Assignment 2
Harsh Shukla
Report
CS6240 12628 Parallel Data Processing SEC 01
Weather Data with Different Design Mapreduce Patterns.

Map Reduce Algorithms:

Approach 1:

No combiner:

The given program is divided into 2 classes:

1) Station
2) Combiner.

I have a **station class** which implements writable interface.

The variables are:

- Text station id;
- Double writable value
- Text type
- IntWritable numbers

```
i.    Check for arguments i.e. is the number of arguments provided matches to that required
      // if arg.value != desiredvalue;
      // error
      // else proceed
ii.   Configure the Hadoop job:
iii.  Setup files for input and output files.
iv.   Mapper:
      Map (stationed, record)
            // parse the values for stations from csv file (temperature data and stations)
            // if (station detail == tmax(Type)
            {
                  set value of Tmax to the station detail
                  context.write(word,obj)
            }
            // else if (station detail == tmin(Type)
            {
                  set value of Tmin to the station detail
                  context.write(word,obj)
            }
v.    Reducer:
            //initialize all the temperature data to zero
            // for (station data:records)
            //accumulate all the sums and counts values for a given key according to the type
                  (maximum and minimum)
            // Calculate average.
            // emit the results.
            // context.write(key,nullableWritable)
```

****

Approach 2:

Combiners:

The given program is divided into 2 classes:

1) Station
2) Combiner.

I have a **station class** which implements writable interface.

The variables are:

- Text station id;
- Double writable value
- Text type
- IntWritable numbers

I have a **combiner class** which performs all the required tasks:

```
i.    Check for arguments i.e. is the number of arguments provided matches to that
      required
      // if arg.value != desiredvalue;
      // error
      // else proceed
ii.   Configure the Hadoop job:
iii.  Setup files for input and output files.
iv.   Mapper:
      a. Map (station id, record)
      // parse the values for stations from csv file (temperature data and stations)
      // if (station detail == tmax(Type) {
              set value of Tmax to the station detail
              context.write(word,obj)
      }
      // else if (station detail == tmin(Type) {
              set value of Tmin to the station detail
              context.write(word,obj)
      }
v.    Combiner:
      a. combiner extends the reducers.
      //combiner(key,records)
      //initialize the temperature(tmax,tmin et al) values to 0
      // for all (station data : records)
      // combine values for similar keys
      // if temp.getType == Tmax
      //increment value of maximum temperature by 1 (maximum temperature +=1)
      // else if temp.getType == Tmax
      //increment value of minimum temperature by 1 (minimum temperature +=1)
      // Pass the accumulated values to object or type record.
      //  context.write(key,max)
      // context.write(key,min)

vi.   Reducer:
Reduce(key,records)
// initioalize temperature(max and min tempratures) values to 0;
// for each (data: records)
// accumulate all the sum and count values for a given key
// Calculate averages for maximum and minimum tempratures.
//emit the values
// context.write(key,nullableWritable)
```

****

Approach 3:

InMapper:

The given program is divided into 2 classes:

1) Station
2) Combiner.

I have a **station class** which implements writable interface.

The variables are:

- Text station id;
- Double writable value
- Text type
- IntWritable numbers
- Check for arguments i.e. is the number of arguments provided matches to that required

```
        // if arg.value != desiredvalue;
        // error
        // else proceed
```

- Configure the Hadoop job:
- Setup files for input and output files.

i. **Mapper**

The class mapper implements the in mapper taking station id as key and station object as value.

```
    // map(stationidkey, station)
    // iniotialize hashmap as the accumulate ing data structure
                private HashMap<String,Station> records;
    // parse the values for stations from csv file (temperature data and stations)
        // if (station detail == tmax(Type) {
                set value of Tmax to the station detail
                update the records
                // records.put(word.toString(), station);

        }
        // else if (station detail == tmin(Type) {
                set value of Tmin to the station detail
                update the records
                // records.put(word.toString(), station);
        }
```

ii. Cleanup

```
        //initialize variables for combining sums and counts
                For each entry in HashMap H{
                //Emit combined data
        emit(stationID,records)
        }
```

iii. Reducer

```
    Reduce(stationId,records)
    // initialize temperature(max and min tempratures) values to 0;
    // for each (data: records)
    // accumulate all the sum and count values for a given key
    // Calculate averages for maximum and minimum tempratures.
    //emit the values
    // context.write(key,nullableWritable)
```

****

Secondary Sort:

Explanation:

The records are emitted according to station ID i.e. in ascending order. I have compared the station ID, if the stationID's are equal then I compare the years. Grouping Comparator Sorts in the increasing order of StationId and sends it to the reducer. As we are using Secondary Sort, all the inputs to the station id in the composite key is going to be same for same reduce while the input for the year in composite key being in ascending order too.

Therefore, we don't have to do that creating our own data structure and have used this MapReduce functionality to sort.

The program implements secondary sort for the weather data analysis. It does the in mapper combining as this shall improve results and efficiency.I have also used Key comparator and group comparator .

Algorithm:

The given program is divided into 2 classes:

1) Station
2) Combiner.
3) Year Station

I have a **station class** which implements writable interface.

The variables are:

- Text station id;
- Double writable value
- Text type
- IntWritable numbers

I have a **Year Station** class which takes station and data for multiple years from a given mapper.

- private Text stationId;
- private IntWritable year;

i. **Year Station**

```
// takes the station data and years from multiple years. And sets it
// compareto(obj)
    // take the data from multiple years
    // calculates the difference between years
    //        if (diffStationId == 0)
              return diffYear;
    else
              return diffStationId;
    }
```

ii. **Secondary Sort implementation:**

```
Check for arguments i.e. is the number of arguments provided matches to that required
                // if arg.value != desiredvalue;
                // error
                // else proceed
    Configure the Hadoop job:
    Setup files for input and output files.
```

iii. **Mapper:**

```
value.
// map(stationidkey, station)
// iniotialize hashmap as the accumulate ing data structure
            private HashMap<String,Station> records;
// parse the values for stations from csv file (temperature data and stations)
    // if (station detail == tmax(Type) {
            set value of Tmax to the station detail
```

```
                        update the records
                        // records.put(word.toString(), station);

                }
                // else if (station detail == tmin(Type) {
                        set value of Tmin to the station detail
                        update the records
                        // records.put(word.toString(), station);
                }

iv.     Cleanup
                //initialize variables for combining sums and counts
                        For each entry in HashMap H{
                        //Emit combined data
                emit(stationID,records)
                }
v.      Key Comparator(key):
        // keyComparator extends WritableComaparable
        // take data from the file
        // check for difference in stationID
        // check for difference in year by comparing present and next year.
        // if{
                station id difference == 0
                return yeardifference
              else {
                return stationdifference}
vi.     Grouping Comparator(key1,key2):
        Sorts in the station ids in increasing order.

vii.     //Reducer
        Reduce(Key,records
        // initialize variable for suma nd count to zero
        // for (temperature :list of stations)
                keep track of current year
                {
        if {
                current year == previous year
                add to acculmulator
                }
        else {
                calculate averages and append the values to the output
                }
        update the current year from the key.
        Reset all accumulators.
         If (Sum of maximum temprature !=0 ||sum of minimum temprature !=0){
        //Calculate averages and add to results
                append([curr_year,tminAvg,tmaxAvg])
                }
                //Emit Calculated Averages
                context.write(key.getStationId(), result);
        }
```
****

Performance Comparison:
1. Was the Combiner called at all in program Combiner? Was it called more than once per Map task?
Ans: Yes the combiner was called in the combiner program.The number of calls made is a bit hard to determine as this depends on different scenarios. The observations can be confirmed by the following logs:

```
Execution 1:
Map input records=8467426
                                  Map output records=2440578
                                  Map output bytes=100063698
                                  Map output materialized bytes=551810
                                  Input split bytes=565
                                  Combine input records=2440578
                                  Combine output records=35512
                                  Reduce input groups=3745
                                  Reduce shuffle bytes=551810
                                  Reduce input records=35512
                                  Reduce output records=3745
                                  Spilled Records=71024
                                  Shuffled Maps =45
                                  Failed Shuffles=0
                                  Merged Map outputs=45
                                  GC time elapsed (ms)=4090
                                  CPU time spent (ms)=46320
                                  Physical memory (bytes) snapshot=6914887680
                                  Virtual memory (bytes) snapshot=58404028416
                                  Total committed heap usage (bytes)=6029312000

Execution 2:
Map-Reduce Framework
                                  Map input records=8467426
                                  Map output records=2440578
                                  Map output bytes=100063698
                                  Map output materialized bytes=551810
                                  Input split bytes=565
                                  Combine input records=2440578
                                  Combine output records=35512
                                  Reduce input groups=3745
                                  Reduce shuffle bytes=551810
                                  Reduce input records=35512
                                  Reduce output records=3745
                                  Spilled Records=71024
                                  Shuffled Maps =45
                                  Failed Shuffles=0
                                  Merged Map outputs=45
                                  GC time elapsed (ms)=3980
                                  CPU time spent (ms)=47790
                                  Physical memory (bytes) snapshot=6917951488
                                  Virtual memory (bytes) snapshot=58391556096
                                  Total committed heap usage (bytes)=5961154560
```

2. Was the local aggregation effective in InMapperComb compared to NoCombiner?
Yes local aggregation was very effective to attain the required purpose. The combining was done at the mapper phase only and therefore we did not require a separate combiner. The data to send in in mapper is very less as compared to no combiner and therefore it has a direct relationship with time taken to process the data.
****

Running Times:

Design Pattern	Execution time1	Execution time2
No Combiner	56seconds	52 seconds
Combiner	54 seconds	54 seconds
In Mapper	52 seconds	52 seconds
Secondary Sort	58 Seconds	58 Seconds

Spark Scala Program:

1. No Combiner:

I have used Pair RDD for data representation. This is because:

a. Pair RDD stores the data in key –value pair and with the data sets that we have for the assignment the best way to represent them is in key value pairs.

Algorithm:

```
// load the input file and parse the data from csv.
// Create pair RDD for maximum and minimum temperature respectively.
// We will use groupby function to group the records based on sums and counts of maximum and
scala > . val variable = sc.textFile("input")
        .filter(field1 => fields(2) for maximum and minimum temperature
        .map(parse data))
//   minimum tempratures as well as stationID
// groupBy[K](f: (A) ⇒ K): immutable.Map[K, Repr]
//  combine and append all the values for min and max tempratures.
// give the data as output.
// averageTemprature.saveASTextFile("output") (for csv use saveAsCSVFile)
```

2. Combiner

I have used Pair RDD for data representation. This is because:

a. Pair RDD stores the data in key –value pair and with the data sets that we have for the assignment the best way to represent them is in key value pairs.

Algorithm:

```
// load the input file and parse the data from csv.
scala > . val variable = sc.textFile("input")
        .filter(field1 => fields(2) for maximum and minimum temperature
        .map(parse data))

// Create pair RDD for maximum and minimum temperature respectively
// Use combineByKey command to aggregate the temperatures and count in accordance to the
//stationID.
// val sumTemprature = temperatureCount.combinebyKey.
// We will receive the list of temperature sum and number of sums as per the station id .
// Count the average
//val averagetemprature == sumTemprature.map (currentValue/Sum)
//give the data as output
averageTemprature.saveASTextFile("output") (for csv use saveAsCSVFile)
```

3. InMapper

I have used Pair RDD for data representation. This is because:

a. Pair RDD stores the data in key –value pair and with the data sets that we have for the assignment the best way to represent them is in key value pairs.

Algorithm:

```
// load the input file and parse the data from csv.
scala >  . val variable = sc.textFile("input")
           .filter(field1 => fields(2) for maximum and minimum temperature
           .map(parse data))

// Create pair RDD for maximum and minimum temperature respectively.
// We will use reduceByKey function to group the records based on sums and counts of maximum and
scala >  . val variable = sc.textFile("input")
           .filter(field1 => fields(2) for maximum and minimum temperature
           .map(parse data))
//   minimum tempratures as well as stationID
     //  reduceByKey(function, [numPartition])
//  combine and append all the values for min and max tempratures.
// give the data as output.
// averageTemprature.saveASTextFile("output") (for csv use saveAsCSVFile)
```

4. Secondary Sort:

```
// load the input file and parse the data from csv.
scala >  . val variable = sc.textFile("input")
           .filter(field1 => fields(2) for maximum and minimum temperature
           .map(parse data))
```