3 Delivery: Hadoop

Objectives

We are introducing Hadoop as a mapReduce platform and the Java classes as the programming framework. Our main objectives are the following:

- Programming a first implementation of word count in python using mapReduce streaming.
- Learning a mapreduce application implementation on Hadoop.
- Launching a job in a Hadoop cluster.

3.1 First implementation: counting words with Python using Hadoop streaming

Working with HDFS

First of all we create a new data folder in the HDFS file system:

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

Now we copy an input file ("El Quijote de la Mancha") from our local folder to HDFS file system:

```
wget https://www.gutenberg.org/cache/epub/2000/pg2000.txt
hdfs dfs -put ~/pg2000.txt /hduser/input
```

To make sure no problems occurred, we check the existence of the file in HDFS:

```
hdfs dfs -ls /hduser/input
```

```
Found 1 items
-rw-r--r--   1 cloudera supergroup    2198927 2018-02-12 07:39
   /hduser/input/pg2000.txt
```

Implementation of the mapper and reducer

In the snippets 1 and 2 below we can see a simple implementation of a mapper and a reducer to count words:

Snippet 1: `word_count_mapper.py` implementation

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
```

```
  # Remove leading and trailing characters
  line=line.strip()

  # Split each line into words
  words = line.split()

  # Print concat(word, 1) for each word
  for word in words:
    print ('%s\t%s' % (word, "1"))
```

One thing we wanted to add to the mapper was to perform a `word.rstrip(',')` for each `word in words` to get rid of trailing comma characters, but the mapreduce job gave errors when doing this. So we decided just to keep the mapper untouched, as this is nothing more than a test.

Snippet 2: `word_count_reducer.py` implementation

```
#!/usr/bin/env python
import sys

word2count= {}

for line in sys.stdin:
  line = line.strip()

  # Save word and count of each concat(word, 1)
  word, count = line.split('\t', 1)

  # Check if count is a number
  try:
    count = int(count)
  except ValueError: # Ignore the word
    continue

  # Add counts of each word in an array
  try: # Normal behaviour: add + 1 to the counter
    word2count[word] = word2count[word] + count
  except: # Add new word to the "database" with value "1"
    word2count[word] = count

for word in word2count.keys():
  print ('%s\t%s' % ( word, word2count[word]))
```

It is always a good idea to make sure the permissions of the scripts are right, so the mapreduce job does not give an error:

```
chmod -R a+rx ~/src
```

Running the mapreduce job

Now we need to use the Hadoop runtime with the `hadoop-streaming` tool to use our Python implementation:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -mapper
    ~/src/word_count_mapper.py -reducer ~/src/word_count_reducer.py
    -input /hduser/input -output /hduser/output-python
```

If no errors occurred during the mapreduce job, we should get an output like this:

```
18/02/12 10:55:15 INFO mapreduce.Job: Counters: 50
  File System Counters
    FILE: Number of bytes read=3688605
    ...
    HDFS: Number of write operations=2
  Job Counters
    Killed map tasks=1
    ...
    Total megabyte-milliseconds taken by all reduce tasks=7725056
  Map-Reduce Framework
    Map input records=37861
    ...
    Total committed heap usage (bytes)=391979008
  Shuffle Errors
    BAD_ID=0
    ...
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=2203023
  File Output Format Counters
    Bytes Written=448894
18/02/12 10:55:15 INFO streaming.StreamJob: Output directory:
    /hduser/output-python
```

Now that we have the final output we can check, for instance, the words that appear the most in the text:

```
hdfs dfs -cat /hduser/output-python/part-00000 | awk '{print $2,
    $1}' FS=" " | sort -n | tail -n5
```

```
9575 a
10200 la
15894 y
17988 de
19429 que
```

One question one may ask is if there can be more than one file on the input folder. The answer is yes, but whilst running the mapreduce they are combined in a same output. This is, of course, the intended behaviour of Hadoop.

3.2 Second implementation: counting words with Java Classes

In this part we will reproduce the previous implementation with Python for the mapreduce job, but using Java classes instead.

For the process we will be using the same input file ("El Quijote de la Mancha"), obtained from Project Gutenberg.

Implementation of the mapper and reducer

The `WordCount` application is quite straight-forward:

(i) The `Mapper` implementation, via the map method, processes one line at a time, as provided by the specified `TextInputFormat`. It then splits the line into tokens separated by white-spaces, via the `StringTokenizer`, and emits a key-value pair of `< <word>, 1>`.

(ii) `WordCount` also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is same as the `Reducer` as per the job configuration) for local aggregation, after being sorted on the keys.

(iii) The `Reducer` implementation, via the reduce method just sums up the values, which are the occurrence counts for each word.

(iv) The `main` method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the `Job`. It then calls the `job.waitForCompletion` to submit the job and monitor its progress.

Snippet 3: `WordCount.java` implementation

```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
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 WordCount {

  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

```

```java
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer 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);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

Creating a .jar binary

The first thing to do is to add these lines to the `.bashrc` file to export the required environment variables:

```bash
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

We need to get the class files from the `WordCount.java` source code:

```
cd ~/src
hadoop com.sun.tools.javac.Main WordCount.java
```

The following files will be created:

- `WordCount.class`
- `WordCount$IntSumReducer.class`
- `WordCount$TokenizerMapper.class`

Now we only need to create the `WordCount.jar` binary:

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

Running the mapreduce job

Now we need to use the Hadoop runtime with the `jar` tool to use our Java implementation:

```
cd
hadoop jar ~/src/WordCount.jar WordCount /hduser/input
    /hduser/output-java
```

If no errors occurred during the mapreduce job, we can check, for instance, the words that appear the most in the text:

```
hdfs dfs -cat /hduser/output-java/part-r-00000 | awk '{print $2,
    $1}' FS="␣" | sort -n | tail -n5
```

```
9575 a
10200 la
15894 y
17988 de
19429 que
```

As expected, we get the same result we got with the implementation in Python. It is important to notice that this mapper and reducer implementation has the same problem the previous one had: it keeps trailing comma characters.

How not to do the previous steps

Notice than in the previous instructions we manually changed the working directory to `~/src` before creating the `.jar` binary; this was a crucial step. Imagine that instead we had done the following:

```
hadoop com.sun.tools.javac.Main ~/src/WordCount.java
jar cf ~/src/WordCountBad.jar ~/src/WordCount*.class
hadoop jar ~/src/WordCountBad.jar WordCount /hduser/input
    /hduser/output-java-bad
```

```
Exception in thread "main" java.lang.ClassNotFoundException:
   WordCount
  at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
  at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
  at java.security.AccessController.doPrivileged(Native Method)
  at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
  at java.lang.Class.forName0(Native Method)
  at java.lang.Class.forName(Class.java:270)
  at org.apache.hadoop.util.RunJar.run(RunJar.java:214)
  at org.apache.hadoop.util.RunJar.main(RunJar.java:136)
```

What is going on here? Let's have a look at the two compiled `.jar` files:

```
jar -tvf ~/src/WordCount.jar
```

```
     0 Sat Feb 17 06:22:42 PST 2018 META-INF/
    68 Sat Feb 17 06:22:42 PST 2018 META-INF/MANIFEST.MF
  1501 Sat Feb 17 06:22:40 PST 2018 WordCount.class
  1739 Sat Feb 17 06:22:40 PST 2018 WordCount$IntSumReducer.class
  1736 Sat Feb 17 06:22:40 PST 2018 WordCount$TokenizerMapper.class
```

```
jar -tvf ~/src/WordCountBad.jar
```

```
     0 Sat Feb 17 15:06:04 PST 2018 META-INF/
    68 Sat Feb 17 15:06:04 PST 2018 META-INF/MANIFEST.MF
  1501 Sat Feb 17 15:06:04 PST 2018 home/cloudera/src/WordCount.class
  1739 Sat Feb 17 15:06:04 PST 2018
     home/cloudera/src/WordCount$IntSumReducer.class
  1736 Sat Feb 17 15:06:04 PST 2018
     home/cloudera/src/WordCount$TokenizerMapper.class
```

The issue here is that if we run the `jar cf` command to create the binary from outside the directory where the `*.class` files are located, the binary will include the full *hard* path of the classes, instead of having a *soft* inclusion of them.

It is, of course, possible to run the mapreduce job with the *bad* `.jar` binary, if we call the class by the right name, but it is just simpler to compile the `.jar` binary properly to avoid any mistakes.

3.3 Third implementation: processing weather data with Python

Processing weather data

Many big data applications start with a large size bulk dataset available at a remote server to download. In this case, we are using data from the National Climatic Data Center (NCDC). Each line in the dataset represents a record where a list of meteorological measures is available. The following is an example of all measures that can be found in a record:

```
0057
332130   # USAF weather station identifier
99999    # WBAN weather station identifier
19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-0128    # air temperature (degrees Celsius x 10)
1        # quality code
-0139    # dew point temperature (degrees Celsius x 10)
1        # quality code
10268    # atmospheric pressure (hectopascals x 10)
1        # quality code
```

Our problem is to find the maximum and mean temperature of each year found in the dataset. Then, we will be focusing on observation date and air temperature parameters of each record. First we are going to introduce the structure of the application and then we are going to focus on the implementation details of each part of the map reduce application.

Before we start working with Python and Java to construct our mapredude implementation, we need to put our data on the HDFS:

```
for (( year = 1901; year <= 1920; year ++ )); do
  hdfs dfs -put ~/data/ncdc-dataset/$year /hduser/input-weather/$year
done
```

The dataset for years 1901–1920 can be downloaded from `https://github.com/aldomann/parallel-and-distributed-systems/blob/master/data/ncdc-dataset.zip`.

The map reduce job

For this process we are interested in three fields from the dataset:

- Year (characters 15–19).
- Air temperature (characters 87–92).
- Quality code for the temperature (characters 92–93).

In the snippet 4 below we can see a simple implementation of a mapper to get the year and the temperature from the NCDC dataset:

Snippet 4: `temp_mapper.py` implementation

```python
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  # Remove heading and tailing characters
  val = line.strip()

  # String split to get data
  (year, temp, q) = (val[15:19], val[87:92], val[92:93])

  # Print concat(year, temp) for each year
  if (temp != "+9999" and re.match("[01459]", q)):
    print ("%s\t%s" % (year, temp))
```

In the snippet 5, 6, and 7 below we can see three different reducers:

- `temp_max_reducer.py` looks for the maximum temperature value for each year.
- `temp_min_reducer.py` looks for the minimum temperature value for each year.
- `temp_mean_reducer.py` calculates the mean temperature for each year.

Snippet 5: `temp_max_reducer.py` implementation

```python
#!/usr/bin/env python

import sys

(last_key, max_val) = (None, 0)
for line in sys.stdin:
```

```python
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    # Print results of non-last key
    print ("%s\t%s" % (last_key, max_val))
    # Starts reducing new key
    (last_key, max_val) = (key, int(val))
  else:
    # Process data
    (last_key, max_val) = (key, max(max_val, int(val)))

if last_key:
  # Print results of last key
  print ("%s\t%s" % (last_key, max_val))
```

Snippet 6: `temp_min_reducer.py` implementation

```python
#!/usr/bin/env python

import sys

(last_key, min_val) = (None, 0)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    # Print results of non-last key
    print ("%s\t%s" % (last_key, min_val))
    # Starts reducing new key
    (last_key, min_val) = (key, int(val))
  else:
    # Process data
    (last_key, min_val) = (key, min(min_val, int(val)))

if last_key:
  # Print results of last key
  print ("%s\t%s" % (last_key, min_val))
```

Snippet 7: `temp_mean_reducer.py` implementation

```python
#!/usr/bin/env python

import sys

(last_key, max_val, count) = (None, 0, 0)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    # Print results of non-last key
    print ("%s\t%s" % (last_key, max_val/count))
    # Starts reducing new key
    (last_key, max_val, count) = (key, int(val), 1)
  else:
    # Process data
```

```
    (last_key, max_val, count) = (key, int(max_val) + int(val),
        int(count) + 1)

if last_key:
  # Print results of last key
  print ("%s\t%s" % (last_key, max_val/count))
```

3.3.1 Running the maximum temperature mapreduce job

As we already did before, we need to use the Hadoop runtime with the `hadoop-streaming` tool to use our Python implementation:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -mapper
    ~/src/temp_mapper.py -reducer ~/src/temp_max_reducer.py -input
    /hduser/input-weather -output /hduser/output-max-temp
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-max-temp/part-00000
```

```
1901  317
1902  244
1903  289
1904  256
1905  283
1906  294
1907  283
1908  289
1909  278
1910  294
1911  306
1912  322
1913  300
1914  333
1915  294
1916  278
1917  317
1918  322
1919  378
1920  294
```

3.3.2 Running the minimum temperature mapreduce job

Now we just need to use the Hadoop runtime with the `hadoop-streaming` tool to use our Python implementation:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -mapper
    ~/src/temp_mapper.py -reducer ~/src/temp_min_reducer.py -input
    /hduser/input-weather -output /hduser/output-min-temp
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-min-temp/part-00000
```

```
1901	-333
1902	-328
1903	-306
1904	-294
1905	-328
1906	-250
1907	-350
1908	-378
1909	-378
1910	-372
1911	-378
1912	-411
1913	-372
1914	-378
1915	-411
1916	-289
1917	-478
1918	-450
1919	-428
1920	-344
```

3.3.3 Running the mean temperature mapreduce job

Now we just need to use the Hadoop runtime with the `hadoop-streaming` tool to use our Python implementation:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -mapper
    ~/src/temp_mapper.py -reducer ~/src/temp_mean_reducer.py -input
    /hduser/input-weather -output /hduser/output-mean-temp
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-mean-temp/part-00000
```

```
1901	46.698507007922
1902	21.659558263518658
1903	48.241744739671326
1904	33.32224247948952
1905	43.3322664228014
1906	47.0834855681403
1907	31.76414576084966
1908	28.836573511543136
1909	26.565303955402175
1910	35.558665794637015
1911	30.719045120671563
1912	16.801145236855803
1913	29.958786491127647
```

```
1914  29.817932296431838
1915  5.098548073625243
1916  21.42393787117405
1917  22.91685727355901
1918  31.36519845111326
1919  27.605149653640048
1920  43.508667830133795
```

3.4 Fourth implementation: processing weather data

The idea here is the same, to implement a mapper and a reducer to find the maximum, mean, and minimum temperature for each year in the dataset.

First we need to define our *main* class `MaxTemperature`, where we define the `main` function that calls to the mapper and the reducer:

Snippet 8: `MaxTemperature.java` implementation

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    @SuppressWarnings("deprecation")
    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(TemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The `MinTemperature.java` and `MeanTemperature.java` source codes are essentially the same, so we will not include them here explicitly. One just needs to be careful to call the classes by the right name.

For all of the mapreduce jobs we will use the same mapper, as the map job just extracts the relevant information from the dataset. The implementation can be seen in snippet 9 below:

Snippet 9: `TemperatureMapper.java` implementation

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TemperatureMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    System.out.println(line);

    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading
         plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

The reducer is the usual maximum calculation algorithm, the main difference with the implementation in Python is that `maxValue` is initialised to `Integer.MIN_VALUE = -2147483648`. The implementation for `MaxTemperatureReducer.java` can be seen in snippet 10 below:

Snippet 10: `MaxTemperatureReducer.java` implementation

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

```

```
  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

For `MinTemperatureReducer.java` the main difference is in the initialisation of `minValue` using `Integer.MAX_VALUE = 2147483647`, and the calculation of the minimum. The changes in the implementation can be seen in snippet 11 below:

Snippet 11: `MinTemperatureReducer.java` implementation

```
int minValue = Integer.MAX_VALUE;
for (IntWritable value : values) {
  minValue = Math.min(minValue, value.get());
}
```

For `MeanTemperatureReducer.java` we need to work with the `DoubleWritable` class, as we need to calculate the ratio between the total sum of temperature values (`totalValue`) and the amount of records per year (**`count`**). The implementation can be seen in snippet 12 below:

Snippet 12: `MeanTemperatureReducer.java` implementation

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MeanTemperatureReducer
  extends Reducer<Text, IntWritable, Text, DoubleWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    double totalValue = 0.0;
    int count = 0;
    for (IntWritable value : values) {
      totalValue += value.get();
      count += 1;
```

```
    }
    context.write(key, new DoubleWritable(totalValue/count));
  }
}
```

Creating the .jar binaries

We need to get the class files from the different `.java` source codes:

```
cd ~/src
hadoop com.sun.tools.javac.Main MaxTemperature.java
    TemperatureMapper.java MaxTemperatureReducer.java
hadoop com.sun.tools.javac.Main MinTemperature.java
    TemperatureMapper.java MinTemperatureReducer.java
hadoop com.sun.tools.javac.Main MeanTemperature.java
    TemperatureMapper.java MeanTemperatureReducer.java
```

The following files will be created:

- `MaxTemperature.class`
- `MinTemperature.class`
- `MeanTemperature.class`
- `TemperatureMapper.class`
- `MaxTemperatureReducer.class`
- `MinTemperatureReducer.class`
- `MeanTemperatureReducer.class`

To create the `.jar` binaries, we use:

```
jar cf MaxTemperature.jar MaxTemperature.class
    TemperatureMapper.class MaxTemperatureReducer.class
jar cf MinTemperature.jar MinTemperature.class
    TemperatureMapper.class MinTemperatureReducer.class
jar cf MeanTemperature.jar MeanTemperature.class
    TemperatureMapper.class MeanTemperatureReducer.class
```

3.4.1 Running the maximum temperature mapreduce job

As we already did before, we need to use the Hadoop runtime with the `jar` tool to use our Java implementation:

```
hadoop jar ~/src/MaxTemperature.jar MaxTemperature
    /hduser/input-weather /hduser/output-max-temp-java
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-max-temp-java/part-r-00000
```

```
1901	317
1902	244
1903	289
1904	256
1905	283
1906	294
1907	283
1908	289
1909	278
1910	294
1911	306
1912	322
1913	300
1914	333
1915	294
1916	278
1917	317
1918	322
1919	378
1920	294
```

3.4.2 Running the minimum temperature mapreduce job

We just need to use the Hadoop runtime with the `jar` tool to use our Java implementation:

```
hadoop jar ~/src/MinTemperature.jar MinTemperature
    /hduser/input-weather /hduser/output-min-temp-java
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-min-temp-java/part-r-00000
```

```
1901	-333
1902	-328
1903	-306
1904	-294
1905	-328
1906	-250
1907	-350
1908	-378
1909	-378
1910	-372
1911	-378
1912	-411
1913	-372
1914	-378
1915	-411
1916	-289
1917	-478
1918	-450
```

```
1919  -428
1920  -344
```

3.4.3 Running the mean temperature mapreduce job

We just need to use the Hadoop runtime with the `jar` tool to use our Java implementation:

```
hadoop jar ~/src/MeanTemperature.jar MeanTemperature
    /hduser/input-weather /hduser/output-mean-temp-java
```

After the mapreduce job finishes successfully, we just need to check the output:

```
hdfs dfs -cat /hduser/output-mean-temp-java/part-r-00000
```

```
1901  46.698507007922
1902  21.659558263518658
1903  48.241744739671326
1904  33.32224247948952
1905  43.3322664228014
1906  47.0834855681403
1907  31.76414576084966
1908  28.836573511543136
1909  26.565303955402175
1910  35.558665794637015
1911  30.719045120671563
1912  16.801145236855803
1913  29.958786491127647
1914  29.817932296431838
1915  5.098548073625243
1916  21.42393787117405
1917  22.91685727355901
1918  31.36519845111326
1919  27.605149653640048
1920  43.508667830133795
```