Homework2

Name: Yang Cai
NUID: 001632759

Map-Reduce Algorithm:

a). NoCombiner:

```
Map(filename, line) {
   station, type, temp = parse(line)
   if (type == “TMAX”) emit(station, (temp, 1, 0, 0));
   else if (type == “TMIN”) emit(station, (0, 0, temp, 1));
}
Reducer(stationId, [r1, r2, ….rn]) {
   minCt = 0, minSum = 0, maxCt = 0, maxSum = 0;
   for each r in [r1….rn]:
      minCt += r[3], minSum += r[2], maxCt += r[1], maxSum += r[0]
   emit(stationId, (maxCt == 0 ? NA : maxSum/ maxCt, minCt == 0 ? NA ? minSum/ minCt)
}
```

b). Combiner:

The Mapper and Reducer is the same as above.

```
Combiner(stationId, [r1, r2, ….rn]) {
   minCt = 0, minSum = 0, maxCt = 0, maxSum = 0;
   for each r in [r1….rn]:
      minCt += r[3], minSum += r[2], maxCt += r[1], maxSum += r[0]
   emit(stationId, (maxSum, maxCt, minSum, minCt))
}
```

c). In Mapper Combiner:

The Reducer is the same.

```
Class Map {
   Map<String, Record> map = new HashMap;
   Map(file, line) {
      station, type, temp = parse(line)
      if (type == “TMAX”) map.put(station, map.getOrDefault(station, (0, 0, 0, 0)) + (temp, 1, 0,
0));
      else if (type == “TMIN”) map.put(station, map.getOrDefault(station, (0, 0, 0, 0)) + (temp, 1,
0, 0));
   }
   CleanUp() {
      for each key in map:
         emit(key, map.get(key))
   }
```

d). Secondary Sort

```
Map(filename ,line) {
   station, year, type, temp = parse(line)
   if (type == “TMAX”) emit((station, year), (year, temp, 1, 0, 0));
   else if (type == “TMIN”) emit((station, year), (year, 0, 0, temp, 1));
}

getPartition(key, value, numPartitions) {
   return abs(key.station.hash) % numPartitions;
}

KeyComparator {
   int compare(k1, k2) {
      int res = k1.station.compareTo(k2.station)
      if (res == 0) res = k1.year.compareTo(k2.year)
      return res;
   }
}

Reduce((station, firstyear), [r1, r2, r3 …..rn]) {
   List = new ArrayList<>();
   minCt = 0, minSum = 0, maxCt = 0, maxCt = 0, year = 0
   for each r in [r1….rn]: {
      if year != r.year {
          lst.add((year, minSum / minCt, maxSum / maxCt))
          minCt = 0, minSum = 0, maxCt = 0, maxCt = 0
      }
      year = r.year
      minCt += r.minCt
      minSum += r.minSum
      maxCt += r.maxCt
      maxSum += maxSum
   }
   lst.add((year, minSum / minCt, maxSum / maxCt))
   emit(station, lst)
}

GroupingComparator {
   int compare(k1, k2) {
      return k1.station.compareTo(k2.station)
   }
}
```

Explanation: For reduce call, it will receive all records for the same station, in increasing order of year. For example, the reduce will receive key (station1, year), so all values will belong to the same station1, in the order of year:1880, 1881 …. 1889.

Spark Scala Program:

Data representation: I mainly use pairRDD and and RDD to manipiulate data. Since pairRDD has key and value, so it is perfectly good for mapreduce framework. We can benefit a lot from pariRDD functions, such as reduceByKey, groupByKey, etc. They can be used to correspond to the Mapreduce's reduce task. RDD is used for some intermediate data. Dataframe is a special kind of DataSet: Dataframe = DataSet[Row]. I don't use DataSet/Dataframe since they are like SQL tables, it is good to manipulate the using SQL, but they are not good match for Mapreduce frame work.

Part 1 Scala program:

The source file is already included in folder, here is the copy, the NoCombiner / Combiner / InMapperCombiner functions are all in the same file, they will be executed according to command line arguments:

```
package CS6240.WeatherScala

import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD
import scala.collection.mutable.HashMap
import org.apache.spark.Partitioner
import org.apache.spark.HashPartitioner
import java.io.File

object Analyze {
  // accumulate data structrue to store temperature information
  case class AccumEntry(maxSum: Int, maxCt: Int, minSum: Int,
minCt: Int)

  // combine two AccumEntry into one AccumEntry
  def combEntry(r1: AccumEntry, r2: AccumEntry): AccumEntry = {
    AccumEntry(r1.maxSum + r2.maxSum, r1.maxCt + r2.maxCt,
r2.minSum + r2.minSum, r2.minCt + r2.maxSum)
  }

  // convert one line in .csv to (station, maxtemp, mintemp)
  def lintToInputEntry(arr: Array[String]): (String, (Option[Int],
Option[Int])) = {
    (arr(0),
      (if (arr(2) == "TMAX") Some(arr(3).toInt) else None,
        if (arr(2) == "TMIN") Some(arr(3).toInt) else None))
  }

  // convert (station, maxtemp, mintemp) to accumulate data
structrue
  def InputToAccumEntry(r: (String, (Option[Int], Option[Int]))):
(String, AccumEntry) = {
    (r._1, if (r._2._1.isEmpty) AccumEntry(0, 0, r._2._2.get, 1)
else AccumEntry(r._2._1.get, 1, 0, 0))
  }

  // compute the average temperature
  def AccumEntryToAve(v: AccumEntry): (Double, Double) = {
    (if (v.maxCt == 0) Double.NaN else 1.0 * v.maxSum / v.maxCt,
      if (v.minCt == 0) Double.NaN else 1.0 * v.minSum / v.minCt)
  }

  // function without combiner
  def NoCombiner(rdd: RDD[(String, (Option[Int], Option[Int]))]):
RDD[(String, (Double, Double))] = {
    rdd.map(InputToAccumEntry)
      .groupByKey()
      // convert list of entries to averages
      .mapValues(entries =>
AccumEntryToAve(entries.foldLeft(AccumEntry(0, 0, 0,
0))(combEntry)))
  }

  // since reduceByKey will automatically combine in each
partition, so this function
  // has combiner implicit
  def WithCombiner(rdd: RDD[(String, (Option[Int], Option[Int]))]):
RDD[(String, (Double, Double))] = {
    rdd.map(InputToAccumEntry)
      .reduceByKey(combEntry)
      .mapValues(AccumEntryToAve)
  }

  // function has in-mapper combiner in each mapper task
  def inMapperCombiner(rdd: RDD[(String, (Option[Int],
Option[Int]))]): RDD[(String, (Double, Double))] = {
    rdd.mapPartitions { records =>
      // set up a hasmap and accumulate result in hashmap
      val map: HashMap[String, AccumEntry] = HashMap()
      records.foreach(r => map += (r._1 ->
combEntry(map.getOrElse(r._1, AccumEntry(0, 0, 0, 0)),
InputToAccumEntry(r)._2)))
      map.iterator
    }.reduceByKey(combEntry).mapValues(AccumEntryToAve)
  }

  def main(args: Array[String]) = {
    val conf = new SparkConf()
      .setAppName("Weather Data")
      .setMaster("local")
    val sc = new SparkContext(conf)
    val spark = SparkSession.builder().getOrCreate()
    val test = sc.textFile(args(1))
    // filter out "TMAX"/"TMIN", and convert to (station. maxtemp,
mintemp)
    val res = test.map(line => line.split(","))
      .filter(arr => arr(2) == "TMAX" || arr(2) == "TMIN")
      .map(lintToInputEntry)
    // map command line argument to each function:
NoCombiner/WithCombiner/inMapperCombiner
    def mapFunctions = Map[String,
      ((RDD[(String, (Option[Int], Option[Int]))]) => RDD[(String,
(Double, Double))])](
        "NoCombiner" -> NoCombiner,
        "WithCombiner" -> WithCombiner,
        "inMapperCombiner" -> inMapperCombiner
        )
    // execute the corresponding function and save to file
    mapFunctions(args(0))(res).saveAsTextFile(args(2))
  }
}
```

Part 1 Scala program Discussion:

a). For NoCombiner:

I use map to first achieve map task, then use groupBy to aggregate all the values for the same key, at last use mapValues to combiner these list of values and compute average temperature.
b). For Combiner:
I still use map to emit each record (simulating map tasks), and then directly call reduceByKey and mapValues to aggragate values. We can do this since Spark's reduceByKey will automatically reduce on each partition, and then reduce on all paritions. In other words, Spark's reduceByKey will do the combiner's job.
c). For In-Mapper Combiner:
We first call mapPartitions to aggregate values within one partition, then reduceByKey to aggregate across all the partitions and mapValues to compute averages. This is similar to in-mapper combiner, since in mapPartion(), I set up a hashmap to accumulate each record and emit these accumulate result at end, so the map tasks wil emit the accumulate values directly, this is the same as in-mapper combiner.

Part 2 Scala program:

This program is also included in the source files, but I still copy it to here:

```
package CS6240.WeatherScala

import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import scala.collection.mutable.HashMap
import org.apache.spark.HashPartitioner
import org.apache.spark.Partitioner

object AnalyzeSecondarySort {
   // accumulate data structrue to store temperature information
  case class AccumEntry(maxSum: Int, maxCt: Int, minSum: Int,
minCt: Int)

  // accumulate data structrue for one year
  case class YearAccumEntry(year: Int, entry: AccumEntry)

  // key in the secondary sort
  case class SecondaryKey(stationId: String, year: Int)

  // key comparator, first sort on station and then by year
  object SecondaryKey {
    implicit def orderByStationAndYear[A <: SecondaryKey]:
Ordering[A] = {
      Ordering.by(e => (e.stationId, -1 * e.year))
    }
  }

  // key partitioner based onlt on station id
  class SecondaryPartitioner(partitions: Int) extends Partitioner {
    require(partitions >= 0, s"Number of partitions ($partitions)
cannot be negative.")

    override def numPartitions: Int = partitions
    val delegate = new HashPartitioner(partitions)
    override def getPartition(key: Any): Int = {
      val k = key.asInstanceOf[SecondaryKey]
      delegate.getPartition(k.stationId)
    }
  }

  // secondary sort, make use of
repartitionAndSortWithinPartitions. And by calling mapPartitions
  // we are aggragating records for each stationa and year
  def SecondarySort(rdd: RDD[(SecondaryKey, AccumEntry)]):
RDD[(String, List[(Int, Double, Double)])] = {
    rdd.repartitionAndSortWithinPartitions(new
SecondaryPartitioner(rdd.partitions.size))
      .mapPartitions { iter =>
        val map = HashMap[String, List[(Int, Double, Double)]]()
        val last = iter.foldLeft(("Dummy", YearAccumEntry(0,
AccumEntry(0, 0, 0, 0)))) {
          (accm, ele) =>
            if (accm._1 == ele._1.stationId) {
              val id = accm._1
              if (accm._2.year == ele._1.year) {
                // if both year and station are the same, we
accumulate the data
                val year = accm._2.year
                val oldAccmEntry = accm._2.entry
                val newAccmEntry = AccumEntry(oldAccmEntry.maxSum +
ele._2.maxSum,
                  oldAccmEntry.maxCt + ele._2.maxCt,
                  oldAccmEntry.minSum + ele._2.minSum,
                  oldAccmEntry.minCt + ele._2.minCt)
                (id, YearAccumEntry(year, newAccmEntry))
              } else {
                // if it is different year but same station, we
comnpute the averages and append
                // to stations's list
                val lst = map.getOrElse(id, List())
                val e = accm._2.entry
                map += (id -> ((accm._2.year, e.maxSum * 1.0 /
e.maxCt, e.minSum * 1.0 / e.minCt) :: lst))
                (id, YearAccumEntry(ele._1.year, ele._2))
              }
            } else {
              // if it a different station, we compute the average
and put a new record in hasmap
              if (accm._1 != "Dummy") {
                val lst = map.getOrElse(accm._1, List())
                val e = accm._2.entry
                map += (accm._1 -> ((accm._2.year, e.maxSum * 1.0 /
e.maxCt, e.minSum * 1.0 / e.minCt) :: lst))
              }
              (ele._1.stationId, YearAccumEntry(ele._1.year,
ele._2))
            }
        }
        map.iterator
      }
  }

  // convert one line in .csv to (station, maxtemp, mintemp)
  def lintToInputEntry(arr: Array[String]): (String, (Option[Int],
Option[Int])) = {
    (arr(0),
      (if (arr(2) == "TMAX") Some(arr(3).toInt) else None,
        if (arr(2) == "TMIN") Some(arr(3).toInt) else None))
  }

  // convert (station, maxtemp, mintemp) to accumulate data
structrue
  def InputToAccumEntry(r: (String, (Option[Int], Option[Int]))):
(String, AccumEntry) = {
    (r._1, if (r._2._1.isEmpty) AccumEntry(0, 0, r._2._2.get, 1)
else AccumEntry(r._2._1.get, 1, 0, 0))
  }
  def main(args: Array[String]) = {
    val conf = new SparkConf()
      .setAppName("WordCount")
      .setMaster("local")
    val sc = new SparkContext(conf)
    val spark = SparkSession.builder().getOrCreate()
    val test = sc.textFile(args(0))
    val res = test.map(line => line.split(","))
      .filter(arr => arr(2) == "TMAX" || arr(2) == "TMIN")
      .map { arr =>
        val secondaryKey = SecondaryKey(arr(0), arr(1).substring(0,
4).toInt)
        val accumEntry =
InputToAccumEntry(lintToInputEntry(arr))._2
        (secondaryKey, accumEntry)
      }
    SecondarySort(res).saveAsTextFile(args(1))
  }
}
```

Performance Comparison:

Performance of part 1:

No Combiner run 1: 84 seconds
No Combiner run 2: 82 seconds
Combiner run 1: 80 seconds
Combiner run 2: 78 seconds
In Mapper Combiner run 1: 76 seconds
In Mapper Combiner run 2: 76 seconds

Question1:

Yes, Combiner is called in program Combiner. It is called more than once in each Mapper task, since in the syslog, Map output records=8798758, and Combine input records=8798758, they are the same, which means there is one combine() call for each key of one Map Task. Also the program Combiner's combine output is exactly the same as In-Mapper Combiner's Reducer's input. These could be used to verify that combiner is called for every key in each Map task, in other words, it is called more than once.

Question2:

Yes, In-Mapper Combiner is effective compared to NoCombiner. Since the time of In-Mapper Combiner is much less than NoCombiner(76 vs 84 seconds). Also for In-Mapper Combiner, the mapper's output is 9846980 bytes, and No-Combiner is 387145352. The in-mapper combiner Has much less data to be transferred.

Question3:

Secondary Sort runtime 1: 56 seconds
Secondary Sort runtime 2: 56 seconds