Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 2
Name: Neha Pradhan

ASSIGNMENT – 2 REPORT

Map Reduce Algorithms:

1. No Combiner pseudo code:
   a. Data structure used for accumulating temperature and calculating averages based on TMIN and TMAX by station id:
   StationStats ← Text obsvType, DoubleWritable temperature

```
map (offset o, recordLine l) {
    //The key o is the offset of the line in the input
    //l is a line of record entries from the input file
    splitLine[] ← l.split(",") //array of tokens from line delimited on ,
    obsvType ← splitLine[2]
    if obsvType equals "TMIN" or "TMAX
            stationID ← splitLine[0]
            temperature ← splitLine[3]
            StationStats stnStat ← new StationStats(obsvType, temperature)
            emit(stationID, stnStat)
}

reduce (Text stationId, List<StationStats> stnStats) {
    for each stat in stnStats
            Accumulate TMIN and TMAX values
```

- Calculate the mean for both TMIN and TMAX based on values accumulated
- Create output string (output) in the required format with the station id, mean TMIN and mean TMAX values

```
    emit(output, null)
}
```

2. Combiner pseudo code:
   a. Data structure used for accumulating temperature and calculating averages based on TMIN and TMAX by station id:
   StationStatsCombiner ← DoubleWritable tMaxSum, DoubleWritable tMaxCount,
   DoubleWritable tMinSum, DoubleWritable tMinCount

```
map (offset o, recordLine l) {
    //The key o is the offset of the line in the input
    //l is a line of record entries from the input file
    splitLine[] ← l.split(",") //array of tokens from line delimited on ,
```

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 2
Name: Neha Pradhan

```
        obsvType ← splitLine[2]
        stationID ← splitLine[0]
        temperature ← splitLine[3]

        if obsvType equals “TMIN”
                Update data structure stnStatCom,
                        tMinSum ← temperature
                        tMinCount ← 1

                emit(stationID, stnStatCom)

        else if obsvType equals “TMAX”
                Update data structure stnStatCom,
                        tMaxSum ← temperature
                        tMaxCount ← 1

                emit(stationID, stnStatCom)
    }

    combine (Text stationId, List<StationStatsCombiner> stnStatsComb) {
        for each stat in stnStatsComb
                if tMaxCount is not 0
                        tMaxSum ← tMaxSum + (tMaxSum of current stat)
                        tMaxCount ← tMaxCount + 1
                else
                        tMinSum ← tMinSum + (tMinSum of current stat)
                        tMinCount ← tMinCount + 1

    }

    reduce (Text stationId, List<StationStatsCombiner> stnStatsComb) {

        for each stat in stnStatsComb
                tMaxSum ← tMaxSum + (tMaxSum of current stat)
                tMaxCount ← tMaxCount + (tMaxCount of current stat)
                tMinSum ← tMinSum + (tMinSum of current stat)
                tMinCount ← tMinCount + (tMinCount of current stat)
        • Create output string (output) in the required format with the station id, mean TMIN
          and mean TMAX values
        emit(output, null)
    }
```

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 2
Name: Neha Pradhan

3. In Mapper Combiner pseudo code:
   a. Data structure used for accumulating temperature and calculating averages based on TMIN and TMAX by station id:
      StationStatsCombiner ← DoubleWritable tMaxSum, DoubleWritable tMaxCount,
      DoubleWritable tMinSum, DoubleWritable tMinCount
   b. Data structure used to store intermediate results:
      HashMap<Text, StationStatsCombiner>

```
Class Mapper {
        HashMap<Text, StationStatsCombiner>
        setup () {
                initialize the HashMap structure H
        }
        map (offset o, recordLine l) {
                //The key o is the offset of the line in the input
                //l is a line of record entries from the input file
                splitLine[] ← l.split(",") //array of tokens from line delimited on ,
                obsvType ← splitLine[2]
                stationID ← splitLine[0]
                temperature ← splitLine[3]

                if obsvType equals "TMIN"
                        if HashMap contains stationID
                                Update HashMap,
                                        tMinSum ← tMinSum from Map + temperature
                                        tMinCount ← tMinCount from Map+1

                        else

                                Insert new record into the HashMap

                else if obsvType equals "TMAX"
                        if HashMap contains stationID
                                Update HashMap,
                                        tMaxSum ← tMaxSum from Map + temperature
                                        tMaxCount ← tMaxCount from Map + 1

                        else

                                Insert new record into the HashMap with new stationID as
                                key and with a StationStatsCombiner (tMaxSum ←
                                temperature, tMaxCount ← 1) as value

        }
```

```
        cleanup () {
            for each entry in HashMap
                emit(H .stationID, H.StationStatsCombinerValue)
        }
    }

    reduce (Text stationId, List<StationStatsCombiner> stnStatsComb) {

        for each stat in stnStatsComb
            tMaxSum ← tMaxSum + (tMaxSum of current stat)
            tMaxCount ← tMaxCount + (tMaxCount of current stat)
            tMinSum ← tMinSum + (tMinSum of current stat)
            tMinCount ← tMinCount + (tMinCount of current stat)
```

- Create output string (output) in the required format with the station id, mean TMIN and mean TMAX values

```
    emit(output, null)

    }
```

4. Secondary Sort
   a. Data structure used for accumulating temperature and calculating averages based on TMIN and TMAX by station id:
      StationStatsCombiner ← DoubleWritable tMaxSum, DoubleWritable tMaxCount, DoubleWritable tMinSum, DoubleWritable tMinCount
   b. Data structure used to create composite key for sorting:
      CompositeKey ← Text stationID, IntWritable year

```
map (offset o, recordLine l) {
    //The key o is the offset of the line in the input
    //l is a line of record entries from the input file
    splitLine[] ← l.split(",") //array of tokens from line delimited on ,
    obsvType ← splitLine[2]
    stationID ← splitLine[0]
    temperature ← splitLine[3]
    year ← splitLine[1].substring(0,4)
    compositeKey ← (stationID, year)
    if obsvType equals "TMIN"
        Update data structure stnStatCom,
            tMinSum ← temperature
            tMinCount ← 1

        emit(compositeKey, stnStatCom)
```

```
        else if obsvType equals "TMAX"
                Update data structure stnStatCom,
                        tMaxSum ← temperature
                        tMaxCount ← 1

                emit(compositeKey, stnStatCom)
        }

    reduce (CompositeKey cKey, List<StationStatsCombiner> stnStatsComb) {

        for each stat in stnStatsComb
```

- Accumulate TMIN and TMAX values till year is the same
  tMaxSum ← tMaxSum + (tMaxSum of current stat)
  tMaxCount ← tMaxCount + (tMaxCount of current stat)
  tMinSum ← tMinSum + (tMinSum of current stat)
  tMinCount ← tMinCount + (tMinCount of current stat)
- When year changes, write the values accumulated above to the output string (output) and start accumulating values for the new year.

```
        emit(output, null)
    }
```

Key Comparator:
This class extends the WritableComparator class and we override the compare method to sort the stations by stationID and for records with the same stationID the year is sorted in ascending order.

compare (key1, key2)

- Compare stationID of both keys, if both are equal, compare years of both keys and return the result of comparison
- If stationID of keys are not equal return result of comparison of stationIDs

Group Comparator:
This class extends the WritableComparator class and we override the compare method to group the stations by stationID.

compare (key1, key2)

- Compare the two keys based on the stationID and return the result of comparison to provide the grouping factor.

HashPartitioner:

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 2
Name: Neha Pradhan

This class extends the Partitioner class and we override the getPartition class to ensure that the stations with the same stationID are sent to the same reducer irrespective of the year which is part of the composite key. The getPartitioner method is the default one used by Hadoop.

Note: In the secondary sort program after the map emits the output records, the partitioner partitions the output records according to the criteria mentioned in the getPartition function after which the key comparator and group comparator sorts and groups the records based on the criteria mentioned in the implementation, in this case the criteria for sorting is the year and for grouping is the stationID.

Performance Comparison:

1. No Combiner
   Time taken in Run 1: 74 seconds
   Time taken in Run 2: 82 seconds

2. Combiner
   Time taken in Run 1: 68 seconds
   Time taken in Run 2: 71 seconds

3. In Mapper Combiner
   Time taken in Run 1: 69 seconds
   Time taken in Run 2: 71 seconds

Questions:

1. Was the Combiner called at all in program Combiner? Was it called more than once per Map task?
   The Combine input records field in the logs has a value of 8,798,758 and the Combine output records has a value of 223,795 which means that the combiner was called in the program and it combined the intermediate data that was output by the Map program. The field Merged Map outputs has a value of 153 which can be an indication that the Combiner was called more than once but there is no way to determine if it was called more than once per Map task.

2. Was the local aggregation effective in In Mapper Combiner compared to No Combiner?
   The local aggregation is very effective as it reduces the number of records that the reducer needs to process and it also ensures that the data transferred across the network is reduced compared to when no combiner is used. As seen in the logs after running on EMR the number of records output by map is reduced from 8,798,758 in no combiner to 223,795 in in mapper combiner which means that the reducer has less processing to do. The Reduce Shuffle Bytes too are reduced from 48,076,572 in no combiner versus

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 2
Name: Neha Pradhan

4,267,412 in in mapper combiner which again means there is less data to be transferred across the network thereby reducing overhead.

Secondary Sort

Time taken in Run 1: 49 seconds
Time taken in Run 2: 47 seconds