# CS 6240 – Parallel Data Processing with Map Reduce
# Section-01, HW-2, Biyanta Vipulbhai Shah

## Map-Reduce Algorithms

### Program 1:

1. No Combiner:

```
map (offset B, line L)
        for each record r in L
                if r contains TMIN or TMAX
                        if contains TMIN then
                                value = {tmin, 0.0, isTmin: true}
                                emit (stationID_key, value)
                        else if contains TMAX then
                                value = {0.0, tmax, isTmin: false}
                                emit (stationID_key, value)

reduce (stationID_key, value_list)
        total_min = 0
        total_max = 0
        count_min = 0
        count_max = 0

        for each x in value_list do
                if record station ID had minimum temperature then
                        total_min += x.tmin
                        count_min += 1

                else
                        total_max += x.tmax
                        count_max += 1

        output = append (stationID_key, total_min/count_min, total_max/count_max)
        emit (output, null)
```

2. Combiner:

```
map (offset B, line L)
    for each record r in L
        if r contains TMIN or TMAX
            if contains TMIN then
                value = {tmin, 0.0, count_min: 1, count_max: 0}
                emit (stationID_key, value)
            else if contains TMAX then
                value = {0.0, tmax, count_min: 0, count_max: 1}
                emit (stationID_key, value)

combiner (stationID_key, value_list)
    total_min = 0
    total_max = 0
    count_min = 0
    count_max = 0

    for each x in value_list do
        total_min += x.tmin
        total_max += x.tmax
        count_min += x.count_min
        count_max += x.count_max

    value_out = {total_min, total_max, count_min, count_max}
    emit (stationID_key, value_out)

reduce (stationID_key, value_list)
    total_min = 0
    total_max = 0
    count_min = 0
    count_max = 0

    for each x in value_list do
        total_min += x.tmin
        total_max += x.tmax
        count_min += x.count_min
        count_max += x.count_max

    output = append (stationID_key, total_min/count_min, total_max/count_max)
    emit (output, null)
```

3. In-Mapper Combiner

```
class Mapper {
        hashmap H;
        setup () {
                H = new hashmap
        }

        map (offset B, line L) {
                for each record r in L
                        if r contains TMIN or TMAX
                                if contains TMIN
                                        if H contains the stationID then
                                                update tmin and count_min in H
                                        else
                                                create a new key-value pair in H

                                else if contains TMAX
                                        if H contains the stationID then
                                                update tmax and count_max in H
                                        else
                                                create a new key-value pair in H
        }

        cleanup () {
                for each stationID s in H do
                        emit (s, H[s])
        }
}

reduce (stationID_key, value_list)
        total_min = 0
        total_max = 0
        count_min = 0
        count_max = 0

        for each x in value_list do
                total_min += x.tmin
                total_max += x.tmax
                count_min += x.count_min
                count_max += x.count_max

        output = append (stationID, total_min/count_min, total_max/count_max)
        emit (output, null)
```

**Program 2:**
Secondary Sort

```
map (offset B, line L)
    for each record r in L
        if r contains TMIN or TMAX
            CompositeKey = {stationID, year}
            if contains TMIN then
                value = {tmin, 0.0, count_min: 1, count_max: 0}
                emit (CompositeKey, value)
            else if contains TMAX then
                value = {0.0, tmax, count_min: 0, count_min: 1}
                emit (CompositeKey, value)

combiner (CompositeKey, value_list)
    total_min = 0
    total_max = 0
    count_min = 0
    count_max = 0

    for each x in value_list do
        total_min += x.tmin
        total_max += x.tmax
        count_min += x.count_min
        count_max += x.count_max

    value_out = {total_min, total_max, count_min, count_max}
    emit (CompositeKey, value_out)


getPartition (stationID, year)
    return myPartition(stationID)

keyComparator (stationID, year)
    // Sorts in increasing order of stationID first
    // if the stationID is equal, sorts in increasing order of year.

groupingComparator (stationID, year)
    // Does not consider the year while sorting
    // Sorts in increasing order of stationID
    // Hence two keys with the same stationID are considered identical no
    // matter the year value
```

```
reduce (CompositeKey, value_list)
        total_min = 0
        total_max = 0
        count_min = 0
        count_max = 0
        prev_year = CompositeKey.year

        for each x in value_list do
                if prev_year != CompositeKey.year then
                        output = append previous year's values in given format
                        set prev_year to current year (CompositeKey.year)

                        total_min = x.tmin
                        total_max = x.tmax
                        count_min = x.count_min
                        count_max = x.count_max
                else
                        total_min += x.tmin
                        total_max += x.tmax
                        count_min += x.count_min
                        count_max += x.count_max

        output = append the last year's value in given format
        emit (output, null)
```

**NOTES:** While calling the reduce method, we sort objects in order of its key, which in our case is a Composite Key of (stationID, year). Output through this reduce call would be ((stationID, year), {tmax, tmin values}) for each record. The output format expected is such that we need to have all station ID's listed for a single year. For the same we use a grouping comparator which compares objects on the basis of year and groups the station temperature details together to display stationID and values with the year.

## Performance Comparison

Using 6 m4.large machines (1 master, 5 workers) in EMR

| Program Name | First Run | Second Run |
|---|---|---|
| No Combiner | 80 seconds | 80 seconds |
| Combiner | 78 seconds | 80 seconds |
| In-Mapper Combiner | 76 seconds | 76 seconds |

## Questions

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

A. **Yes**, the combiner has been called in the program Combiner.

The proof of this is the following records:
Map input records=30868726
Map output records=8798241
**Combine input records=8798241**
**Combine output records=223783**
Reduce input records=223783
Reduce output records=14135

The Combine input and output records are not zero, which shows that the Combiner was called in the program Combiner.
However, we cannot come to know whether it was called more than once per Map task because calling the Combiner is not in our hands. The Combiner is scheduled by the MapReduce task at its own convenience.

2. What difference did the use of a Combiner make in Combiner compared to NoCombiner?

A. In the Combiner; use of the Combiner, combined the map output records and the reducer thus had less input records to perform computation on. Partial computation was performed by the Combiner which gave less overhead to the reducer. However, in the NoCombiner, the number of output records of the Map and the input records of the reducer is the same, showing no aggregation of records.

The proof of this is in the following records:

| **Combiner** | **No Combiner** |
|---|---|
| Map input records=30868726<br>**Map output records=8798241**<br>Combine input records=8798241<br>Combine output records=223783<br>**Reduce input records=223783**<br>Reduce output records=14135 | Map input records=30868726<br>**Map output records=8798241**<br>Combine input records=0<br>Combine output records=0<br>**Reduce input records=8798241**<br>Reduce output records=14135 |

We can see that **the Map Output Records for both the cases is the same**, but the **Reduce Input Records is much less in the Combiner** compared to the NoCombiner.

3. Was the local aggregation effective in InMapperComb compared to NoCombiner?

**A. Yes,** the local aggregation was effective in InMapperComb compared to NoCombiner.
The following results prove so:

| In-Mapper Combiner | No Combiner |
|---|---|
| **Map input records=30868726**<br>**Map output records=223783**<br>Combine input records=0<br>Combine output records=0<br>Reduce input records=223783<br>Reduce output records=14135 | **Map input records=30868726**<br>**Map output records=8798241**<br>Combine input records=0<br>Combine output records=0<br>Reduce input records=8798241<br>Reduce output records=14135 |

The Map input records for both the In-Mapper combiner and the No Combiner are exactly the same. However, if you note the Map output records for the both of them, you will notice that the In-Mapper Combiner has much lesser output records than the No Combiner. This proves that local aggregation was effective. The Reduce Output Records for both of the programs are the same though. Thus there is no change in the final result, just a change in how an In-Mapper Combiner and No Combiner does the intermediate computation.

4. Which one is better, Combiner or InMapperComb? Briefly justify

A. In this case, The InMapperComb and the Combiner number of Reduce input records are the same. This means that the InMapperComb via local aggregation and the Combiner via the Combiner class have combined the same number of records. However, taking into consideration the time. The InMapperComb takes less time (76 seconds) as compared to the Combiner (80 seconds). Thus InMapperComb is effective here.
   There is though one drawback of the InMapperComb. The InMapperComb uses a Hashmap for the local aggregation, this creates the objects on heap, so it's not very memory efficient. Thus in the ideal case where there is memory available abundantly, we should use the InMapperCombiner.

5. How do the running times and accuracy of these MapReduce programs compare to the sequential implementation of per-station mean temperature?

A. The run time for the NoCombiner MapReduce program is 80 seconds, while the run time for the Sequential average is 9.421 seconds.
   The input data for these programs was 1GB which was very less compared to the MapReduce capabilities.
   On AWS, data is distributed and we transfer this data from the Mapper to the Reducer class while there is no such transfer in the Sequential program.
   Thus for small static data, the sequential program will be faster, but when we are using real time data which is huge, MapReduce program will be more efficient.

Using 6 m4.large machines (1 master, 5 workers) in EMR

| **Program Name** | **Running Time** |
|---|---|
| Secondary Sort | 54 seconds |