CASE STUDY 1 **MOVIE RATING**

In this case study, we have a movie lens dataset, which contains the following data file:

1. genome-scores.csv

2. genome-tags.csv

3. links.csv

4. movies.csv

5. ratings.csv

6. readme.csv

7. tags.csv

From above datasets, we will going to load movies.csv and ratings.csv file in HDFS folder.

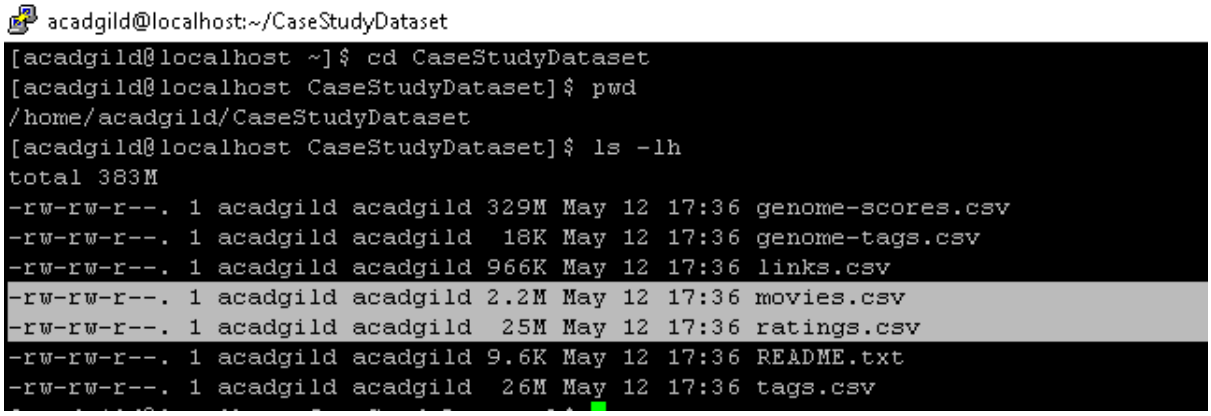
From these two datasets, we will try to implement the following use cases-

1. What are the movie titles that the user has rated.

2. How many times the user has rated a movie.

3. In use case 2 above, what is the average rating given for a movie.

We have a CaseStudyDataset folder contains movies.csv and ratings.csv file in acadgild home directory



Make a directory Movie_rating in HDFS folder

acadgild@localhost:~

```
[acadgild@localhost ~]$ hdfs dfs -mkdir /Movie_Rating
18/05/12 17:44:26 WARN util.NativeCodeLoader: Unable to load native-hadoop library
[acadgild@localhost ~]$ hdfs dfs -ls /
18/05/12 17:44:35 WARN util.NativeCodeLoader: Unable to load native-hadoop library
Found 7 items
drwxr-xr-x   - acadgild supergroup          0 2018-05-12 17:44 /Movie_Rating
drwxr-xr-x   - acadgild supergroup          0 2018-05-12 15:10 /hbase
drwxr-xr-x   - acadgild supergroup          0 2018-04-21 15:14 /spark
drwxr-xr-x   - acadgild supergroup          0 2018-02-02 12:49 /sqoopout111
drwxrwx---   - acadgild supergroup          0 2018-02-09 11:35 /tmp
drwxr-xr-x   - acadgild supergroup          0 2018-02-24 15:49 /tweetdata
drwxr-xr-x   - acadgild supergroup          0 2018-04-27 22:48 /user
[acadgild@localhost ~]$
```

Load these data in HDFS folder Movie_Rating by using following commands:

hdfs dfs -put /home/acadgild/CaseStudyDataset/movies.csv /Movie_Rating
hdfs dfs -put /home/acadgild/CaseStudyDataset/ratings.csv /Movie_Rating
hdfs dfs -ls /Movie_Rating

acadgild@localhost:~

```
[acadgild@localhost ~]$ hdfs dfs -put /home/acadgild/CaseStudyDataset/movies.csv  /Movie_Rating
18/05/12 17:51:32 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
[acadgild@localhost ~]$
[acadgild@localhost ~]$
[acadgild@localhost ~]$
[acadgild@localhost ~]$ hdfs dfs -put /home/acadgild/CaseStudyDataset/ratings.csv  /Movie_Rating
18/05/12 17:51:43 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
[acadgild@localhost ~]$
[acadgild@localhost ~]$
[acadgild@localhost ~]$
[acadgild@localhost ~]$ hdfs dfs -ls /Movie_Rating
18/05/12 17:51:54 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
Found 2 items
-rw-r--r--   1 acadgild supergroup    2283388 2018-05-12 17:51 /Movie_Rating/movies.csv
-rw-r--r--   1 acadgild supergroup   25389624 2018-05-12 17:51 /Movie_Rating/ratings.csv
You have new mail in /var/spool/mail/acadgild
[acadgild@localhost ~]$
```

These two data set contains following details:

Now we use map-reduce program to find all the use-case of these data.

movies.csv: movieID, title, genres

ratings.csv: userID, movieID, rating, timestamp

To calculate above use cases, we need movieID, titles from movies.csv and movieID, rating from rating.csv.

We will performing map-reduce programming by using **reduce-side join** operation to get desire results.

We have a separate mapper for each of two datasets i.e. one mapper for movies data input and one mapper for ratings data input.

Mapper program for movies data

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MoviesMapper extends Mapper<LongWritable, Text, Text, Text>
       {
           public void map(LongWritable key, Text value, Context context)
              throws IOException, InterruptedException {
try
     {
          if (key.get() == 0 && value.toString().contains("movieId"))
       {
            return;
         }
else
   {
            String record = value.toString();
            String[] parts = record.split(",");
            context.write(new Text(parts[0]), new Text("movies\t" + parts[1]));
         }
 }
catch (Exception e)
{
         e.printStackTrace();
      }
           }
}
```

- We will read the input taking one tuple at a time
- Then, tokenize each word in that tuple and fetch the movieID along with the titles.
- The movieID will be key of the key-value pair that mapper will generate.
- We will also add a tag “movies” to indicate that this input tuple is of movies type.
- Therefore, mapper for movies type will produce following intermediate key-value pair:
- Finally, the output of mapper for ratings will be of the following format:

Key – Value pair: [movieID, titles]

Mapper program for ratings data

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class RatingsMapper extends Mapper<LongWritable, Text, Text, Text>
{
                public void map(LongWritable key, Text value, Context context)
                                throws IOException, InterruptedException
{
        try
            {
              if (key.get() == 0 && value.toString().contains("userId"))
             {
                return;
              }
              else
                        {
                           String record = value.toString();
                           String[] parts = record.split(",");
                           context.write(new Text(parts[1]), new Text("ratings\t" + parts[2]));
                                }
            }
                  catch (Exception e)
           {
              e.printStackTrace();
            }

        }
}
```

- Like mapper for movies data, we will follow the similar steps for ratings data.
- We will fetch the movieID and ratings
- Here we use ratings as a tag
- Therefore, the movieID will be key of the key-value pair that the mapper will generate.
- Finally, the output of mapper for ratings will be of the following format:

  Key – Value pair: [movieID, titles]

The sorting and shuffling phase will generate an array list of values corresponding to each key. In other words, it will put together all the values corresponding to each unique key in the intermediate key-value pair.

Now, the framework will call reduce() method (reduce(Text key, Iterable<Text> values, Context context)) for each unique join key (movieID) and the corresponding list of values.

Then, the reducer will perform the join operation on the values present in the respective list of values to calculate the desired output.

Therefore, the number of reducer task performed will be equal to the number of unique movieID.

Reducer program with reduce-side join operation

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


public class MovieRatingReducer extends Reducer<Text, Text, Text, Text> {

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException

{
       String titles = "";
       double total = 0.0;
       int count = 0;
      System.out.println("Text Key    =>"+key.toString());

        for (Text t : values)
          {
                    String parts[] = t.toString().split("\t");
                    System.out.println("Text values =>"+t.toString());

                    if (parts[0].equals("ratings"))
                {
                     count++;
                    String rating = parts[1].trim();
                    System.out.println("Rating is =>"+rating);
                    total += Double.parseDouble(rating);
                                        }
           else if (parts[0].equals("movies"))
                    {
                                titles = parts[1];
                                        }
                                }

                     double average = total / count;
                    String str = String.format("%d\t%f", count, average);
                    context.write(new Text(titles), new Text(str));
                    }
          }
```

Our aim is to find what are the movies user has rated, how many times rated and what is the average rating of those movies.

Therefore, following steps will be taken in each of the reducers to achieve the desired output:

In each of the reducer we have a key & list of values where the key is nothing but the movieID.

The list of values will have the input from both the datasets i.e. titles from movies and rating from ratings.

Now, we will loop through the values present in the list of values in the reducer.

Then, split the list of values and check whether the value is of movies type or ratings type.

If it is of the ratings type, perform the following steps:

- Increase the counter value by one to calculate the no of times movie rated.
- Cumulatively update the amount value to calculate the total of rated value of that movie

On the other hand, if the value is of movies type, store it in a string variable, so that we will assign the titles as key in output key-value pair.

Next to get the average rating of a movie, we divide the total of rated value of a movie with no of time of that movie rated

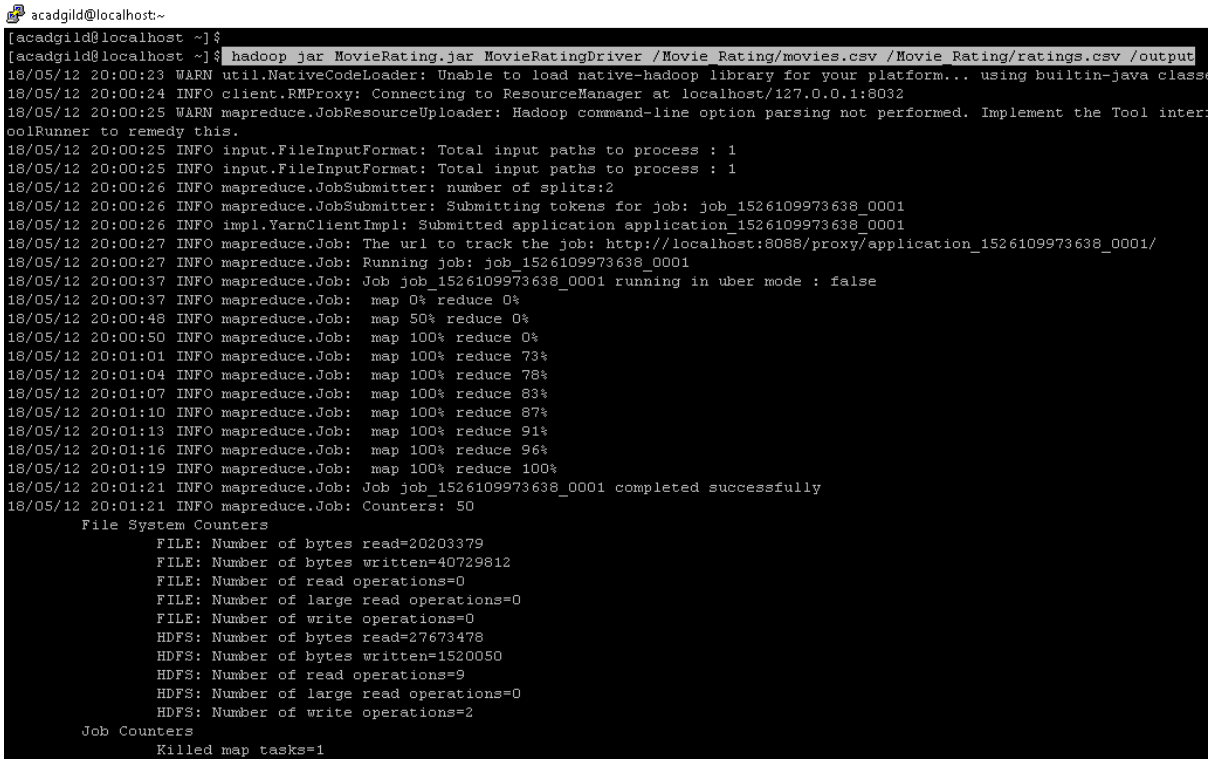
Finally, we will write the output key-value pair in the output folder in HDFS.

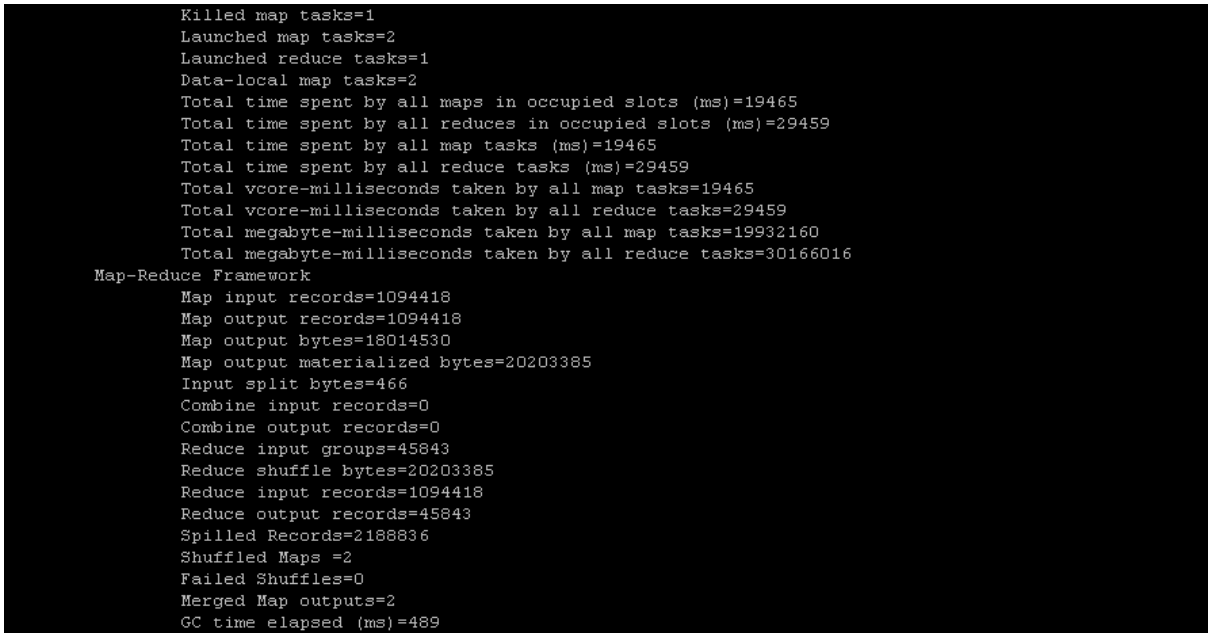
Driver program

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MovieRatingDriver {
                                        @SuppressWarnings("deprecation")
public static void main(String[] args) throws Exception
      {
           if (args.length != 3)
{
    System.err.println("Usage: MovieRatingDriver <input path1> <input path2> <output path>");
    System.exit(-1);
  }
   //Job Related Configurations

           Configuration conf = new Configuration();
           Job job = new Job(conf, "MovieRatingDriver");
           job.setJarByClass(MovieRatingDriver.class);

//Since there are two input, specifying two input path, input format and mapper

           MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, MoviesMapper.class);
           MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, RatingsMapper.class);

//Set the reducer

           job.setReducerClass(MovieRatingReducer.class);


 //set the out path

           Path outputPath = new Path(args[2]);
           FileOutputFormat.setOutputPath(job, outputPath);
           outputPath.getFileSystem(conf).delete(outputPath, true);

 //set up the output key and value classes

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(Text.class);
    //execute the job
   System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Now we export the above class files into a .jar file and save it by name MovieRating.jar and run the map-reduce by following Hadoop commands:

hadoop jar MovieRating.jar MovieRatingDriver /Movie_Rating/movies.csv /Movie_Rating/ratings.csv /output

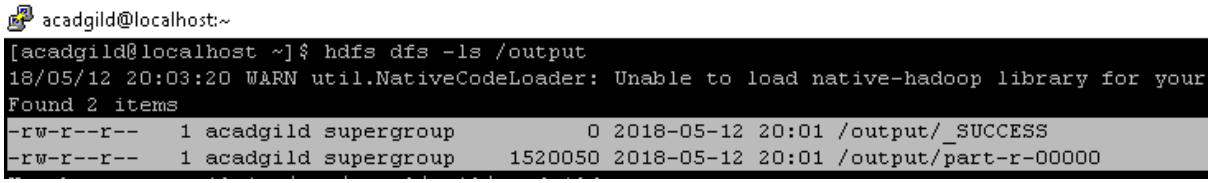


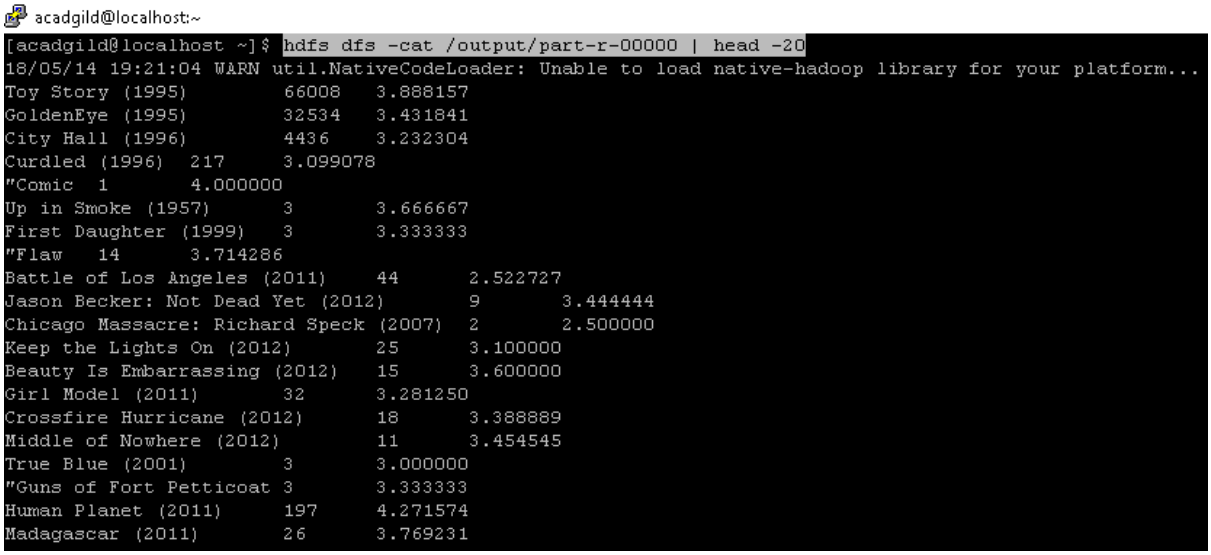


```
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=489
                CPU time spent (ms)=34100
                Physical memory (bytes) snapshot=701927424
                Virtual memory (bytes) snapshot=6259212288
                Total committed heap usage (bytes)=509083648
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=0
        File Output Format Counters
                Bytes Written=1520050
```

Output-

Output file is present in the /output path of HDFS as shown below in screen shot.





Above screen, shot shows the top-20 movies with their title, no. of times movie rated and average rating.