A4 Review ankita-ushang-xia

Navya-Shabbir-Yu

10/17/2017

Code

- Missing structure in code:

It is a good practice to organize multiple java files into function specific sub-packages. And one have one entry point java file. This structure is clear and easy to understand. Eg:

```
.
+-- org.neu.pdpmr
|   +-- drivers
|       +-- Driver1.java
|   +-- mappers
|       +-- Mapper1.java
|   +-- reducers
|   +-- util
|   +-- Main.java
```

ActiveAirportAirlines.java

Two classes are embeded in a wrapper class. Unnecessary use of inner public static classes. Use namespaces instead.

Overall job:

Suggestion for saving 2 passes at whole corpus: - One job that will spit cleaned flight records. And send flight delays with frequency. - The same job can also create side-effect by storing flight count in memory for carrier and destinations as there are very few of them. Then in the cleanup stage of the job emit these records in the same stream but with a secondary sort key. This way in the second reduce only job you just check the first record has a secondary key say "-1", if it is there, that means following records are from top N airlines or destinations.

AverageMonthlyDelayMapper.java

```
@Override
protected void setup(Context context) throws IOException, InterruptedException {
   //Read the previous file and call the helper function to get top 5
   readFile("/part-r-00000",topAirlines,context);
   readFile("/part-r-00001",topAirports,context);
}
```

This is not the most reliable way to read the part files. Either loop through all the parts available in the directory or generate only one merged file using hdfs util.

ActivePartitioner

```
public int getPartition(Text text, IntWritable intWritable, int i) {
    if(text.toString().substring(0,2).equals("AL")) return 0;
    return 1;
}
```

Too many conversions. A better way is the have a secondary key and create a writable class for that key.

Consider replacing

```
Iterator it = airlineCount.entrySet().iterator();
while(it.hasNext()){
    Map.Entry pair = (Map.Entry)it.next();
    context.write(new Text(String.valueOf("AL_"+pair.getKey())),
                  new IntWritable((int)pair.getValue()));
}
```

with

```
for(Map.Entry pair: airlineCount.entrySet()){
    context.write(new Text(String.valueOf("AL_"+pair.getKey())),
                  new IntWritable((int)pair.getValue()));
}
```

or Java 8

```
airlineCount.foreach((k,v) -> {
    try{
        context.write(new Text(String.valueOf("AL_"+k)),new IntWritable((int)v));
    } catch (Exception e){}
});
```

ActiveAirportAirlines

At a lot of places, comments are added where they might not be entirely necessary. Here is an example of `reduce` method in the `ActiveAirportAirlines` class. Comments about exceptions and context can be clearly removed as they don't add any significant information in this case.

```
/**
 * The reduce function adds up the total count of the airport and the lines sent by the mapper
 * @param key       It is the key sent by mapper in format AL_<AIRLINE-NAME> or AP_<AIRPORT-NAME>
 * @param values    The list of values of counts for each key
 * @param context   The context of this jobs execution
 * @throws IOException
 * @throws InterruptedException
 */
public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {}
```

AverageMonthlyDelay

Inconsistent parathensis placement can be seen across the `AverageMonthlyDelay` class. Lines 63-67 vs 75-79 show while loop conditions being used with and without spaces. For the sake of consistency it can be suggested that one convention be followed throughout the entire code.

```
while (line != null) {
  String keyValue[] = line.split("\\t");  // Since the reducer writes with a tab DELIMITER
  t.put(keyValue[0].substring(3),Integer.parseInt(keyValue[1]));  //remove the "AL_" or "AP_"
  line = br.readLine();
}
```

vs

```
while(i.hasNext() && count<5){
  Map.Entry pair = (Map.Entry)i.next();
  target.add(pair.getKey().toString());
  count++;
}
```

Also this class seems to contain logical components that can be modularized for following a better code structure. All the code performing sanity checks and data cleaning (eg isNotCancelled, timezoneCheck etc) can be moved into a utility class.

The `isNotCancelled` method below can be refactored. Values are read multiple times, instead they can be read once and stored in variables local to the method (Eg Integer.parseInt(record[41]), Integer.parseInt(record[51])) are evaluated twice.

```
if(Integer.parseInt(record[41])>0 && Integer.parseInt(record[30])>0 &&
    Integer.parseInt(record[51])>0 && Double.parseDouble(record[42])>Integer.MIN_VALUE){
    // timeZone = CRSArrTime - CRSDepTime - CRSElapsedTime;
    int timeZone = Integer.parseInt(record[40]) - Integer.parseInt(record[29])
                    - Integer.parseInt(record[50]);
    int res = Integer.parseInt(record[41]) - Integer.parseInt(record[30])
                - Integer.parseInt(record[51]) - timeZone;
    if(res == 0)
      return true;
    else
      return false;
}
```

Possible refactoring can be as follows

```
int arrTime = Integer.parseInt(record[41]);
int deptTime = Integer.parseInt(record[30]);
int actElapsedTime = Integer.parseInt(record[51]);
if(arrTime > 0 && deptTime > 0 && actElapsedTime > 0 &&
    Double.parseDouble(record[42])>Integer.MIN_VALUE){
    // timeZone = CRSArrTime - CRSDepTime - CRSElapsedTime;
    int timeZone = Integer.parseInt(record[40]) - Integer.parseInt(record[29])
                    - Integer.parseInt(record[50]);
    if((arrTime - deptTime - actElapsedTime - timeZone) == 0) return true;
    return false;
}
```