# Assignment 2 - CS 6240

- Sriharsha Srinivasa Karthik Kaipa, Sec 01

## Running the code:

The source code for this assignment is available in the folder named “assignment2”. The “src” folder contains the final code. Make sure the inputs for all Combiner related programs are in the folder “`inputCombiner`” alongside the “src” folder and the inputs for Secondary Sort are in the folder “`inputSecondarySort`”. Run instructions for the code are as follows:

- For cleaning the project, building the jar and then running it from the command line, please use the following rules:
  ```
  NoCombiner:~$ make nocombiner
  Combiner:~$ make combiner
  InMapperCombiner:~$ make inmappercombiner
  SecondarySort:~$ make secondarysort
  ```
- For cleaning the project, use the following rule
  ```
  make cleancombiner
  make cleansecondarysort
  ```
- For building the jar, use the following rule
  ```
  make jar
  ```

## Pseudo Code:

### No Combiner:

**Mapper Task:**

```
map (line) {
     emit(stationID, {temp, type});
}
```

**Reducer Task:**

```
reduce (stationID, list = [{temp1, type1}, {temp2, type2}...]) {
     for {temp, type} in list {
          if type == TMAX {
               tmax += temp;
               tmaxCount += 1;
          } else {
               tmin += temp;
               tminCount += 1;
          }
     }
     emit(stationID, {tmin/tminCount, tmax/tmaxCount});
}
```

Combiner:

**Mapper Task:**

```
map (line) {
     emit(stationID, {temp, type, 1});
}
```

**Combiner Task:**

```
reduce (stationID, list = [{tp1, ty1, v1}, {tp2, ty2, v2}...]) {
     for {temp, type, val} in list {
          if type == TMAX {
               tmax += temp;
               tmaxCount += val;
          } else {
               tmin += temp;
               tminCount += val;
          }
     }
     emit(stationID, {{TMIN, tmin, tminCount);
     emit(stationID, {{TMAX, tmax, tmaxCount);
}
```

**Reducer Task:**

```
reduce (stationID, list = [{ty1, tp1, v1}, {ty2, tp2, v2}...]) {
     for {type, temp, val} in list {
          if type == TMAX {
               tmax += temp;
               tmaxCount += val;
          } else {
               tmin += temp;
               tminCount += val;
          }
     }
     emit(stationID, {tmin/tminCount, tmax/tmaxCount});
}
```

In-Mapper Combiner:

**Mapper Task:**

```
setup () {
      HashMap hs = new HashMap();
}
map (line) {
      if hs.contains(stationID) {
            tp = hs.get(stationID);
            if type == TMAX {
                  tp[2] += temp;
                  tp[3] += 1;
            } else {
                  tp[0] += temp;
                  tp[1] += 1;
            }
            hs.put(stationID, tp);
      } else {
            tp = double[4];
            if type == TMAX {
                  tp[2] = temp;
                  tp[3] = 1;
            } else {
                  tp[0] = temp;
                  tp[1] = 1;
            }
            hs.put(stationID, tp);
      }
}
cleanup() {
      for each {stationID, tp} in hs {
            emit(stationID, {TMIN, tp[0], tp[1]);
            emit(stationID, {TMAX, tp[2], tp[3]);
      }
}
```

**Reducer Task:**

```
reduce (stationID, list = [{ty1, tp1, v1}, {ty2, tp2, v2}...]) {
      for {type, temp, val} in list {
            if type == TMAX {
                  tmax += temp;
                  tmaxCount += val;
            } else {
                  tmin += temp;
                  tminCount += val;
            }
      }
```

```
     emit(stationID, {tmin/tminCount, tmax/tmaxCount});
}
```

Secondary Sort:

**Mapper Task:**

```
map (line) {
     emit(stationID_year, {year, type, temp});
}
```

**Partitioner Task:**

```
getPartition (stationID_year, value, #reduceTasks) {
     return abs(stationID.hashCode % #reduceTasks);
}
```

**Key Comparator Task:**

```
compare (stationID1_year1, stationID2_year2) {
     st = staionID1.compare(stationID2);
     if st == 0
          return year1.compare(year2);
     return st;
}
```

**Grouping Comparator Task:**

```
compare (stationID1_year1, stationID2_year2) {
     return staionID1.compare(stationID2);
}
```

**Reducer Task:**

```
reduce (stationID_year, list = [{y1, ty1, tp1}...]) {
     LinkedHashMap hs = new LinkedHashMap();
     for {year, type, temp} in list {
          if hs.contains(year) {
               tp = hs.get(year);
               if type == TMAX {
                    tp[2] += temp;
                    tp[3] += 1;
               } else {
                    tp[0] += temp;
                    tp[1] += 1;
               }
               hs.put(year, tp);
          } else {
               tp = double[4];
               if type == TMAX {
                    tp[2] = temp;
                    tp[3] = 1;
               } else {
                    tp[0] = temp;
                    tp[1] = 1;
               }
```

```
                hs.put(year, tp);
            }
        }
        String s;
        for each {year, tp} in hs {
            s + year + “,” + tp[0]/tp[1] + “,” + tp[2]/tp[3] + “;”;
        }
        emit(stationID, s);
}
```

Every reduce call in secondary sort program will receive the largest stationID_year for a single stationID as the key and all the years’ data sorted based on the year. For example, for staionID ABC, the key it would receive would be ABC_1889 and the values it would receive would be [{1880, type1, temp1}, {1881, type2, temp2}, {1881, type3, temp3}...{1889, typen, tempn}]. The keys that the reduce function would receive would be in sorted order based on staionID.

## Performance Comparison:

### No Combiner:

Run 1: 72.839 seconds
Run 2: 87.749 seconds

### Combiner:

Run 1: 72.927 seconds
Run 2: 76.499 seconds

### In-Mapper Combiner:

Run 1: 70.554 seconds
Run 2: 72.927 seconds

Q.1 Was the Combiner called at all in program Combiner? Was it called more than once per Map task?
A. The combiner was called in the “combiner” program. The proof for this is in these log file records:

```
Combine input records=8798241
Combine output records=445200
```

Also from the records, it is clear that a total of 20 map tasks and 10 reduce tasks were created and since we used 5 worker cores, that means there were 4 map tasks and 2 reduce tasks per core.

The ratio of “Combine input records” and “Combine output records” comes out to be approximately 19.76 which is close to 20, the number of map tasks. From this, it is clear that one combiner was called per map task.

Q.2 What difference did the use of a Combiner make in Combiner compared to NoCombiner?
A. Looking at the following log entries

```
Reduce input records=8798241 - NoCombiner
Total time spent by all reduce tasks (ms)=130757 - NoCombiner
Reduce input records=445200 - Combiner
Total time spent by all reduce tasks (ms)=114622 - Combiner
```

It is quite clear that the use of combiner has significantly reduced the number of records input to the reduce tasks and also the total time spent by all reduce tasks has decreased by 15 seconds.

Q.3 Was the local aggregation effective in InMapperComb compared to NoCombiner?
A. Local aggregation was highly effective in InMapperCombiner when compared to NoCombiner just by looking at the run times of each of the two types of programs. On an average, there is approximately 8.5535 seconds difference between the two, the NoCombiner version being the slower of the two.

Q.4 Which one is better, Combiner or InMapperComb? Briefly justify your answer.
A. InMapperCombiner is much better when compared to Combiner. This may be because of the overhead in records read/write to the file when a combiner is involved in the process. This can be seen from the following records:

```
Physical memory (bytes) snapshot=13027385344 - InMapperCombiner
Physical memory (bytes) snapshot=13168611328 - Combiner
```

As can be seen, there is a difference of 141 MB between InMapperCombiner and Combiner in physical memory utilization. This is happening because the mapper first writes records to memory and combiner again writes records to memory based on the records from mapper. This is a slow read/write process which takes a lot of time. But this step is skipped because InMapperCombiner directly writes the combined records to memory which is a faster operation. Hence, InMapperCombiner is better.

Q.5 How do the running times and accuracy of these MapReduce programs compare to the sequential implementation of per-station mean temperature?
A. The average run times of NoCombiner, Combiner and InMapperCombiner are 80.294 seconds, 74.713 seconds and 71.7405 seconds. The average run time of Sequential for the same data is 5.47 seconds. The output data is more or less of the same accuracy.