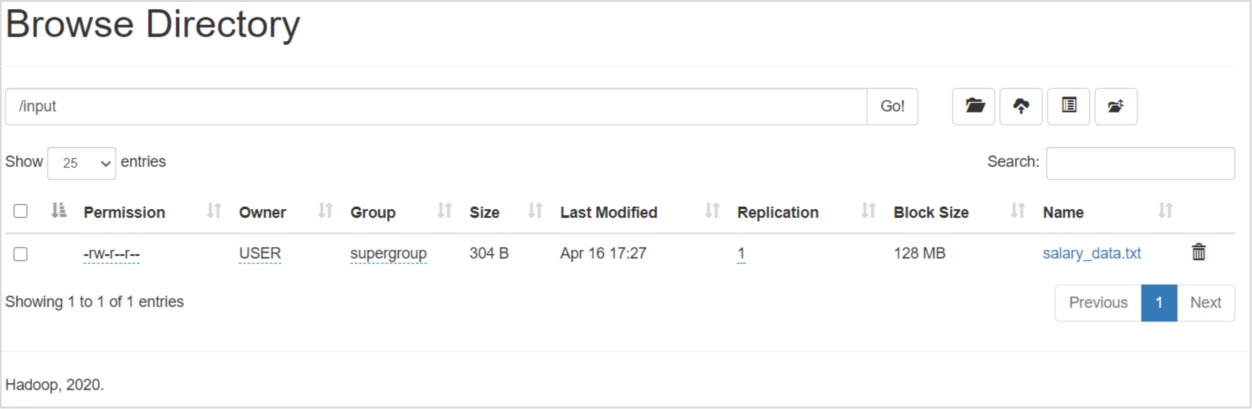
Step 2.1: Create the folder lab02 contain the source code file avg*Salary.java* and the given *salary_data.txt.*

Run dfs and yarn services as administrator by the following command

```
%HADOOP_HOME%\sbin\start-dfs.cmd
%HADOOP_HOME%\sbin\start-yarn.cmd
```

Then creating the *input* folder in dfs then upload *salary_data.txt* to this folder.

```
Hadoop fs –mkdir /input
hdfs dfs –copyFromLocal salary_data.txt /input
```



Browse Directory

/input

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
☐	-rw-r--r--	USER	supergroup	304 B	Apr 16 17:27	1	128 MB	salary_data.txt	🗑

Showing 1 to 1 of 1 entries

Hadoop, 2020.

Figure 6.2: The result after uploading the data file into input folder on dfs.

Note: Because we don’t have any dataset for testing this program, so we create the content of this file.

Description: there 4 attributes of the data from left to right; ID, Name, Day of Birth, salary, each value separated by the tab command “\t”.

```
001       nguyen van a       11/01/2000        17000000
002       nguyen van b       12/02/2000        19700000
003       nguyen van c       13/03/2000        14000000
004       nguyen van d       14/04/2000        15500000
005       nguyen van e       15/05/2000        16000000
006       nguyen van f       16/06/2000        19000000
007       nguyen van g       17/07/2000        25000000
008       nguyen van h       18/08/2000        18000000
```

Step 2.2: Prepare Source code: **self-implement**.

Explain the source code:

- First include the library for program such as:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
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;




public class Salary {
  public static class SalaryMapper extends Mapper<Object,
            Text,
            IntWritable,
            DoubleWritable> {
```

```
    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
      String line = value.toString();
      String tmp[] = line.split("\t");
      double salary = Double.parseDouble(tmp[3]);
      context.write(new IntWritable(1), new DoubleWritable(salary));
    }
  }

  public static class SalaryReducer
       extends Reducer<IntWritable,DoubleWritable,IntWritable,DoubleWritable> {
      public void reduce(IntWritable key, Iterable<DoubleWritable> values, Context context)
       throws IOException, InterruptedException {
        double sum = 0;
        int count = 0;
        int key_tmp = key.get();
        for(DoubleWritable val: values){
          sum += val.get() * key_tmp;
          count += 1;
        }
        count *= key_tmp;
        context.write(new IntWritable(count), new DoubleWritable(sum/count));
  }
}

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Salary");
    job.setJar("Salary.jar");
    job.setJarByClass(Salary.class);
    job.setMapperClass(SalaryMapper.class);
    job.setCombinerClass(SalaryReducer.class);
    job.setReducerClass(SalaryReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(DoubleWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

- Define class Salary contains 2 classes and 1 main function:
  - SalaryMapper: contain the map phase, which prepare the data for reduce phase, in particularly, extracting the salary from each line.
    - We consider the input key type is the object, the input value is each line in the salary data file.
    - The output key is the number of people who has the sample wage, and its datatype is *IntWritable*.
    - The output value is average salary. Its datatype is *DoubleWritable*. Because the total wage is usually massive.
    - In the map function, first we convert assign the *line* to be the converted String of values, which means the data of one line. The *tmp* variable is the array contains values of the line, which separated by tab command, and we using split function to do this job.

```
            String line = value.toString();
            String tmp[] = line.split("\t");
```

    - We convert the string salary to double datatype by Double.parseDouble function, then saving it into *salary* variable.

```
    double salary = Double.parseDouble(tmp[3]);
```

    - In map phase, we count each wage as one, 1 is the key and salary is the value.

```
            context.write(new IntWritable(1), new DoubleWritable(salary));
```

    - The result of map phase including the output key and output value. As explained we have some instance for clear, such as:

      e.g: (1, 15000000)

      (1, 29300000)

  - SalaryReducer: In reduce phase, we combine the wages to make the average wage, and create the new count.

    For example:

Map phase 1: [100, 200 300] → reduce phase → 3:[200]

        - The input key and input values from map phase is IntWritable and Iterable<*DoubleWritable*>. Because we have the average wage and the number of people that calculated, thus we can easily to calculate the total wage by multiply the key and the average wage.
        - We save the total wage into the sum variable and the count variable saves the number of people, which is the size of the array values multiply with the key. Using for loop to calculate those variables.

```
  public static class SalaryReducer
    extends Reducer<IntWritable,DoubleWritable,IntWritable,DoubleWritable> {
   public void reduce(IntWritable key, Iterable<DoubleWritable> values, Context context)
    throws IOException, InterruptedException {
     double sum = 0;
     int count = 0;
     int key_tmp = key.get();
     for(DoubleWritable val: values){
       sum += val.get() * key_tmp;
       count += 1;
     }
     count *= key_tmp;
     context.write(new IntWritable(count), new DoubleWritable(sum/count));
 }
}
```

        - Then we write the count and average wage by divided sum to count then using write function to write the ouput to context.

    - Main function: we set up the config of the map reduce, setJar and setJarByClass whole purpose to idenfy the Salary.jar and Salary.class for run the job avoid the related java.lang errors. Then we set up class for the Mapper, Combiner and Reducer of MapReduce by the setMapperClass, setCombinerClass, setReducerClass function. As t
        - Set up output key is the IntWritable class and output values is the average wage is the DoubleWritable class.
        - The file input and output are pass to arg, when we run the .exe and pass arguments including input path and output path into the command.

```
     Configuration conf = new Configuration();
```

```
Job job = Job.getInstance(conf, "Weather");
job.setJar("Weather.jar");
job.setJarByClass(Weather.class);
job.setMapperClass(WeatherMapper.class);
job.setCombinerClass(WeatherReducer.class);
job.setReducerClass(WeatherReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FloatWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

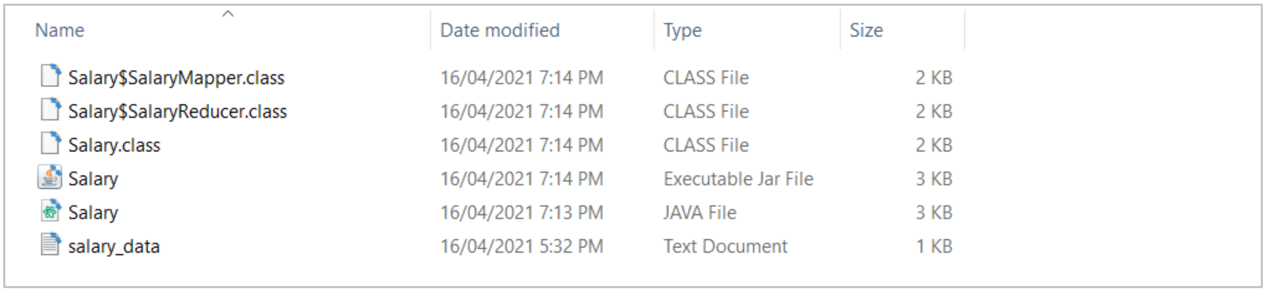
Step 3: Combine to the jar and executable file by using the following command.

Create file Salary.jar and Salary*.class:

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

Then create the executable file Salary.exe:

```
jar cf Salary.jar Salary*.class
```



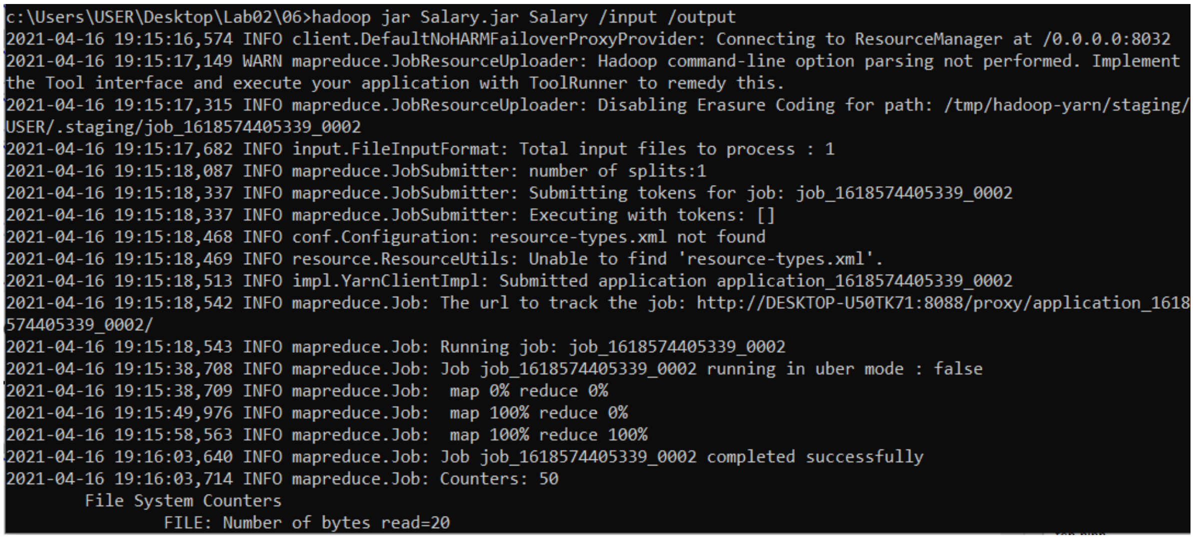
Name	Date modified	Type	Size
Salary$SalaryMapper.class	16/04/2021 7:14 PM	CLASS File	2 KB
Salary$SalaryReducer.class	16/04/2021 7:14 PM	CLASS File	2 KB
Salary.class	16/04/2021 7:14 PM	CLASS File	2 KB
Salary	16/04/2021 7:14 PM	Executable Jar File	3 KB
Salary	16/04/2021 7:13 PM	JAVA File	3 KB
salary_data	16/04/2021 5:32 PM	Text Document	1 KB

Figure 6.3: The files after run the previous commands.

Step 4: Run the map reduce and print the result on the screen.

To run the map reduce we use this command: this command will run the Salary.exe with the input data is the input folder and the result returning are saving into the file *part-r-00000* the output folder.

```
hadoop jar Salary.jar Salary /input /output
```

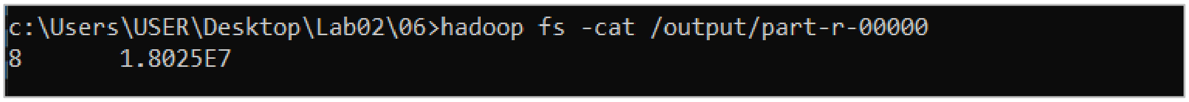


```
c:\Users\USER\Desktop\Lab02\06>hadoop jar Salary.jar Salary /input /output
2021-04-16 19:15:16,574 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2021-04-16 19:15:17,149 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement
the Tool interface and execute your application with ToolRunner to remedy this.
2021-04-16 19:15:17,315 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/
USER/.staging/job_1618574405339_0002
2021-04-16 19:15:17,682 INFO input.FileInputFormat: Total input files to process : 1
2021-04-16 19:15:18,087 INFO mapreduce.JobSubmitter: number of splits:1
2021-04-16 19:15:18,337 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1618574405339_0002
2021-04-16 19:15:18,337 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-04-16 19:15:18,468 INFO conf.Configuration: resource-types.xml not found
2021-04-16 19:15:18,469 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-04-16 19:15:18,513 INFO impl.YarnClientImpl: Submitted application application_1618574405339_0002
2021-04-16 19:15:18,542 INFO mapreduce.Job: The url to track the job: http://DESKTOP-U50TK71:8088/proxy/application_1618
574405339_0002/
2021-04-16 19:15:18,543 INFO mapreduce.Job: Running job: job_1618574405339_0002
2021-04-16 19:15:38,708 INFO mapreduce.Job: Job job_1618574405339_0002 running in uber mode : false
2021-04-16 19:15:38,709 INFO mapreduce.Job:  map 0% reduce 0%
2021-04-16 19:15:49,976 INFO mapreduce.Job:  map 100% reduce 0%
2021-04-16 19:15:58,563 INFO mapreduce.Job:  map 100% reduce 100%
2021-04-16 19:16:03,640 INFO mapreduce.Job: Job job_1618574405339_0002 completed successfully
2021-04-16 19:16:03,714 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=20
```

Figure 6.4: Runing map reduce process.

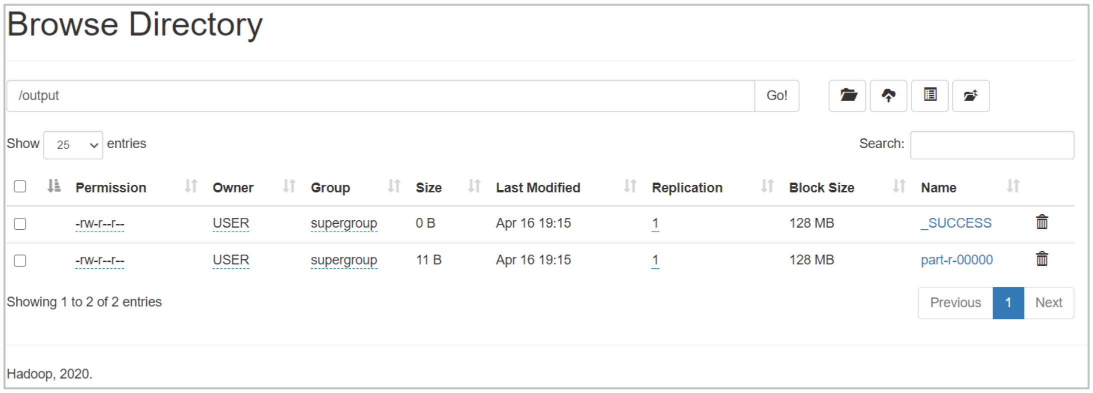
After run successfully, we run the following command to watch the result on the command screen.

```
hadoop fs –cat /output/part-r-00000
```



```
c:\Users\USER\Desktop\Lab02\06>hadoop fs -cat /output/part-r-00000
8       1.8025E7
```

Figure 6.5: The result saved in part-r-00000 file in the output folder on dfs.



Browse Directory

/output

Show 25 entries

Search:

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
☐	-rw-r--r--	USER	supergroup	0 B	Apr 16 19:15	1	128 MB	_SUCCESS	🗑
☐	-rw-r--r--	USER	supergroup	11 B	Apr 16 19:15	1	128 MB	part-r-00000	🗑

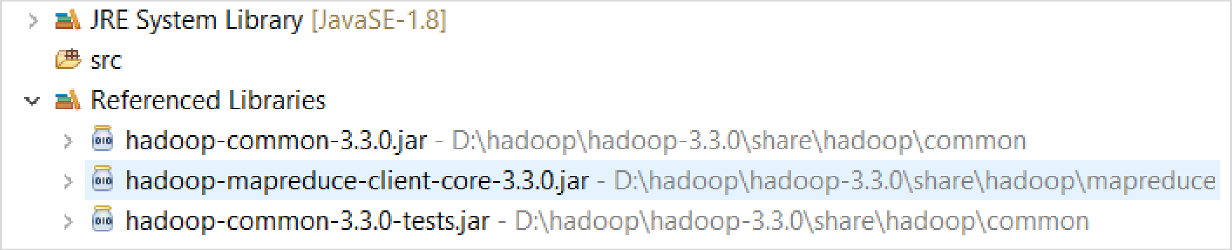
Showing 1 to 2 of 2 entries

Hadoop, 2020.

Figure 6.6: The destination of the result file on dfs.

TELECOM CALL DATA RECORD PROGRAM

Step 1: Include the necessary library like the picture below:



JRE System Library [JavaSE-1.8]
src
Referenced Libraries
hadoop-common-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common
hadoop-mapreduce-client-core-3.3.0.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\mapreduce
hadoop-common-3.3.0-tests.jar - D:\hadoop\hadoop-3.3.0\share\hadoop\common

Figure 6.1: Include necessary jar library from hadoop.

Step 2: Prepare data and source code for run map reduce on Hadoop:

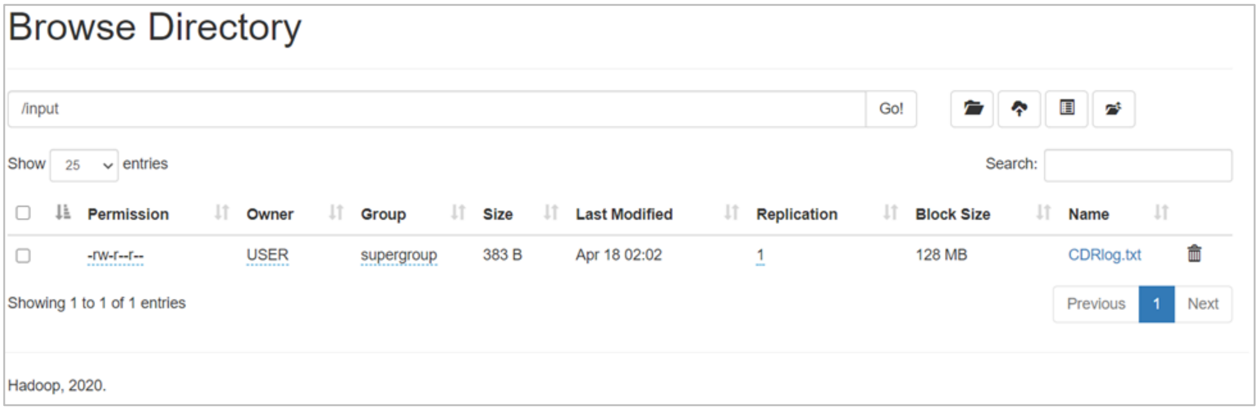
Step 2.1: Create the folder lab02 contain the source code file Telecom*.java* and the given *CDRlog.txt.*

Run dfs and yarn services as administrator by the following command

```
%HADOOP_HOME%\sbin\start-dfs.cmd
%HADOOP_HOME%\sbin\start-yarn.cmd
```

Then creating the *input* folder in dfs then upload *salary_data.txt* to this folder.

```
hadoop fs –mkdir /input
hdfs dfs –copyFromLocal  CDRlog.txt /input
```



Browse Directory

/input

Show 25 entries Search:

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
☐	-rw-r--r--	USER	supergroup	383 B	Apr 18 02:02	1	128 MB	CDRlog.txt

Showing 1 to 1 of 1 entries

Hadoop, 2020.

Figure 9.2: The result after uploading the data file into input folder on dfs.

Step 2.2: Prepare Source code: **self-implement**.

Explain the source code:

- First include the library for program such as:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import java.time.LocalDateTime;
import java.time.Duration;
import org.apache.hadoop.io.Text;
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;


public class Telephone {

 public static class TelephoneMapper extends Mapper<Object,
           Text,
           Text,
           LongWritable> {
  public void map(Object key, Text value, Context context)
   throws IOException, InterruptedException {
   String line = value.toString();
   String tmp[] = line.split("[|]");
   String FromPhone = tmp[0];
   int STDflag = Integer.parseInt(tmp[4]);
   long CallingTime = 0;
   if(STDflag == 1 ){
    CallingTime = cacl_calling_time(tmp[2], tmp[3]);
   }
   context.write(new Text(FromPhone), new LongWritable(CallingTime));
  }
  long cacl_calling_time(String from, String to){
   // First is date, second is time
   String from_data[] = from.split(" ");
```

```
    String to_data[] = to.split(" ");
    // contains data about year, month, day in order from left to right
    String from_tmp[] = from_data[0].split("-");
    String to_tmp[] = to_data[0].split("-");
    int from_date[] = new int[3];
    int to_date[] = new int[3];
    for(int i =0; i < 3; i++){
      from_date[i] = Integer.parseInt(from_tmp[i]);
      to_date[i] = Integer.parseInt(to_tmp[i]);
    }
    // contains data about hour, minute, second inoder from left to right
    String from_tmp1[] = from_data[1].split(":");
    String to_tmp1[] = to_data[1].split(":");

    int from_time[] = new int[3];
    int to_time[] = new int[3];
    for(int i = 0; i < 3; i++){
      from_time[i] = Integer.parseInt(from_tmp1[i]);
      to_time[i] = Integer.parseInt(to_tmp1[i]);
    }

    LocalDateTime Start = LocalDateTime.of(from_date[0], from_date[1], from_date[2],
                             from_time[0], from_time[1], from_time[2]);
    LocalDateTime End = LocalDateTime.of(to_date[0], to_date[1], to_date[2],
                            to_time[0], to_time[1], to_time[2]);

    long total_time = Duration.between(Start, End).toMinutes();

    return total_time;
  }
}


public static class TelephoneReducer
     extends Reducer<Text,LongWritable,Text,LongWritable> {
      public void reduce(Text key, Iterable<LongWritable> values, Context context)
       throws IOException, InterruptedException {
       long total_time = 0;
       for(LongWritable val: values){
         total_time += val.get();
       }
       if(total_time > 60){
         context.write(key, new LongWritable(total_time));
       }
      }
}

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Telephone");
    job.setJar("Telephone.jar");
    job.setJarByClass(Telephone.class);
    job.setMapperClass(TelephoneMapper.class);
    job.setCombinerClass(TelephoneReducer.class);
    job.setReducerClass(TelephoneReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

- Define class Telephone contains 2 classes and 1 main function:
    - TelephoneMapper: contain the map phase, which prepare the data for reduce phase, in particularly, extracting the value of “From Phone Number”, “Call Start Time”, “Call End Time” from each line.
        - We consider the input key type is the object, the input value is each line in the data file *CDRlog.txt*.

- The output key is the phone number of the caller who makes a call, and its datatype is *Text*.

- The output value is the calling time of that call. Its datatype is *LongWritable*. Because the history call of a phone number can be very large as possible when the user may be a seller or others, and they made a lot of call.

- In the map function, first we convert assign the *line* to be the converted String of values, which means the data of one line. The *tmp* variable is the array contains values of the line, which separated by "|" charater, and we using split function to do this job.

```
String line = value.toString();
 String tmp[] = line.split("[|]");
```

- We extract the caller's number from *tmp* array, which is the first element of that array, then saving it into *FromPhone* variable. Then we also extract the STD flag by convert the last element of *tmp* array to integer datatype by using *Integer.parseInt function*. After that, we save that value to *STDflag* variable.

```
String FromPhone = tmp[0];
int STDflag = Integer.parseInt(tmp[4]);
```

- Initialize the CallingTime variable equaling zero because if the *STDflag* equals zero, the default value of *CallingTime* will be zero. Then we consider the condition when *STDflag* equals one, we implement the function *calc_calling_time* takes two arguments the time when the caller called and the ending time of that call that are the third and the fourth element in the *tmp* array. (note: the order in array started at zero).

```
long CallingTime = 0;
 if(STDflag == 1 ){
		CallingTime = cacl_calling_time(tmp[2], tmp[3]);
 }
```

- Explaination about the *calc_calling_time* function: the argument explained before. Firstly, we separate the date and the time when started the call and ended the call into 2 string array; *from_data* stored the stared date and started time; *to_data* stored the ended date and the ended time.

```
long cacl_calling_time(String from, String to){
	// First is date, second is time
	String from_data[] = from.split(" ");
```

```
      String to_data[] = to.split(" ");
      // from_tmp and to_tmp contains data about year, month, day in order
from left to right
```

- After that we convert the started date including year, month, day to the string array *from_tmp*, which separated by "-" charater by using the *split* function (do the same with ended date in the next 2 lines in source code).

```
      String from_tmp[] = from_data[0].split("-");
      String to_tmp[] = to_data[0].split("-");
```

- Then we initialize 2 integer array to save the integer value of the stared and ended date, those are *from_date* and *to_date,* Using *for* loop and the Integer.parseInt function to convert the values, which have the same order in the new integer array.

```
      int from_date[] = new int[3];
      int to_date[] = new int[3];
      for(int i =0; i < 3; i++){
        from_date[i] = Integer.parseInt(from_tmp[i]);
        to_date[i] = Integer.parseInt(to_tmp[i]);
      }
```

- Then we do the same with the stared time and the ended time like the implementation below.

```
      // contains data about hour, minute, second inoder from left to right
      String from_tmp1[] = from_data[1].split(":");
      String to_tmp1[] = to_data[1].split(":");

      int from_time[] = new int[3];
      int to_time[] = new int[3];
      for(int i = 0; i < 3; i++){
        from_time[i] = Integer.parseInt(from_tmp1[i]);
        to_time[i] = Integer.parseInt(to_tmp1[i]);
      }
```

- Aftet that we defined the 2 variable *Start* and *End* whole datatype is *LocalDateTime* – for computing the calling time. Passing 6 arguments in order like year, month, day, hour, minute, second into *LocalDateTime.of* function to save its information. Then we call the *Duration.between* function (Supported by LocalDateTime and Duration library) to calculate the calling time and extract it into minutes by *toMinutes* function, then saving that value into *total_time* variable and return this. (Supported by *LocalDateTime* and *Duration* library)

```
      LocalDateTime Start = LocalDateTime.of(from_date[0], from_date[1],
from_date[2],
                                             from_time[0], from_time[1],
from_time[2]);
      LocalDateTime End = LocalDateTime.of(to_date[0], to_date[1],
to_date[2],
                                           to_time[0], to_time[1],
to_time[2]);

      long total_time = Duration.between(Start, End).toMinutes();

      return total_time;
```

- To sum up, in map phase, each caller's number is the key and the calling time (with condition *SDFflag equals 1*) is the value.

```
context.write(new Text(FromPhone), new LongWritable(CallingTime));
```

- The result of map phase including the output key and output value. As explained we have some instance for clear, such as:

  e.g: ("123456", 90) SDFflag = 1.

  ("981293", 0) SDFflag = 0.

- TelephoneReducer: In reduce phase, we combine all the time of the calls to make the total calling time and write the result if the total time higher than 60 minutes.

  For example:

  Map phase "9812221": [0, 20, 54] → reduce phase → "9812221": 74

  - The input key and input values from map phase is *Text* and Iterable<*LongWritable*>.
  - We save the total calling time into the *total_time* variable Using for loop to calculate the total time of each caller's number.

```
public static class TelephoneReducer
 extends Reducer<Text,LongWritable,Text,LongWritable> {
   public void reduce(Text key, Iterable<LongWritable> values, Context context)
    throws IOException, InterruptedException {
     long total_time = 0;
     for(LongWritable val: values){
       total_time += val.get();
     }
     if(total_time > 60){
       context.write(key, new LongWritable(total_time));
     }
   }
}
```

    - Then we write the caller’s number and total calling time, if total calling time higher than 60 minutes. Finally, using write function to write the ouput to context.

  - Main function: we set up the config of the map reduce, setJar and setJarByClass whole purpose to idenfy the Salary.jar and Salary.class for run the job avoid the related java.lang errors. Then we set up class for the Mapper, Combiner and Reducer of MapReduce by the setMapperClass, setCombinerClass, setReducerClass function.

    - Set up output key is the Text class and output values is the total calling time is the LongWritable class.

    - The file input and output are pass to arg, when we run the .exe and pass arguments including input path and output path into the command.

```
 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "Telephone");
  job.setJar("Telephone.jar");
  job.setJarByClass(Telephone.class);
  job.setMapperClass(TelephoneMapper.class);
  job.setCombinerClass(TelephoneReducer.class);
  job.setReducerClass(TelephoneReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(LongWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
```

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

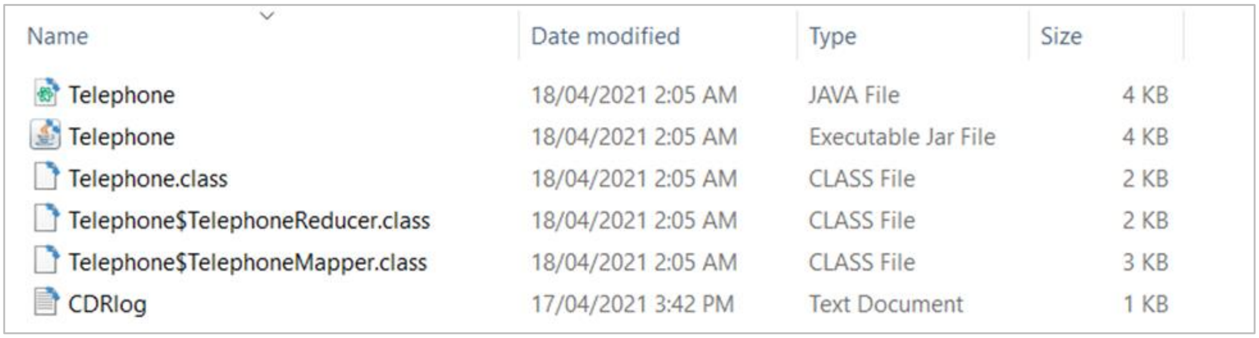
Step 3: Combine to the jar and executable file by using the following command.

Create file Telephone.jar and Telephone*.class:

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

Then create the executable file Telephone.exe:

```
jar cf Telephone.jar Telephone*.class
```



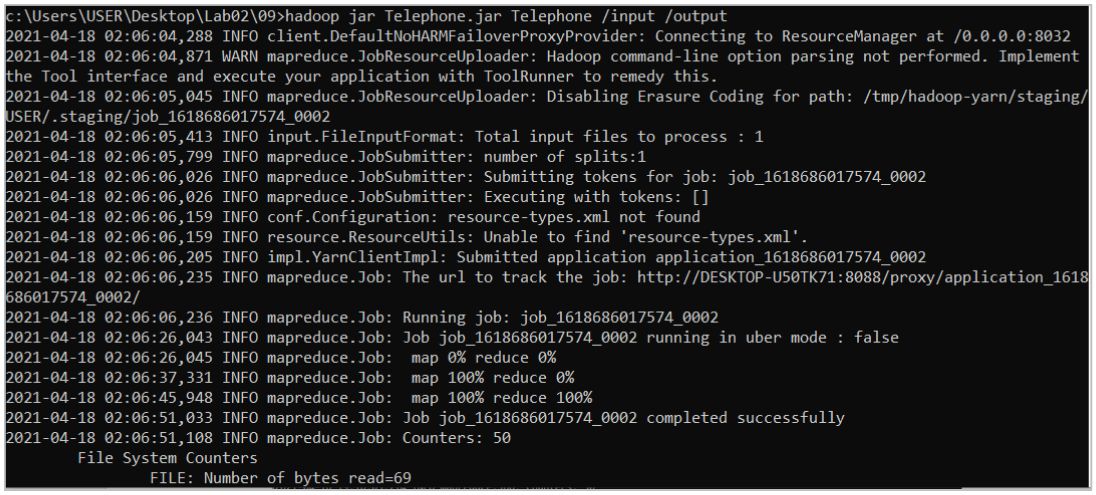
Name	Date modified	Type	Size
Telephone	18/04/2021 2:05 AM	JAVA File	4 KB
Telephone	18/04/2021 2:05 AM	Executable Jar File	4 KB
Telephone.class	18/04/2021 2:05 AM	CLASS File	2 KB
Telephone$TelephoneReducer.class	18/04/2021 2:05 AM	CLASS File	2 KB
Telephone$TelephoneMapper.class	18/04/2021 2:05 AM	CLASS File	3 KB
CDRlog	17/04/2021 3:42 PM	Text Document	1 KB

Figure 9.3: The files after run the previous commands.

Step 4: Run the map reduce and print the result on the screen.

To run the map reduce we use this command: this command will run the Telephone.exe with the input data is the input folder and the result returning are saving into the file *part-r-00000* the output folder.

```
hadoop jar Telephone.jar Telephone /input /output
```

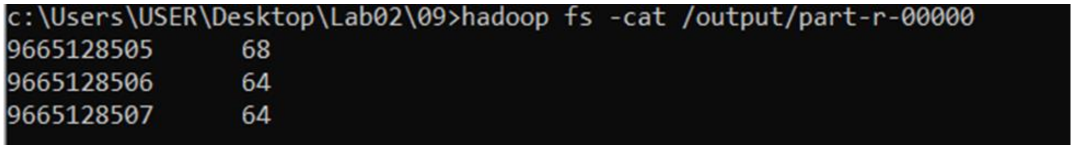


```
c:\Users\USER\Desktop\Lab02\09>hadoop jar Telephone.jar Telephone /input /output
2021-04-18 02:06:04,288 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2021-04-18 02:06:04,871 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement
the Tool interface and execute your application with ToolRunner to remedy this.
2021-04-18 02:06:05,045 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/
USER/.staging/job_1618686017574_0002
2021-04-18 02:06:05,413 INFO input.FileInputFormat: Total input files to process : 1
2021-04-18 02:06:05,799 INFO mapreduce.JobSubmitter: number of splits:1
2021-04-18 02:06:06,026 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1618686017574_0002
2021-04-18 02:06:06,026 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-04-18 02:06:06,159 INFO conf.Configuration: resource-types.xml not found
2021-04-18 02:06:06,159 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-04-18 02:06:06,205 INFO impl.YarnClientImpl: Submitted application application_1618686017574_0002
2021-04-18 02:06:06,235 INFO mapreduce.Job: The url to track the job: http://DESKTOP-U50TK71:8088/proxy/application_1618
686017574_0002/
2021-04-18 02:06:06,236 INFO mapreduce.Job: Running job: job_1618686017574_0002
2021-04-18 02:06:26,043 INFO mapreduce.Job: Job job_1618686017574_0002 running in uber mode : false
2021-04-18 02:06:26,045 INFO mapreduce.Job:  map 0% reduce 0%
2021-04-18 02:06:37,331 INFO mapreduce.Job:  map 100% reduce 0%
2021-04-18 02:06:45,948 INFO mapreduce.Job:  map 100% reduce 100%
2021-04-18 02:06:51,033 INFO mapreduce.Job: Job job_1618686017574_0002 completed successfully
2021-04-18 02:06:51,108 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=69
```

Figure 9.4: Runing map reduce process.

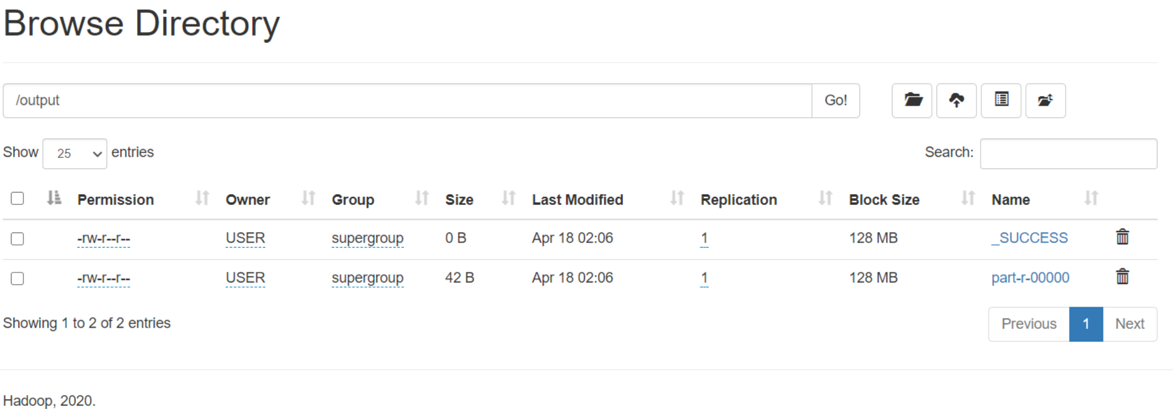
After run successfully, we run the following command to watch the result on the command screen.

```
hadoop fs –cat /output/part-r-00000
```



```
c:\Users\USER\Desktop\Lab02\09>hadoop fs -cat /output/part-r-00000
9665128505      68
9665128506      64
9665128507      64
```

Figure 9.5: The result saved in part-r-00000 file in the output folder on dfs.



Browse Directory

/output Go!

Show 25 entries Search:

□	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
□	-rw-r--r--	USER	supergroup	0 B	Apr 18 02:06	1	128 MB	_SUCCESS	🗑
□	-rw-r--r--	USER	supergroup	42 B	Apr 18 02:06	1	128 MB	part-r-00000	🗑

Showing 1 to 2 of 2 entries

Hadoop, 2020.

Figure 9.6: The destination of the result file on dfs.