ASSIGNMENT 5

PROBLEM STATEMENT:

Dataset is sample data of songs heard by users on an online streaming platform.

Description of data set attached in musicdata.txt is as follows: -

1st Column - UserId

2nd Column - TrackId

3rd Column - Songs Share status (1 for shared, 0 for not shared)

4th Column - Listening Platform (Radio or Web - 0 for radio, 1 for web)

5th Column - Song Listening Status (0 for skipped, 1 for fully heard)

TASK 1:

Write a MapReduce program to find the number of unique listeners in the data set.

TASK 2:

Write a MapReduce program to find out what are the number of times a song was heard fully.

TASK 3:

Write a MapReduce program to find out what are the number of times a song was shared.

SOLUTION:

A mapper class is written which would emit trackId and userIds and intermediate key value pairs. A constants class is created as shown below

```
public class LastFMConstants {

     public static final int USER_ID = 0;
     public static final int TRACK_ID = 1;
     public static final int IS_SHARED = 2;
     public static final int RADIO = 3;
     public static final int IS_SKIPPED = 4;

}
public class LastFMConstants {

     public static final int USER_ID = 0;
     public static final int TRACK_ID = 1;
     public static final int IS_SHARED = 2;
     public static final int RADIO = 3;
     public static final int IS_SKIPPED = 4;
```

Now, a mapper class is created which would emit intermediate key value pairs as (TrackId, UserId) as shown below

```
public static class UniqueListenersMapper extends
Mapper< Object , Text, IntWritable, IntWritable > {
        IntWritable trackId = new IntWritable();
        IntWritable userId = new IntWritable();

public void map(Object key, Text value,
    Mapper< Object , Text, IntWritable, IntWritable > .Context context)
        throws IOException, InterruptedException {

    String[] parts = value.toString().split("[|]");
    trackId.set(Integer.parseInt(parts[LastFMConstants.TRACK_ID]));
    userId.set(Integer.parseInt(parts[LastFMConstants.USER_ID]));
        if (parts.length == 5) {
        context.write(trackId, userId);
    } else {
        // add counter for invalid records
        context.getCounter(COUNTERS.INVALID_RECORD_COUNT).increment(1L);
    }
    }
}
```

We are using a counter here named INVALID_RECORD_COUNT , to count if there are any invalid records which are not coming the expected format.

Now a Reducer class is written to aggregate the results. Here we simply cannot use sum reducer as the records we are getting are not unique and we have to count only unique users. Here is how the code would look like-

```
public static class UniqueListenersReducer extends
    Reducer< IntWritable , IntWritable, IntWritable, IntWritable> {
    public void reduce(
        IntWritable trackId,
        Iterable< IntWritable > userIds,
        Reducer< IntWritable , IntWritable, IntWritable,
        IntWritable>.Context context)
        throws IOException, InterruptedException {
        Set< Integer > userIdSet = new HashSet< Integer >();
        for (IntWritable userId : userIds) {
        userIdSet.add(userId.get());
        }
        IntWritable size = new IntWritable(userIdSet.size());
        context.write(trackId, size);
    }
}
```

Here we are using Set to eliminate duplicate userIds. Now we can create the driver class as follows-

```
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: uniquelisteners < in > < out >");
            System.exit(2);
        }
        Job job = new Job(conf, "Unique listeners per track");
        job.setJarByClass(UniqueListeners.class);
        job.setMapperClass(UniqueListenersMapper.class);
        job.setReducerClass(UniqueListenersReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        org.apache.hadoop.mapreduce.Counters counters = job.getCounters();
        System.out.println("No. of Invalid Records :"
                + counters.findCounter(COUNTERS.INVALID_RECORD_COUNT)
                        .getValue());
    }
```

COMMANDS USED AND OUTPUT:

-hadoop fs -ls / - Lists all the directories under root in HDFS

-hadoop fs -ls /output - Lists all the directories under the output directory

-hadoop fs -cat /part-00000 – Displays the result or content of the directory part-r-00000

Unique Listeners –

```
111117|223|0|1|1
```

Song was fully heard-

```
111117|223|0|1|1
```

Number of times a song was shared-

```
111113|225|1|0|0
111115|225|1|0|0
```

We can check the output of these files on the HDFS as well by browsing the filesystem by typing localhost:50700/

The entire program is stored in MR5.txt