CS236 DBMS Project
Hadoop Mapreduce

ReadMe
Niveditha Anand 862392888 nanan010@ucr.edu

I have done the whole project alone. So all the work is done only by me.

Softwares Installed :
1. Hadoop. Version= 3.3.4
2. Java. Version =1.8
3. NetBeans. Version = 15

MapReduce:
MapReduce is a technique where the input data is processed in a Map and a Reduce phase. The map reduce is implements with the help of Jobs. Jobs are set to a Mapper class and a Reducer class. The Mapper processes the data from the input file and maps them in a <Key, Value> pair and sends them to the reducer where the input is grouped by the key.

Problem Statement:
Since there are 3 problem statement, I have separated them as 3 jobs.
The first job is combining both the files.
The next job is to find the maximum temperature and minimum temperature for each state.
The last job is to sort them.

1. Find the average temperature of each states in the US
There are 2 types of files, one .csv file where the input data contains- the station id, country and state. The other file is a .txt file in which contains the station id, average temperature and the number of recordings.

Since the input will be split in the mapper side which makes it **quicker** to join, I have chose to do mapper side join and produce the result to the reducer.

The **station id** is the **primary** key, which is present in both files. Using the station ID we can **join** the two files with corresponding state and average temperature and the number of recordings.
I have implemented **join** in the **mapper side.** The data on the mapper side is easier to join as we get each line of the file and we join with the data of the other table with the help of primary key.
The weighted **average** computation is performed in the **reducer** class. The data in the reducer class is grouped as <key, value> so we can compute the value from the values of each key.

Implementation:
A job is created to implement the mapper and the reducer.
The smaller files (2006.txt, 2007.txt, 2008.txt and 2009.txt) are put into the **cache** by the job, which can be used by the mapper class to join with respect to the bigger file(.csv).

Setup():
When the map starts, **setup** method is first executed which reads the files from the cache. The data(TimeStamp for the month, temperature, no. of recordings) are read from the files. We need the only the **month** from the timestamp field(YYYYMMDD).
I have used a SimpleDateFormat to convert to the month.
All the data are stored in a **Hashmap** which can be referenced in the map() to compare with the station id and join them into a file.
The input data contains the column header in-between the data lines, so each line is checked for this to be ignored.
The delimiter used is the white spaces of different length, so we spilt the data using (**\ \s+**).
Since we need a weighted **average**, the multiply the average temperature and the number of recordings which can be combined with the other temperatures from the same state same month for the average temperature of that month for that state. This is also stored in the map.

Map():
The data is received from the .csv file the path is specified in the terminal.
Each line is processed by splitting using (“,”) as a delimiter. The first header line and the column headers are ignored by doing a string comparison for the each line. The data is returned with doubled quotes, so I have taken the substring of the input ignoring the quotes.
The input from the .csv file is compared to the map data containing <station id, list> from the .txt files.
As the goal is to combine the state with the months with their appropriate average, the map is checked for station id, if present, the month, average temperature and the number of recordings are joined with the state from the input.
The key is set as **state** and **month** and the other fields(average temp, number of recordings, weighted temperature) are set as the value and the output is sent to the reducer.

ReducerJoin() : reducer():
Each <key, value> pair is passed to the reducer grouping all the values for each key. Meaning for each state and for every month, the values are grouped which can thus be processed to calculate the average. The weighted average was calculated by weighted temperature/total number of recordings for that month for that particular state. This is done in the reducer phase. To do this for each value in the reduce() I am adding the number of records value and the weighted sum temperature value repeatedly until the last element of the key-value pair is reached(hasNext). When it is the last element, the

average temperature is calculated by dividing the weighted sum by the total number of records. The key is set as **state** with **calculated average** and **month** as the value. This key value pair is written to the file present in the /user/out/part1.

Execution Time :
Total time spent by all maps in occupied slots (ms)=124084
Total time spent by all reduces in occupied slots (ms)=5492
Total time spent by all map tasks (ms)=124084
Total time spent by all reduce tasks (ms)=5492

2. Find the maximum and minimum temperature and month for each state
I have to calculate the minimum and maximum temperature for each state comparing each month, so I chose to do a separate job with a mapper and reducer. The completion of the first job is check before starting this job as it uses the output file of the first job as input for this job. Once completed, the job is started by setting the mapper and reducer classes and specifying the input file location. And the output is stored in the /user/out/part2.
Map():
The map sets the **state** as key and the month and the average temperature as the value and passed to the reducer.
reduce():
The reducer takes each key-value pair grouped by the key which is the state and compares them with global value max and min which repeatedly updates the max and min value, if the max and min value is found for that state based on (no hasNextValue()) is present. We write the temperature value and the month value for max and min temperature and the difference for each state to the output file with state as their key value.
The difference is calculated with max and min value for the states.
Note :
I have set min value initially to 200 as the temperature can't rise above to 200 given it C or F. I tried to get the first element as we get the last element using .hasNext() but I couldn't succeed with that so I've set a random feasible value for min=-100 and max set to 0 and computed with temperature value for each state.

Execution Time :
Total time spent by all maps in occupied slots (ms)=4034
Total time spent by all reduces in occupied slots (ms)=4658
Total time spent by all map tasks (ms)=4034
Total time spent by all reduce tasks (ms)=4658

3. Order the elements based on the difference :
I have used a separate job for the sorting as mapper and reducer shuffle and sort by default in their implementation. I have used the difference as the key as we need to sort it using the difference of the temperature.

Implementation :
After the completion of the two jobs, the third job is started. With the input file from / user/out/part2 and the output file is in user/out/part3.
map():
The mapper set the key as **difference** and the other fields as value. Since the difference is a **float** value the mapper outputs a FloatWritable.
reduce():
The reducer gets the FloatWritable as input with the difference value in it and it is shuffled and sort it using this difference as the key and the reducer uses this key-value to sort it using the difference and written to a file.

Execution time:

Without a combiner:

Total time spent by all maps in occupied slots (ms)=4231
Total time spent by all reduces in occupied slots (ms)=4827
Total time spent by all map tasks (ms)=4231
Total time spent by all reduce tasks (ms)=4827

With Combiner:

Total time spent by all maps in occupied slots (ms)=4126
Total time spent by all reduces in occupied slots (ms)=4694
Total time spent by all map tasks (ms)=4126
Total time spent by all reduce tasks (ms)=4694

Things tried to implement:

Since for the second job I set the state as the key, and we need to sort the data using the difference. I did some research and found the **secondary sort** which is a map reduce concept with custom comparator. I implemented for the SortMapper and the SortReducer using a custom key class **SortKey** and a **sortComparator** and a customer partitioner SortPartitioner and a custom groupComparator called **SortGroupComapator.** But I was getting EOFExeception when I try to readFloat which the difference. I was stuck with this and thus moved on to write a sorted hash map.

cleanup(): The cleanup() sorts the difference as the key and write it to the file with the sorted data. The map is sorted based on the difference and thus makes data stored. But I was getting the output values twice : 1 from reducer and 1 from cleanup. I didn't know how to flush the context in the cleanup() to write the output from the sorted which could maintain the data order.

Combiner class :
I tried implementing combiner class for the first jobs but I got errors for using the Reducer class as the combiner class. So I did not use a combiner for the first two jobs.

Things I learnt :
While splitting the input I tried to print the values. The output can be printed using System.out.println() which we can find in the /localhost:9870/logs/userlogs/{job_id}/{completed_task}. When I was doing this I was trying to print all the result. Since the

value was too many, the map was running for a long time, I leant that we can see the yarn application -list can be used to list all the running jobs with the status. We can then kill the application using yarn -kill {id}.

The Secondary sort was the best thing I learnt in the map reduce. But unfortunately I faced an error and since there were only 50 states, using a hash map sort was easier.

The third job is not essential. We can store the reducer output value in the hash-map and sort it using the TreeMap in the cleanup phase. But my cleanup was giving some bogus data to the file as well. Also I wanted to write it in 3 files as the problem statement contains 3 parts.

Input files are in :
/user/join

OutPut files can be found in :
/user/out/part1
/user/out/part2
/user/out/part2