1. Map-reduce Algorithm

(1) no combine

```
map(offset key, string value) {
    emit(stationId, (tempType, tempValue));
}

reduce(stationId, [(tempType1, tempValue1), ...]) {
    maxTotal = 0;
    maxNums = 0;
    minTotal = 0;
    maxNums = 0;

    for each pair in inputList
          if (pair.tempType = "MAX"){
             maxTotal += pair.tempValue,
             maxNums++;
          }
          if (pair.tempType = "MIN"){
              minTotal += pair.tempValue,
              minNums++;
          }
    emit(stationId, maxTotal / maxNums, minTotal / minNums)
}
```

(2) with combiner

```
map(offset key, string value) {
    emit(stationId, (tempType, tempValue));
}

combiner: code is exactly same as reducer

reduce(stationId, [(tempType1, tempValue1), ...]) {
    maxTotal = 0;
    maxNums = 0;
    minTotal = 0;
    maxNums = 0;

    for each pair in inputList
          if (pair.tempType = "MAX"){
             maxTotal += pair.tempValue,
```

```
            maxNums++;
        }
        if (pair.tempType = "MIN"){
            minTotal += pair.tempValue,
            minNums++;
        }
    emit(stationId, maxTotal / maxNums, minTotal / minNums)
}
```

(3) in mapper combining

```
class Mapper() {
  HashMap H;

  setup(){
     H = new HashMap()<string, int[]>
  }

   map(offset key, string line) {
      for each line
         split line into 3 part (stationId, tempType, tempValue)
         construct map by definition
   }

   cleanup() {
      for each entry in map
         emit(entry.key, entry,value)
   }
}

class Reducer{
   reduce(stationId, [[maxTotal0, maxNums0, minTotal0, minNums0], ..., ...]){
      maxNums = 0
      maxTotal = 0
      minNums = 0
      minTotal = 0
      for each array in inputList
         maxTotal += array[0]
         maxNums += array[1]
         minTotal += array[2]
         minNums += array[3]
       emit(stationId, maxTotal / maxNums, minTotal / minNums)
   }
}
```

(4) time series

```
map(offset key, string line) {
   split line into 4 part (stationId, year, tempType, tempValue)
      emit((stationId, year), (tempType, tempValue, year))
}

getPartition() {
   records with same stationId will go to same reducer
}

group() {
   records with same stationId will be processed in one reduce call
}

// records in same reduce call is sorted by year
reduce((stationId, year), [(tempType, tempValue, year)]) {
   curYear = key.year;

   for each value in inputList
      if(value.year == curYear) {
      // means current value is recorded in same year as previous value
            accumulate the maxTotal, maxNums, minTotal, minNums
      } else {
      // means current value is recorded in a new year

          calculate the result of previous station's records
          start recording the records in new station
      }
     emit (stationId, [(year1, maxMean1, minMin1), (...), ..., ...])
}
```

2.Spark Scala Program

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

basically we can consider the transformation of data during the program, hdfs -> mapper, mapper -> reducer, reducer -> hdfs
(1) hdfs -> mapper: for this phase, I think we should use DaraFrame, because dataFrame is a structured data, it is organized into named column, like a table. so it is convenient for mapper to process.

(2) mapper -> reducer: for this phase, I think we can use dataset, because dataset is a collection of strong-typed JVM objects, so we can maintain the consistency of the mapper output and reducer input.
(3) reducer -> hdfs: I think in the phase, we should use RDD as data representation, because as definition says, RDD is immutable distributed collections of data, so it is suitable for hdfs

2. 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.

no combiner

```
val input = sc.textFile("input")
val maxData = input.map(line => line.split(","))
                        .filter(fields => fields(2) == "TMAX" )
val minData = input.map(line => line.split(","))
                        .filter(fields => fields(2) == "TMIN")
val numMax = maxTemps.count
val numMin = minTemps.count
val summaxs = maxTemps.map(fields => Integer.parseInt(fields(3)))
                        .reduce((sum, temp) => sum + temp)
val summins = minTemps.map(fields => Integer.parseInt(fields(3)))
                        .reduce((sum, temp) => sum + temp)
println("stationId " + fields(0) + "Avg max: " + summax / numMax) + " Avg min :  " + summin
/ numMins
```

with combiner

```
val input = sc.textFile("input")
val maxData = input.map(line => line.split(","))
                        .filter(fields => fields(2) == "TMAX" )
                        .combineByKey()
val minData = input.map(line => line.split(","))
                        .filter(fields => fields(2) == "TMIN")
                        .combineByKey()
val numMax = maxTemps.count
val numMin = minTemps.count
val summaxs = maxTemps.map(fields => Integer.parseInt(fields(3)))
                        .reduce((sum, temp) => sum + temp)
val summins = minTemps.map(fields => Integer.parseInt(fields(3)))
                        .reduce((sum, temp) => sum + temp)
println("stationId " + fields(0) + "Avg max: " + summax / numMax) + " Avg min :  " + summin
/ numMins
```

3. 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?

I think reduceByKey() implements these things, because from the name of this function, we can know this function act as reduce call in map reduce function, which is used to aggregation

4. 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.

3. Performance Comparison

no combiner:

run 1:

```
Total time spent by all maps in occupied slots (ms)=33936864
Total time spent by all reduces in occupied slots (ms)=18500640
Total time spent by all map tasks (ms)=707018
Total time spent by all reduce tasks (ms)=192715
Total vcore-milliseconds taken by all map tasks=707018
Total vcore-milliseconds taken by all reduce tasks=192715
Total megabyte-milliseconds taken by all map tasks=1085979648
Total megabyte-milliseconds taken by all reduce tasks=592020480
```

run 2:

```
Total time spent by all maps in occupied slots (ms)=32364624
Total time spent by all reduces in occupied slots (ms)=13820640
Total time spent by all map tasks (ms)=674263
Total time spent by all reduce tasks (ms)=143965
Total vcore-milliseconds taken by all map tasks=674263
Total vcore-milliseconds taken by all reduce tasks=143965
Total megabyte-milliseconds taken by all map tasks=1035667968
Total megabyte-milliseconds taken by all reduce tasks=442260480
```

with combiner:

run1:

```
Total time spent by all maps in occupied slots (ms)=39342912
Total time spent by all reduces in occupied slots (ms)=12164256
Total time spent by all map tasks (ms)=819644
Total time spent by all reduce tasks (ms)=126711
Total vcore-milliseconds taken by all map tasks=819644
Total vcore-milliseconds taken by all reduce tasks=126711
Total megabyte-milliseconds taken by all map tasks=1258973184
```

```
Total megabyte-milliseconds taken by all reduce tasks=389256192
```

run2:

```
Total time spent by all maps in occupied slots (ms)=39530256
Total time spent by all reduces in occupied slots (ms)=12082272
Total time spent by all map tasks (ms)=823547
Total time spent by all reduce tasks (ms)=125857
Total vcore-milliseconds taken by all map tasks=823547
Total vcore-milliseconds taken by all reduce tasks=125857
Total megabyte-milliseconds taken by all map tasks=1264968192
Total megabyte-milliseconds taken by all reduce tasks=386632704
```

In Mapper Combine

run1:

```
Total time spent by all maps in occupied slots (ms)=24117456
Total time spent by all reduces in occupied slots (ms)=7564608
Total time spent by all map tasks (ms)=502447
Total time spent by all reduce tasks (ms)=78798
Total vcore-milliseconds taken by all map tasks=502447
Total vcore-milliseconds taken by all reduce tasks=78798
Total megabyte-milliseconds taken by all map tasks=771758592
Total megabyte-milliseconds taken by all reduce tasks=24206745
```

run2

```
Total time spent by all maps in occupied slots (ms)=23397552
Total time spent by all reduces in occupied slots (ms)=7168608
Total time spent by all map tasks (ms)=487449
Total time spent by all reduce tasks (ms)=74673
Total vcore-milliseconds taken by all map tasks=487449
Total vcore-milliseconds taken by all reduce tasks=74673
Total megabyte-milliseconds taken by all map tasks=748721664
Total megabyte-milliseconds taken by all reduce tasks=229395456
```

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

we can compare the results of two combiner program to see if we can get some idea

first run:

```
Map input records=30870343
Map output records=30870343
Combine input records=30870343
Combine output records=468620
Reduce input records=468620
Reduce output records=28981
```

second run:

```
Map input records=30870343
Map output records=30870343
Combine input records=30870343
Combine output records=468620
```

```
Reduce input records=468620
Reduce output records=28981
```

as we can see, every item is exactly same, which means the time to execute combiner in this two program is same, so it's hard to determine if the combiner called more than once, only thing we can make sure is the combiner do called in program, because the combine input is not 0

Was the local aggregation effective in InMapperComb compared to NoCombiner?

first let me list the metrics for Inmapper comb and no combiner

noCombiner:

```
Map input records=30870343
Map output records=30870343
Combine input records=0
Combine output records=0
Reduce input groups=28981
Reduce input records=30870343
Reduce output records=28981
```

inMapperComb:

```
Map input records=30870343
Map output records=223795
Combine input records=0
Combine output records=0
Reduce input groups=14136
Reduce input records=223795
Reduce output records=14136
```

from metrics above we can safely observe that "map output record" decreases significantly, as the result, the reduce input groups and records also decrease, from the perspective of network traffic, in mapper combing is more effective than noCombiner

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.

```
Total time spent by all maps in occupied slots (ms)=10362336
Total time spent by all reduces in occupied slots (ms)=8683776
Total time spent by all map tasks (ms)=215882
Total time spent by all reduce tasks (ms)=90456
Total vcore-milliseconds taken by all map tasks=215882
Total vcore-milliseconds taken by all reduce tasks=90456
Total megabyte-milliseconds taken by all map tasks=331594752
Total megabyte-milliseconds taken by all reduce tasks=277880832
```