CS 6240: Project Final Report

Team Members

Name	Email	Class
Frank Philip	philip.f@husky.neu.edu	Tues 6 – 9 PM
Puneeth Nettekere Rangaswamy	nettekererangaswam.p@husky.neu.edu	Fri 1:35 – 3:15 PM
Umang Mehta	mehta.u@husky.neu.edu	Fri 1:35 – 3:15 PM

Introduction

We were interested in doing different analytics on this popular data set that would seem interesting to a movie buff or maybe even the big brass at Netflix! With this objective, we set out to analyze our data and then decided on the different technologies we would use to achieve the results.

Our first task was to identify the top 5 movies of each year. For the years ranging from 1890 to 2005, we aimed to identify the top 5 movies of a year based on the average rating of each movie. This would be an interesting task as we hope to see which were the best movies released during different years. Years closer to the 20th century, with more movie releases, would reveal some interesting competition while movies in the earlier 19th century would reveal classics that are still considered worthy by an audience that rated them from1998 to 2005. We slightly deviated from our initial goal as our data set revealed only a handful of movies for the earlier part of the 19th century. For this computation we leveraged HBase and Hive and performed a comparison of their execution. Since HBASE cannot compute joins, we used intermediate join output from the plain MapReduce program, which were stored in the HBase tables and retrieved later to compute the top five movies We also wrote a plain MapReduce program to accomplish the task. The details of the implementation and comparison between the two will be provided in the further sections.

Our second task we created a system based on clustering the movies based on the average rating given by all the users for a particular movie. We clustered movies based on average rating. We believe this is interesting as we basically created a recommendation system based on the clusters created. Hence the user will have a chance to utilize the output of this MapReduce job to find out movies which have similar average rating as they lie in the same cluster.Using K Means clustering we aim to create multiple clusters concurrently in MapReduce program

Our final task was to find the opening strength for the movies. Instead of going with the ratings of the movie, we were interested in analyzing how well the movie faired when it launched. This would show the “opening strength” of a movie and is irrespective of how users reviewed it over the years. This task was executed on both HBase and PigLatin. In the HBase execution we had to setup a MapReduce as a helper task to our HBase tables and related computation. The population of the tables was painfully long with just half the data set clocking 1.5 hours of execution time on AWS. PigLatin also came with its own challenges where we had had helper tasks to convert the input files into CSV format so as to use them effectively with Pig.

Dataset

We worked with the Netflix Prize data set. We obtained the data set from http://www.lifecrunch.biz/archives/207, a link provided in the project proposal requirements. We dealt with two data sets – training data set & movie file description data set. The movie rating files contains over 100 million ratings from 480 thousand randomly chosen, anonymous Netflix customers over 17770 movie titles. The data was collected between October 1998 and December 2005 and reflect the distribution of all ratings received during this period.

The training data set contains 17770 files, one per movie. The first line of each file contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the following format: "CustomerID,Rating,Date"

- 'MovieID's range from 1 to 17770 sequentially
- 'CustomerID's range from 1 to 2649429, with gaps. There are 480189 users
- 'Rating's are on a five star (integral) scale from 1 to 5
- Date of rating have the format YYYY-MM-DD

Movie information in "movie_titles.txt" is in the following format:
"MovieID,YearOfRelease,Title"

- 'MovieID' do not correspond to actual Netflix movie ids or IMDB movie ids
- 'YearOfRelease' can range from 1890 to 2005 and may correspond to the release of corresponding DVD, not necessarily its theaterical release
- 'Title' is the Netflix movie title and may not correspond to titles used on other sites
- Titles are in English

The data was could not be used in the form we received it in. The training data set consisted of multiple text files, which presented a challenge when working with Hive and PigLatin. We wrote a python program to convert the data set files into a single well-formed CSV file. Even when working with MapReduce and HBase, the multiple text files proved to be a hurdle as uploading 17770 files onto S3 took a long time. We had to split up the data set into smaller chunks so as to upload in parallel. This also proved to advantageous in the HBase program as populating the entire data set would have taken more than 3 hours and we were able to upload half of it to do the necessary computations.

Samples

Training Data set
1:
1488844,3,2005-09-06
822109,5,2005-05-13
885013,4,2005-10-19
30878,4,2005-12-26

Movie File Description set
1,2003,Dinosaur Planet
2,2004,Isle of Man TT 2004 Review
3,1997,Character
4,1994,Paula Abdul's Get Up & Dance
5,2004,The Rise and Fall of ECW

Technical Discussion

Task 1: Identify the top 5 movies of each year

- From the Netflix dataset described above, the task was to retrieve top 5 movies of each year.

  Using Hive
  To perform the task, we created two tables movie_reviews and movie_details. To populate movie_revies, we first had to convert all the available movie review files into a CSV file, using create_csv.py which will be provided in the source code.
  The following is the Hive query file used to perform this operation. Here reviewfile is the CSV file generated from all the ratings files and moviefile is the movie_titles.txt file provided by the dataset.

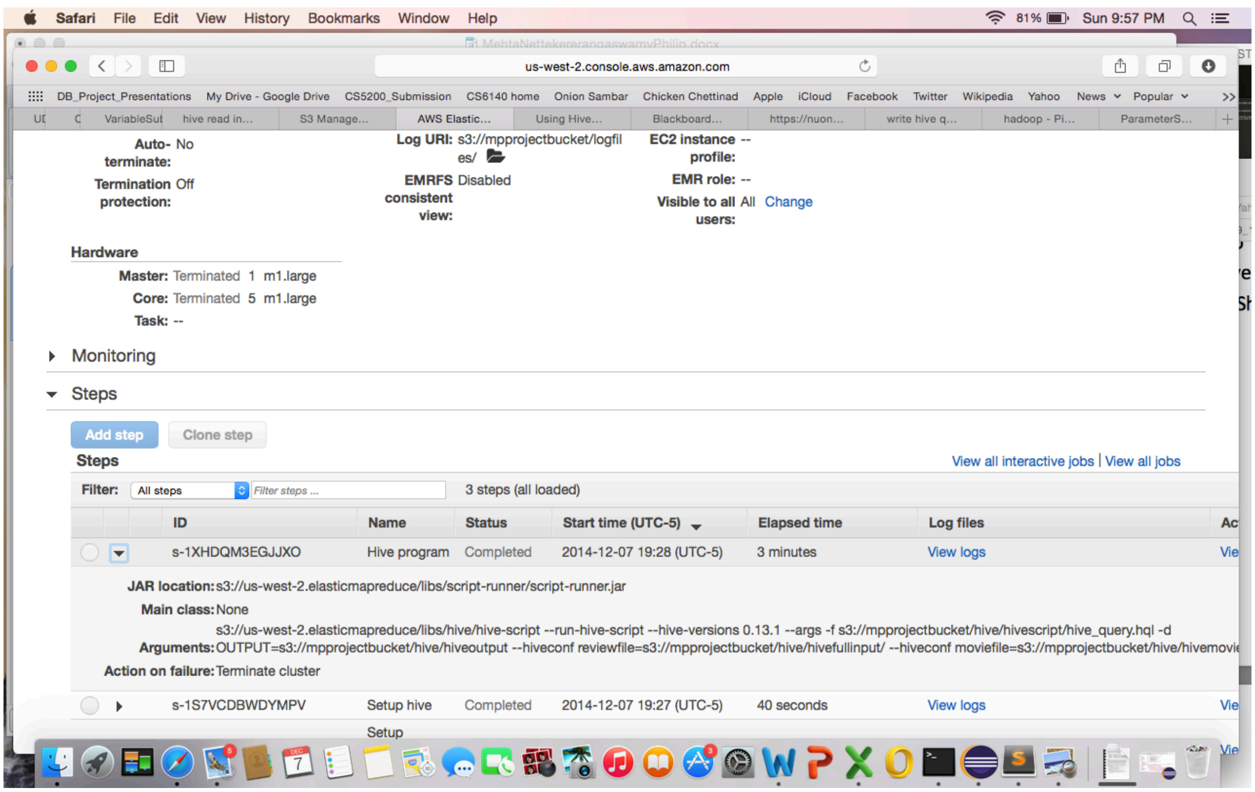
  ```
  //First create the table movie reviews
  create external table movie_reviews(movie_id INT,customer_id double,rating float,
  date TIMESTAMP) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
  LOCATION '${hiveconf:reviewfile}';

  //Create the table moviedetails
  create external table movie_details(movie_id INT,year_of_release String,movie_name
  String) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
  LOCATION '${hiveconf:moviefile}';

  //Query to compute top movies.
  select * from (select  m.year_of_release,
  m.movie_id,avg.avg_rating,m.movie_name,RANK() OVER (PARTITION BY
  ```

m.year_of_release ORDER BY avg.avg_rating desc) ranking from (select movie_id,sum(rating)/count(rating) avg_rating from movie_reviews group by movie_id order by movie_id,avg_rating) avg,movie_details m where m.movie_id=avg.movie_id) ratings where ranking <=5;

It was not easy getting the HiveQL file to run on EMR. Various issues were faced which will be explaned in the next few sections.
To run this query on EMR, we had to provide inputs as seen in the following screen shot



Plain Mapreduceprogram
As provided in the interim project report, the plain mapreduce program logic remains same, except that a custom petitioner has been added when finding the top five movies
Mapper1
  Emit(MovieID,averagerating)

Mapper2
    If(entry from mapper1)
        Emit(MovieID,"Average File"+rating);
    Else

```
        Emit(MovieID,”MovieFIle”+release year+movie name)

Reducer2
        //execute join logic
        emit(NULL,movieID+”@”,Rating+”@”,+Year+”@”+”Name
        //Used @ asdelimiter since movie names will have spaces and commas

Mapper3
        Emit((year,rating),(movie ID,name))

customPartinioner(key,values){
  year = key.getfirst();
                        if(year <1920){
                                return 0;
                        }
                        else if(year>=1920 && year <1940){
                                return 1;
                        }
                        else if(year>=1941 && year <1960){
                                return 2;
                        }
                        else if(year>=1961 && year <1980){
                                return 3;
                        }
                        else if(year>=1981 && year <2000){
                                return 4;
                        }
                        else if(year>=2000 && year <2020){
                                return 5;
                        }
}

KeyComparator
    //Sort by year first and then by rating in descending order

GroupComparator
        //Group by year first
        //Order by rating descending.

Reducer3
        Emit(NULL,Year+Rating+MovieID+MovieName)
```

Using HBASE

Using Hbase was not very efficient since the data needed to be populated had to be first joined using the plain mapreduce program.

The Hbase table was then created using a rowkey of year+rating+movieID and moviename in the column info.

After running the Hive program on the entire dataset, the sample output is provided below, in the format “year, movie ID, average_rating, movie name,serial number for that year”
The complete output is present in the task1_hive_output.txt file which is the stdout file in the logs of EMR job
2000 4238 4.554434413170473 Inu-Yasha 1
2000 7664 4.512657568496718 Gladiator: Extended Edition 2
2000 14302 4.459673246100392 The Sopranos: Season 2 3
2000 8226 4.405324587633838 Buffy the Vampire Slayer: Season 5 4
2000 1072 4.3984575835475574 As Time Goes By: Series 8 5
2001 7230 4.716610825093296 The Lord of the Rings: The Fellowship of the Ring: Extended Edition 1
2001 5018 4.538913362701909 Fruits Basket 2
2001 12834 4.527519289221416 Family Guy: Vol. 2: Season 3 3
2001 15296 4.513328357413233 Band of Brothers 4
2001 4427 4.473692145333937 The West Wing: Season 3 5
2002 7057 4.702611063648014 Lord of the Rings: The Two Towers: Extended Edition 1
2002 17085 4.48234947616443 24: Season 2 2
2002 8468 4.472177076077703 CSI: Season 3 3
2002 8116 4.467343562831555 The Sopranos: Season 4 4
2002 1418 4.464824120603015 Inu-Yasha: The Movie 3: Swords of an Honorable Ruler 5
2003 14961 4.723269925683507 Lord of the Rings: The Return of the King: Extended Edition 1
2003 8964 4.6 Trailer Park Boys: Season 4 2
2003 14791 4.6 Trailer Park Boys: Season 3 2
2003 13 4.552 Lord of the Rings: The Return of the King: Extended Edition: Bonus Material 4
2003 14240 4.5451207887760265 Lord of the Rings: The Return of the King 5
2004 3456 4.6709891019450955 Lost: Season 1 1
2004 9864 4.638809387521466 Battlestar Galactica: Season 1 2
2004 15538 4.605021432945499 Fullmetal Alchemist 3
2004 12398 4.592084006462035 Veronica Mars: Season 1 4
2004 7833 4.582389367165081 Arrested Development: Season 2 5
2005 3033 4.586363636363636 Ghost in the Shell: Stand Alone Complex: 2nd Gig 1
2005 16875 4.521739130434782 Ah! My Goddess 2
2005 7749 4.36312692630989 Curb Your Enthusiasm: Season 4 3

2005 11607 4.30578807731875 Hotel Rwanda 4
2005 8355 4.282574568288854 UFC 52: Ultimate Fighting Championship: Randy Couture vs. Chuck Liddell 5

Some of the interesting patterns are that Lord of the Rings was the top rated movie for three consecutive years 2003,2002 and 2001

Running the Hive program on AWS with 10 large machines took as less as 3 minutes.
We were not able to run on small machines because small machines use version 0.11.0.2 version while the program was written in 0.13.1. When run on small machines we received the error "FAILED: ParseException line 4:3 mismatched input ',' expecting) near 'avg' in subquery source "
While the same query ran on large versions with Hive version 0.13.1

A table with analysis of time between three tasks is provided below

Plain Map reduce	Hive	HBase
(Last Time): 800 Input files Config:1 small master, 10 small machines Time: 7 minutes 10 seconds		
5000 input files Config: 1 large master 10 large machines Time: 30 minutes	Entire input file Config: 1 large master 10 large machines Time: 3 minutes	Joined input file from first mapreduce program Config:1 large master 10 large machines Time: 1 minute for populate 1 minute for compute

Looking at the running times, Hive program turned out to be the best. It was easy to write up code. Once we figured out the way to run it on AWS, it ran quite fast and is the best solution in this case.

Although Hbase programs ran fast, most of the logic of joins had to be done in the map reduce program whose output is used to populate the hbase tables.

Task 2: Create a recommendation system

- **Purpose:**

Parallel K Means algorithm on the training dataset based on MapReduce. Concurrently creating and updating clusters based on the average rating by all the users for a particular movie. The goal is to devise an algorithm at the end of which all the movies in a particular range of rating lie in the nearest cluster. The value of k is 5 as the ratings are {1,2,3,4,5}

- **Data Preparation:**
  Initially we were trying to create an Inverted Index of input dataset but we later realized we could work with a new dataset with key as the Movie_Id and value as combination of average rating (the average of the total customer rating) for the movie with the above mentioned Movie_Id ,year and Movie_Name. We are reusing the joined dataset from the First Task.

  The joined dataset is organized as follows:
  - One Movie_ ID data per line,
  - One line is formatted as <Movie_Id>@<average_Rating>@<movie_Released_Year>@ <Movie_Name>(line components are separated by @)
  - For example, in this line
    1@3.7472527472527473@2003@Dinosaur Planet
    10@3.1774193548387095@2001@Fighter

    The pseudocode for performing the inverted index on the newly formed combined input file is provided below.
- **Pseudo Code**

```
 Mapper
 {Creating a HashMap cluster consisting of cluster ID and its related movie.
  The movie being another HashMap movieDetails.
  Creating a double array kCluster[] of length 5. //Represents the total number of clusters
formed

   public void setup()
        {Initializing the kCluster[] with random initial values;
         Initialize the HashMap  cluster.
         }
   public void map(Object key, Text value,Context context)
        {
         //Splitting the input value to obtain the rating and the Movie_Id
         //Calculating the distance matrix for new value of cluster by
         //Creating a list of difference between movie rating and cluster assigned rating
         //Code snippet for calculating the distance matrix

         List<Double> mod = new ArrayList<Double>();
         for(double d:kClusters)
       {
       mod.add(Math.abs(rating-d));
```

```
        }
        int index =mod.indexOf(Collections.min(mod));
        double minimum=kClusters[index];

         // Finding the cluster in which the movie with the least distance
         //Initialize the movieDetails HashMap
         //Code snippet for checking if the movieDetails is being created for the first time or
not

         HashMap<Integer,String> movieDetails;
        if(cluster.get(index) != null){
          movieDetails=cluster.get(index);
        }else{
          movieDetails= new HashMap<Integer,String>();
        }

     //Creating a StingBuilder object and updating both movieDetails and cluster HashMap
based on the   revised values
     //Calculate the average rating of movie in the cluster and supply the new revised
average rating to                   the kCluster[index(which is the index which is calculated
using the distance matrix)]
    //Hence on every map call I will have new average movie rating in the clusters

         public void cleanup()
        {//Iterate over both the HashMaps and emit
          Emit(Cluster_ID,(Average_movie_rating,Movie_Name))

}

 Reducer//will group Cluster_ID based on inherent Reducer property
  Emit(Cluster_ID,Movie_Name)
```

A sample output of the program is shown below:

1
Love Reinvented
Lost in the Pershing Point Hotel
.
.
2
Pitcher and the Pin-Up
WWE: Armageddon 2003
.
.
3
Dinosaur Planet

Isle of Man TT 2004 Review
.
4
Invader Zim
Aqua Teen Hunger Force: Vol. 1
.
5
.

- Interesting thing to be observed here is that the cluster number 5 which will have movies with rating close to 5 is least full as there are very less movies which have such high standards.These are eventually the best contenders for the movie which have been hit on box office.
- The parallel k-Means algorithm is implemented studying the approach followed in the below mentioned research paper.
  http://www.cs.ucsb.edu/~veronika/MAE/parallelkmeansmapreduce_zhao.pdf
- The source code is named as kMeans.java

Configuration	Time
5000 Input files Config:1 small master, 5 small machines	2minutes and 59 seconds
5000 Input files Config:1 large master, 5 large machines	30 seconds

- Initially we had created Inverted Index which had almost the same information but the Movie_ID was repeated for all the given customer_ID. As there are more than one customer who rate a particular movie. The current approach yielded better result as the movie_ID is unique and we are able to get the average rating from the Task 1.
- Also initially we had a map only task with an InMapperReducer but it often ran out of Heap space. Hence having a Reducer gave us an edge having all the movies lying in the same cluster reduce in the same call.
- As a future improvement we could use Weka to generate a recommendation based system in which if a User A rates Movies 1,2,3 and 4 and User B rates Movies 1,2 and 3.The system would recommend User B ,Movie 4.

Task 3: Analyze opening strength of movie

The approach was to obtain the launch date of the movie. Since the data set provided only the year of the release we decided to use the first review of the movie as the launch date as it more than likely to be close to the launch date. Then the close date is computed as 30 days from the launch date. The records are iterated and reviews within the close date are counted for every movie and presented as the opening strength of a particular movie.

In Reducer of NetflixCOM.java (compute) of HBase program

Pseudocode:

```
class MovieReducer (Text, Text, Text, IntWritable) {
	function reduce(key, values) {
		openingStrength = 0
		launchDate = values[0];

		// Closing date 30 days from launch
		closingDate = launchDate + 30;
		// Iterating over values and incrementing "openingStrength" only if review
		// date is before closing date
		emit(movieID, openingStrength)
	}
}
```

PigLatin program movie_open.pig

Code:

```
-- Join column having launch date with sorted movie records to filter by date
join_results = JOIN sorted_movie_records BY MovieID, movie_launch_record BY MovieID;
-- Filter records which are only 30 days after launch
filter_results = FILTER join_results BY DaysBetween($1, $3) < 30;
-- Parse and group filtered records by 'MovieID'
opening_strength_records = FOREACH filter_results GENERATE $0 AS MovieID, $1 AS
ReviewDate;
grouped_opening_strength = GROUP opening_strength_records BY MovieID;
```

The HBase source code files are NetflixCOM.java and NetflixPOP.java. The PigLatin source file is movie_open.pig

The task was executed on a 10 large machine core machines and 1 large master machine. The reason for only considering this configuration was because the population of the HBase table proved to be very time consuming. We were looking at 3 hours to populate the HBase table even with this configuration and therefore did not chose to run in a smaller cluster configuration. This problem was not faced in the PigLatin execution but we decided to keep the same configuration for consistency. Following are the performance results obtained from HBase and PigLatin executions of the task with half the data set (8000 movies):

Program	PigLatin	HBase
Population	Nil	1 hour 31 minutes
Computation	17 minutes	4 minutes

The reported execution times can be verified in the controller text files submitted for both HBase and PigLatin executions for Task 3. Though HBase clearly edges out PigLatin in the computation execution time, we believe PigLatin would be the most apt for this task. Populating the HBase table proves to be a large overhead for the computation and is clearly not scalable. Extrapolating the results for the full data set, PigLatin would complete the task in less than 40 minutes which is still less than 50% of the time taken by HBase to populate half the data set for its computation!

Setup challenges

Task 1:

The major set up challenge was to set up hive in local machine and then run the hive program on EMR

- On the local machine, hive uses a default warehouse at /user/hive/warehouse which does not exist on a mac local machine. To overcome this, a new warehouse location needs to be created in hive-site.xml in the conf folder of the hive installation as shown below

```
<configuration>
<property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/Users/puneeth/user/hive/warehouse</value>
  <description>location of the warehouse directory</description>
</property>
        </configuration>
```

- Once setup, tables and data can be loaded with ease.
- Now, to run on EMR, we needed to pass the input file paths as arguments to the hive program. This took a while to figure out and this can be done using the – hiveconf <property=value> parameter in the arguments and these parameters can be accessed using { hiveconf:property} in the hiveql file.
- Another issue faced was that the hiveql file provided in the source code did not run on small machines since they use Hive 0.11.02 and the program gave an error. SO, we had use only large machines with Hive version 0.13.1

Following is a useful link which helped:
http://blog.mustardgrain.com/2010/09/30/using-hive-with-existing-files-on-s3/

After all these challenges were overcome, the Hive program ran successfully on EMR with the full dataset. The log have been provided in task1_hive_stderr.txt and the final output in task1_hive_output.txt

Task 2:

We tried to use Weka for performing k-means clustering but it failed to provide accurate results as our input was restricted to only two attributes. We are mainly concerned with the average rating for a given movie along with its name. We have enclosed two jpeg files one with the implementation as testing with only 80 attributes and the corresponding visualized representation of the cluster.

Task 3:

Issues were faced in PigLatin. The program initially failed to parse a chararray which was in the right format. Weirdly, the program failed to parse only days from 24 to 30. This issue was fixed by explicitly specifying a format for the datetime.

Conclusion

In our project we have performed various concepts learnt in this course.
In task 1, we have implemented equijoin, Hbase populate and compute and Hive implementation on EMR. This output can be expanded to compute the top n movies in a year based on the customer rating and can be displayed to the user suggesting them to watch these movies.

In Task2, we have created clusters of movies with similar average ratings. This can be used as a recommendation system, where in a user can be suggested movies within the same cluster. If further attributes like movie genre, movie length are available, these can be grouped together and a more refined suggestion can be given to the user.

We believe Task 3 was a great task as we analyzed the performance of the movie irrespective of the quality, genre, launch date and target audience. Considering 30 days as the average box office running time for a movie, our objective was to analyze how strong the release was for the movie irrespective of other parameters. For instance, the movie may have been poorly rated but several people could have watched it when it released. This seems like a good analysis as production houses are more concerned about the bottom line and the money they reel in when a movie is released rather than audience response over a period of time.