Map-Reduce Algorithm

NoCombiner Pseudocode:

```
map (…, Line L) {
        extract stationID, type and temperature from line        //type can either be TMAX or TMIN
        emit [stationID, {type, temperature}]
}
reduce (stationID, List of [type, temperature]) {
         Calculate TMAXSum and TMAXCount if type = TMAX;
        Calculate TMINSum and TMINCount if type = TMIN;
        Calculate MeanTMAX = TMAXSum/TMAXCount;
        Calculate MeanTMIN = TMINSum/TMINCount;
        emit [stationID, {MeanTMIN, MeanTMAX}];
}
```

Combiner Pseudocode:

```
map (…, Line L) {
        extract stationID, type and temperature from line        //type can either be TMAX or TMIN
        emit [stationID, {type, temperature}]
}
combine (stationID, List of [type, temperature]) {
         Calculate localTMAXSum and localTMAXCount if type = TMAX;
        Calculate localTMINSum and localTMINCount if type = TMIN;
        emit [stationID, {type, localTMINSum, localTMINCount}];
        emit [stationID, {type, localTMAXSum, localTMAXCount}];
}
reduce (stationID, List of [type, localTempSum, localCount]) {
         Calculate totalTMAXSum and totalTMAXCount if type = TMAX;
        Calculate totalTMINSum and totalTMINCount if type = TMIN;
        Calculate MeanTMAX = totalTMAXSum/totalTMAXCount;
        Calculate MeanTMIN = totalTMINSum/totalTMINCount;
        emit [stationID, {MeanTMIN, MeanTMAX}];
}
```

InMapperComb Pseudocode:

```
Class Mapper {
HM = HashMap<StationID, List<TMINtemp, TMINCount, TMAXtemp, TMAXCount>>();
setup () {
    HM = new HashMap();
    }
map (…, Line L) {
    Extract stationID and type from line L;
    For stationID,
    Calculate TMAXSum and TMAXCount if type = TMAX;
    Calculate TMINSum and TMINCount if type = TMIN;
    Update HM;
    }
cleanup () {
    For each stationID in HM,
        emit [stationID, {type, TMINSum, TMINCount}];
        emit [stationID, {type, TMAXSum, TMAXCount}];
    }
}
reduce (stationID, List of [type, localTempSum, localCount]) {
     Calculate totalTMAXSum and totalTMAXCount if type = TMAX;
    Calculate totalTMINSum and totalTMINCount if type = TMIN;
    Calculate MeanTMAX = totalTMAXSum/totalTMAXCount;
    Calculate MeanTMIN = totalTMINSum/totalTMINCount;
    emit [stationID, {MeanTMIN, MeanTMAX}];
}
```

SecondarySort Pseudocode:

```
map (…, Line L) {
    extract stationID, type, year and temperature from line L        //type can either be TMAX or
    TMIN
    emit [{year, stationID}, {type, temperature, year}]
}
partitioner ({year, stationID} k, {type, temperature, year} v) {
    for each stationID from key k,
        partition=calculate hashcode() of stationID % numberofreducetask;
    return partition;
}
keysortcomparator ({year, stationID} k1, {year, stationID} k2) {
    extract stationID from k1 and k2; //sid1, sid2
    compare sid1 == sid2
    if compare == 0,
                return year1.compareTo(year2);
}
secondarysortgroupingcomparator({year, stationID} k1, {year, stationID} k2) {
        extract stationID from k1 and k2; //sid1, sid2
        return sid1.compareTo(sid2)
}
LinkedHM = HashMap<year, List<TMINtemp, TMINCount, TMAXtemp, TMAXCount>> ();
reduce ({year, stationID} k, List of {(type, temperature, year) v} Li) {
        Extract year from each value v in List Li
        For each year,
                Calculate TMAXSum and TMAXCount if type = TMAX;
                Calculate TMINSum and TMINCount if type = TMIN;
                Update LinkedHM with year as key

        For each year in LinkedHM,
                Calculate meanMinTemp and meanMaxTemp;

        Extract stationID from key k;
        emit [ stationID, {year, meanMinTemp, meanMaxTemp}];
}
```

Spark-Scala Programs

Briefly discuss where in your program—and why—you chose to use which of the following data representations: RDD, pair RDD, DataSet, DataFrame.

In NoCombiner, Combiner and InMapperCombiner, I would use the RDD (Resilient Distributed Data) because the input to the map, is unstructured or semi-structured. For example, the columns contain different types of data, like for example, it has TMAX, PRCP, TMIN values all in the same column.
RDD would be better to use, since I wouldn't require imposing a schema, such as columnar format as conditions in my programs are being handled and calculations are done.
The csv file is loaded into text format by Hadoop, and it is stored in a list of lines as an RDD which is immutable and also because the data semi-structured **map()** method is used to split the lines by ','
We extract only those which have temperature type as either "TMAX" or "TMIN". We use the **filter()** method to achieve this.
Now, we need to convert RDD to pairRDD, where in stationID is key and temperature is value.
In Secondary Sort, I would use PairRDD since the key I'm using in the mapper and reducer is a combination of stationID and year which makes it easier to manipulate the calculation.

Show the Spark Scala programs you wrote for part 1 (mean min and max temperature for each station in a single year). If you do not have a fully functional program, discuss the Scala commands your program should use.

NoCombiner, Combiner, and InMapperCombiner

Line.split(",").foreach, command to split the line and perform the emit function in the mapper.
.collect(key, value), command to emit (Key, Value) pair
values.toList.reduceLeft(val1, val2) command to find the sumMinTemp, minCount, sumMaxTemp, maxCount based on the type of temperature condition
find, command to filter the results based on temperature type(TMAX and TMIN)

Discuss the choice of aggregate function for the first problem (see step 3 above). In particular, which Spark Scala function(s) implement(s) NoCombiner, Combiner, and InMapperComb; and why?

NoCombiner

groupByKey() function can be used on the pairRDD which would result in partition of data, all shuffled over the network as (key, value).

Combiner

combineByKey() which can be used to implement our own combiner.

InMapperCombiner

reduceByKey() can be used which returns dataset of (k,v) pairs, i.e., list of (k,v) pairs.
After the above functionality, which calculates the sum for TMAX and TMIN. Then average can be calculated.

Show the Spark Scala programs you wrote for part 2 (10-year time series per station). If you do not have a fully functional program, discuss the Scala commands your program should use.

The file is read splitting is done on each line of input using the map() function.
We create a combination of key that consists of "stationID" and "year". This is represented by creating a **case** class that would return a CompositeKey.
Then we can use a function which would return list containing only the values we will use to calculate the sum of MAXTemp and MINTemp.
For sorting, Spark is told explicitly that data needs to be sorted first by "stationID" and then by "year" if the stationID matches.
We create a helper object for Composite Key and define an implicit ordering method which orders based on year.
Extending "Ordered" class would imply sorting only by a single 'natural' value and not on a composite key.
After this, meanMINTemp and meanMAXTemp are calculated.

Performance Comparison

Program	**Running Time 1 (in seconds)**	**Running Time 2 (in seconds)**
NoCombiner	86	80
Combiner	84	82
InMapperComb	80	78

NoCombiner

```
Map input records=30870343
Map output records=8798758
Map output bytes=182702250
Map output materialized bytes=50899494
Input split bytes=1683
Combine input records=0
Combine output records=0
Reduce input groups=14136
Reduce shuffle bytes=50899494
Reduce input records=8798758
Reduce output records=14136
```

Combiner

Map input records=30870343
Map output records=8798758
Map output bytes=182702250
Map output materialized bytes=5590788
Input split bytes=1683
Combine input records=8798758
Combine output records=447590
Reduce input groups=14136
Reduce shuffle bytes=5590788
Reduce input records=447590
Reduce output records=14136

InMapperComb

Map input records=30870343
Map output records=447590
Map output bytes=12889297
Map output materialized bytes=5643645
Input split bytes=1683
Combine input records=0
Combine output records=0
Reduce input groups=14136
Reduce shuffle bytes=5643645
Reduce input records=447590
Reduce output records=14136

Description:

- **Map input records** are the number of lines or data in the 1991.csv file.
- **Map output bytes** is the number of bytes of uncompressed output produced by all the maps in the job. Since the mapper in NoCombiner and Combiner is exactly same, the value is same and is much larger than the value in InMapperComb because the values are combined inside the map.
- **Reduce Input records** are the number of values that go into the reducer. The reducer input is same for Combiner and InMapperComb since the records are combined in the two and are much less than the value for NoCombiner where no records are combined.
- **Reduce Input Groups** are the number of unique keys that were fed to the reducer. It is the same for all the programs since there are that many number of unique stationID in the input file.
- **Combine Output Records** are the number of values passed to the combiners. The value is zero in case of NoCombiner and InMapperComb since there are no Combiner calls.
- **Map Output Records** are the number of records that the mapper emits. Since the mapper in NoCombiner and Combiner is exactly same, the value is same and is much larger than the value in InMapperComb because the values are combined inside the map.

Was the Combiner called at all in program Combiner? Was it called more than once per Map task?

Yes, the combiner was called in the program Combiner. No, it wasn't called more than once per Map task. It was exactly once per Map task.

Was the local aggregation effective in InMapperComb compared to NoCombiner?

In terms of running time, the InMapperComb was more effective in comparison to NoCombiner since a local aggregation was done in the Mapper class where values of same stationID were grouped together in the same map task unlike NoCombiner where multiple numbers of records having same key were sent to reducer call thus increasing the running time.

Run the program from part 2 (secondary sort) above in Elastic MapReduce (EMR), using six m4.large machines (1 master, 5 workers). Report its running time.

Secondary Sort running time = 58 seconds.