Name: Chandrika Sharma
CS: 6240
HW: #2

Map Reduce Algorithms:

Task1: Map Reduce Programs for Climate Analysis

Following is the pseudo code of the implementation of Climate Analysis program:
INPUT:
This program takes input one or more csv files.
OUTPUT:
Mean minimum temperature and the mean maximum temperature, by station, for a single year of data in the following format:
FORMAT:
(StationId1, AverageMinTemp1, AverageMaxTemp1)
(StationId2, AverageMinTemp2, AverageMaxTemp2)

Pseudo Codes:

NO-COMBINER IMPLEMENTATION

Map Phase:

```
Class Mapper:
        map (offset b, Record r){
        //b: Is the Byte Offset of the current line of text in the input file
        //r: Single record at that offset in the csv file
        //Note that each map call would receive each map call would receive each
        record of the input
        for each record do{
                //Pre-Processing: split record by ','
                records.toString().split(",")[0]<- (StationID,typeOfRecord,Temp)
                //Emiting the key -> StationId and value->HashMap {('TMAX':
        temp),('TMIN':temp)}
                emit(StationId,[( t1max,0 ,1,0),( 0,t2min,0,1)......])
        }
```

Reduce Phase:

```
Class REDUCER{
        Method Reduce(StationId s, [(t1max,0,1,0),(0,t2min,0,1).....]){
//Each entry in the list[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)...] is
the max and min temperatures of station s.
                for each h in the input list
                        total_maxTemp ← maxTemp +h.tmax
                        total_minTemp ← minTemp+ h.tmin
                        count +=1/0
        avgmax ← total_maxTemp/total_count
        avgmin ← total_maxTemp/total_count
        emit(s, avgmin, avgmax)}
```

COMBINER IMPLEMENTATION

Map Phase:

```
Class Mapper:
        Method map (offset b, Record r){
        //b: Is the Byte Offset of the current line of text in the input file
        //r: Single record at that offset in the csv file
        //Note that each map call would receive each map call would receive each
        record of the input
        //1/0 means either one or zero depending upon which type of record
        (tmax/tmin) was in that record.
        for each record in r do
                //Pre-Processing: split record by ‘,’
                records.toString().split(",")[0]<- (StationID,typeOfRecord,Temp)
                //Emiting the key -> StationId and value->HashMap {(‘TMAX’:
        temp),(‘TMIN’:temp)}
                emit(StationId,(tmax,1))or emit(StationId,(tmax,1))
        }
```

Combine Phase:

//Combine takes the same input as the reducer, It returns a more compact key value pair by aggregating the map outputs in an optimal manner

<u>Class Reducer:</u>

```
Method Combine(StationId s, [(t1max,1),(t2max1),(t3min,1)…])
    for each h in the input list
        total_maxTemp ← maxTemp +h.tmax
        total_minTemp ← minTemp+ h.tmin
        tmaxcount+=1
        tmincount +=1
    emit(s, pairs(tmax, tmin, tmaxcount, tmincount))
}
```

<u>Reduce Phase:</u>

Class Reduce:

//Each entry in the list[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)…] is the max and min temperatures of station s

```
Method Reduce(StationId s,
[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)…]){
    for each h in the input list
        total_maxTemp ← maxTemp +h.tmax
        total_minTemp ← minTemp+ h.tmin
        count +=1
    avgmax ← total_maxTemp/total_count
    avgmin ← total_maxTemp/total_count
    emit(s, pair(avgmin, avgmax))
}
```

IN-MAPPER COMBINER IMPLEMENTATION:

<u>Map Phase</u>:

```
Class Mapper:
    Method Setup:
        H← Global HashMap
    Method map (offset b, Record r){
```

```
//b: Is the Byte Offset of the current line of text in the input file
//r: Single record at that offset in the csv file
//Note that each map call would receive each map call would receive each
record of the input
//1/0 means either one or zero depending upon which type of record
(tmax/tmin) was in that record.
for each record in r do
    //Pre-Processing: split record by ','
    records.toString().split(",")[0]<- (StationID,typeOfRecord,Temp)
    //Emiting the key -> StationId and value->HashMap {('TMAX':
temp),('TMIN':temp)}
    for each h in the input list
        total_maxTemp ← maxTemp +h.tmax
        total_minTemp ← minTemp+ h.tmin
        tmaxcount+= maxcount
        tmincount +=mincount
        H{S:tmax,tmin,maccount,mincount}← update values for each
station

For all terms in the csv file, a Hash Map gets updated as and when the
record is processed.
    emit(StationId,HashMap H)
}
```

Reduce Phase:
Class Reduce:

```
//Each entry in the list[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)…] is
the max and min temperatures of station s
Method Reduce(StationId s,
[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)…]){
    for each h in the input list
        total_maxTemp ← maxTemp +h.tmax
        total_minTemp ← minTemp+ h.tmin
        totalTmaxCount+=1
        totalTminCount+=1
    avgmax ← total_maxTemp/ totalTmaxCount
    avgmin ← total_maxTemp/ totalTmaxCount
emit(s, avgmin, avgmax)}
```

2. Following is the pseudo code of the implementation of Climate Analysis program:

INPUT:
This program takes input one or more csv files.

OUTPUT:
Mean minimum temperature and the mean maximum temperature, by station, for a multiple years of data in the following format:

FORMAT:
(StationId1, Year: AverageMinTemp1, AverageMaxTemp1)
(StationId2, Year: AverageMinTemp2, AverageMaxTemp2)

<u>SECONDARY-SORT IMPLEMENTATION</u>

A solution for secondary sorting involves doing multiple things. First, instead of simply emitting the Station Id as the key from the mapper, we need to emit a composite key, a key that has multiple parts. The key will have the Station ID and the Year.

Now we want that the following happens:
Map –> ({Station ID, Year},Type, Temprature,Count)
Reduce({Station ID, Year},list[Type, Temperature, Count]) –>(StationID, AvgTmax, AvgTmin)
{Station ID, Year} is a composite key, but inside it, the Station ID is referred to as the “natural” key. It is the key which values will be grouped by.

<u>Map Phase</u>:
```
Map(StationId, date, temperatures){
        Emit((StationId, date.year),
[(t1max,t1min,1,1),(t2max,t2min,1,1),(t3max,t3min,1,1)…])}
```

<u>Partition Phase</u>:
```
getPartition(StationId,year){
        return myPartition(StationId)
}
```

<u>Grouping Comparator Phase</u>
```
GroupingComparator(StationId,Year){
//Sorts in increasing order of the StationId first and then Year
//Hence 2 keys with same value are considered same
}
```

Reduce Phase

```
reduce((StationId), list[Type, Temperature, Count]){
        as data grouped now, simply calculate the avg temp max and min and

emit (StationId, AvgTmin, AvgTmax)

}
```

SPARK-SCALA PROGRAMS:

In the following programs I have used the following types of aggregation methods:

1. **ReduceByKey**: This aggregation function aggregates all input records based on the key passed. This looks like out reduce function in Map Reduce.
2. **Map**: is a transformation that passes each dataset element through a function and returns a new RDD representing the results
3. **CombineByKey**: Acts as a combiner, Both InMapper Combiner or Combiner as it does the job of a MapReduce combiner. What ever key is passed on to it, it combines the values of that key based on an operation. In Out Case it Summed up all the Tempratures.

NO-COMBINER IMPLEMENTATION

```
object Combiner {
  def main(args: Array[String]) = {
   //Start the Spark context

    val conf = new SparkConf()
      .setAppName("Combiner")
      .setMaster("local")
      val sc = new SparkContext(conf)
//Reading file—————————————————————————
    val pairRDD = sc.textFile("/Users/chandrikasharma/Downloads/1991.csv")

    val maxTemps = pairRDD.map(line => line.split(","))
                      .filter(fields => fields(2) =="TMAX" ||fields(2) =="TMIN")
                      .map(fields => ((fields(0),fields(2)), Integer.parseInt(fields(3))))
//————————-used ReduceByKey Function
    val sumByKey=maxTemps.reduceByKey((sum,temp) => temp + sum)
//————————-used FoldByKey Function

    val countByKey = maxTemps.foldByKey(0)((ct, temp) => ct + 1)
//————————-used CombineByKey Function
    val combinerI = avgByKey.combineByKey(
       value => (value,1),
       (acc1:(Double,Double),x)=>(acc1._1+value.toDouble, acc1._2+1),
       (acc2:(Double, Double), acc3:(Double, Double)) => (acc1._1 + acc2._1, acc1._2
+ acc2._2))
    val avgByKey = combinerI.map(t => (t._1, t._2._1 / t._2._2.toDouble))

    val maxKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMAX").map(t=>(t._1._1,t._2))

    val minKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMIN").map(t=>(t._1._1,t._2))
    val result = maxKeyVal.fullOuterJoin(minKeyVal).map {
           case (id, (left, right)) =>(id, (left.getOrElse(0.00)).toString + ( ","+
right.getOrElse(0.00)).toString)}



    result.saveAsTextFile("Combiner")
    sc.stop

  }
}
```

NO-COMBINER IMPLEMENTATION:

```
object NoCombiner {
  def main(args: Array[String]) = {
   //Start the Spark context

    val conf = new SparkConf()
      .setAppName("NoCombiner")
      .setMaster("local")
      val sc = new SparkContext(conf)
    val pairRDD = sc.textFile("/Users/chandrikasharma/Downloads/1991.csv")
//    val pairRDD = sc.textFile("abc.csv")
    val maxTemps = pairRDD.map(line => line.split(","))
                      .filter(fields => fields(2) =="TMAX" ||fields(2) =="TMIN")
                      .map(fields => ((fields(0),fields(2)), Integer.parseInt(fields(3))))
    val sumByKey=maxTemps.reduceByKey((sum,temp) => temp + sum)
    val countByKey = maxTemps.foldByKey(0)((ct, temp) => ct + 1)
    val sumCountByKey = sumByKey.join(countByKey)
    val avgByKey = sumCountByKey.map(t => (t._1, t._2._1 / t._2._2.toDouble))
    avgByKey.foreach(println)
    val maxKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMAX").map(t=>(t._1._1,t._2))

    val minKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMIN").map(t=>(t._1._1,t._2))
    val result = maxKeyVal.fullOuterJoin(minKeyVal).map {
             case (id, (left, right)) =>(id, (left.getOrElse(0.00)).toString + ( ","+
right.getOrElse(0.00)).toString)}



    result.saveAsTextFile("NoCombiner")
    sc.stop

  }

}
```

SECONDARY SORT IMPLEMENTATION

```
object secondarySort {
 def main(args: Array[String]) = {
  //Start the Spark context

   val conf = new SparkConf()
    .setAppName("SecondarySort")
    .setMaster("local")
    val sc = new SparkContext(conf)
   val pairRDD = sc.textFile("/Users/chandrikasharma/Downloads/SecSordFolder”)

   val maxTemps = pairRDD.map(line => line.split(","))
                      .filter(fields => fields(2) =="TMAX" ||fields(2) =="TMIN")
                      .map(fields => ((fields(0),fields(2)),(fields(1),
       Integer.parseInt(fields(3)))))
//———————————————————using GroupByKey

   val secSort =  maxTemps.groupByKey()
                       .sortBy(_._1, ascending=true)

//———————————————————using RedeuceByKey

   val sumByKey=maxTemps.reduceByKey((sum,temp) => temp + sum)

//———————————————————using foldByKey

   val countByKey = maxTemps.foldByKey(0)((ct, temp) => ct + 1)

//———————————————————using sumByKey

   val sumCountByKey = sumByKey.join(countByKey)
   val avgByKey = sumCountByKey.map(t => (t._1, t._2._1 / t._2._2.toDouble))
   val maxKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMAX").map(t=>(t._1._1,t._2))
   val minKeyVal =  avgByKey.filter(t=>(t._1._2)=="TMIN").map(t=>(t._1._1,t._2))
   val result = maxKeyVal.fullOuterJoin(minKeyVal).map {
```

```
            case (id, (left, right)) =>(id, (left.getOrElse(0.00)).toString + ( ","+
right.getOrElse(0.00)).toString)}

    result.saveAsTextFile("SecondarySort")
    sc.stop }}
```

PERFORMANCE COMPARISIONS:

Program	Time taken attempt 1	Time Taken attempt 2	
No Combiner	124 seconds	124 Seconds	
Combiner	116 seconds	112 seconds	
In-Mapper	86 seconds	86 seconds	

No-combine

```
Map input records=31688662
Map output records=9213198
Map output bytes=515939088
Combine input records=0
Reduce input records=9213198
Reduce input groups=14723
Combine output records=0
```

Combine

```
Map input records=31688662
Map output records=9213198
Map output bytes=414593910
Combine input records=9213198
Combine output records=233643
Reduce input records=233643
Reduce input groups=14723
```

In-Mapper:

```
Map input records=31688662
Map output records=233643
Map output bytes=17055939
Combine input records=0
Combine output records=0
Reduce input groups=14723
Reduce shuffle bytes=5442761
Reduce input records=233643
Reduce output records=14723
```

Important Terms in the Sys-log:

Map input records:

This is the number of Records inputted into the Mapper Function. It remains same throughout each program.

Map output records:

This is the number of Records outputted from the Mapper Function. This reduces drastically when we run Combiner and In-Mapper as the local aggregation occurs.

Map output bytes:

This is very interesting, This value changes drastically from No-Combiner to Combine. Also then from Combine to In Mapper combine. This could be because Of how in Combiner, only the combination happens locally and not globally but in In-Mapper the combination task is global and hence the Bytes transferred from The In-Mapper to reducer is much lesser than the bytes passed on to the reducer from a combiner step.

Q1. From the data above, I can deduce that in Combiner Implementation since the number of records inputted into the combiner are clearly mentioned, its definitely been called once. But it is difficult to predict whether the combiners are being called more than once. This could depend on a lot of factors. Also in the syslog It only mentions the number of Map Tasks and no mention of the number of Combiner Calls per map Task so it’s difficult to say.

Q2:Yes the local aggregation was effective in In-Mapper.
As per the following information:
No-Combiner - Map output records=9213198
In-Mapper - Map output records=233643
By looking at the data mentioned, we can easily say that the records are being aggregated locally in the In-Mapper because the map’s output is much lesser in comparision with the No-Combine implementation

Secondary Sort program gave me a casting error on EMR however I have attached the result of the local running of my program for partial grades. You can find the output in the MRAssignment2Results.