**Name: Atindra Mardikar**
**Class: Nat Tuck (Tue/Fri 1:35-3:15pm)**
**HW-02 Report**

---

## Map Reduce Source Code:

### 1) NoCombiner:

**Mapper**{

```
map()
{
    Read file; Break the file by lines;
    For each line {
        Break into tokens;
        StationId=token[0];
        If(token[3]==Tmax){
            emit(stationID,new Mean(negativeInfinity,token[3]));
        }
        else If(token[3]==Tmin){
            emit(stationID,new Mean(token[3],positiveInfinity));
        }
    }
 }
}
```

**Reducer**{

```
reduce()
{
    TminSum=0;TmaxSum=0;TminCount=0;TmaxCount=0;
    For each Mean in values{
        If(Mean.Tmax!=positiveInfinity){
            TmaxSum+=Mean.Tmax;
            TmaxCount++;
        }
        else If(Mean.Tmin!=negativeInfinity){
            TminSum+=Mean.Tmin;
            TminCount++;
        }
        Calculate MeanTmin and MeanTmax;
        Emit(StationId, new Text(MeanTmin+",”+MeanTmax))
    }
 }
}
```

## 2) Combiner:

**Mapper**

```
 {
 map()
 {
        Read file; Break the file by lines;
        For each line {
                Break into tokens;
                StationId=token[0];
                If(token[3]==Tmax){
                        emit(stationID,new Mean(negativeInfinity,token[3],1));
                }
                else If(token[3]==Tmin){
                        emit(stationID,new Mean(token[3],positiveInfinity,1));
                }
        }
    }
}
```

**Combiner**

```
   {
                TminSum=0;TmaxSum=0;TminCount=0;TmaxCount=0;
                For each Mean in values{
                        If(Mean.Tmax!=positiveInfinity){
                                TmaxSum+=Mean.Tmax;
                                TmaxCount++;
                        }
                        else If(Mean.Tmin!=negativeInfinity){
                                TminSum+=Mean.Tmin;
                                TminCount++;
                        }
            Emit(StationId, new Mean(TminSum,positiveInfinity,TminCount));
            Emit(StationId, new Mean(negativeInfinity,TmaxSum,TmaxCount));
                }
}
```

**Reducer**

```
   {
    reduce()
    {
                TminSum=0;TmaxSum=0;TminCount=0;TmaxCount=0;
                For each Mean in values{
                        If(Mean.Tmax!=positiveInfinity){
                                TmaxSum+=Mean.Tmax;
```

```
                    TmaxCount++;
                }
                else If(Mean.Tmin!=negativeInfinity){
                    TminSum+=Mean.Tmin;
                    TminCount++;
                }
                Calculate MeanTmin and MeanTmax;
                Emit(StationId, new Text(MeanTmin+","+MeanTmax))
            }
        }
}
```

## 3) InMapper Combiner:

**Mapper**

```
{
  setup()
  {
        HashMap TminMap=new HashMap<String, Double[]>();
        HashMap TmaxMap=new HashMap<String, Double[]>();
   }
   map()
   {
        Read file; Break the file by lines;
        For each line {
            Break into tokens;
            StationId=token[0];
            If(token[3]==Tmax){
                If(TmaxMap.containsKey(token[3]){
                    Update the sum and count in double array;
                }
                else{
                    TmaxMap.put(stationID,[token[3],1]);
                }
            }

            else If(token[3]==Tmin){
                If(TminMap.containsKey(token[3]){
                    Update the sum and count in double array;
                }
                else{
                    TminMap.put(stationID,[token[3],1]);
                }
            }
        }
    }
```

```
        cleanup()
        {
                for (String key : TmaxMap){
                emit(key, new MeanCombiner(negativeInfinity,
                        TmaxMap.get(key)[0], TmaxMap.get(key)[1]);
                }
                for (String key : TminMap){
                emit(key, new MeanCombiner(TminMap.get(key)[0],
                        positiveInfinity,TminMap.get(key)[1]);
                }
   }
```

**Reducer**

```
   {
    reduce()
    {
        TminSum=0;TmaxSum=0;TminCount=0;TmaxCount=0;
        For each Mean in values{
                If(Mean.Tmax!=positiveInfinity){
                        TmaxSum+=Mean.Tmax;
                        TmaxCount++;
                }
                else If(Mean.Tmin!=negativeInfinity){
                        TminSum+=Mean.Tmin;
                        TminCount++;
                }
                Calculate MeanTmin and MeanTmax;
                Emit(StationId, new Text(MeanTmin+","+MeanTmax))

        }
    }

}
```

## 4) Secondary sort:

**Mapper**

```
{
  map()
  {
        Read file; Break the file by lines;
        For each line {
              Break into tokens;
              Update Hashmap;
        }
        for each entry in Hashmap
              emit((stationId,year),new Station(Tmin,Tmax));
   }
}
```

**KeyComparator(stationId,year)**

```
{
    // sorts in increasing order of stationId first.
   //  if stationId is equal sorts it in increasing order of year.
}
```

**GroupingComparator(stationId,year)**

```
{
    // sorts in increasing order of stationId.
   //  Does not consider yeaer for sorting.
  //  Hence two keys with same stationId are considered identical irrespective
      of their year
}
```

**Reducer**

```
  {
   reduce()
   {
        TminSum=0;TmaxSum=0;TminCount=0;TmaxCount=0;
        For each station in values{
              //Update all the local variables
        }
              Calculate MeanTmin and MeanTmax;
              Create result key;
              Emit(resultkey, NullWritable);
        }
   }
}
```

So the groupComparator is responsible for grouping the composite key by their stationID so That all the keys with same stationId goes to one reducer.

Key compartor is responsible for sorting the key on both stationId and year.

So the reducer will receive all the records with same StationId and it can calculate the mean for each year.

## Performance Comparison:

Running Times (taken from the controller log):

1) NoCombiner:
   1 run: 82 seconds
   2 run: 84 seconds

2) Combiner:
   1 run: 80 seconds
   2 run: 98 seconds

3) InMapper Combiner:
   1 run: 78 seconds
   2 run: 78 seconds

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

  On checking Combine Input records, In both the runs the combiner was called for every Map task. Yes it was called once for per Map task as the output as the Map output records is equal to Combine input records. So for every record set emitted by the mapper there was a call to combiner.

- **What difference did the use of a Combiner make in Combiner compared to NoCombiner?**

  So using the combiner reduces the number of input records to the reducer significantly(combiner: 447564, Nocombiner:8798241). If we don't use any combiner all the output records from the mapper are fed into the reducer. On the other hand the use of combiner reduces the load on the reducer.

- **Was the local aggregation effective in InMapperComb compared to NoCombiner?**

So using the InMappercombiner significantly reduces the number of output records from the mapper to the reducer (InMapperCombiner: 445200, Nocombiner:8798241). Also the output bytes from the mapper are vry less compared to no combiner (InMappercombiner: 16027200, Nocombiner:246350748). If we don't use any combiner all the output records from the mapper are fed into the reducer. On the other hand the use of InMappercombiner reduces the load on the reducer.

- **Which one is better Combiner or InMapperCombiner? Briefly justify your answer.**

  According to me InMapperCombiner is a better option than using combiner because even though we specify the combiner its up to the map reduce whether to call it for every map call. So we need to write the combiner logic in reducer as well so that if for a map call combiner is not called we don't lose any data. Also according to the running times InMapper Combiner are faster than Combiner.
  The only disadvantage of InmapperCombiner is complexity and sometimes space inefficiency. But overall InMapperCombiner is a better option and is guranteed to reduce work load on the reducer.

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

  Running Sequntial on such a large file take a long time, around 22 mins on my computer, which is 1320 seconds. Mapreduce programs take appron $1/16^{th}$ time than that which is way faster. Result wise both the programs give identical results.