CS 6240 Parallel Data Processing in MapReduce - 02
Homework 2 Report
Dheeraj Joshi

Pseudocode for 1:

a. No Combiner

```
//Mapper
Map(record){
      //Map reads one record at a time
      //every record reads as "StationId,Date,TMAX/TMIN,Value,…."
      //stationData contains running sum and count for TMAX & TMIN
      //if record specifies TMAX ,add value to stationData
      //or if record specifies TMIN, add value to stationData
      //Emit Data with StationId as Key
      emit(stationId,stationData)
}
//Reducer
Reduce(stationId,stationDatas[]){
      //initialize variables for accumulating sums and counts
      For each stationData in stationDatas[]{
            //accumulate all sum and count values for key
      }
      //Emit Calculated Averages
      emit(stationId,[tmaxAvg,tminAvg])
}
```

b. Combiner

```
//Mapper
Map(record){
      //Map reads one record at a time
      //every record reads as "StationId,Date,TMAX/TMIN,Value,…."
      //stationData contains running sum and count for TMAX & TMIN
      //if record specifies TMAX, add value to stationData
      //or if record specifies TMIN, add value to stationData
      //Emit Data with StationId as Key
      emit(stationId,stationData)
}
//Combiner
Combine(stationId,stationDatas){
      //initialize variables for combining sums and counts
      For each stationData in stationDatas[]{
            //combine all sum and count values for key
      }
      //Emit combined data
      emit(stationId,combinedStationData)
}
//Reducer
Reduce(Key,combinedStationDatas[]){
      //initialize variables for accumulating sums and counts
      For each stationData in combinedStationDatas[]{
            //accumulate all sum and count values for key
      }
      //Emit Calculated Averages

      emit(Key,[tmaxAvg,tminAvg])
}
```

CS 6240 Parallel Data Processing in MapReduce - 02
Homework 2 Report
Dheeraj Joshi

c. InMapper Combiner

```
//Mapper
Setup(){
      //Initialize HashMap H
}
Map(record){
      //Map reads one record at a time
      //every record reads as "StationId,Date,TMAX/TMIN,Value,…."
      //stationData contains running sum and count for TMAX & TMIN
      //if record specifies TMAX, add value to stationData
      //or if record specifies TMIN, add value to stationData
      if(H.contains(stationID))
            //combine new values to stationData
      else
            H.add(stationID,stationData)
}
Cleanup(){
      //initialize variables for combining sums and counts
      For each entry in HashMap H{
            //Emit combined data
            emit(stationID,combinedStationData)
      }
}

//Reducer
Reduce(stationID,combinedStationDatas[]){
      //initialize variables for accumulating sums and counts
      For each stationData in combinedStationDatas[]{
            //accumulate all sum and count values for key
      }
      //Emit Calculated Averages
      emit(stationID,[tmaxAvg,tminAvg])
}
```

CS 6240 Parallel Data Processing in MapReduce - 02
Homework 2 Report
Dheeraj Joshi

Pseudocode for 2:

The secondary sort is used in cases where we need to sort records based on a certain value other than the key value. The mapper processes each record line by line for its input chunk and generates the output for each line encountered. The key class implements the WritableComparable interface and therefore, has a comparing function that can be defined according to our criteria using the stationId and year. The partitioner returns a partition number based on the hashcode of the stationId in our case and is divided by the number of reduce tasks determined by the Hadoop master. These key,value pairs are then grouped based on the grouping comparator and and are sent as a group to the reducer. The key year values do change in the iteration of values as we have grouped them on the stationId. Therefore, although sorted in terms of the year, the keys are iterated for all years and accumulated to get the final desired output. The sequence of these values is trivial because of the key comparator that ensures that the values are sorted for all years of one particular stationId.

```
//WritableComparable Key Class
Class Key{
      String stationID;
      Int year;
}

//Mapper
Map(record){
      //Map reads one record at a time
      //every record reads as "StationId,Date,TMAX/TMIN,Value,…."
      //stationData contains running sum and count for TMAX & TMIN
      //if record specifies TMAX, add value to stationData
      //or if record specifies TMIN, add value to stationData
      //Contruct Object Key with stationID and year and emit
      emit(Key,stationData)
}

//Partitioner
getPartition(Key,int numOfPartitions){
      //partition data for each reducer on stationID
      return(Key.stationID.getHashCode()*INTEGER.MAX_VAL)
      % numOfPartitions;
}

//KeyComparator
compare(Key){
      //compare stationID's
      int res = stationID.compareTo(Key.stationID)
      if(res==0){
            //if stationId's are same, compare year
            return year.compareTo(Key.year)
      }
      return res;
}
```

```
//GroupingComparator
compare(Key key1, Key key2){
      //compare to groupby similar stationID's
      return key1.stationID.compareTo(key2.stationID);
}
```

```
//Reducer
Reduce(Key,stationDatas[]){
      //initialize an array list for keeping data for all years
      list = []
      //initialize variables for accumulating sums and counts
      //Keep track of current year
      curr_year=Key.year;
      For each stationData in stationDatas[]{
            if(curr_year != Key.year){
                  //Calculate averages and add to list
                  list.add([curr_year,tminAvg,tmaxAvg])
                  //update current year from the key
                  curr_year=Key.year;
                  //reset all accumulators
            }
            //accumulate all sum and count values for key
      }
      //For last year with one object
      if(tmaxsum !=0 || tminsum !=0){
            //Calculate averages and add to list
            list.add([curr_year,tminAvg,tmaxAvg])
      }
      //Emit Calculated Averages
      emit(Key,list);
}
```

Questions:

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

Ans: The Combiner is definitely called as observed in the JARSyslog at line #79. 8798241 records from the mapper were provided as input to the Combiner which in turn produces 223783 records after combining data for similar stationId's. The Reducer takes exactly 223783 records as input and produces 14135 records as the output. However, the number of calls made to the combiner per map task is hard to infer as the call to the combiner class can be made by the master at any point in the execution based on load and time based scenarios.

CS 6240 Parallel Data Processing in MapReduce - 02
Homework 2 Report
Dheeraj Joshi

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

Ans: The Combiner combined records that were emitted by different map tasks with the same stationId as key. This makes sure that lesser (combined) data is transferred across to the reducers and in turn makes it easy to for the reducers to process lesser number of values per key. As seen in the logs, NoCombiner did not combine any of the map output values and therefore, the total time taken by all reduce tasks was significantly higher than the time taken by the Combiner version.

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

Ans: Yes, it was quite effective as the local data for common stationId's was combined by the map task itself and thereby, eliminating the need for a Combiner task. Also, the reducer read lesser records and required lesser time to calculate the averages for the output. The NoCombiner is a granular approach and doesn't use a lot of heap space but sends a lot of data across to the reducers. On the other hand, the InMapperCombiner uses some of the heap space but makes sure that lesser data is sent across to the reducers and therefore, lesser time is taken to process and write the output.

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

Ans: The Combiner is also called as an intermediate reducer for combining map output records and does a better job than having no combiner mechanism at all for recurring data. However, the call to a Combiner is never guaranteed. Sometimes, the key, value pairs might be stored locally and combined later by Hadoop thereby, increasing the cost of I/O tasks. The combining might also be restricted to data that exists in the same buffer and therefore, it can never be assumed to combine all possible records from the map output records. The InMapperCombiner does use some of the heap space but does make sure that the local records are combined at the map task level and considerably reduces the overhead of sending large data and processing multiple key,value pairs at the reduce task level. Therefore, the InMapperCombiner offers better control as well as makes it certain that a minimum number of key,value pairs are sent across to the reducers for processing and writing the output.

CS 6240 Parallel Data Processing in MapReduce - 02
Homework 2 Report
Dheeraj Joshi

5. How do the running times and accuracy of these MapReduce programs compare to the sequential implementation of per-station mean temperature? Modify, run, and time the *sequential* version of your HW1 program on the 1991.csv data. Make sure to change it to measure the *end-to-end* running time by including the time spent reading the file. Tip: Modify your code to read and process the data line by line (i.e., instead of reading it all into memory). Finally, compare the MapReduce output to the sequential program output to verify and report on its correctness.

Ans: The running time for sequential version (33 seconds recorded) when the input file i.e 1991.csv is read line by line is much lower than the MapReduce programs (64 seconds recorded on average). This is because of the additional Hadoop setup overhead of breaking the files and executing in parallel across several workers. Also, according to the observations, the accuracy of the output is the same for both the versions of the program.

Running Times:

Program 1:

a. NoCombiner: *Run 1 – 69 seconds Run 2 – 72 seconds*
b. Combiner: *Run 1 – 66 seconds Run 2 – 69 seconds*
c. InMapperCombiner: *Run 1 – 62 seconds Run 2 – 63 seconds*

Program 2:

Total running Time: 45 seconds