Name : Sushil Thasale
Homework : 02
Section : 01

PSEUDO CODES:

* SumCountPair is a custom class used to store TMIN and TMAX values along with their counts.

```
class SumCountPair{
    - tminSum
    - tminCount
    - tmaxSum
    - tmaxCount
}
```

Pseudo code for No-Combiner:

```
class Mapper{

    Map(offset B, line L):
        - parse line(L) and extract stationID, temperature
        - SP = new SumCountPair
        - if L contains "TMIN":
            add temperature to SP.tminSum
            increment SP.tminCount
        - else if L contains "TMAX":
            add temperatre to SP.tmaxSum
            increment SP.tmaxCount
        - emit(stationID, SP)
}

class Reducer{

    Reduce(stationID, list(SP)):
        - for s in list(SP):
            aggregate s.tminSum
            aggregate s.tminCount
            aggregate s.tmaxSum
            aggregate s.tmaxCount
        - compute average TMIN
        - compute average TMAX
        - emit(stationID, average TMIN, average TMAX)
}
```

Pseudo Code for Custom Combiner:

```
class Mapper{

    Map(offset B, line L):
        - parse line(L) and extract stationID, temperature
        - SP = new SumCountPair
        - if L contains "TMIN":
            add temperature to SP.tminSum
            increment SP.tminCount
        - else if L contains "TMAX":
            add temperatre to SP.tmaxSum
            increment SP.tmaxCount
        - emit(stationID, SP)
}

class Combiner{

    Reduce(stationID, list(SP)):
        - for s in list(SP):
            aggregate s.tminSum
            aggregate s.tminCount
            aggregate s.tmaxSum
            aggregate s.tmaxCount
        - SP = new SumCountPair
        - put aggregated values in SP
        - emit(stationID, SP)
}

class Reducer{

    Reduce(stationID, list(SP)):
        - for s in list(SP):
            aggregate s.tminSum
            aggregate s.tminCount
            aggregate s.tmaxSum
            aggregate s.tmaxCount
        - compute average TMIN
        - compute average TMAX
        - emit(stationID, average TMIN, average TMAX)
}
```

Pseudo Code for InMapper Combiner:

```
class Mapper{

    * Hashmap<stationID, SumCountPair> combiner

    Setup():
        combiner = new HashMap()

    Map(offset B, line L):
        - parse line(L) and extract stationID, temperature
        - if L contains "TMIN":
            combiner[stationID].addTmin(temperature)
        - else if L contains "TMAX":
            combiner[stationID].addTmax(temperature)

    Cleanup():
        - for stationID, SP in combiner:
            emit(stationID, SP)
}

class Reducer{

    Reduce(stationID, list(SP)):
        - for s in list(SP):
            aggregate s.tminSum
            aggregate s.tminCount
            aggregate s.tmaxSum
            aggregate s.tmaxCount
        - compute average TMIN
        - compute average TMAX
        - emit(stationID, average TMIN, average TMAX)
}
```

Pseudo Code for Secondary Sort:

```
* StationYear is custom class used to store stationID and Year

class Mapper{

    * Hashmap<StationYear, SumCountPair> combiner

    Setup():
        combiner = new HashMap()

    Map(offset B, line L):
        - parse line(L) and extract stationID, year, temperature
        - SY = new StationYear(stationID, year)
        - if L contains "TMIN":
            combiner[SY].addTmin(temperature)
        - else if L contains "TMAX":
            combiner[SY].addTmax(temperature)

    Cleanup():
        - for StationYear, SumCountPair in combiner:
            emit(StationYear, SumCountPair)
}

class Partitioner{
    getPartition(StationYear, #partitions):
        - partition based on stationID
}

class KeyComparator{
    compare():
        - Sort in ascending order of stationID
        - if stationID is equal, then sort in ascending order of year
}

class GroupingKeyComparator{
    compare():
        - Sort in ascending order of stationID
        - does not consider year for sorting
        - Hence, two keys with same stationID are considered identical
}
```

```
class Reducer{
    Reduce(StationYear SY, list(SumCountPair)):
        - currentYear = ""
        - summary = ""
        - for SP in list(SumCountPair):
            if SY.year == currentYear:
                keep aggreagting tmin and tmax values for currentYear
            else:
                compute average tmin and tmax for currentYear
                append currentYear, average tmin and tmax to summary
                change currentYear to SY.year
                aggregate tmin and tmax for new currentYear

        - emit(SY.stationID, summary)
}
```

All the keys with same station IDs (e.g. (station-1, year-1), (station-1, year-2), (station-1, year-3)…) and its corresponding values will be input to same reduce function call. These keys will be sorted in ascending order year. The first key (with smallest year) will be representative of all the other keys. As, we iterate over the input values, the corresponding year within the key changes, so we need to keep track of current-year.

Running Times for each program:

	Runtime-1	Runtime-2
No Combiner	57	58
Custom Combiner	51	54
InMapper Combiner	50	45
Secondary Sort	28	

(All times are in seconds)

Data obtained from syslog file for 1 run:

	Map Output Bytes	Map Output Records	Combine Input Records
No Combiner	387122604	8798241	0
Custom Combiner	387122604	8798241	8798241
InMapper Combiner	9846408	223782	0

	Combine Output Records	Reduce shuffle bytes	Reduce Input Records
No Combiner	0	55945304	8798241
Custom Combiner	223782	4209634	223782
InMapper Combiner	0	4209634	223782

1. For map-reduce program with custom combiner, 8798241 records were output by mapper. And, 223782 records were input to reducer. This proves that custom combiner was called during job execution. However, the number of records (also bytes) input to reducer were different for different runs. So, I guess, the combiner was called inconsistently for different map tasks.

2. In case of map-reduce program with no-combiner, 8798241 records were input to reducer and 55945304 bytes were shuffled between mapper and reducer. On the other hand, only 4209634 bytes were shuffled for combiner program. The combiner considerably reduced the shuffling overhead and this might be the reason why the combiner program was faster than no-combiner program.

3. Yes, local aggregation in case of in-mapper combiner was very effective. It considerably reduced the number of records and thus the bytes output from map tasks. In case of no-combiner program 387122604 bytes (8798241 records) were output by map tasks. Whereas, only 9846408 bytes (223782 records) were output by map tasks with in-mapper combining. This reduced the data shuffled and sorted between mapper and reducer, thereby increasing efficiency.

4. For first run (reported above), the performance of custom combiner and in-mapper combiner were same. However, in subsequent runs, the number of records (also bytes) combined by custom combiner were different i.e. custom combiner was called inconsistently for different map tasks. Because of this inconsistency in performance, I would not prefer custom-combiner over in-mapper combiner.

5. The average end-to-end running time (reading file to printing output) of sequential version was 25 seconds. This time is considerably low as compared to the map-reduce versions. This might be because the data is not big enough for map-reduce to be effective. Also, map-reduce programs involve additional tasks like splitting that data, transferring the data to mapper machines, shuffling and sorting map output and transferring it to reducer machines. However, I'm confident that map-reduce would perform much better, if the given data is large. The map-reduce output was same as the sequential version.