Saavn MapReduce Project

PROJECT GOAL

The goal of the project is to identify 100 top trending songs from the Saavn Data File (of size 44 GB) for each day of December which lies between 25th December to 31st December (both days inclusive).

IMPLEMENTATION APPROACH

The project is implemented using Hadoop MapReduce framework available within the Cloudera Hadoop Platform.

The program has been tested on a Single Node Cloudera Hadoop Cluster installed on a Amazon EC2 Instance.

The implementation reads the Saavn Data File from the Amazon AWS S3 bucket (https://s3.amazonaws.com/mapreduce-bde/part-00000) and produces 07 outputs one for each day lying between 25th December to 31st December (both days inclusive). Each of the file contains 100 top trending songs for that particular day. These 07 days are referred to as Trending Days, further in the documentation.

Keeping in mind the I/O overhead associated with MapReduce framework and the associated volume of data, the program uses single MapReduce to solve the problem.

The program consists of the following classes:

Driver Class
com.upgrad.pgpbde.project.map.reduce.SaavnDriver

Mapper Class
com.upgrad.pgpbde.project.map.reduce.SaavnStreamingMapper

Partitioner Class
com.upgrad.pgpbde.project.map.reduce.SaavnStreamingPartitioner

Map Output Key Class
com.upgrad.pgpbde.project.map.reduce.SaavnStreamingKey

Map Output Value Class
com.upgrad.pgpbde.project.map.reduce.SaavnStreamingValue

PSUEDOCODE FOR THE PROGRAM

The Driver class takes the following inputs as command-line arguments:

1. Input File Location
2. Output File Location
3. Trending Window Duration (In Days)

It creates a Job with name EXTRACT_100_TRENDING_SONGS_FOR_SAAVN and sets the following Job configuration before it submits the job:

Trending Window Duration is the duration which has to be specified in days. For each trending day i.e. 25th December to 31st December, the trending window defines the scope of days within which the streamed songs have to be accounted for trending. For example, if the trending window is of 3 days then the songs streamed on

22nd , 23rd and 24th will be considered for 25th December,
23nd , 24rd and 25th will be considered for 26th December,
24th , 25th and 26th will be considered for 27th December, etc.

The Mapper class, the Partitioner class and the Reducer class are set as parameters to the job. The total number of reducers are set to be 7 i.e. one for each day for which the 100 top trending songs have to be identified.

Note: We will discuss about this later as we progress.
The Map Output Key and the Map Output Value are set. The Map Output Key is a Composite Key.

The File Input Format and the File Output Format are set which takes values from the command-line arguments to read a file and write the final output of the program to a set of files.

Note: The input path is location of the Saavn Data File available in public S3 bucket and the output path could be any path on a private / public S3 bucket. Before we proceed to the Mapper class, let’s understand the Map Output Key and the Map Output Value.

The Map Output Key is a composite key defined with a custom class which implements WritableComparable interface. The composite key for map consists of the following properties:

trendingDay - is the day for which 100 top trending songs have to be identified.

songId - identifies the song associated with the stream

The custom class overrides the *hashCode()* and the *compareTo()* method.

The Map Output Value is a custom class which implements the Writable interface. The value for map consists of the following properties:

trendingDay - is the day for which 100 top trending songs have to be identified.

songWeight – identifies the weight associated with an instance of a streamed song.

Note: The *trendingDay* property is going to be used in the Partitioner class.

The Mapper class reads each line of data from the data file as a String. Each line contains comma separated values which are tokenized using the Split(“,”) method which takes , (comma) as the separator.

The tokenized values of *songId*, *streamingHour* and *streamingDate* are stored in local variables. The *steamingDate* String is further tokenized based on the - (hyphen) separator to retrieve the streamingDay.

Before we move ahead, we want to make sure that the data retrieved from the file is correct and does not include any noise.

In order handle the varacity aspect of data, we check whether size of the streamed record is equal to 5 fields, the length of the streamingDate String array is equal to 3, the songId is not null, if yes then only we proceed to the mapper logic.

The next step is to calculate the *songWeight*. For a particular instance of a song streamed on a particular day, the weight of the song is calculated with the following formula:

*weight = streamingHour * streamingDay * 1*

Now we retrieve the value of the *Trending Window Duration* which was set as a parameter to the Job Configuration object in the Driver class.

Based on the Trending Window Duration, the trendDurationLowerLimit is calculated using the following formula:

trendDurationLowerLimit = FIRST_TRENDING_DAY – trendDuration

Based on this value, the *writeMap()* method is called which will perform the job of writing streamed songs data based on the Trending Duration.

For example, if the trendDuration is 3 (i.e. 3 days) then

trendDurationLowerLimit = 25 – 3 = 22

Now the *writeMap()* method will be called for all streaming days which fall between 22nd December and 31st December.

The following parameters are passed as input to the writeMap() method:

songId

streamingDay

songWeight

trendDuration

context

In the *writeMap()* method, we calculate the *startTrendingDay* and *endTrendingDay*.

startTrendingDay = streamingDay + 1
(i.e. the next day from the streamingDay)

For example, 26 = 25 + 1

endTrendingDay = streamingDay + trendDuration

For example, 28 = 25 + 3

This means the songs streamed on 25th December is considered for the trendingDays 26th December, 27th December and 28th December.

If startTrendingDay <= FIRST_TRENDING_DAY (i.e. 25th December) then

startTrendingDay = FIRST_TRENDING_DAY

Else If endTrendingDay > LAST_TRENDING_DAY (i.e. 31st December) then

endTrendingDay = LAST_TRENDING_DAY

Now we write the Mapper Output i.e. Map Output Composite Key and Map Output Value.

The trendingDay values in the 07 records will be 25, 26, 27, 28, 29, 30, 31 for each different Map Output Value.

The Partitioner class defines a HashMap daymap with 07 Key-Value pairs as:

25-0, 26-1, 27-2, 28-3, 29-4, 30-5, 31-6.

Each *trendingDay* value like 25, 26 and so on corresponds to a reducer number to which it belongs. This means all the Map Outputs with value 25 in its trendingDay property will be processed at reducer number 0 i.e. all the streams data to be analysed for 100 top trending songs for the day of 25th December will be available at the reducer number 0 for aggregation purpose.

The purpose of overriding the *hashCode()* and *compareTo()* methods in the Map Output Composite Key custom class was to make sure the partitioner distributes the keys correctly to the respective reducer.

The Reducer class or Reducer instance (at run-time) receives a specific set of map output pertaining to a particular trendingDay i.e. the day for which the 100 top trending songs have to be identified.

The Reducer aggregates (sums up) the count of total number of times a song was played within the Trending Duration by adding *songWeight* associated with a *songId*.

The Reducer class contains a HashMap *songStream* which is defined as a member of the class i.e. it will be accessible to the reduce() method on a particular reducer instance (like 0, 1, 2... 6).

Once the aggregated count of total number of times a song is played within a trending window is available, we want to add the stream to the HashMap with values songId and the total count.

On a particular reducer say reducer 0 which contains all the data for the trendingDay 25, we want to see if a song with a particular songId is already added or not. If not, we will add the song with songId and its count.

If the song with the songId is already added, we want to check if the total count of stream within the trending duration is greater than or equal to the total count of stream which is already available in the HashMap.

If this is true, then the total count which is greater is updated for that particular songId in the HashMap.

At the end of processing all Keys and Iterable<Values> on a reducer instance for a trendingDay, the HashMap will contain all the songs which have seen positive increase in stream hits.

In the Reducer class, we have overriden the *cleanup()* method which is executed on each reducer after the *reduce()* method is executed.

Please Note: We are not performing the *context.write()* in the reduce method. Instead, it is performed in the *cleanup()* method.

After the *reduce()* method is successfully executed, the *cleanup()* method is executed. The HashMap containing the songId as Key and their counts as Value is sorted in descending order on value i.e. counts. The sorted HashMap is iterated in a loop to find the top 100 trendings songs.

Once these songs are identified, we will perform a *context.write()* in the *cleanup()* method for the top 100 songs with their songIds.

The Reducer Output Key is the songId and the Value is NullWritable i.e. we do not want to write any value as the output. We just want the Key i.e. the songId to be written to the file.

COMMANDS USED FOR PROGRAM EXECUTION

```
hadoop jar SaavnMapReduceProject.jar
com.upgrad.pgpbde.project.map.reduce.SaavnDriver s3a://mapreduce-
bde/part-00000 s3a://saavnmapreduceoutput/output 3
```

(The above command assumes that the SaavnMapReduceProject.jar is available in the same location from where the command is executed)